



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Specification and Automated Reasoning for Datastructure Comprehensions

Bachelor's Thesis

Tierry Hörmann

October 2, 2018

Advisors: Prof. Dr. Peter Müller, Dr. Alexander J. Summers, Arshavir Ter-Gabrielyan
Department of Computer Science, ETH Zürich

Abstract

With quantified permissions, Viper added native support for encoding iterated separating conjunctions; a way of specifying datastructures in permission logics, particularly in separation logic, without imposing an order of traversal. For reasoning about the content of such datastructures, quantified permissions are, however, impractical, since they only allow conjunctions of direct assertions about the elements of the datastructure. A more common and general way to reason about datastructures is to create a comprehension over the content of a datastructure and to reason about program properties via such functions. In this thesis, we provide a system for specifying comprehensions for Satisfiability Modulo Theories (SMT) along with an axiomatization whose goal is to reason automatically about comprehension. We have implemented the technique as a native feature into the Viper framework for the verification condition generating verifier.

Acknowledgements

I would like to thank my supervisors, Alexander J. Summers and Arshavir Ter-Gabrielyan for their immense support. They were always available when help was needed and always helped out to solve the problem. Our long(!) weekly meetings were very interesting and provided good inputs for me to think about my progress from a different perspective. I would also like to thank Prof. Peter Müller for the opportunity to work in this fascinating research area. I really enjoyed working on the project.

Contents

Contents	iii
1 Introduction	1
1.1 Motivation	1
1.2 Comprehension	2
1.3 Background	2
1.3.1 Viper	2
1.3.2 Permission-based verification	3
1.3.3 SMT-solving	3
1.4 Notation	5
1.5 Related Work	5
2 Formal Specification	7
2.1 Components of a comprehension	7
2.2 Properties of the components	8
2.3 Definition of a comprehension	10
2.4 Definition of the value of a comprehension	10
3 The Singleton Case	13
3.1 Overview	13
3.2 Simplification of comprehension value definition	14
3.3 Automatic deduction	15
3.4 Incompleteness and matching loops	18
3.5 Summary	22
4 The General Case	25
4.1 Overview	25
4.2 Fundamental limitations	26
4.2.1 Automatic induction	26
4.2.2 Losing permission	27

CONTENTS

4.3	Automatic deduction	28
4.4	Incompleteness	31
4.4.1	Multiple heaps	31
4.4.2	Equal filters	31
4.4.3	Combining filters	32
4.5	Matching loops	36
4.5.1	General Axiom	36
4.6	Summary	40
5	Analysis	43
5.1	Matching Loop Freedom	43
5.2	Performance	45
5.3	Limitations	46
6	Implementation	49
6.1	Syntax	49
6.2	Well-Definedness checks	50
6.3	Assumptions	51
7	Evaluation	53
8	Conclusion and Future Work	55
A	Appendix	57
	Bibliography	61

Introduction

1.1 Motivation

Modern realistic programs usually make extensive use of datastructures to manage the program heap. For example arrays are one of the most fundamental components of many programming languages, and their correct usage is taught early on when learning such a language. For a program verifier to be useful in practice, it is therefore important to provide means of describing such datastructures, as well as techniques for reasoning about the content.

Consider the following example of a simple C function that swaps two values in an array:

```
void swap(int array[], int size, int slot1, int slot2) {
    assert(slot1 >= 0 && slot2 >= 0 && slot1 < size && slot2 < size);
    int tmp = array[slot1];
    array[slot1] = array[slot2];
    array[slot2] = tmp;
}
```

If we could prove now that the content of the array has not changed after a call to `swap`, we could also prove that a sorting algorithm, using `swap` as its only method of altering the array, did not change its content. Such results could then be useful for example in an optimizer, which could prove that the sum over the array has not changed after sorting, and therefore cache the value for the sum in a previous call to use it for subsequent calls.

The goal of this thesis is to create a technique for encoding comprehensions into SMT and provide support for comprehensions in Viper.

1.2 Comprehension

We define a *comprehension* as a function over a set of values, returning a value describing the content of this set according to the semantics of the function. Examples of comprehensions would be a sum over a set of integers, a product over a set of integers, a minimum of a set of comparable values, etc. We will distinguish between a *comprehension*, which is the operator per se, and the *value of a comprehension*, which is the value we get when evaluating the comprehension on a specific set of values.

We define a datastructure comprehension as the comprehension over the set of values of a specific datastructure. So, for example, a function which calculates the sum over all elements of an array would be a datastructure comprehension.

The approach developed in this thesis (and the technical solution) are more general than datastructure comprehensions. We will interpret a datastructure as an abstraction of a part of the program heap and will provide a mechanism for describing and reasoning about comprehensions over the mathematical set of values in a particular state of such a part of the program heap. We will therefore not restrict ourselves to specific datastructures, but allow a more general way of abstracting memory.

Note that since a comprehension depends on some function (e.g. the sum between two elements) and is itself a function, it carries the semantics of a higher-order function. This is noteworthy, since we encode comprehensions for SMT, which is in first-order logic.

1.3 Background

1.3.1 Viper

Viper [10] is a verification infrastructure developed at ETH Zürich. It has an intermediate verification language which supports permission-based reasoning natively, to argue about the program heap. The Viper intermediate language translates to SMT which is finally verified by Z3 [1]. In this thesis we will focus on supporting comprehensions for Viper. This will not only be relevant in the practical part, the implementation, but also in the theoretical part, since we will focus on creating axioms which will be encoded into SMT. Sometimes it will also be relevant in the theoretical part that we will use Z3 to solve SMT problems, because we will rely on certain features of SMT-solvers.

1.3.2 Permission-based verification

The Viper language is procedural and not object oriented. Objects can, however, still be modeled easily with fields and references. Fields in Viper are top-level declarations, however they don't have a unique value, but the value of a field is rather bound to a reference. The program heap in Viper is therefore modeled as a set of *heap locations*, which are fields of references.

Viper has support for permission-based reasoning, which means that for every state, every heap location has an assigned permission value between 0 and 1. 1 indicates full permission, while 0 indicates no permission. To read a heap location, there must be at least some permission to the location. To write to a heap location, there must be full permission. Indicating whether there is permission to a heap location can be done via the access predicate `acc(r.v, p)`, which indicates that there is a permission amount of p for heap location $r.v$. A Viper assertion can include access predicates. Instead of assuming and asserting, we *inhale* and *exhale* an assertion. Inhaling an assertion gains the permission from the access predicates, i.e. it adds the permission amounts to the current state, and asserts the remaining expression. Exhaling an assertion checks, whether the permission amounts of the access predicates are currently held and if so, gives them up, i.e. it removes the permission amounts from the state. It then checks whether the remaining expression currently holds. For encoding this permission model, Viper uses a *permission mask*, a map which maps from heap locations to permission amounts, to indicate which heap location has which permission amount in the current state. The state of a Viper program consists of the permission mask, as well as the heap, which is modeled as a map from heap locations to values.

To specify permission for multiple heap locations, we can use a feature called *quantified permissions* [9]. Quantified permissions are Viper's way of expressing *iterated separating conjunctions* [11], quantifiers over a set of assertions. We specify such an assertion as follows

$$\forall x : T. c(x) \rightarrow acc(e(x).f, p(x))$$

where $c(x)$ is a boolean expression, $e(x)$ a reference-typed, injective expression and $p(x)$ an expression denoting a permission amount.

1.3.3 SMT-solving

Since Viper reasons about verifiability by encoding a program into SMT and solving it with the SMT-solver Z3 [1], we will ultimately describe how to encode our axioms into SMT as well. We will rely our axioms on the *e-matching* and *trigger-matching* feature of Z3 for handling forall quantifiers, as well as on *skolemization* for handling existential quantifiers.

With e-matching, Z3 supports a so called *e-graph*, which is a graph, holding a node for every constant. In the e-graph, equality and inequality which were previously proven, are stored between constants. The e-graph makes it possible to quickly find all equivalent constants, which is necessary for trigger matching.

With trigger matching, Z3 takes a heuristic approach on forall quantifier instantiation. A trigger contains at least one term, which is a function application of an uninterpreted function. Every quantified variable must appear at least once in a term of the trigger. The quantifier gets instantiated, iff all the terms in the trigger occur somewhere in a clause without any quantified variables (the quantified variables are instantiated). If this is the case, we say that the quantifier got triggered by the respective terms. The respective instantiated variables in the found terms will then be used to instantiate the quantifier. A quantifier can have multiple triggers, to allow different terms to trigger the instantiation. In this case every match of one of the triggers instantiates the quantifier. As an example, we can have the following quantifier

$$\forall x : Int :: \{foo(x), boo(x)\} x < 10$$

where $\{foo(x), boo(x)\}$ denotes the trigger of the quantifier. As soon as the SMT-solver learns terms which contain the two expressions $foo(x)$ and $boo(x)$, for example with the term $foo(5) < boo(5)$, the quantifier gets instantiated and the SMT-solver learns $5 < 10$. If we now had an additional trigger for the axiom, for example $\{hoo(x)\}$, the quantifier would also be instantiated for a term like $hoo(foo(42))$, which gives us the instantiation $foo(42) < 10$.

This introduction to triggers might a reader conclude that it is good to just allow generous triggers, because then we cause more instantiations. However triggers must be selected very carefully, because trigger-matching gives rise to the problem of so called *matching loops*. Imagine we had an axiom

$$\forall x : Int :: \{foo(x)\} foo(foo(x)) < 10$$

The body of the quantifier introduces a new term, which matches the trigger of the same axiom. This is called a matching loop, and instantiation will go on, until we can prove or disprove something else, or the verifier gets terminated. If we would provide the more restrictive trigger $\{foo(foo(x))\}$, we would prevent the matching loop.

Skolemization is a technique of handling existential quantifiers. With skolemization, the quantified variables will be replaced by constants and the existential will be replaced with its body. If the existential is inside of a forall quantifier, the quantified variables will be instead replaced by new functions over the quantified variables of the forall quantifier. The function has the respective type of the variable.

Another important feature of Z3 to control instantiations of quantifiers, is that the right-hand-side of an implication will only be learned, if the left-hand-side could be proven. This means that when instantiating an implication, only terms from the left-hand-side will be used for trigger matching, until the right-hand-side will be learned by proving the left-hand-side. This feature will be extensively used in our axiomatizations.

1.4 Notation

When specifying SMT axioms, we will use the following mathematical notation to specify quantifiers.

$$\forall x : T. \{t_1(x)\}\{t_2(x)\} \text{ cond}(x)$$

where $\{t_1(x)\}$ and $\{t_2(x)\}$ are the triggers of the quantifier.

To refer to a heap location access, we will use the notation

$$r \cdot_h v$$

to refer to an access for reference r and field v in heap h .

To specify a mathematical function, we use the notation

$$f : D \rightarrow T$$

where f is the name of the function, D is the domain type and T is the type of the image.

To specify a comprehension value, we will use the following notation

$$c(h, f)$$

where h is a heap and f is a filter.

1.5 Related Work

In terms of supporting comprehensions for program verification, there are only very few previous works. We could actually only find one paper, which goes roughly in the same direction as this thesis. For the Spec# program verifier [4], there are implementations for sequence comprehensions, such as sum, count, product, min and max [6]. However this approach is in many ways less general than our approach. First of all the underlying datastructure is required to impose an order on its elements. Another major difference is, that the axiomatization in this approach is based on the idea of automated induction. The comprehension value can only be deduce if we know the value for the previous case, which is the case where the top or the

bottom of the iterable range, over which the comprehension was calculated, is decreased or increased respectively. For example to deduce the value of a comprehension for an array $[0 \dots n]$, the solver would need to have an assertion about the value of the sub-array $[1 \dots n]$ or $[0 \dots n - 1]$. This obviously restricts the order, by which algorithms can traverse the datastructure to still be verifiable. In addition to all of the above, this technique does not provide any explicit care to the heap. This means that for example features such as framing of comprehension values across different heaps won't be supported.

However the presented approach makes better effort in combining, in our definition different comprehensions by a certain axiom which proves that two comprehension values from possibly different comprehensions are the same, when they cover the same set of values and have the same basic operator. Also the provided approach is more lightweight than our approach.

Formal Specification

In section 1.2 we informally described what we mean by a comprehension. In this chapter, we try to formalize this intuition and provide formal definitions for every component. First, we will try to gather, what components we will need to provide a formal definition for, in order to be able to define a comprehension, and what properties those components must have. Afterwards, we will be able to define a comprehension formally. Next, we will define, what the value of a comprehension is, and what is needed to evaluate it.

2.1 Components of a comprehension

In this section we present the components of a comprehension and why they are considered as such. In the next section we will then provide restrictions on the components and explain the motivation behind those restrictions.

Binary operator In order to define a comprehension we must fix some core operator, which describes how the value of the comprehension will be calculated. For example, for a sum comprehension, this would simply be the binary plus between two elements, while for a minimum comprehension this would be a relation $<$. From now on we will use the symbol $*$ to refer to this operator and call it the *binary operator* of the comprehension. The reason why the operator is binary will be explained in the next section.

Receiver expression Our next component comes with the fact that arguing about the heap is in many cases unnatural with references. For example, arrays provide in many cases a more practical way of arguing about heap locations, which is why they are one of the most common datastructures in use. We can interpret such a domain shift as an abstraction of the heap. In Viper the usual way to encode such an abstraction for arrays is via the func-

tion $\text{loc}(a, i)$, which maps from arrays and integers to references. Since comprehensions are evaluated over heap locations, it makes sense to provide such a way of abstraction to the user for comprehensions as well. This is why we include an expression e , mapping from some argument of type A (which might be a tuple) to references, as a component of a comprehension. We call e the *receiver expression* and the type A the *argument type* of the comprehension, of which instances are called *comprehension arguments*. So, for example, $\text{loc}(a, i)$ could be a receiver expression, mapping from argument type $(Array, Int)$ to Ref .

Body In addition to the receiver expression e , we will need to know what values of the heap locations we want our comprehension to comprehend over. For this, the user will need to specify a field v , which is also a component of the comprehension. Together with the receiver expression, we now get a field-access expression $e.v$, which we call the *body* of the comprehension.

Unit Lastly, we will also need the user to specify a value 1 , which we call the *unit* of the comprehension. For example, for a sum comprehension, the unit will be 0 . What this value exactly is and why we need it will be explained in the next section.

2.2 Properties of the components

Arity of the operator $*$ In the previous section we already stated, that the core operator $*$ should be a binary operator, however did not explain why. This property is merely a design choice oriented towards properties of similar concepts, like the *fold* operator in functional programming. While it might be interesting to generalize the core operator of a comprehension to more general functions, we restrict ourselves here on the simpler (and probably more common) case of a binary operator.

Associativity and commutativity Note that in our informal description we stated that the comprehension ranges over a *set* of values. Since a set has no particular order, and the comprehension is a function (which means, that its value is unique for a given set), the order in which the value of the comprehension is calculated should not be relevant for the result. For the binary operator $*$, this means that it must be *associative* and *commutative*.

Identity Since $*$ is binary, we will also need to specify what happens when we evaluate a comprehension over an empty set of references or over a singleton set. This is where we introduce the unit 1 of a comprehension. It will fill the space of the missing arguments of the binary operator in this case.

So for the singleton set of references $\{e(a)\}$ (remember that $e(a)$ denotes the receiver expression on argument a , which is reference-typed) and the empty set, we would evaluate the comprehension with $e(a).v * \mathbf{1}$ and $\mathbf{1} * \mathbf{1}$ respectively. If we now require $\mathbf{1}$ to be an *identity element* in terms of $*$, we get $e(a).v * \mathbf{1} = e(a).v$ and $\mathbf{1} * \mathbf{1} = \mathbf{1}$. This requirement now makes sense in two ways:

1. The value of the comprehension over a singleton set is equal to the value of the single heap location of the set, which is something we naturally expect.
2. If we have two sets A, B of references, we can now evaluate the comprehension over the set $A \cup B$ by simply evaluating the comprehension over A and B separately and applying $*$ to the two results. From a different perspective, this means also that we can evaluate a comprehension with a divide and conquer approach. This feature will be very relevant when we try to find methods on how to automatically derive the value of a comprehension.

Together with the properties of $*$ we can now conclude that $(T, *)$ is a commutative monoid with identity element $\mathbf{1}$, where T denotes the type of the comprehension.

Injectivity For each comprehension we require the receiver condition e to be *injective*. This has, among other things, to do with quantified permissions. Since during evaluation of a comprehension, we will obviously access areas of the program heap, we will need to check, whether we currently have permission to those heap locations. In Viper, this is done with quantified permissions by quantifying over the set of heap locations $e(a).v$ we evaluate the comprehension over. Within the quantifier we assert for them the expression `acc(e(a).v, 1/2)`. This leads to an assertion in the form of

```
forall a: A :: cond(a) => acc(e(a).v, 1/2)
```

where `cond(a)` is a condition, which restricts the abstract representation of the heap location a to the heap locations we wish the comprehension to be evaluated over. Note that the permission amount of `1/2` is completely arbitrary and can be replaced with any amount between 0 and 1. With an amount between 0 and 1 we ensure that we only have read permission, which is enough for comprehensions. Viper's system for quantified permission requires $e(a)$ to be injective into the set of heap locations [9] (under the condition $cond(a)$). Therefore, it makes sense to pose this as a requirement for comprehensions as well. We will also later see that in certain axioms we will want to quantify over references, with which we will want to specify the comprehension arguments. This will then be possible with inverse functions, which require receiver injectivity.

Types There are two types which occurred so far in our discussion

- A refers to the type which abstracts heap locations, for example the tuple $(Array, Int)$ for arrays. We will sometimes use a to refer to an instance of A and a_i to refer to the i 'th entry in the tuple of a . We call A the argument type of the comprehension.
- T refers to the type of the value of the comprehension and we will therefore call it simply the type of the comprehension.

With our discussion above we can now impose types on our components. Obviously, v must be of type T . Because $*$ is binary, associative, and commutative, it must be of type $(T, T) \rightarrow T$. This now means that $\mathbf{1}$ has type T as well. Lastly, e needs to be of type $A \rightarrow T$.

2.3 Definition of a comprehension

The above discussion now allows us to formally define a comprehension.

Definition 2.1 (Comprehension) *A comprehension is a tuple $(e, v, *, \mathbf{1}, A, T)$, where A, T are types, e is an injective expression of type $A \rightarrow Ref$, v is a field of type T , $\mathbf{1}$ is a value of type T , $*$ is a binary function of type $(T, T) \rightarrow T$ and $(T, *)$ is a commutative monoid with identity element $\mathbf{1}$.*

2.4 Definition of the value of a comprehension

After providing a formal definition for a comprehension, we will now try to do the same for the value of a comprehension. For evaluation of a comprehension, we need two components.

The first component is a set f of comprehension arguments a with

$$f := \{a : A \mid \text{cond}(a)\}$$

We call f a *filter* of comprehension c with argument type A and $\text{cond}(a)$ the *filtering condition* of filter f . We say that a filter f *covers* a heap location $r.v$, iff there exists a comprehension argument a , such that $\text{cond}(a)$ and $e(a) == r$. Viper already has a way to specify sets. The Viper language includes a built-in type `Set` with appropriate functions and built-in axiomatization. However, to be able to fully control the axiomatization, we introduce a new type *Filter* which abstractly expresses sets like f . Since filters are essentially sets, we will also use common set notations, such as relations like \in, \subset, \setminus , etc. Later on we will take a closer look at those operators and provide axiomatizations for them. We will say that a filter instance is bound to a comprehension, because they range over type A of some comprehension.

In principle we could relax this dependency a bit and only require a dependency of a filter to type A . However since filters will only be specified by the user for comprehension calls, and we will provide an own axiomatization for every comprehension, it makes sense to uphold this dependency, such that we can talk about filters of a comprehension. The type *Filter* however is independent of any comprehension and can be interpreted as the set of all possible filters for all possible comprehensions.

The second component for evaluation of a comprehension is a heap h in which the comprehension is evaluated.

To define now the value of a comprehension, we will make use of the insight we gained in section 2.2 on that we can evaluate a comprehension with a divide and conquer approach.

Definition 2.2 (Comprehension value) *A function $\varphi_c(h, f)$ is called the value of the comprehension $c = (e, v, *, \mathbf{1}, A, T)$ for $h : \text{Heap}$ and $f : \text{Filter}$, iff for all $f' \subset f$*

$$\varphi_c(h, f) = \begin{cases} \mathbf{1} & |f| = 0 \\ e(a) \cdot_h v, \quad a \in f & |f| = 1 \\ \varphi_c(h, f') * \varphi_c(h, f \setminus f') & |f| > 1 \end{cases} \quad (2.1)$$

This recursive definition allows the comprehension value to be described via splitting up the filter into two parts and describing the comprehension value over those two parts. One base case is the empty filter. The empty filter can only be split up into empty filters again. As we have discussed above, it makes sense to set the comprehension value over an empty filter as $\mathbf{1}$. The other base case is the singleton filter. The singleton filter can only be split up into an empty filter and the same singleton filter. The singleton base case incorporates, what we actually comprehend over, which are the heap locations covered by the filter. So the comprehension value over a singleton filter is equal to the value of the heap location covered by the filter.

The Singleton Case

We will now start our analysis on how we can encode this specification into an SMT problem, such that we can automatically solve it. For this we start with a more restrictive scenario, which can later be generalized to the general case.

3.1 Overview

In this chapter we restrict ourselves to programs which are not allowed to contain method calls. This would mean that heap updates only occur via assignments, in other words, only singleton heap updates (updates to a single location at once) are possible. In addition, we require that at the entry point of a program, the permission mask is empty, which means that there is no permission to any heap location. This requirement induces that at every program point (for every heap) we know the value of every heap location we have permission to (just by looking at the most recent assignment to the location). Evaluation of a comprehension is then trivial, because we can simply recursively apply the definition to every heap location. In addition this means that the number of heap locations we have permission to is finite, because we can only inhale permission to single heap locations.

To get a more concrete idea about this scenario, consider the following Viper program. At the end of the method, we know the heap locations to have the values `r1.val == 0`, `r2.val == 0`, `r3.val == -2`, so the sum comprehension over those heap locations would obviously be `-2`. We were able to deduce this by simply looking at the different occurrences of `val`. We will now take a look at how such a reasoning can be done automatically. For this example this means, how a user could write an assertion at the end of method `singleton` to express the value of the comprehension after program execution, and how such an assertion could be checked automatically via observing location accesses to single heap locations.

Source Code 3.1: Singleton updates

```

1  field val: Int
2  method singleton() {
3      var r1: Ref
4      var r2: Ref
5      var r3: Ref
6      inhale acc(r1.val) && acc(r2.val) && acc(r3.val)
7      r1.val := 0
8      r2.val == r1.val
9      r3.val := r2.val - 2
10 }

```

3.2 Simplification of comprehension value definition

Let us tackle this problem by first providing an alternative formula for the value of a comprehension (for equation 2.1).

Lemma 3.1 *For all $h, f, a \in f, f' \subset f$, the value $\varphi_c(h, f)$ of the comprehension $c := (e, v, *, \mathbf{1}, A, T)$ is equal to*

$$\Phi_c(h, f) := \begin{cases} \mathbf{1} & |f| = 0 \\ \Phi_c(h, f \setminus \{a\}) * e(a) \cdot_h v & |f| > 0 \end{cases} \quad (3.1)$$

The above equation now expresses the value of a comprehension in a similar way to a *fold* operation (without imposing an ordering), by an inductive definition. We provide a simple proof by induction of the above lemma.

Proof (by induction over the number of elements in f) We prove the property that $\varphi_c(h, f) = \Phi_c(h, f)$ for some fixed c, h .

- *Base case $|f| = 0$:* $\varphi_c(h, f) = \mathbf{1} = \Phi_c(h, f)$
- *Base case $|f| = 1$:* Let $a \in f$ be arbitrary (that is $f = \{a\}$). We get

$$\begin{aligned} \varphi_c(h, f) &= e(a) \cdot_h v \\ &= \mathbf{1} * e(a) \cdot_h v \\ &= \Phi_c(h, \{a\}) * e(a) \cdot_h v \\ &= \Phi_c(h, f \setminus \{a\}) * e(a) \cdot_h v \\ &= \Phi_c(h, f) \end{aligned}$$

- *Induction hypothesis:* We assume that for fixed f with $0 \leq n := |f|$ the definitions 3.1 and 2.1 are equal.

- *Step case* $|f| = n + 1$: Let $f' \subset f$ be arbitrary. We restrict without loss of generality $f' \neq \{\}$, fix an arbitrary $a \in f'$ and get

$$\begin{aligned}
\varphi_c(h, f) &= \varphi_c(h, f') * \varphi_c(h, f \setminus f') \\
&= \varphi_c(h, \{a\}) * \varphi_c(h, f' \setminus \{a\}) * \varphi_c(h, f \setminus f') \\
&= \varphi_c(h, \{a\}) * \varphi_c(h, f' \setminus \{a\}) * \varphi_c(h, (f \setminus f') \setminus \{a\}) \\
&= \varphi_c(h, \{a\}) * \varphi_c(h, f' \setminus \{a\}) * \varphi_c(h, (f \setminus \{a\}) \setminus f') \\
&= \varphi_c(h, \{a\}) * \varphi_c(h, f' \setminus \{a\}) * \varphi_c(h, (f \setminus \{a\}) \setminus (f' \setminus \{a\})) \\
&\stackrel{(2.1)}{=} \varphi_c(h, \{a\}) * \varphi_c(h, f \setminus \{a\}) \\
&= \varphi_c(h, f \setminus \{a\}) * e(a)_{.h}v \\
&\stackrel{\text{IH}}{=} \Phi_c(h, f \setminus \{a\}) * e(a)_{.h}v \\
&= \Phi_c(h, f)
\end{aligned}$$

For $a \in f \setminus \{f'\}$ the above steps would be similar, hence we get $\varphi_c(h, f) = \Phi_c(h, f)$. \square

To manually prove the value of a comprehension for our simple programs, we can now just apply formula 3.1 inductively to every heap location of interest, since we know the value of the heap location. How we can encode this into SMT to make such a deduction automatic is shown in the next sections.

3.3 Automatic deduction

From formula 3.1 we can now create a first draft of two quantifiers (for the two cases of the definition), without triggers, to axiomatize the value of a comprehension. The draft will later be refined and provided with triggers.

$$\forall h : \text{Heap}, f : \text{Filter}. \quad \text{empty}(f) \rightarrow c(h, f) = \mathbf{1} \quad (3.2)$$

$$\begin{aligned}
\forall h : \text{Heap}, f : \text{Filter}, a : A. \quad a \in f \rightarrow \\
c(h, f) = c(h, f \setminus \text{filterCreate}(a)) * e(a)_{.h}v \quad (3.3)
\end{aligned}$$

Here we have used the expression $c(h, f)$ to refer to the value of a comprehension, which is a notation we will use from now on in the context of SMT encodings. In the above axioms we have used some functions, which we have not yet defined in this context (but whose meaning should be clear), in particular *empty*, \in , \setminus and *filterCreate*. We will need to show how to encode and axiomatize them. In addition we will need to find a way of defining filters.

Filter Let us start with the filter encoding. As already stated, a filter should be a variable (or literal) of type *Filter* that represents a set of comprehension arguments. A common way to describe sets is the set-builder notation, e.g., $f := \{a : A \mid \text{cond}(a)\}$. We will use this notation to describe filters. We will call the filter condition of filter f as *cond*. Since A is fixed for a comprehension, $\text{cond}(a)$ will define the content of a filter. To encode this, let us provide a built-in function:

$$\text{filtering} : (\text{Filter}, A) \rightarrow \text{Bool}$$

that represents the filter condition for a given filter. To define a filter f , we can now declare a variable or constant f_0 for which we assume the following:

$$\forall a : A. \{\text{filtering}(f_0, a)\} \quad \text{filtering}(f_0, a) \leftrightarrow \text{cond}(a), \quad (3.4)$$

where $\text{cond}(a)$ is a boolean-typed expression.

empty We can now see that *filtering* provides semantics for \in , so we are left with encoding *empty*, \setminus , and *filterCreate*. *empty*(f) should encode a check that filter f is empty, i.e., has no elements. This suggests that empty has the following signature

$$\text{empty} : \text{Filter} \rightarrow \text{Bool}$$

Checking that a filter has no elements contained is equivalent to checking, whether the *filtering* function for this filter is false for every comprehension argument, which is an UNSAT query (a boolean formula, which should be proven to be unsatisfiable) of the expression *cond*. This can be encoded by either introducing a new constant a and checking $\neg \text{filtering}(f, a)$, or (as we will do), by directly checking the condition:

$$\forall a : A. \neg \text{filtering}(f, a)$$

It is easy to see that, with skolemization, the two approaches are equivalent. Finally, we get the following axiom for the *empty* function:

$$\forall f : \text{Filter}. \{\text{empty}(f)\} \quad \text{empty}(f) \leftrightarrow (\forall a : A. \{\text{filtering}(f, a)\} \neg \text{filtering}(f, a)) \quad (3.5)$$

The triggers of the above axiom are chosen such that the body of the outer quantifier gets instantiated when *empty* is applied, and such that the body of the inner quantifier gets instantiated, when *filtering* for f is called, creating a 'shortcut', to a possibly very complex formula to check *filtering* for f . For example consider a filter f , which has a very complex filtering condition, taking immensely long to prove *filtering* for some argument a . As soon as we get to know (f), every call to $\text{filtering}(f, a)$ can then simply be answered by instantiating the inner quantifier of the above axiom, where we learn directly $\neg \text{filtering}(f, a)$.

Narrow Now, we consider the encoding of \setminus and *filterCreate*. Note that in the axiomatization of a comprehension value from above, the two functions only appear together to describe a filter f' as a filter f without a specific argument. Such a filter 'constructor' can also be represented by a single function, which we call *narrow*, with the following signature:

$$\textit{narrow} : (\textit{Filter}, \textit{Ref}) \rightarrow \textit{Filter}$$

Note that we use *Ref* in the signature, instead of the more obvious A . The reason for this lies within triggering, which we will explore in the next paragraph. So, the *narrow* function can now be interpreted as a function that outputs a filter that covers every reference, covered by another filter, except for one specific reference. The following SMT axiom encodes this interpretation.

$$\begin{aligned} \forall f : \textit{Filter}, r : \textit{Ref}, a : A. \{ \textit{filtering}(\textit{narrow}(f, r), a) \} \\ \textit{filtering}(\textit{narrow}(f, r), a) \leftrightarrow \textit{filtering}(f, a) \wedge e(a) \neq r \end{aligned} \quad (3.6)$$

Triggering Since we have our components now defined and axiomatized, we can think about triggers for the two axioms for the value of a comprehension. A good selection of triggers is vital for our automatic deduction, since triggers define where the axioms get instantiated, and therefore where our axioms can be applied. Selecting too restrictive triggers can cause that we cannot prove desired properties anymore, while on the other hand, too loose triggers can cause matching loops and bad performance [6] [8]. The challenge therefore is to find minimally necessary triggers to be able to prove exactly what we would like to. We will make a deeper analysis on the capabilities of our system in the next section, while we try to keep this requirement in mind when coming up with triggers.

For our axioms 3.2 and 3.3, the triggers are fortunately very straight-forward. Both of the axioms will need to have the comprehension call as a triggering-term, since whenever we call a comprehension we request its value, and those two axioms define the value. For axiom 3.2 this is even the only term we need, since it already contains all the quantified variables, so for this axiom we get the trigger $\{c(h, f)\}$. Axiom 3.3 however includes an additional quantified variable. Now remember that we would like to deduce the value of a comprehension via location accesses, so it makes sense to trigger on a location access. However a location access does not necessarily mention a value of type A , so we cannot use it as a trigger for our quantified variable of type A . So instead of now using a variable of type A (an abstract representation of a reference) as a quantified variable, we could just use a variable r of type *Ref* and use a location access of the form $r.v$ as a triggering-term.

This now obviously implies that we need to change the body of the quantifier and replace the occurrence of a . For the term $f \textit{filterCreate}(a)$ we

already have a solution to replace a by using the *narrow* function, which we defined earlier, and which does not depend on an instance of A , but on a reference. For the other two terms with a contained, remember that we required the receiver expression of the comprehension to be injective, which means that it has an inverse $e^{-1}(r)$, which we will use instead of a in the axiom. With all of this being said, we now get the following axiom which will replace our draft formula 3.3

$$\begin{aligned} \forall h : \text{Heap}, f : \text{Filter}, r : \text{Ref}. \{c(h, f), r \cdot_h v\} \\ \text{filtering}(f, e^{-1}(r)) \rightarrow c(h, f) = c(h, \text{narrow}(f, r)) * r \cdot_h v \end{aligned} \quad (3.7)$$

3.4 Incompleteness and matching loops

We have now created an encoding of the definition of a comprehension value for SMT. However we would not be able to prove a lot of cases in our restricted scenario. Consider again example 3.1, where we have two heap updates. Let's say we want to prove $c(h_2, f) = -2$, with h_2 being the heap at the end of the program and f being a filter including r_1, r_2 and r_3 , we would get the following deduction.

$$\begin{aligned} c(h_2, f) &= c(h_2, \text{narrow}(f, r_3)) * r_3 \cdot_{h_2} v \\ &= c(h_2, \text{narrow}(f, r_3)) * -2 \end{aligned}$$

Since $\text{narrow}(f, r_3)$ is obviously not empty and we don't have any further location access in heap h_2 , we will not learn anything about $c(h_2, \text{narrow}(f, r_3))$ and therefore won't be able to prove our assertion. The problem here is that we focus ourselves on a single heap and try to deduce the value of the comprehension only based on information from the current heap, because we restrict h in axiom 3.7 to the heap used in the comprehension call via the trigger. To avoid this, we want to instantiate axiom 3.7 in the heap, where the heap location update happened, independently from the heap in which the comprehension was called. We can do this by defining a new function

$$\text{dummy} : (\text{Filter}) \rightarrow \text{Bool}$$

, which we can interpret as a function that indicates, whether the given filter is used in a comprehension call. We can axiomatize the *dummy* function easily as follows.

$$\forall h : \text{Heap}, f : \text{Filter}. \{c(h, f)\} \quad \text{dummy}(f) \quad (3.8)$$

When we use now the trigger $\{\text{dummy}(f), r \cdot_h v\}$ for axiom 3.7, the axiom gets instantiated for every filter used as an argument for a comprehension call, whenever we have a heap location update.

This alone however won't be enough. We can now instantiate the axiom in different heaps, other than the one mentioned in the comprehension call, but when instantiating the axiom, we also get an assertion about the comprehension in a different heap. That the value of the comprehension in this different heap is the same as in the current heap, is not always true, so we need another axiom for this. To make this point more clear consider again the deduction for example 3.1, but this time with our new trigger.

$$\begin{aligned} c(h_2, f) &= c(h_2, \text{narrows}(f, r_3)) * r_{3 \cdot h_2} v \\ &= c(h_2, \text{narrows}(f, r_3)) * -2 \end{aligned}$$

$$\begin{aligned} c(h_1, \text{narrows}(f, r_3)) &= c(h_1, \text{narrows}(\text{narrows}(f, r_3), r_2)) * r_{2 \cdot h_1} v \\ &= c(h_2, \text{narrows}(\text{narrows}(f, r_3), r_2)) * 0 \end{aligned}$$

$$\begin{aligned} c(h_1, \text{narrows}(\text{narrows}(f, r_3), r_2)) \\ &= c(h_1, \text{narrows}(\text{narrows}(\text{narrows}(f, r_3), r_2), r_1)) * r_{1 \cdot h_1} v \\ &= c(h_2, \text{narrows}(\text{narrows}(\text{narrows}(f, r_3), r_2), r_1)) * 0 \end{aligned}$$

$$c(h_2, \text{narrows}(\text{narrows}(\text{narrows}(f, r_3), r_2), r_1)) = \mathbf{1}$$

We can easily verify by hand that for example $c(h_1, \text{narrows}(f, r_3))$ is equal to $c(h_2, \text{narrows}(f, r_3))$, but obviously we won't get this automatically. This problem is called *framing* and has luckily already been encoded for Viper for custom functions. However since our function will be generated, and to have more control, we provide our own framing axiom.

$$\begin{aligned} \forall f : \text{Filter}, h_1 : \text{Heap}, h_2 : \text{Heap}. \{c(h_1, f), c(h_2, f)\} \\ (\forall r : \text{Ref}. \{r \cdot_{h_1} v, r \cdot_{h_2} v\} \text{filtering}(f, e^{-1}(r)) \rightarrow r \cdot_{h_1} v = r \cdot_{h_2} v) \\ \rightarrow c(h_1, f) = c(h_2, f) \quad (3.9) \end{aligned}$$

Matching loop With the framing axiom, we have now however introduced the possibility for matching loops. Let f be an unbounded filter (e.g. with $\text{filtering}(f, a) \leftrightarrow \text{true}$). Now let there be two comprehension calls $c(h_1, f)$, $c(h_2, f)$ for two different heaps h_1, h_2 , where $c(h_1, f) \neq c(h_2, f)$. The two calls now trigger axiom 3.9 and instantiates the following formula, which can be interpreted as follows: If for all heap locations covered by f , the value is the same in h_1 and h_2 , then the value of the comprehension calls must be the same in the two heaps.

$$\begin{aligned} (\forall r : \text{Ref}. \{r \cdot_{h_1} v, r \cdot_{h_2} v\} \text{filtering}(f, e^{-1}(r)) \rightarrow r \cdot_{h_1} v = r \cdot_{h_2} v) \\ \rightarrow c(h_1, f) = c(h_2, f) \end{aligned}$$

The left-hand side of the implication is a forall-quantifier, which will become an existential-quantifier during SMT-solving. Remember that when solving SMT-problems with existentials, the solver will skolemize the existential and create fresh variables (or functions). In this example, a fresh function of type *Ref* will be created, let's call it ρ . Since the two comprehension calls have different values, some heap locations covered by the filter, must have different values in h_1 and h_2 and therefore the SMT-solver will learn $\text{filtering}(f, e^{-1}(\rho(f, h_1, h_2)))$, which means that we have an expression for a reference, which is covered by f and for which we know that its value is not the same in h_1 and h_2 . Now the SMT-solver will instantiate axiom 3.7 for $f, \rho(f, h_1, h_2)$ and for both h_1 and h_2 once and learns

$$\begin{aligned} & \text{filtering}(f, e^{-1}(\rho(f, h_1, h_2))) \\ & \rightarrow c(h, f) = c(h, \text{narrows}(f, \rho(f, h_1, h_2))) * \rho(f, h_1, h_2).h.v \end{aligned}$$

where h is once h_1 and once h_2 .

Since the SMT-solver just learned the left-hand side of the implication, the SMT-solver also learns the right-hand side and therefore the filter $f' := \text{narrows}(f, \rho(f, h_1, h_2))$ will be in scope. Since f was unbounded, f' is again unbounded and the comprehension calls for h_1 and h_2 with f' will again trigger the framing axiom 3.9, which closes the matching loop.

The problem however lies not only within the framing axiom, indeed every assertion of a similar form to

$$\begin{aligned} & \forall f : \text{Filter}, h : \text{Heap}. \\ & \exists r : \text{Ref}. \text{filtering}(f, r) \wedge \text{cond}_1(r.h.v) \wedge \text{cond}_2(c(h, f)) \end{aligned} \quad (3.10)$$

will introduce a matching loop, where cond_1 and cond_2 are boolean expressions, with at least one occurrence of their argument and the triggers are avoided for simplicity. This is because such an axiom will generate a new function from skolemization for every instantiation, and because the body of the existential mentions all triggering terms of the axiom 3.7, which again generates a new filter, instantiating the above axiom.

Let's first state a few things we cannot do to avoid the matching loop, before we will propose a solution that works. One idea could be to somehow restrict the instantiation of the existential of 3.10 to disallow instantiations for subsequently narrowed down filters. However in some cases this restricts us too heavily. Suppose we extend our scenario to the case, that the value of a comprehension can be arbitrary (but known) at the start of the program. In this case, we would not be able to deduce the value of the comprehension anymore, after we have done two heap updates to different locations, since the framing axiom 3.9 can no longer deduce that the comprehension over the remaining locations has not changed, because the filter describing

the remaining locations is of the form $narrow(narrow(f, r_1), r_2)$, i.e. it is narrowed down twice. We could now say that this scenario does not matter to us, however in the general case this will be one of the most basic scenarios we would like to support.

The next idea might be to not use references in the existential of 3.10 or in the axiom 3.7, but for framing we really need to check, whether all our heap locations have remained the same value, which requires quantification over references in some way. For the axiom 3.7 we also need the reference, since we want to cover all possible accesses of heap locations and don't want to require a certain way of accessing / abstracting a heap location for instantiating the axiom.

Since the matching loop only occurs, because the SMT-solver creates fresh reference-typed functions, an idea, which will actually work, would be to restrict the instantiation of axiom 3.7 to references which were not generated via skolemization. To exclude instantiation, if we have a certain term in scope, we could introduce a new feature for SMT-solvers, which we will call now anti-triggers: triggers which prevent instantiation if the term is in scope. However Viper uses Z3 which does not support anti-triggers currently. This means that we need to go the other way around and create something for every reference typed expression ourselves and only trigger, if this something is in scope as well. One possibility to realize this, is by using a new dummy function

$$userMentioned : Ref \rightarrow Bool$$

, which indicates whether a reference is created by the user (and not by the SMT-solver), or not. We can then assume $userMentioned$ for every reference, except for the one in the framing axiom 3.9 and add $userMentioned(r)$ as an additional triggering term to axiom 3.7.

Instead of the $userMentioned$ function, a possibility is also to introduce a new level of indirection for references via a function in a way that references will be translated to valid references with a function

$$translate : Ref \rightarrow Ref$$

Then every expression e of type Ref in the program (also in triggers), except for r in the framing axiom, can be replaced with $translate(e)$, which will prevent the skolemized reference to instantiate anything. For the remainder of this thesis we will however use the approach with the $userMentioned$ function.

No matching loops We have now prevented the above matching loop. However one might think that we still have a matching loop, especially in axiom 3.7, because the body of the quantifier creates a new filter, which can

instantiate the same axiom again. However the crucial part here is that the creation of the new filter is on the right-hand side of the implication. For implications, Z3 will only learn the right-hand side, if it can prove the left-hand side. Indeed the axiom will get instantiated again for filter $\text{narrows}(f, r)$, reference r and heap h , but the right-hand side will never be learned, because the left-hand side checks whether r is covered by $\text{narrows}(f, r)$, which is clearly not the case, as we can see from the axiomatization of narrows 3.6. This mechanism prevents the matching loop.

3.5 Summary

For this section, we rewrite and provide a summary for the axioms we have created so far. In the end we will reason about why those axioms are complete in our restricted scenario, illustrated in section 3.1, that is why the axioms ensure that we can prove automatically what we wish to prove.

Note that even with the unique numbers for the formulas in this section, the formulas are actually not new. Some have simply not been specified in this way, because, for example, their trigger got changed in a subsection, while others are repeated here and will be used as reference points later on in the document.

We have created the two axioms which describe the value of a comprehension, according to formula 3.1.

$$\forall h : \text{Heap}, f : \text{Filter}. \{c(h, f)\} \text{ empty}(f) \rightarrow c(h, f) = \mathbf{1} \quad (3.11)$$

$$\begin{aligned} \forall h : \text{Heap}, f : \text{Filter}, r : \text{Ref}. \{dummy(f), r._h v, userMentioned(r)\} \\ filtering(f, e^{-1}(r)) \rightarrow c(h, f) = c(h, narrows(f, r)) * r._h v \end{aligned} \quad (3.12)$$

For filters we have created an axiomatization of narrows , which produces a filter by excluding a reference from a given filter, and of empty , which indicates whether a filter covers any elements.

$$\begin{aligned} \forall f : \text{Filter}, r : \text{Ref}, a : A. \{filtering(narrows(f, r), a)\} \\ filtering(narrows(f, r), a) \leftrightarrow filtering(f, a) \wedge e(a) \neq r \end{aligned} \quad (3.13)$$

$$\begin{aligned} \forall f : \text{Filter}. \{empty(f)\} \\ empty(f) \leftrightarrow (\forall a : A. \{filtering(f, a)\} !filtering(f, a)) \end{aligned} \quad (3.14)$$

Lastly we have created the framing axiom and the axiomatization of dummy .

$$\begin{aligned} \forall f : \text{Filter}, h_1 : \text{Heap}, h_2 : \text{Heap}. \{c(h_1, f), c(h_2, f)\} \\ (\forall r : \text{Ref}. \{r._{h_1} v, r._{h_2} v\} filtering(f, e^{-1}(r)) \rightarrow r._{h_1} v = r._{h_2} v) \\ \rightarrow c(h_1, f) = c(h_2, f) \end{aligned} \quad (3.15)$$

$$\forall h : \text{Heap}, f : \text{Filter}. \{c(h, f)\} \text{ dummy}(f) \quad (3.16)$$

Completeness We will now prove via induction over the number of heap locations covered by a filter that the above feature is complete in our scenario, by comparing it to the definition of a comprehension from formula 3.1. We define completeness by the possibility of an SMT-solver to prove an assertion over the value of a comprehension at some point in a program, *iff* it could be proved manually.

First consider an assertion over the value of a comprehension for an empty filter. According to the comprehension definition, the value of the comprehension should then be **1**. Since such an assertion contains an occurrence of a comprehension call, the axiom 3.11 will be triggered which then triggers axiom 3.14. So we get an implication of the form

$$(\forall a : A. !\text{filtering}(f, a)) \rightarrow c(h, f) = 1$$

Because f is empty, the left-hand side can be proved by the SMT solver and we learn that the comprehension has indeed a value of **1**. Note that this is the only way of proving the value of the comprehension with our axioms, since the only other axiom which could describe the value of a comprehension is axiom 3.12, whose left-hand side, however, cannot be true, because the filter is empty.

Now let us consider an assertion over the value of a comprehension $c(h, f)$ for a filter f which covers $n + 1$ elements ($|f| = n + 1$) while assuming that the axioms are complete for a comprehension with a filter covering n elements. First let us observe that axiom 3.16 gets triggered by $c(h, f)$. Axiom 3.11 will obviously also get triggered by $c(h, f)$, however, the left-hand side will be disproved, since the filter is not empty (and therefore when trying to prove the right-hand side of axiom 3.14, the solver will find an argument a , for which $\text{filtering}(f, a)$ is *true*). So, the only way to gain knowledge about the value of the comprehension is via axiom 3.12. Now there are two possibilities. The first one is that the value of every heap location covered by the filter is unknown, because we either don't have permission, or there was never a value assigned to the location. The second possibility is that the value of some heap location is defined, in which case there was an assignment to the location, since assignments are the only possibility to update the heap in our scenario.

If the value of every heap location of the filter is unknown, then obviously an assertion about the value of a comprehension will be false. This is also the case for our axioms, since the right-hand side of axiom 3.12 will not be instantiated, because there will not be a heap access for a location covered by the filter (or if there would be, the solver will fail to verify the program,

because there is not enough permission). Therefore the solver could not prove the assertion and would fail.

If the values of some heap locations covered by the filter are defined, then there will be assignments to those locations. Consider the heap location of the most recent heap location update relative to h . Let h_0 indicate the heap, in which the update happened. The corresponding heap update will now trigger axiom 3.12 and will also instantiate the right-hand side of the implication, so we will learn the following formula:

$$c(h_0, f) = c(h_0, \text{narrows}(f, r)) * r.h_0.v.$$

Because r is the reference of the heap location with the most recent update (in terms of h) of all the heap locations covered by f , we will now learn with the framing axiom 3.15 that $c(h, f) = c(h_0, f)$. Now the solver will try to derive the value of $c(h, \text{narrows}(f, r))$, which can be done *iff* the value can be proved manually, according to our induction hypothesis.

This showed for the scenario of this chapter, that an assertion over $c(h, f)$ can be automatically proved *iff* it can be manually proved.

The General Case

4.1 Overview

In the previous chapter we have considered a restricted scenario, in which we only allowed singleton heap updates. We also required that we don't have access to any part of the heap at the beginning of the program, which resulted in a scenario, in which we knew at any time the value of every heap location, to which we assigned value to once. In that scenario we were able to provide an axiomatization for the value of a comprehension, for which we also provided an informal proof about completeness. In this chapter we will eliminate any restrictions on our programs, which means that we will now additionally allow heap updates to multiple locations at once, and we will allow the heap to be arbitrary in the beginning of the program. In Viper we encode a heap update to multiple locations via quantified permissions. With quantified permissions we can exhale permission to a certain set of heap locations and inhale permission to this set again later. Since the program lost permission to those heap locations, we however cannot guarantee that the value of the locations didn't change, so the comprehension over a filter, covering parts of this set of heap locations, might have changed as well. One way in which such a generalization will be applied is for method calls. A method can take permission to an arbitrary number of heap locations and modify those locations. When calling a method, we however simply exhale the precondition and afterwards inhale the postcondition, so we won't learn anything about the method body.

To illustrate this scenario, consider the following example. Note that at the beginning of method `general`, we don't have any information about the content of the heap locations in `s`, so we cannot state anything about the value of the sum-comprehension over `s`. After line 12, the only statement we can pose about the comprehension, is that it has increased by 2. This should already be possible with our axioms from the singleton case. After

Source Code 4.1: General updates

```
1 field val: Int
2
3 method modify(s: Set[Ref])
4   requires forall r: Ref :: r in s ==> acc(r.val)
5   ensures forall r: Ref :: r in s ==> acc(r.val)
6
7 method general(s: Set[Ref], r1: Ref)
8   requires forall r: Ref :: r in s ==> acc(r.val)
9   requires r1 in s
10  ensures forall r: Ref :: r in s ==> acc(r.val)
11  {
12    r1.val := r1.val + 2
13    modify(s)
14  }
```

line 13 however, we lost every information about the comprehension again, since `modify` could have arbitrarily modified the heap for the locations in `s`. This means that method `modify` should have some postcondition, describing the change that happened to the heap locations, from which we can deduce the value of the comprehension at the end of method `general`. Now the problem however is that we don't have a singleton heap update anymore at line 13, which means that with our current axioms we wouldn't be able to deduce the value of the comprehension, even if we knew how the heap changed. In this chapter we will now provide further axioms to our current ones, to support such a general case.

4.2 Fundamental limitations

Before even starting the discussion about how we would provide an axiomatization for the general scenario, we can already state some limitations the system will have. By stating those limitations now, we avoid having to worry about some cases later and have a more concrete idea of why the axiomatization we will provide will serve for what we want to support.

4.2.1 Automatic induction

Our first limitation has to do with how we specify changes to multiple heap locations. For this we have multiple possibilities. We can quantify over heap locations and impose a condition on every location, we can specify changes via recursive predicates, or we can write how a comprehension has changed. If we want to use this specification to deduce the value of a comprehension we will however run into problems for most of the above approaches.

As already stated in the introduction, recursive predicates are a feature for different scenarios, especially for modelling data-structures which are traversed in a specific order. With comprehensions we want to support data-structures which can be traversed in multiple directions. Therefore we won't focus on compatibility with recursive predicates here.

Next, let's consider what happens when we specify the changes for a comprehension other than the comprehension whose value we wish to deduce. For example, we could state that the maximum comprehension (i.e. the comprehension expressing the maximum value over a set of heap location values) has value 5 after some method call over some filter f . From this we could intuitively require from the axioms that the sum comprehension after the call over f is at most $5 * |f|$. Such a deduction would however most certainly require a proof by induction and since Viper does not support automatic deduction via induction, we consider this problem out of scope.

The same principle applies for the approach with a custom forall quantifier. We can even interpret a forall quantifier as a comprehension, which comprehends over the truth values of the expression of the quantifier (and in this case not over heap locations), and uses the logical and as the binary expression, together with *true* as the unit. So for arbitrary quantifiers, we would also need a proof by induction and won't support this approach as well. To illustrate this in an example, consider a quantifier, which specifies that every heap location has at least a value of 0. We could then argue that the sum comprehension over a filter f must always be positive. However this is not trivial to deduce, it actually requires a proof by induction, which is why we won't support deductions from arbitrary quantifiers.

This means that the only possibility to specify changes to the heap, if we want to deduce the value of a comprehension c later, is by specifying how the value of c has changed.

4.2.2 Losing permission

Imagine we want to deduce the value of the sum comprehension over filter f and we lose and gain permission to a single heap location covered by f . At first sight this might not be a real problem, since this is only a single heap location. However this single heap location can have an arbitrary value x , which might be huge, or very small. In such a case, our sum comprehension would have a completely different value, depending on whether this heap location is in f , or not and we won't be able to prove anything concrete about the value of the comprehension over f (other than that the value has changed for this part of the comprehension), if we lost permission to this location and did not learn how the value of the location changed. This means that for the general case, we would need to specify every modification to heap locations covered by f , to be able to deduce the value of a comprehension.

Together with the previous limitation this means that for every modification to heap locations of a filter f , we typically need to specify the change by expressing the value of the comprehension for those locations after the modification. If the modification happens however to a single heap location, we don't have to mention the comprehension, since this is already covered by the axiomatization from the singleton case.

4.3 Automatic deduction

This time we go back again to the original formula 2.1 for the value of a comprehension.

$$\varphi_c(h, f) = \begin{cases} \mathbf{1} & |f| = 0 \\ e(a) \cdot_h v, \quad a \in f & |f| = 1 \\ \varphi_c(h, f') * \varphi_c(h, f \setminus f') & |f| > 1 \end{cases}$$

for all $f : \text{Filter}, h : \text{Heap}, f' \subset f$

Notice how with this formula, we can split up filter f into 'subfilters' f' and $f \setminus f'$ of f , to inductively define the value of the comprehension. This split will be practical for us in the general scenario, where we, for example, know the value of the comprehension over f' . In this case, the problem of deducing the value of the comprehension over f is reduced to deducing the value of the comprehension over $f \setminus f'$.

Let us again create a draft for an axiom to support the third case of the above value definition.

$\forall h : \text{Heap}, f : \text{Filter}, f' : \text{Filter}.$

$$f \subseteq f' \rightarrow c(h, f) = c(h, f') * c(h, f \setminus f') \quad (4.1)$$

At first sight this might seem to be incorrect, since we don't check for $|f| > 1$ and check for $f \subseteq f'$ instead of $f \subset f'$. However, as one can easily prove, this modification does not change the definition, because if $|f| \leq 1$, then f' or $f \setminus f'$ will be the empty set while the other subfilter will be f , and because of the first case in the definition, we would therefore get $\varphi_c(h, f) * \varphi_c(h, \emptyset) = \varphi_c(h, f)$. This means that removing the condition $|f| > 1$ does not alter the definition, but it makes it more redundant. The reason on why we do it this way, is because \subseteq is easier to axiomatize than \subset . Also the \subseteq relation can be used in some other axioms. Additionally since we can get rid of the check $|f| > 1$, this approach is probably even better for performance.

Now we have again introduced some operators, for which we don't yet have an axiomatization, which we will need to provide, concretely the operators \subseteq (which we will call *subfilter relation*) and \setminus .

Filter minus First we will consider the axiomatization of the \setminus operator, which we will encode as a function with the following signature.

$$\text{minus} : (\text{Filter}, \text{Filter}) \rightarrow \text{Filter}$$

We can axiomatize *minus*, by stating the filtering condition of the result of the function.

$$\forall f : \text{Filter}, f' : \text{Filter}, a : A. \{ \text{filtering}(\text{minus}(f, f'), a) \} \\ \text{filtering}(\text{minus}(f, f'), a) \leftrightarrow \text{filtering}(f, a) \wedge \neg \text{filtering}(f', a) \quad (4.2)$$

The trigger is chosen such that the axiom is instantiated whenever the filtering condition is called, i.e. whenever something about the content of the result of *minus* is requested.

Subfilter relation In set theory, the subset-relation is often defined as follows [7]

$$A \subseteq B : \iff \forall x (x \in A \rightarrow x \in B) \quad (4.3)$$

For our subfilter relation, a straight-forward approach would therefore be, to use this definition directly to write an axiomatization. The problem however is that such an axiom would introduce a forall quantifier as the left-hand side of an implication, which would turn into an existential quantifier during SMT-solving. Existential quantifiers have several problems, one of which is performance. Because an existential will be skolemized, the SMT-solver will introduce a new free variable to its problem, increasing the problem difficulty and therefore increasing the runtime. Remember that our way of defining a filter is by defining membership of elements via a condition for a type A . We encoded this via the *filtering* function, which expresses membership of a value $a : A$ of a filter, so the *filtering* encodes the \in relation. The following theorem describes that detecting the subfilter relation between two filters is difficult when the filters are defined via such conditions.

Theorem 4.1 (Subset is NP-hard) *Given sets R and S with members of type A (possibly a tuple), defined by membership conditions $\text{cond}_R : A \rightarrow \text{Bool}$ and $\text{cond}_S : A \rightarrow \text{Bool}$ respectively, such that*

$$\forall a : A. (a \in R \leftrightarrow \text{cond}_R(a)) \wedge (a \in S \leftrightarrow \text{cond}_S(a))$$

Deciding whether $R \subseteq S$ is NP-hard.

Proof (by reduction) We prove the theorem with a reduction to SAT. Let $\Phi(a)$ be an arbitrary boolean formula for free variable a , which is a tuple of *Bool* of size n . We get

$$\neg \Phi(a) \iff \neg \text{true} \vee \neg \Phi(a) \\ \iff \text{true} \rightarrow \neg \Phi(a)$$

Let now $cond_R(a) := true$ and $cond_S(a) := \neg\Phi(a)$, then

$$\begin{aligned} R \subseteq S &\stackrel{(4.3)}{\iff} \forall a : A. (cond_R(a) \rightarrow cond_S(a)) \\ &\iff \forall a : A. \neg\Phi(a) \\ &\iff \neg\exists a : A. \Phi(a) \end{aligned}$$

Note that the expression $\exists a : A. \Phi(a)$ expresses SAT for the n free boolean variables in a . \square

The above result might motivate us to still use the definition 4.3 to directly write an axiom for subfilters, since we anyhow cannot expect nice performance. However there is still a different option. We can eliminate the problematic existential and reduce the problem to the *empty* check, which already incorporates an existential. First we define a new function with the following signature to encode the \subseteq relation.

$$subfilter : (Filter, Filter) \rightarrow Bool$$

Now with the above idea we get the following axiom.

$$\begin{aligned} \forall f : Filter, f' : Filter. \{subfilter(f', f)\} \\ subfilter(f', f) \leftrightarrow empty(minus(f', f)) \end{aligned} \quad (4.4)$$

That this axiom is sound can be verified with a simple set-theoretical derivation.

$$\begin{aligned} A \subseteq B &\iff \forall x. x \in A \rightarrow x \in B \\ &\iff \forall x. x \notin A \vee x \in B \\ &\iff \forall x. \neg(x \in A \wedge x \notin B) \\ &\stackrel{\text{def}}{\iff} \forall x. \neg(x \in A \setminus B) \\ &\stackrel{\text{def}}{\iff} A \setminus B = \emptyset \end{aligned}$$

Triggering Remember that we wish to instantiate our new axiom, whenever we get to learn something about the value of a comprehension over a subfilter f' of the filter f , for which we want to deduce the value of the comprehension. This happens whenever we have a call of the form $c(h, f')$. So a natural choice for the trigger would be the two terms $c(h, f)$ and $c(h, f')$, which would then form the trigger $\{c(h, f), c(h, f')\}$. With those two triggers we will however, similar to the previous chapter, limit ourselves to deductions in a single heap h . In the next section we will see, how we can prevent this from happening.

4.4 Incompleteness

4.4.1 Multiple heaps

By using the trigger $\{c(h, f), c(h, f')\}$ for our axiom 4.1, we limit ourselves to deductions in the heap h , similar to our problem we had for the singleton case. This means that we can rely on the method that we used to solve the similar problem earlier, in order to find triggers in the general case. Remember that we replaced the term $c(h, f)$ with the term $dummy(f)$, to not restrict the heap to the one we are trying to deduce the value of our comprehension in. This principle is applicable here as well. If we use the trigger $\{dummy(f), c(h, f')\}$, the deduction of the value of c over f will be attempted in every heap h , in which a call to c happened over some subfilter f' of f . The framing axiom 3.9 will then possibly deduce that the value of the comprehension over f in the two different heaps has not changed.

4.4.2 Equal filters

For our next problem, imagine we had a simple scenario, where we knew the values $c(h, f_1)$ and $c(h, f_2)$, we knew that $f_1 \cup f_2 = f$ (from the filtering conditions), and we wanted to deduce $c(h, f)$. Axiom 4.1 will then be instantiated for f and f_1 (or similar for f and f_2) and the solver will learn the term

$$c(h, f) = c(h, f_1) * c(h, f \setminus f_1)$$

For the solver, it is however not clear that $f \setminus f_1 = f_2$, hence we cannot deduce the value $c(h, f)$. The problem is that there is no axiom which tries to prove that two filters are equal. To solve this, let us first define a new function

$$equiv : (Filter, Filter) \rightarrow Bool$$

, which indicates whether two filters are semantically equivalent to each others and therefore encodes the \equiv relation between two filters. Semantical equivalence means that the filtering conditions of the two filters have the same truth values for the same parameters, so we could directly write an axiom which would however, similar to the subfilter relation, again introduce a forall quantifier as the left-hand side of an implication. To avoid this, we choose a similar approach as for the subfilter relation and define filter equivalence as follows.

$$f \equiv f' :\iff f \subseteq f' \wedge f' \subseteq f$$

From this definition we can directly write the following axiom

$$\forall f : Filter, f' : Filter. \{equiv(f, f')\} \\ equiv(f, f') \leftrightarrow subfilter(f, f') \wedge subfilter(f', f) \quad (4.5)$$

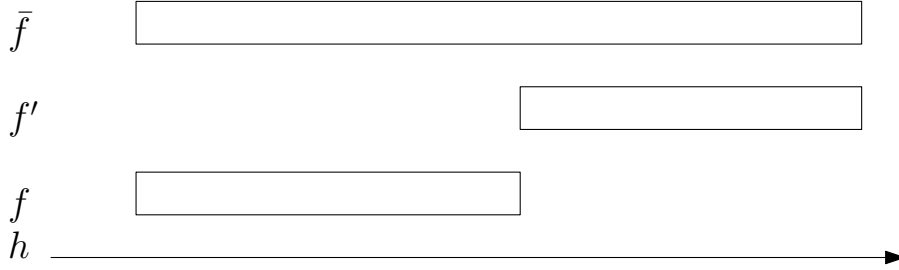


Figure 41: Example of a problem where f is a subfilter

With *equiv* axiomatized, we can provide an axiom, which will try to prove filter equivalence for every two filters whose comprehension values we are interested in.

$$\forall f : Filter, f' : Filter. \{dummy(f), dummy(f')\} equiv(f, f') \leftrightarrow f = f' \quad (4.6)$$

This solves our problem from above, because the solver can now prove that $f \setminus f_1 = f_2$ and therefore that $c(h, f \setminus f_1) = c(h, f_2)$.

4.4.3 Combining filters

A very common case in practice for the binary operator $*$ is that it has an inverse for every element and therefore $(*, 1)$ is a group. This means that we could deduce the value $c(h, f)$ if we knew the equation $c(h, \bar{f}) = c(h, f) * c(h, \bar{f} \setminus f)$ and knew the value of $c(h, \bar{f})$ and $c(h, \bar{f} \setminus f)$. In this subsection we will provide axioms to support this case as well.

Currently we have only cared about splitting up filters to derive the value of a comprehension over a filter through the value of the comprehension over its subfilters. This is a natural approach if we consider the definition 2.1 of the value of a comprehension. However in our new scenario, we might want to consider deductions based on ‘superfilters’ \bar{f} as described above. Figure 41 illustrates such a possible scenario. The heap h is illustrated as a continuous sequence of heap locations, while the filters f , f' and \bar{f} are illustrated as boxes, indicating which part of h they cover. We will use such illustrations further on as well, to argue about specific scenarios.

In this example we assume that we know $c(h, \bar{f})$ and $c(h, f')$ and want to derive $c(h, f)$. This might seem to not be possible at first sight, because we cannot split up f to derive $c(h, f)$. But what the axioms allow is that \bar{f} is split up by f' via axiom 4.1 into f' and $\bar{f} \setminus f'$ and we learn the equation

$$c(h, \bar{f}) = c(h, f') * c(h, \bar{f} \setminus f')$$

which is enough to prove $c(h, f)$, since with axiom 4.6 we can derive that $\bar{f} \setminus f' = f$.

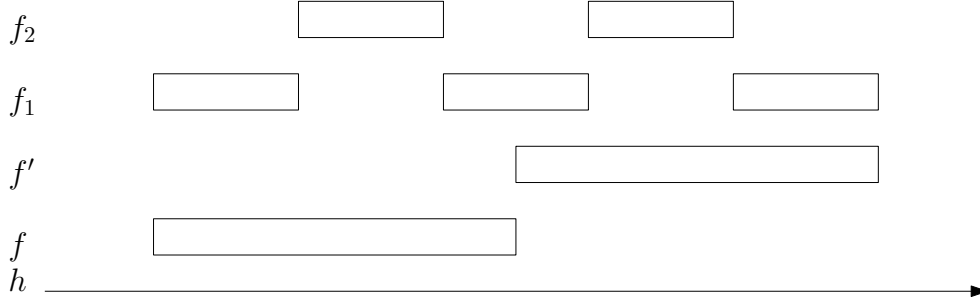


Figure 42: Example of a problem where there is no subfilter relation

The real problem however occurs, when we don't know $c(h, \bar{f})$, but would be able to deduce it via different filters. Figure 42 shows such an example, where the comprehension value over f_1 , f_2 and f' are known in h . Note that there is no subfilter relation between any of the filters, so the axiom 4.1 won't be instantiated. However we could manually derive $c(h, f)$, because

$$c(h, f_2) * c(h, f_1) = c(h, f') * c(h, f)$$

So if we had an axiom, which was able to combine filters to express the comprehension over the filter union, we would have solved this problem. Let's start with a draft for such an axiom.

$$\forall f_1 : Filter, f_2 : Filter. \{dummy(f_1), dummy(f_2)\} dummy(f_1 \cup f_2) \quad (4.7)$$

Remember that function $dummy(f)$ indicates whether we are interested in the value of the comprehension over f in some heap. Obviously this draft makes use of the operator \cup , for which we don't have an encoding yet. This is done with the following function

$$union : (Filter, Filter) \rightarrow Filter$$

which we axiomatize with a similar approach to the one for the axiomatization of *minus*.

$$\forall f_1 : Filter, f_2 : Filter, a : A. \{filtering(union(f_1, f_2), a)\} \\ filtering(union(f_1, f_2), a) \leftrightarrow filtering(f_1, a) \vee filtering(f_2, a) \quad (4.8)$$

The draft 4.7 is however not really good. First of all we introduced a matching loop, because we generate a new term $dummy(f_1 \cup f_2)$, which can replace the trigger term $dummy(f_2)$, such that we get a sequence of instantiations of the form

$$dummy(f_1 \cup f_2), dummy(f_1 \cup f_1 \cup f_2), dummy(f_1 \cup f_1 \cup f_1 \cup f_2), \dots$$

In addition we will allow the union of every pair of filters in scope, which even if we solved the matching loop, would give us way more instantiations than we actually needed. If we look at our problem scenario, we can see that our problem is very specific to the case where we have two sets A and B of disjoint filters, with $\bigcup_{f \in A} f = \bigcup_{f \in B} f$ (the union of all filters in A and the union of all filters in B are equal). In our problem we would want to deduce the value of the comprehension over a filter in A from the value of the comprehension over all other filters in A and all filters in B . This can be achieved by combining all filters in B to deduce $c(h, \bigcup_{f \in B} f)$. With this result we could then apply our general axiom 4.1 multiple times recursively on $c(h, \bigcup_{f \in B} f)$ for filters in A . In our above example, A would be $\{f', f\}$ and B would be $\{f_1, f_2\}$.

To quickly summarize the scenario: We have two sets A and B of *disjoint*, non-empty filters, with $\bigcup_{f \in A} f = \bigcup_{f \in B} f$ for which we know the comprehension in h , except for one $f \in A$, for which we want to deduce the comprehension value. The non-empty requirement of the sets in A and B was not discussed yet, but makes sense in our scenario, since adding an empty filter to either A or B does not change the union over all filters. From this we learn that every filter in B must have a non-empty intersection with some filter in A .

Lemma 4.2 *Let A and B be non-empty sets of disjoint sets of non-empty sets with $\bigcup_{f \in A} f = \bigcup_{f \in B} f$. Then for every $b \in B$, there exists $a \in A$, such that $a \cap b \neq \emptyset$.*

Proof (by contradiction) Assume that there exists $b \in B$ such that for all $a \in A$, $a \cap b = \emptyset$. Then we get

$$\begin{aligned} \bigcup_{f \in A} f &= \bigcup_{f \in B} f \\ \left(\bigcup_{f \in A} f \right) \setminus b &= \left(\bigcup_{f \in B} f \right) \setminus b \end{aligned}$$

Since for all $a \in A$ we know that $a \in A$, $a \cap b = \emptyset$, we get

$$\left(\bigcup_{f \in A} f \right) \cap b = \emptyset$$

and therefore

$$\left(\bigcup_{f \in A} f \right) \setminus b = \bigcup_{f \in A} f$$

This now means that

$$\begin{aligned} \bigcup_{f \in A} &= \left(\bigcup_{f \in B} f \right) \setminus b \\ \left(\bigcup_{f \in B} f \right) \setminus b &= \bigcup_{f \in B} f \\ \left| \left(\bigcup_{f \in B} f \right) \setminus b \right| &= \left| \bigcup_{f \in B} f \right| \end{aligned}$$

Since B and b are non-empty, we however also get

$$\left| \left(\bigcup_{f \in B} f \right) \setminus b \right| < \left| \bigcup_{f \in B} f \right|$$

which is clearly a contradiction. \square

The lemma requires that A and B are non-empty, which is the case for our scenario, since we require that there is a filter in A , for which we don't know the value of the comprehension. We can even restrict ourselves more that A and B must have a size of at least 2, since otherwise we would have a scenario similar to figure 41. Those restrictions can be incorporated into the draft 4.7 to fit the axiom to our scenario.

$$\begin{aligned} &\forall f : \text{Filter}, f_1 : \text{Filter}, f_2 : \text{Filter}. \{ \text{dummy}(f), \text{dummy}(f_1), \text{dummy}(f_2) \} \\ &\neg \text{empty}(f \cap f_1) \wedge \neg \text{empty}(f \cap f_2) \wedge \text{empty}(f_1 \cap f_2) \rightarrow \text{dummy}(f_1 \cup f_2) \quad (4.9) \end{aligned}$$

To make the connection to our formalized scenario, note that f_1 and f_2 would be filters of set B and f would be a filter of set A . With multiple recursive applications of the axiom we would therefore finally learn $\text{dummy}(\bigcup_{f \in B} f)$ which would make the remaining axioms (especially axiom 4.1) to deduce the value of $c(h, \bigcup_{f \in B} f)$. The two conditions $\neg \text{empty}(f \cap f_1)$ and $\neg \text{empty}(f \cap f_2)$ check whether f_1 and f_2 have a non-empty intersection with an element in A and the condition $\text{empty}(f_1 \cap f_2)$ checks whether f_1 and f_2 are disjoint. To be precise, we made the axiom a bit more restrictive in the sense that we require for every filter $f_1 \in B$ to have a non-empty intersection with a filter $f \in A$ and that there is another filter $f_2 \in B$ which also has a non-empty intersection with f . This is in contrast to the restriction that we only require every element $f_1 \in B$ to have a non-empty intersection with an element $f \in A$. This restriction however does not restrict the capabilities of our axioms, because the only case it disallows now, is that there is an element $f_1 \in B$ and an element $f \in A$, such that f_1 and f have a non-empty intersection,

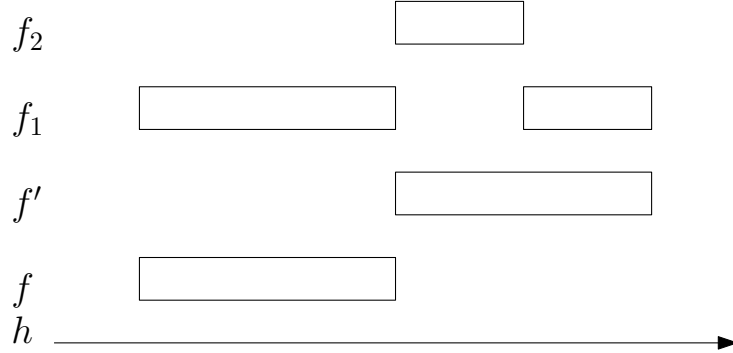


Figure 43: Example of a problem where only f_1 has a non-empty intersection with f

and every element different from f_1 in B has an empty intersection with f . Figure 43 shows an example of such a case. Note that f must now be a subfilter of f_1 , because our scenario still requires that $\bigcup_{f \in A} f = \bigcup_{f \in B} f$. This means that for this special case, we can use axiom 4.1 and split f_1 up into f and $f_1 \setminus f$ and therefore reduce the problem to deriving the value of the comprehension for $f_1 \setminus f$.

We have not yet provided an encoding and an axiomatization for \cap , which we will do now. This is a very straight-forward task, similar to the encoding and axiomatization of \setminus and \cup , based on the common definitions from set-theory. The function encoding the operator has the following signature

$$\text{intersect} : (\text{Filter}, \text{Filter}) \rightarrow \text{Filter}$$

and the following axiomatizations

$$\begin{aligned} \forall f : \text{Filter}, f' : \text{Filter}, a : A. \{ \text{filtering}(\text{intersect}(f, f'), a) \} \\ \text{filtering}(\text{intersect}(f, f'), a) \leftrightarrow \text{filtering}(f) \wedge \text{filtering}(f') \end{aligned} \quad (4.10)$$

4.5 Matching loops

With the new axioms and triggers we now have introduced some matching loops, which we will need to prevent.

4.5.1 General Axiom

First we have introduced a matching loop in the general axiom for the value of a comprehension 4.1. Remember that our current trigger for this axiom is $\{\text{dummy}(f), c(h, f')\}$. Now let's say that the axiom gets instantiated for filters f and f' with $f' \subseteq f$. From the instantiation, the solver will learn

$$c(h, f) = c(h, f') * c(h, f \setminus f')$$

Now the axiom will however get instantiated again from the terms $dummy(f)$ and $c(h, f \setminus f')$. The solver will then be able to prove that $f \setminus f' \subseteq f$ and therefore it will learn

$$c(h, f) = c(h, f \setminus f') * c(h, f \setminus (f \setminus f'))$$

This goes on forever, because the solver does not know $f \setminus (f \setminus f') = f'$.

We could solve this by demanding yet another feature from the SMT-solver. If there was a possibility to prioritize quantifiers when there are multiple possible instantiations, we could prioritize the axiom 4.6 which would prove that $f \setminus (f \setminus f') = f'$ before the general axiom gets instantiated. With this we would avoid this instantiation because of e-matching.

However Z3 does not include such a feature, so we will use a different approach for this. The axiom relies on two filters: a *base filter* (which will be split up by the axiom) and a *subfilter*. The axiom then creates a new filter by subtracting the subfilter from the base filter and therefore creating a diminished version of the base filter. We get a matching loop, because we reuse the diminished version of the base filter as the subfilter in the axiom. However intuitively it makes sense to only allow filters which were created by the user (via calling the comprehension) as the subfilter. If we then recursively apply the general axiom for multiple filters S , for which we know the comprehension value, we eventually have the original base-filter split up into all filters in S and one single filter, whose comprehension value is unknown, which is the recursively diminished version of the original base-filter. So we introduce a new function

$$userCreated : Filter \rightarrow Bool$$

which indicates whether a filter was created by the user, or by the SMT-solver. We can now assume the *userCreated* function for every filter that was indeed created by the user (i.e. every filter that appears in the form of $c(h, f)$ in the program) and trigger the general axiom 4.1 only upon appearance of $userCreated(f')$ for subfilter f' , which leads us to the new trigger $\{dummy(f), c(h, f'), userCreated(f')\}$.

Now there is still an additional matching loop. Let f' be an empty filter. In this case, f' would satisfy the subfilter relation with any f , which means that when the general axiom gets instantiated for base filter f and subfilter f' , the solver would learn

$$c(h, f) = c(h, f') * c(h, f \setminus f') = c(h, f \setminus f')$$

So the general axiom could get instantiated again for base filter $f \setminus f'$ and subfilter f' , since the solver does again not necessarily know that $f \setminus f' = f$. Since this is however a very specific case, we can simply avoid this matching

loop by adding an empty check for the subfilter to the left-hand side of an implication. With those insights we now almost get our final version of the general axiom.

$$\forall h : \text{Heap}, f : \text{Filter}, f' : \text{Filter}. \{ \text{dummy}(f), c(h, f'), \text{userCreated}(f') \} \\ \neg \text{empty}(f') \wedge \neg \text{equiv}(f', f) \wedge f' \subseteq f \rightarrow c(h, f) = c(h, f') * c(h, f \setminus f') \quad (4.11)$$

We have not discussed yet the occurrence of the term $\neg \text{equiv}(f', f)$ on the left-hand side of the implication. This term is simply there for performance reasons. First of all it is obvious that the case where $\equiv (f, f')$ does not contribute anything meaningful to the derivation of $c(h, f)$, the solver would simply learn

$$c(h, f) = c(h, f') * c(h, f \setminus f') = c(h, f) * c(h, f \setminus f) = c(h, f)$$

so this instantiation would be completely unnecessary. Next we know from previous examples that we usually want to know the equivalence relation between two filters as soon as possible, to allow e-matching.

Filter combinations From the axiom for combining filters 4.9 we also get matching loops. The difference here is that we don't get a matching loop only because of this axiom, since the left-hand side of the implication is already restricting enough that the right-hand side won't produce a filter which was used before for this axiom. This is because the right-hand side allows neither f_1 , nor f_2 to be empty, so the resulting filter will have a strictly larger cardinality than f_1 or f_2 .

However by creating a new filter $f_1 \cup f_2$, and learning $\text{dummy}(f_1 \cup f_2)$, the solver will instantiate the general axiom 4.11 for $f_1 \cup f_2$ as base filter and either f_1 or f_2 as subfilter. This will result in a diminished filter $(f_1 \cup f_2) \setminus f_1$ ($(f_1 \cup f_2) \setminus f_2$ respectively) which is again not obviously equivalent to f_1 and will therefore again instantiate axiom 4.9.

A first approach might be to create a new function

$$\text{base} : \text{Filter} \rightarrow \text{Bool}$$

which is assumed for all filters which were not generated by axiom 4.9. To avoid the matching loop, we could then add the term $\text{base}(f)$ to the trigger of the general axiom 4.11. The problem with this approach is however that we have now eliminated the possibility to deduce the value for the comprehension over a subfilter of the combined filter, since we can not split up the combined filter anymore.

Instead we will now use the userCreated function again to prevent a filter, which was not user created, but generated by diminishing a base filter, to

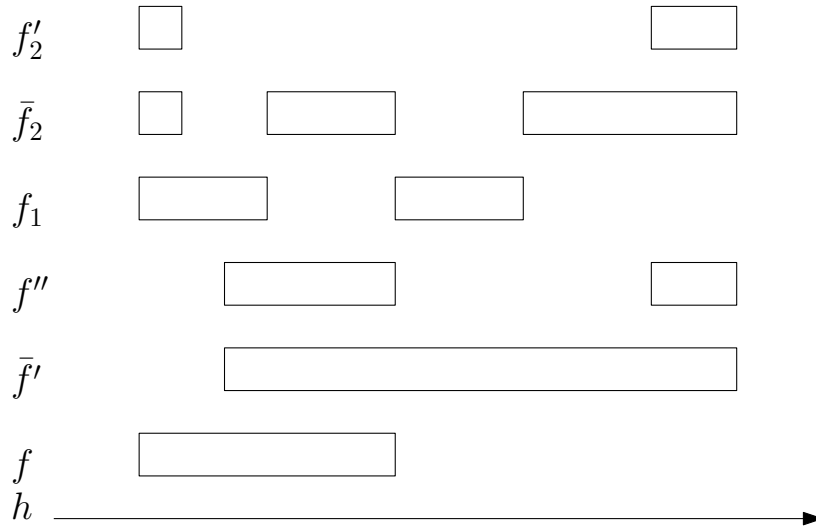


Figure 44: An example showcasing the new limitation

be combined. Since we however rely with the axiom for filter combinations on the possibility to be instantiated recursively, we will also learn from the axiom that the combined filter is user created, therefore rendering the name 'user created' for the function inaccurate. The new axiom now looks as follows.

$$\begin{aligned}
 & \forall f : \text{Filter}, f_1 : \text{Filter}, f_2 : \text{Filter}. \\
 & \{ \text{dummy}(f), \text{dummy}(f_1), \text{dummy}(f_2), \text{userCreated}(f_1), \text{userCreated}(f_2) \} \\
 & \quad \neg \text{empty}(f \cap f_1) \wedge \neg \text{empty}(f \cap f_2) \wedge \text{empty}(f_1 \cap f_2) \\
 & \quad \rightarrow \text{dummy}(f_1 \cup f_2) \wedge \text{userCreated}(f_1 \cup f_2) \quad (4.12)
 \end{aligned}$$

New limitation While avoiding the matching loop for the filter union axiom, we also introduced a limitation: we can no longer combine filters, which are not user created. To see that there are cases, for which this is relevant, consider the example 44. First notice that there is no pair of filters, which are disjoint, so the filter union axiom won't get instantiated. Next notice that there are only two pairs of filters, which form a subfilter relation, which are $f'_2 \subseteq \bar{f}_2$ and $f'' \subseteq \bar{f}'$. Those two pairs will therefore instantiate the general axiom and will generate two new filters $f_2 := \bar{f}_2 \setminus f'_2$ and $f' := \bar{f}' \setminus f''$, which are not user created. Figure 45 shows the result of this subtraction. Now we would like to combine f_1 and f_2 together to then subtract f' and deduce the comprehension value for f . However such a combination is not possible now, because the filters f_2 and f' are not user created.

How relevant this limitation in practice really is, is however hard to tell. Because it is rather complex, one might suppose that this will be rather a

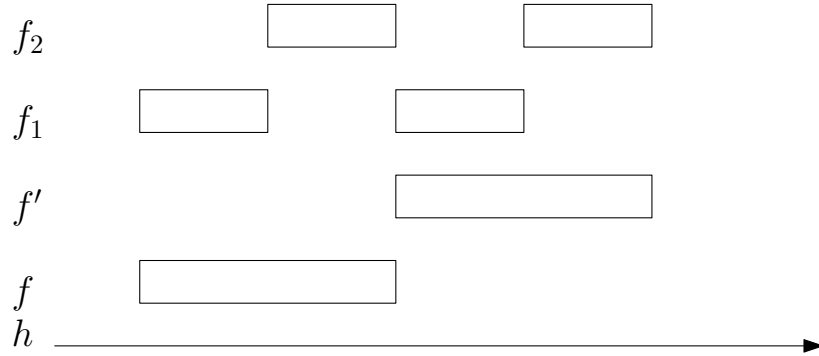


Figure 45: The example of figure 44, after subtraction

theoretical result and that most of the use-cases in practice won't be affected by it.

4.6 Summary

This section again provides a summary of the axioms and declarations of this chapter. Remember that those axioms complement / replace the axioms from the previous chapter.

First we created the general axiom to describe the value of a comprehension.

$$\begin{aligned}
 \forall h : \text{Heap}, f : \text{Filter}, f' : \text{Filter}. \\
 \{ \text{dummy}(f), c(h, f'), \text{userCreated}(f') \} \\
 \neg \text{empty}(f') \wedge \neg \text{equiv}(f', f) \wedge f' \subseteq f \\
 \rightarrow c(h, f) = c(h, f') * c(h, f \setminus f') \quad (4.13)
 \end{aligned}$$

We have created the new filter generating function *minus*, *union*, *intersect*, along with their axiomatizations.

$$\begin{aligned}
 \forall f : \text{Filter}, f' : \text{Filter}, a : A. \{ \text{filtering}(\text{minus}(f, f'), a) \} \\
 \text{filtering}(\text{minus}(f, f'), a) \leftrightarrow \text{filtering}(f, a) \wedge \neg \text{filtering}(f', a) \quad (4.14)
 \end{aligned}$$

$$\begin{aligned}
 \forall f_1 : \text{Filter}, f_2 : \text{Filter}, a : A. \{ \text{filtering}(\text{union}(f_1, f_2), a) \} \\
 \text{filtering}(\text{union}(f_1, f_2), a) \leftrightarrow \text{filtering}(f_1, a) \vee \text{filtering}(f_2, a) \quad (4.15)
 \end{aligned}$$

$$\begin{aligned}
 \forall f : \text{Filter}, f' : \text{Filter}, a : A. \{ \text{filtering}(\text{intersect}(f, f'), a) \} \\
 \text{filtering}(\text{intersect}(f, f'), a) \leftrightarrow \text{filtering}(f) \wedge \text{filtering}(f') \quad (4.16)
 \end{aligned}$$

Also for filters, we have created and axiomatized the two filter relations *subfilter* and *equiv*.

$$\forall f : \text{Filter}, f' : \text{Filter}. \{ \text{subfilter}(f', f) \}$$

$$\text{subfilter}(f', f) \leftrightarrow \text{empty}(\text{minus}(f', f)) \quad (4.17)$$

$$\forall f : \text{Filter}, f' : \text{Filter}. \{ \text{equiv}(f, f') \}$$

$$\text{equiv}(f, f') \leftrightarrow \text{subfilter}(f, f') \wedge \text{subfilter}(f', f) \quad (4.18)$$

Next we have introduced the new dummy function *userCreated* to prevent matching loops. We have created an axiom for proving that two filters are equal.

$$\forall f : \text{Filter}, f' : \text{Filter}. \{ \text{dummy}(f), \text{dummy}(f') \} \text{equiv}(f, f') \leftrightarrow f = f' \quad (4.19)$$

And we have created an axiom to support filter union.

$$\forall f : \text{Filter}, f_1 : \text{Filter}, f_2 : \text{Filter}.$$

$$\{ \text{dummy}(f), \text{dummy}(f_1), \text{dummy}(f_2), \text{userCreated}(f_1), \text{userCreated}(f_2) \}$$

$$\neg \text{empty}(f \cap f_1) \wedge \neg \text{empty}(f \cap f_2) \wedge \text{empty}(f_1 \cap f_2)$$

$$\rightarrow \text{dummy}(f_1 \cup f_2) \wedge \text{userCreated}(f_1 \cup f_2) \quad (4.20)$$

Completeness It might be striking that we only considered one single case of the definition 2.1 of a comprehension and think that we are done. The reason for this is that the axioms from the singleton case already cover the first two cases of the definition.

The first case in the definition is exactly the same as it was for the simplified definition 3.1 in the singleton case. This case was covered by the comprehension axiom 3.11, and will obviously cover it in the general case as well.

For the second case of the definition, consider the following thought. There are always two possibilities to describe the value of a comprehension over a singleton filter f : either by stating the value directly, or by stating the value of the heap location, which f covers. The derivation of the value of a comprehension for f for the first possibility is then covered by axiom 4.13, while the second one is covered by axiom 3.12. The second possibility describes the second case of the definition, but with our encoding we don't require that the value of every heap location is known. Instead we can summarize the value of multiple heap locations via the comprehension value.

To reason about completeness for the general case is hard, since it is already hard to define what 'complete' means. We could argue that we would be

complete, iff we want to prove the value of a comprehension over f in some heap, for which we knew the value of the comprehension for some filters, with which we could express f in some way. The question then is, what ways of expressing f from other filters would we allow. It gets obvious quite soon, that we could only support expressions with applications of \cup and \setminus (and \setminus only on filters with a subfilter relation), because other set operators don't fit our definition 2.1 for the value of a comprehension. This also makes sense intuitively, if we consider an operator, such as \cap . Imagine we knew the comprehension value for f and f' and want to derive the value for $f \cap f'$. We would not be able to do it solely on this knowledge, because we don't know the comprehension value of the complement of $f \cap f'$ respective to f or f' . So for example if we knew that the sum comprehension of f was 5 and the sum comprehension over f' was 4, we have still no idea, what the sum comprehension of $f \cap f'$ was. However if f and f' were disjoint, then we could state that the sum comprehension of $f \cup f'$ is 9, or if $f' \subseteq f$, that the sum comprehension of $f \setminus f'$ is 1. Both of those cases are covered in our axioms, as we have already discussed.

Such a definition of completeness would now however not take into account that want to deduce the value of a comprehension in some heap via values of comprehensions in different heaps, a case we have also covered.

A question that remains is, whether we really need the value of the comprehension for different filters to be able to deduce the value for the desired filter, or whether we can also deduce the value of a comprehension via other means. Indeed a different way is already supported by our axioms: we can state the value for every heap location covered by the desired filter and from this deduce the value of the comprehension. Other possible means, which would require proofs by induction were already discussed in section 4.2 and were considered out of scope for this thesis.

Chapter 5

Analysis

In this chapter, we will provide an informal analysis of the axiomatization we created in the previous In chapters 4 and 3, we already reasoned about the completeness of the singleton case and of the general case. Therefore we will not talk about completeness in this chapter. While we have already partly argued about matching loops in the previous chapters, we were there mainly concert about avoiding specific matching loops. In this chapter we will however discuss, why our axiomatization should not have any matching loops. After the discussions about matching loops, we will take a closer look at the performance of the axiomatization. In the end of the chapter we will summarize all limitations together, which have resulted from our encoding.

5.1 Matching Loop Freedom

One way to reason about matching loops, is to look at what new triggering terms the axioms create during instantiation, and observe which other quantifiers those triggering terms might instantiate.

Filters We start by looking at which axioms create new filters. The only way to create new filters is via the filter generating functions *minus*, *union*, *intersect*, and *narrow*. The only axioms which use them to create new filters are:

- The general axiom 4.13 with *minus*
- The singleton axiom 3.12 with *narrow*
- The filter combining axiom 4.20 with *intersect* and *union*
- The subfilter axiom 4.17 with *minus*

The subfilter axiom creates the term $empty(minus(f', f))$. This term only instantiates the empty axiom 3.14, instantiating an existential quantifier. The

existential will then be skolemized, such that we gain a new argument instance $a : A$, for which we learn the term $filtering(minus(f', f), a)$, which will only trigger the axiomatization 4.14 of *minus*. This instantiation will create the new terms $filtering(f', a)$ and $filtering(f', a)$, which however will not trigger any other axiom.

The general axiom, which is triggered by $dummy(f)$, $base(f)$, $c(h, f')$, and $userCreated(f')$, creates the terms $empty(f')$, $equiv(f', f)$ and $subfilter(f', f)$, and (for a verifiable left-hand side) the term $c(h, minus(f, f'))$.

The term $subfilter(f', f)$ will trigger the subfilter axiom, for which we know from the previous paragraphs, that it does not introduce matching loops.

The term $equiv(f', f)$ will trigger the equivalence axiom 4.18, creating the term $subfilter(f, f')$, for which does not cause a matching loop, as we have just seen.

The term $empty(f')$ triggers the empty axiom, which also does not introduce matching loops, as we have seen for the subfilter axiom.

The term $c(h, minus(f, f'))$ triggers the dummy axiom 3.16, creating the term $dummy(minus(f, f'))$.

It also triggers the empty comprehension axiom 3.11, which however only instantiates the new term $empty(f)$, which will not cause a matching loop.

Additionally it triggers the framing axiom 3.15, which will create an instance of an existential quantifier. This existential will again be skolemized, creating a fresh constant r , for which the term $filtering(f, e^{-1}(r))$, $r.h.v$ will be learned, which will not trigger anything in this case.

The terms $dummy(minus(f, f'))$, $c(h, f')$, and $userCreated(f')$ will now again trigger the general axiom, but the left-hand side will not be verified, because $f' \not\subseteq minus(f, f')$. The terms that were instantiated on the left-hand side will not cause any matching loops.

The singleton axiom, triggered by $dummy(f)$, $r.h.v$, and $userMentioned(r)$ creates terms $filtering(f, e^{-1}(r))$, $c(h, f)$ and $c(h, narrow(f, r))$. The first term will not trigger anything. The second term will trigger the framing axiom, which, as we have seen for the general axiom, will not introduce a matching loop in this case. Otherwise, it will only trigger the dummy axiom and the empty comprehension axiom, which will not introduce anything interesting. The third term will trigger the same axioms as the second term, however this time we learn a new term $dummy(narrow(f, r))$, which will instantiate the singleton axiom again, but this time we will not instantiate the right-hand side, since $\neg filtering(narrow(f, r), e^{-1}(r))$.

The filter combining axiom instantiates three terms which check for *empty*, which we have already seen, does not introduce a matching loop. The interesting part here is the term $dummy(f_1 \cup f_2)$, which triggers the general

axiom. However, the newly created filter from the general axiom will not trigger the filter combining axiom again, and, therefore, there will not be a matching loop. The remaining instantiations are cases that were already discussed.

Others Every other function that occurs in the axioms has return type *Bool*. So, the only new constants that matter for matching loops are the ones generated by skolemization of existentials. There are two cases where we have existentials: in the empty axiom, which was already shown to not be a problem, and in the framing axiom.

The framing axiom creates the terms $filtering(f, e^{-1}(f))$, $r.h_1v$, and $r.h_2v$ via skolemization. These terms however do not trigger anything. For the heap access terms this is because there is no term $userMentioned(r)$, and, therefore, the singleton axiom will not be instantiated.

5.2 Performance

When we talk about performance, we are mainly interested in the number of instantiations the axioms will produce on a specific program. For the singleton case we would like to know the performance based on the number of heap location specifications. We will quickly realize that the number of instantiations is in the worst case exponential to the number of heap location accesses. Imagine we have n heap location accesses via n heap updates and want to deduce the value of a comprehension for a filter f which covers exactly those n heap locations. The singleton axiom 3.12 will then instantiate for f and for all n heap location specification in the heap where the heap location update happened. Therefore we will get n filters with cardinality $n - 1$, which will instantiate the axiom again for all the heap locations which are still covered by the new filter. This goes on, until the resulting filters are empty. In the end we have a series of instantiations for every permutation of the n heap locations, resulting in a complexity of $\mathcal{O}(n^n)$.

If we now look at which permutations can actually be used to prove the value of the comprehension for f , we see that there is exactly one, namely, the permutation in the reverse order of the heap location updates, since, for all other permutations, framing could not be proved in between the n heaps. However there does not seem to be an easy way to prevent this bad performance, since we don't have any possibility for Z3 to prioritize instantiations.

If we look at the general case, we have the exact same problem. With the general axiom, we will get with n specifications of comprehension values for subfilters after heap updates, again $\mathcal{O}(n^n)$ instantiations.

5.3 Limitations

Even though our design is very general, it still has its limitations and cannot support special cases, which a user might want to have. For some limitations we will already provide some approaches for solutions, which will also be discussed in more detail in chapter ??.

Limitations due to the encoding A limitation we already remarked and discussed was the limitation, induced by the prevention of matching loops for the filter combining axiom in subsection 4.4.3. There are certain scenarios, for which the value of a comprehension cannot be deduce automatically, but could be deduced 'by hand'. This is due to the mechanism we proposed for avoiding matching loops. However if we had control over the order of instantiations, if there are multiple possible instantiations, we could prioritize the equal filters axiom 4.19 above all other axioms to avoid the matching loop. Since the matching loop occurred, because we generated a filter, which already instantiated the general axiom 4.13, but which was specified differently during the first instantiation (and therefore the SMT-solver will not automatically interpret it as the same as previously), which instantiates the general axiom again. If we could prove before the second instantiation of the general axiom that the two version of the filter are indeed equal, the second instantiation will not happen. With our current approach we limit the capability of our axiom, because we add additional restrictions to instantiations with a more refined trigger. This approach however would not restrict anything in terms of capabilities, it only specifies which axioms will be instantiated in which order. Therefore this approach would eliminate this limitation. Note that with such control over the SMT-solver, we could eliminate the *userCreated* function completely, since we only created it to prevent a matching loop for the general axiom, where we had the same problem that the generated filter will not be recognized as a filter causing instantiation earlier.

Another limitation we have in the implementation comes with the fact that the *userMentioned* function will only be assumed on reference-typed expressions outside of quantifiers, as will be presented in the next chapter. This limits the capability of, for example, a reference-typed expression instantiated from a forall-quantifier to trigger the singleton axiom 3.12, which might miss an opportunity to prove the comprehension value over a singleton filter, which might be relevant for the comprehension value over a larger filter. The *userMentioned* function is again a way to prevent matching loops, particularly caused by the generation of skolemized references from the framing axiom 3.15. In section 3.4 we already presented two alternative way of avoiding this matching loop. One again requires a special feature from the SMT-solver, while the other doesn't and might actually cause less problems

for the implementation.

The last and probably most grave issue with our axiomatization is the performance issue, discussed in the previous section. While this is technically no limitation in the sense that we cannot deduce comprehension values, it still is a limitation in practice. We could avoid exponential performance for the presented case again, if we had control over the order of instantiations, but this time not in between quantifiers, but for the order of instantiations for available constants which would trigger the quantifiers of the general and the singleton axioms. We could create a relation between heaps, which indicates for two heaps, which heap was part of the state before the other heap (i.e. which heap is more recent). We then prioritize the quantifiers for more recent heaps, giving us the correct permutation straight ahead, resulting in a linear number of instantiations. This however does not mean that the resulting implementation would have a linear runtime.

Limitations due to the design Our comprehension definition already gives rise to a number of limitations, especially from the requirements on the components of a comprehension. The first limitation was already discussed in the introduction. Our design only allows comprehensions over heap locations. This means that everything we want to comprehend over, requires us to specify it on the heap. For example in the implementation we cannot comprehend directly over the values of a built-in Viper set. Instead we would need to let the set contain references and specify the content of the set on the heap via such a set of references. We could solve this issue by allowing arbitrary expressions for the body of a comprehension. This will however be no trivial generalization, since the axiomatization currently depends on the injectivity of the receiver expression.

The next limitation we have is that the receiver expression of a comprehension cannot be heap-dependent. This prevents some interesting applications of comprehensions. For example we could specify a multiset as a set of references where the content of the multiset would be the values of the heap locations of those references. A multiset of references would then be specified with reference-typed fields. Figure 51 illustrates such a multiset of references. The first row of arrows indicates the references specifying the multiset, while the second row of arrows indicates the references (the content) of the multiset. There might be scenarios, where we would want to comprehend over the values of the final heap locations (the column of pure dots in the figure). However since we don't allow the receiver expression to be heap-dependent, there is no trivial way to do this.

Another limitation we might get is based on the design of the implementation. As we will see in the next chapter, we do not allow so called *outer variables*, variables other than the comprehension arguments to be used in

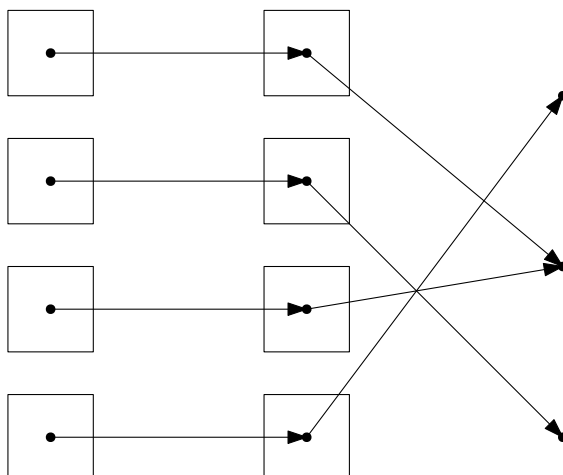


Figure 51: Illustration of a multiset of references

the body. However this supposed limitation can be circumvented by adding an additional comprehension argument a of the same type as the outer variable o we wish to mention in the body and setting $a = o$ in the filter. We can then use a in the body where we would use o .

The next limitation has to do with the binary operator. We make certain requirements on the binary operator, which however all intuitively seem to be basic requirements for any comprehension definition. Still we could make those requirements a bit less tight, such that they are only required for the values which are evaluated while evaluating the comprehension value. For example if we knew that the value of every integer heap location is at least 0, we could require from the binary operator to be commutative only for positive integers. This relaxation would not interfere with any axioms we have provided and could easily be implemented. However for the user this might cause confusing failures, when the comprehension value fails to be deduced because some heap location didn't have a correct value.

One of the more problematic limitations was discussed in section 4.2. We discovered that we only support derivations of comprehension values via direct assertions of heap locations, or via assertions of comprehension values for different filters, but for the same comprehension. We argued that this is because of the requirement of automatic induction proofs. This is definitely no limitation that can be trivially solved.

Implementation

The previous chapters provided a technique, encoding comprehensions for SMT. In this chapter, we will look at how we could build a feature for comprehension support for the Viper infrastructure and discuss the implementation of such a feature for the Viper verifier, based on verification condition generation for Boogie [5]. We will however not explain in detail how the feature is implemented, only specific challenges of the implementation process will be discussed. An interested reader could look into the sources of the implementation, which can be found at [3] [2].

6.1 Syntax

For simplicity, we will only provide a syntax for specifying the value of a comprehension, with which we will provide all necessary information about a comprehension. From the comprehension calls, the implementation will then detect which comprehensions occur and will generate an axiomatization for every comprehension. A comprehension call will be translated into the simple function $c(h, f)$. The syntax for specifying the value of a comprehension looks as follows:

```
comp[b, u] a: A :: { e(a).v | f(a) }
```

b denotes the binary operator, which is passed as a function name of an interpreted function. u is the unit, an expression with a static value. $e(a).v$ is the *body* of the comprehension. $f(a)$ is the filter, a boolean-typed expression. $a: A$ declare the arguments of the comprehension, the 'quantified' variables.

The part of the syntax after $::$ should remind of the set-builder notation for sets. The set described in this part is the set of the values the comprehension comprehends over.

Translation of filters We do not impose any restrictions on filters, which means that they can contain arbitrary variables and even be heap-dependent. To make this possible, we will create a new function $filt : V \rightarrow Filter$ for every filter, which uses *outer variables* $v : V$, which are all the variables used, other than the comprehension arguments. The filtering expression will be assumed for $filt$ with the following axiom.

$$\forall v : V. \{filt(v)\} \forall a : A. \{filtering(filt(v), a)\} filtering(filt(v), a) \leftrightarrow f(a)$$

If the filter has no outer variables, then it will simply be translated into a constant $filt : Filter$, which we axiomatize with the inner quantifier of the above axiom:

$$\forall a : A. \{filtering(filt, a)\} filtering(filt, a) \leftrightarrow f(a)$$

6.2 Well-Definedness checks

The definition of a comprehension imposes some requirements on the properties of the components. Some requirements can be checked statically, while others must be checked in the program. The requirements that can be checked statically are

- Types of the components
- Arity of the operator
- Static value of the unit (i.e. the unit does not contain variables)
- Comprehension arguments as the only variables in the body
- Heap-independence of the body

Note that the requirement that the body does not contain outer variables was not required by the comprehension definition, but makes sense, because we don't want to let the receiver expression depend on the state, for the same argument as why we require heap-independence. This requirement is however not a real limitation, as discussed in the previous chapter in section .

For the requirements that need to be checked via verification conditions, we have a few, which can be checked in a separate procedure via assertions. The first one is the commutativity check of the binary operator, which can be checked by asserting the following quantifier.

$$\forall x_1 : T, x_2 : T. x_1 * x_2 = x_2 * x_1$$

The next check is the associativity check, which can be checked by asserting the following quantifier.

$$\forall x : T, y : T, z : T. ((x * y) * z) = (x * (y * z))$$

Next we have the unit check, which can be checked by asserting the following quantifier.

$$\forall x : T. x * \mathbf{1} = x$$

The next check is the receiver injectivity check. At this point we however weaken the restriction, making the feature more powerful. We check for every comprehension call, whether the receiver is injective on the provided filter. We could easily verify that this weakening does not conflict with anything we have said so far, since only require the injectivity of the receiver to be able to create an inverse function, which will only be used on references, covered by the filter of a comprehension call. This weakening however also means that we will make the injectivity check locally, for every comprehension call. This can again be done by asserting a quantifier that looks like this.

$$\forall a_1 : A, a_2 : A. \text{filtering}(f, a_1) \wedge \text{filtering}(f, a_2) \wedge a_1 \neq a_2 \rightarrow e(a_1) \neq e(a_2)$$

6.3 Assumptions

Next to the assertions for the definedness checks, there will be a few assumptions. First we have the axiomatization of the inverse function. Since this axiomatization does now depend on a filter, it will be assumed alongside every comprehension call. There are two assumptions axiomatizing the inverse function of a receiver.

$$\forall a : A. \{e(a)\} e^{-1}(e(a)) = a$$

$$\forall r : \text{Ref}. \{e^{-1}(r)\} e(e^{-1}(r)) = r$$

The next assumption that will cause technical difficulties, is the *userMentioned* assumption. It should be assumed for all reference typed expressions from the user. However there is no obvious way to support reference-typed expressions inside quantifiers, since they might depend on the quantified variables. For this reason, we will only assume the *userMentioned* function for 'top-level' reference-typed expressions. This will, of course, impose some limitations on the feature, which we have, however, already explored in the previous section.

Evaluation

For the evaluation of our implementation, we created some unit tests, which can be found at [3]. A selection of those tests can be found in the appendix A. The code A.1 shows our standard encoding of an array in Viper. Comprehensions over arrays are most commonly used, since they are one of the most common datastructures and fit well into this scheme for comprehensions.

Our first test A.2 is a very basic test, incorporating only a single heap update to a single location. We use here a sum comprehension over an array as the comprehension of choice. However the value of the comprehension is assumed in the precondition of the method via two complementing filters over the array. In the end the comprehension value over the full array is asserted. This example verifies, and if the comprehension is asserted to a different number in the end, the example fails to verify. With this we already showed that the singleton and the general axiom behave as assumed for at least some cases.

Next up we have a simple test A.3, where we update the heap via a method call. We again use a sum comprehension over an array. The change of the comprehension value is specified in the post-condition of the method. The comprehension at the start of the test again gets assumed via the precondition with two filters. The method however has a different footprint than the two filters in the precondition, therefore we don't have a simple comprehension value update, but the value of the comprehension over the full array must be deduced before the method call, in order to be able to register the change across the method call. This example also verifies, and fails to verify for a wrong assertion. We have also tested method updates together with singleton updates, which works as expected.

The last example A.4 in the appendix is a test for the limitation we have for the union axiom, discussed in subsection 4.4.3 and in section 6.2. As expected, the example won't verify. However when testing the union axiom,

we got a new limitation, based on our axiomatization. The SMT-solver will fail to prove the right-hand-side of the implication of axiom 4.20, especially the two requirements, indicating that the intersection between f and f_1 or f_2 must be empty, fail to be verified. The reason for this seems to be a problem with quantifier instantiation. With some transformations, there will eventually be a forall quantifier, which gives us, when instantiated, the right-hand-side of the implication. However this requires a 'witness' to trigger the quantifier, in our case we need a comprehension argument, for which we know that it is covered by the intersection between f and f_1 and an argument, which is covered by the intersection between f and f_1 . Since there will often be no such witness, we have a more major limitation in our encoding.

Conclusion and Future Work

Conclusion In this thesis we have provided a new definition of a comprehension. The provided definition is in many ways more general than previous definition approaches from other projects. We have then put together a solid axiomatization for SMT to automatically reason about comprehension values. Alongside with the SMT-encoding, we provided an in-depth analysis about the axiomatization, where we discussed the completeness, the absence of matching loops, the performance, and the limitations of the encoding. Together with this axiomatization we induced a way of specifying a comprehension value in SMT. We then implemented this axiomatization as a new feature for the Viper language with a syntax to specify the value of a comprehension in Viper. The axiomatization of the comprehension value is then implemented to be generated by the verification condition generating verifier of Viper and tested. The implementation is tested on different unit-test, which served as a proof of concept that the feature can be used in practice. While the resulting encoding is very powerful, it must be used with care, because of its supposed performance issues.

Future work With this thesis we have provided a technique for encoding a very general form of a comprehension. There are however still a lot of possible extensions, which could be explored further. The limitations section 6.2 in chapter 5 already provided a list of possible extensions with basic approaches how to realize them, to eliminate the limitations of the current technique. In this section we will provide another powerful extension to our system, which would allow the specification of more advanced comprehensions, such as sortedness of an array.

One major problem of the current system is that the values of the heap locations are fed directly into the binary operator to calculate the value of a comprehension. In some cases it might be useful to preprocess those values before giving them to the binary operator. This could be done with an

8. CONCLUSION AND FUTURE WORK

additional function *transform*, which would be part of the comprehension definition. We give this function the following signature.

$$\textit{transform} : V \rightarrow T$$

where V would be a new type, also part of the comprehension definition. Fields would no longer have type T but now they would have type V , while T is still the comprehension type. We call such a function *transformer* of a comprehension.

If we would like to specify the sort comprehension over an array, we could encode the array, such that every entry holds a tuple (c, n) with the value of the current entry and the value of the next entry. A sort comprehension over integer arrays would then use a transformer of type $(Int, Int) \rightarrow Bool$, which indicates whether the tuple of an entry is sorted (e.g. the first entry is smaller or equal to the second entry). The binary operator of the comprehension would then be \wedge and the unit would be *true*.

The next step would however be to create a solid test suite for the proposed feature, since only rough unit tests could be done until now.

In terms of implementation for the Viper project, the next step would be to encode the feature for the verifier based on symbolic execution.

Appendix A

Appendix

Source Code A.1: Standard encoding of an array

```
1 domain Array {
2   function loc(a: Array, i: Int): Ref
3   function len(a: Array): Int
4   function array(r: Ref): Array
5   function index(r: Ref): Int
6
7   axiom allDiff {
8     forall a: Array, i: Int :: {loc(a, i)}
9       array(loc(a, i)) == a && index(loc(a, i)) == i
10  }
11
12  axiom lenNonNeg {
13    forall a: Array :: {len(a)}
14      len(a) >= 0
15  }
16 }
17
18 define in_range(i, a)
19   i >= 0 && i < len(a)
```

Source Code A.2: Test for singleton update

```
1  field val: Int
2
3  function add(l: Int, r: Int): Int
4      ensures result == l + r
5
6  // test singleton heap update
7  method test1(a1: Array)
8      requires forall i: Int :: {loc(a1, i).val}
9          in_range(i, a1) ==> acc(loc(a1,i).val)
10     requires (comp[add, 0] i: Int, a: Array ::
11         {loc(a, i).val | in_range(i, a) && i%2 == 0 && a == a1}) == 5
12     requires (comp[add, 0] i: Int, a: Array ::
13         {loc(a,i).val | in_range(i, a) && i%2 == 1 && a == a1}) == 2
14 {
15     var i1: Int
16     assume in_range(i1, a1) && i1 % 2 == 0
17     var r: Ref
18     assume loc(a1, i1) == r
19     r.val := r.val + 5
20     assert (comp[add, 0] i: Int, a: Array ::
21         {loc(a,i).val | in_range(i, a) && a == a1}) == 12
22 }
```

Source Code A.3: Test for method updates

```
1  field val: Int
2
3  function add(l: Int, r: Int): Int
4      ensures result == l + r
5
6  method m(a1: Array)
7      requires forall i: Int :: {loc(a1, i).val}
8          i >= 0 && i <= len(a1)  $\square$  2 ==> acc(loc(a1,i).val)
9      ensures forall i: Int :: {loc(a1, i).val}
10         i >= 0 && i <= len(a1)  $\square$  2 ==> acc(loc(a1, i).val)
11     ensures (comp[add, 0] i: Int, a: Array ::
12         {loc(a,i).val | i >= 0 && i <= len(a)  $\square$  2 && a == a1}) ==
13         old(comp[add, 0] i: Int, a: Array ::
14             {loc(a,i).val | i >= 0 && i <= len(a)  $\square$  2 && a == a1}) - 5
15
16 // test general heap update
17 method test(a1: Array)
18     // access to full array
19     requires forall i: Int :: {loc(a1, i).val}
20         in_range(i, a1) ==> acc(loc(a1,i).val)
21     requires len(a1) > 0
22     requires (comp[add, 0] i: Int, a: Array ::
23         {loc(a, i).val | in_range(i, a) && i%2 == 0 && a == a1}) == 5
24     requires (comp[add, 0] i: Int, a: Array ::
25         {loc(a,i).val | in_range(i, a) && i%2 == 1 && a == a1}) == 2
26 {
27     m(a1)
28     assert (comp[add, 0] i: Int, a: Array ::
29         {loc(a,i).val | in_range(i, a) && a == a1}) == 2
30 }
```

Source Code A.4: Test for exploring the limitation of the union axiom

```
1 field val: Int
2
3 function add(l: Int, r: Int): Int
4     ensures result == l+r
5
6 method test1(a: Array)
7     requires forall i: Int :: {loc(a, i).val}
8         i >= 0 && i < 7 ==> acc(loc(a, i).val)
9     requires len(a) == 7
10    requires comp[add, 0] i: Int, a1: Array ::
11        {loc(a1, i).val | a1 == a &&
12            (i == 0 || i == 6)} == 5 // f_2'
13    requires comp[add, 0] i: Int, a1: Array ::
14        {loc(a1, i).val | a1 == a &&
15            (i == 0 || i == 3 || i == 5 || i == 6)} == 5 // \bar{f_2}
16    requires comp[add, 0] i: Int, a1: Array ::
17        {loc(a1, i).val | a1 == a &&
18            (i == 0 || i == 1 || i == 2 || i == 4) } == 5 // f_1
19    requires comp[add, 0] i: Int, a1: Array ::
20        {loc(a1, i).val | a1 == a &&
21            (i == 2 || i == 3 || i == 6) } == 5 // f''
22    requires comp[add, 0] i: Int, a1: Array ::
23        {loc(a1, i).val | a1 == a &&
24            i >= 2 && i < 7} == 5 // f'
25    {
26        //:: ExpectedOutput(assert.failed:assertion.false)
27        assert comp[add, 0] i: Int, a1: Array ::
28            {loc(a1, i).val | a1 == a &&
29                i >= 0 && i < 4} == 5 // f
30    }
```

Bibliography

- [1] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
- [2] Thierry Hörmann. Carbon bitbucket. <https://bitbucket.org/tierrimator/carbon>, 2018. Accessed: 2018-10-02.
- [3] Thierry Hörmann. Silver bitbucket. <https://bitbucket.org/tierrimator/silver>, 2018. Accessed: 2018-10-02.
- [4] K. R. M. Leino and P. Müller. Using the spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *Lecture Notes in Computer Science*, pages 91–139. Springer-Verlag, 2010.
- [5] Rustan Leino. This is boogie 2. Microsoft Research, June 2008.
- [6] Rustan Leino and Rosemary Monahan. Reasoning about comprehensions with first-order smt solvers. March 2009.
- [7] Ueli Maurer. Diskrete mathematik. ETH Zürich Departement Informatik, 2015.
- [8] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT '09*, pages 20–29, New York, NY, USA, 2009. ACM.
- [9] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.

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

- [11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.



Declaration of originality

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

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

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

Title of work (in block letters):

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

First name(s):


With my signature I confirm that

- I have committed none of the forms of plagiarism described in the ['Citation etiquette'](#) information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date

Signature(s)

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