ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Checking Termination of Abstraction Functions

Bachelor's Thesis

P. Gruntz

September 1, 2017

Advisors: Malte Schwerhoff, Peter Müller

Chair of Programming Methodolgy
Department of Computer Science, ETH Zürich

Contents

1 Introduction **1**
- 1.1 Introduction to Viper . . . 1
- 1.2 Motivation . . . 3
- 1.3 Proving Termination . . . 4
  - 1.3.1 Variants . . . 4
  - 1.3.2 Heap Based Termination . . . 4
  - 1.3.3 Definitional Axioms . . . 4
  - 1.3.4 Viper-to-Viper transformation . . . 4
  - 1.3.5 Failures . . . 4
- 1.4 Chapter Overview . . . 5

2 Decreasing Clause **7**
- 2.1 Defining a Decreasing Clause . . . 8
- 2.2 Well-Founded Relation . . . 8
  - 2.2.1 Built-in Types . . . 9
  - 2.2.2 Tuples . . . 9
  - 2.2.3 Decreasing Accessible Heap Location . . . 10
- 2.3 User Defined Types . . . 11

3 Program Transformation **13**
- 3.1 Transformations . . . 14
  - 3.1.1 Generation of the proof methods . . . 14
  - 3.1.2 Function Arguments . . . 15
  - 3.1.3 Short-Circuiting Evaluation . . . 15
- 3.2 Predicates . . . 16
  - 3.2.1 Representing Predicate Instances . . . 17
  - 3.2.2 Assumptions about Nested Predicates . . . 18
- 3.3 Implementation . . . 19

4 Definitional Axioms **23**

4.1 Example 24
4.2 Implementation of the dummy functions 25
4.3 Implementation 26

5 Error Message Transformations **27**
5.1 Variant not Decreasing 28
5.2 Variant not Bounded 28
5.3 `Decreases *` 29
5.4 Decreasing Clause not defined 30

6 Evaluation **33**
6.1 Testing 33
6.2 Outcome 33
6.2.1 Testing the additional time for the generation 33
6.2.2 Testing the verification time for the generated termination checks 34
6.3 Conclusion 35

7 Future Work **37**
7.1 Limitations 37
7.1.1 Multiple precondition errors 37
7.1.2 Functions in the decreasing clause 38
7.2 Extensions 40

Bibliography **43**

Chapter 1

Introduction

In deductive software verification, the programs to verify are usually annotated with specifications (or contracts) in order to express what the program is supposed to do. A method specification, for example, usually consists of a precondition and a postcondition, which summarizes the effects of an invocation of that method. A framework where programs plus specifications written in a source language (for example Java) can be encoded as programs in an intermediate language, is the Viper verification infrastructure [1], which has been developed at ETH Zürich.

A source program is encoded in the intermediate verification language Viper and can then be verified either by Carbon, a verifier which is based on verification condition generation or by Silicon which is based on symbolic execution. Both back-ends then use the Z3 SMT solver, a theorem prover from Microsoft Research (see figure 1.1). The key property of such a encoding is that the source program is correct (with respect to its specifications) if the encoding in Viper verifies.

1.1 Introduction to Viper

Listing 1.1: Implementation of a simple (abstraction) function in Viper

```
1 function sum(xs:Seq[Int]):Int
2 requires |xs| >= 0
3 {
4     |xs| == 0 ? 0 : xs[0]+sum(xs[1..])
5 }
```

The example in listing 1.1 shows a function encoded in Viper that computes the sum of the elements of a sequence. The ternary operation `(...?...:...)` is semantically equivalent to the conditional expression `(if...then...else...)`.

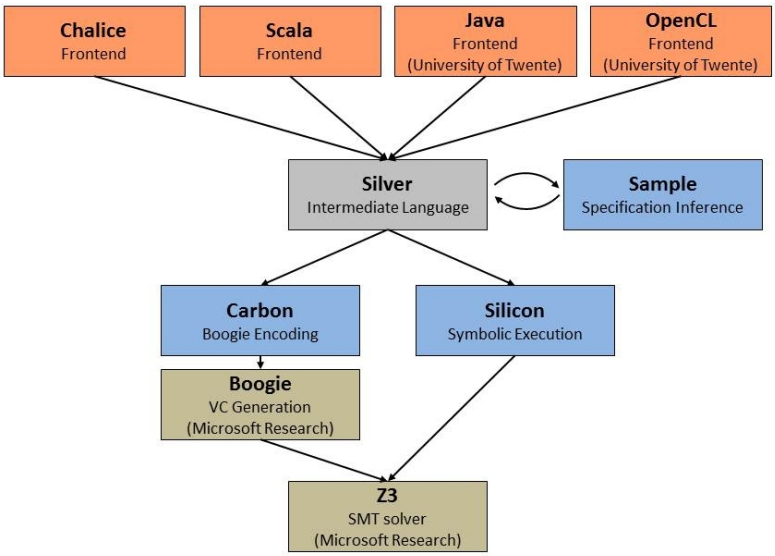

Figure 1.1: The Viper infrastructure

`|xs|` represents the number of elements in the sequence and `xs[i]` returns the *i*th element of `xs`. The argument `xs[1..]` is the sequence `xs` without the first element. The method declaration also includes a precondtion to constrain the argument to lists with at least one element.

To reason about heap-manipulating programs, Viper supports *permissions*. They define the parts of the heap that can be accessed by a function or a method. A heap location can only be accessed if the corresponding permission is known to be currently held. This can be denoted by accessibility predicates, e.g. in preconditions.

Listing 1.2: Simple example of accessing a heap location

```
1 field val: Int
2
3 method incr(ref : Ref)
4   requires acc(ref.val)
5 {
6   ref.val := ref.val + 1
7 }
```

In listing 1.2., in the precondition of the method `incr` an accessible predicate `acc(ref.val)` is used to specify read and write permissions to the location `ref.val`. The method is then able to increment the location and store it again on the heap.

To express access to a recursively defined heap structure we can use *pred-*

icates. Holding an instance of a predicate is analogous to holding a permission to a heap location. The assertion `unfolding P(...)in...` temporarily exchanges the predicate instance `P(...)` for its body.

Listing 1.3: Implementation of adding elements on the heap together

```
1  field val: Int
2  field next: Ref
3
4  predicate list(ref: Ref) {
5     ref != null ==> acc(ref.val) && acc(ref.next) && list(ref.next)
6  }
7
8  function list_sum(x: Ref): Int
9     requires list(x)
10 {
11    unfolding list(x) in x == null ? 0 : x.val + list_sum(x.next)
12 }
```

In listing 1.3 the predicate `list` represents permissions to the fields `val` and `next` and furthermore recursively to the tail of the list starting at `ref.next`. The function `list_sum` requires an instance of the `list` predicate for its parameter `x`. With `unfolding list(x) in (...)` the function temporarily loses permission to the predicate instance but gains the permission of its body. Thus it holds the permissions to `x.val`, `x.next` and `list(x.next)` (assuming that x is not null). With that the function can call `list_sum(x.next)` which requires access to `list(x.next)`.

1.2 Motivation

It is common practice to use abstraction functions, such as the aforementioned functions `sum` or `list_sum`, in program specifications in order to abstract over implementational details. These two abstraction functions can be used for example in the postcondition of an append method to state that the sum of all elements of the resulting sequence is the sum of the input sequence plus the newly appended value. Each verifier in the Viper framework has to check that user-provided abstraction functions are well-defined [2], e.g. check for possible division by zero. Another property of well-definedness is the termination of functions, but this check is currently not implemented at all.

The main goal of this Bachelor's thesis is therefore to define proof obligations in Viper such that the termination of functions can be proven.

1.3 Proving Termination

1.3.1 Variants

In order to show termination of functions, Viper can be used to prove that a given value - a so-called variant [3] - decreases at every recursive function call. This variant has to be strictly decreasing and bounded. In listing 1.1, the length of the sequence (i.e. `|xs|`) would be an example of such a variant.

1.3.2 Heap Based Termination

To show termination it is also possible to implement a variant to show that the number of heap locations accessible to the function is bounded and decreases with every function call. This variant can then be used to prove termination of heap-dependent functions. In listing 1.3 for example termination of the function `list_sum` can be shown by proving that the amount of accessible locations is reduced by each function call. The recursive call is only able to access the list's tail starting at `x.next`, which is only a part of the whole accessible list. The amount of accessible locations on the heap will therefore be reduced with every function call and since the heap is finite, the functions will terminate.

1.3.3 Definitional Axioms

To verify the correctness of functions the verifier uses different axioms (definitional axioms). One is an implication that says, that when the preconditions (conditions which hold before an execution) of a function holds then the postconditions (conditions which hold after the execution) must also hold. This axiom however is not valid when the function won't terminate. Therefore these axioms have to be ignored until the function is guaranteed to terminate. In general we have to make sure that these *definitional axioms* are available only after we verified termination.

1.3.4 Viper-to-Viper transformation

Since Viper is an expressive language, this thesis will present a Viper-to-Viper transformation that encodes, for a given Viper program, the termination-related proof obligations as another Viper program. The abstraction functions in the original Viper program are then guaranteed to terminate if the generated Viper program verifies this obligations, which consists of checking that a variant decreases and is bounded.

1.3.5 Failures

The Viper-to-Viper transformation is generated automatically and transparent to users. Hence can errors in the termination-related proof obligations,

which consist of e.g. a message saying that a variant is not decreasing, be irritating to users. We therefore implemented meaningful error-messages such that the user can understand what the corresponding termination failure is. For example, the reported error for the recursive function `fun` in listing 1.4 is: *"Termination of function f might not hold. The decreasing clause might not decrease"*

Listing 1.4: Example of a non-terminating function

```
1 function fun(x: Int):Int
2 requires x > 0
3 {
4     fun(x)
5 }
```

1.4 Chapter Overview

The main part of the thesis will be discussed in chapter 2 and 3, where the implementation of the Viper-to-Viper transformation is explained. It will cover how a user can define a decreasing clause consisting of several variants (2.1 and 2.3) and how termination for different types of variants is be proven (2.2). Chapter 3 gives an overview of how the transformation will look like. It will cover termination proofs with variants (3.1) and heap based termination measures (3.2). Chapter 4 will explain an mechanism that ensures that the definitional axioms for a function are available only once the function is known to terminate. In chapter 5 the mechanism for translating error messages as well as the additionally implemented error messages for termination-related failures are covered. Chapter 6 will be an evaluation of the implementation with known termination problems and will demonstrate the performance and expressiveness of the chosen approach. In the last chapter some ideas for further improvements regarding termination checks are explained.

Chapter 2

Decreasing Clause

To show termination of a function we have to verify, at every recursive call, that a defined variant monotonically decreases. This variant should decrease with respect to a *well-founded relation*. *A* well-founded relation is a (partial order) relation on a set *A*, where every non-empty subset of *A* has a least element with respect to this relation. Hence the variant cannot decrease forever.

We implemented a well-founded relation with the help of two other relations for every type in Viper: the *decrease relation* ensures that a variant always strictly decreases at every recursive call at least with some amount and the *bounded relation* ensures that the variant doesn't decrease infinitely. The user can specify a variant decreasing clause at the beginning of a function, which then will be used for proving termination by verifying that it is in its corresponding decreases relation.

This chapter will give an overview how a user can define a variant in a so-called decreasing clause, how it will be interpreted and how the decrease and bounded relations are implemented in Viper. A simple example is demonstrated in listing 2.1, where in line two a decreasing clause is specified.

Listing 2.1: Simple terminating recursive function

```
1 function fun(i:Int):Int
2   decreases i
3 {
4    i < 5 ? 0 : fun(i-1)
5 }
```

2.1 Defining a Decreasing Clause

We changed the parser of Viper such that the user can specify a decreasing clause. The position in the program where such a decreasing clause can be defined is after the post-conditions and before the body of a function.

Basically, there are three possibilities how a user can specify a decreasing clause:

1. The user can specify a sequence of expressions: `decreases e1, e2, e3...` Variables appearing in any of these expressions have to be parameters of the function. If there are more than one expression, the decreasing clause will be interpreted as an n-tuple (sequence of n expressions).
2. The user can write `decreases *`, which will prevent any termination checks of this function. It can be used if the user supposes that this function might not terminate or Viper can't prove its termination. Every other function not annotated with a `decreases *` that will call this function will report an error (see chapter 5.1.4).
3. The user defines no decreasing expressions (no `decreases` at all). In this case the body will be checked for possible direct or indirect recursions and an error will be reported if there is at least one (see Chapter 5.1.3).

In listing 2.2. the decreasing clause will be interpreted as a tuple of the value i and j.

Listing 2.2: Decreasing tuples

```
1 function fun(i:Int, j:Int):Int
2   requires i>=0 && j>=0
3   decreases i,j
4 {
5    i < 1 ? 1 : (j < 5 ? fun(i-1,j) : fun(i,j-1))
6 }
```

2.2 Well-Founded Relation

In order to prove termination of the function `fun` in listing 2.2 we have to define a well-founded relation for the decreasing clause and therefore we need the decrease and the bounded relation for the type `Int`. The relations for all relevant Viper types T have the form:

$$\ll : T \times T \rightarrow \mathit{Boolean} \text{ (decrease relation)}$$
$$\prec : T \rightarrow \mathit{Boolean} \text{ (bounded relation)}$$

The definitions of these functions can be imported from Viper's standard library or be defined by the user (see Chapter 2.4: User defined Types)

2.2.1 Built-in Types

The default decrease relations of the Viper built-int types are the following: (where *null* is an empty reference and *write* represents full permission)

- Integers: $i_1 \ll i_2 \Leftrightarrow i_1 < i_2$
- Booleans: $b_1 \ll b_2 \Leftrightarrow b_1 \wedge \neg b_2$
- Sequences: $s_1 \ll s_2 \Leftrightarrow |s_1| < |s_2|$
- Sets: $s_1 \ll s_2 \Leftrightarrow |s_1| < |s_2|$
- Multisets: $m_1 \ll m_2 \Leftrightarrow |m_1| < |m_2|$
- References: $r_1 \ll r_2 \Leftrightarrow r_1 = null \wedge r_2 \neq null$

Permission are represented in Viper by any positive rational number, where the value 1 can also be written with the keyword *write* and the value 0 with the keyword *none*. We can't use the same relation as we used for integers, since for example if we half the permission on every recursive call, the permission would decrease but we can't define a bound which it would reach in any case. Therefore we ensure the well-founded relation by defining that a permissions decreases exactly when it shrinks by the amount of 1.

- Permissions: $p_1 \ll p_2 \Leftrightarrow p_1 \leq p_2 - write$

The bounded relations are only explicitly defined for integers and permissions. The other types (booleans, sequences, sets, multisets and references) are bounded due to their implementation in Viper and therefore, all expressions with one of these types T hold:

- T: $\prec t \Leftrightarrow true$

For integers and permissions hold:

- Integers: $\prec i_1 \Leftrightarrow 0 \leq i_1$
- Permissions: $\prec p_1 \Leftrightarrow none \leq p_1$

2.2.2 Tuples

To prove termination of functions with tuples defined in their decreasing clause (see listing 2.2) we have to define what the decrease and bounded relation means for tuples.

Let t_1 and t_2 be two n-tuples with the sequences $t_1 = (v_1, v_2, ..., v_n)$ and

$t_2 = (w_1, w_2, ..., w_n)$, then the decreasig-relation of two n-tuples is defined as:

$$t_1 \ll t_2 \Leftrightarrow \exists k \geq 1 \wedge k \leq n \bullet (v_k \ll w_k \wedge \forall j \geq 1 \wedge j < k \bullet (v_j = w_j))$$

This ordering is also called *lexicographical ordering*. For a tuple to be in the bounded relation every value has to be bounded:

$$\lessdot t_1 \Leftrightarrow \ \lessdot v_1 \wedge \lessdot v_2 \wedge \ldots \wedge \lessdot v_n$$

A 2-tuple therefore decreases if the first value in the tuple decreases or if the second one decreases while the first one remains the same, and it is bounded if the first and second value are bounded.

In our example in listing 2.2 there are two recursive function calls. We see that in both calls (`fun(i-1,j)` and `fun(i,j-1)`) the decreasing clause, according to the decrease relation of tuples and integers, decreases. And since $i \geq 1$ and $0 \leq j < 5$ holds (using preconditions) before the first recursive call and $i \geq 1$ and $j \geq 5$ holds before the second recursive call, the decreasing clause is in both times bounded. Hence the defined decreasing clause is a valid variant in a well-ordered relation and thus the function will terminate.

2.2.3 Decreasing Accessible Heap Location

Functions in Viper are pure and can therefore not lose or gain accessible heap locations during their execution. But with the expression `unfolding in (...)` functions are able to call themselves recursively with fewer accessible heap locations. Let us now consider the example we introduced in listing 1.3. After the unfolding the function `list_sum` holds the permission to `x.val`, `x.next` and `list(x.next)`. The recursive call of function `list_sum` can then only access `list(x.next)` but neither `x.val` nor `x.next`. We can argue now that if a function always passes fewer accessible heap locations to another function the overall call chain will terminate.

Decrease Relation for Predicates

For the time being we assume well-definedness of predicates, e.g. the predicate can not be unfolded infinitely and we assume termination of predicates. In general we can say that if there is a reflexive dependency of a predicate, meaning that if we would unfold successively assuming a predicate's body we would receive again an instance of this predicate, it follows that the amount of permissions accessible to that predicate decreased. We can this property illustrate via the transitive relation *nested*, which is defined as

follows:

Let p_1 and p_2 be two predicate instances, then the relation is defined as:

$$nested(p1, p2) \Leftrightarrow p_1 \text{ is part of the body of } p_2$$

When a function unfolds a predicate p_1 it loses the permission of the predicate instance and gains the permissions of its body, thus it gains also all permissions of the predicates p_i for which holds: $nested(p_i, p_1)$.

For the function `list` in listing 1.3 it holds after the unfold that *nested(list(x.next), list(x)*. Using the definition of predicates we can follow that the permission accessible to the predicate instance `list(x.next)` has to be less than the amount of permission accessible to `list(x)`. Hence we can conclude:

$$p_1 \ll p_2 \Leftrightarrow \text{nested}(p1, p2)$$

We know that the heap is finite and so is the amount of heap location, thus for all predicates p_i:

$$\prec p_i \Leftrightarrow true$$

This will imply a well-founded relation and therefore a way to prove termination by showing decreasing amount of accessible heap location.

In the example of listing 1.3 holds that $nested(list(x.next), list(x))$. Since these predicates instances are exactly the same which are required for the original function and for the recursive function call it follows that the function will lose amount of accessible heap location in each recursive call and the function will terminate.

2.3 User Defined Types

The definitions of the decreasing and bounded relations and the corresponding axioms for all relevant types as well as for predicates are stored in different Viper files. They nee to be imported if a termination check should be applied.

File	Description
`dec.vpr`	Declaration of the decreasing and bounded-relation
`int_decreases.vpr`	Default axioms for the specific types
`...`	bool, int, perm, ref, pred, seq, set or
`multiSet_decreases.vpr`	multiSet
`pred_support.vpr`	Declaration of nested-relation as well as other axioms needed for proving decreasing predicates

Table 2.1: Files to Import

If users don't want to use these default implementation (see Chapter 2.2.1 - 2.2.3) they can for example only import the file `dec.vpr` and define the axioms for types themselves.

Chapter 3

Program Transformation

The termination-related proof obligations are generated with a Viper-to-Viper transformation. Since an implementation of the checks in each verifier separately would duplicate the necessary effortsn't and that each implementation would have to be adapted if the rules for checking termination change. In this chapter we give an overview of the implementation of this transformation and how it is implemented. We will present an example and explain step by step how the generated Viper code will look and how the generated code will imply termination of functions.

The architecture of the Viper framework consists the following four main parts:

$$Parser \rightarrow Resolver/Typechecker \rightarrow Translator \rightarrow Verifier$$

The Parser, Resolver and the Translator are responsible for parsing the Viper code into an AST as well as doing several checks including the type checks. Afterwards the AST will be passed to the verification back-end e.g. the verifiers Silicon or Carbon. The part where the termination checks are generated is after the translator, where we know that the program is syntactically correct and won't generate further incorrect code. The transformation checks all functions which should be verified and adds, for each function, a method (called `funName_termination_proof`), in which the checks are encoded. The generation of these methods is grouped in a single module whose main tasks are to detect recursion, to add the termination checks and to define which error messages should be generated if the verification of these termination checks fails.

3.1 Transformations

The transformation starts with traversing the body of every function and scanning for function calls. When a recursive function call is detected, we have to prove that the arguments, according to the decreasing clause, are in the corresponding decrease and bounded relation. These two relations are implemented as functions, which return a boolean:

$$decreasing(arg1 : T, arg2 : T) \rightarrow Boolean$$
$$bounded(arg1 : T) \rightarrow Boolean$$

These functions are implemented in Viper as abstract functions (functions without a body) and their output is defined by axioms dependent of the argument type. For example are the axioms for integers:

$$\forall i_1, i_2 : Integer_{\bullet}(i_1 < i_2 \Leftrightarrow decreasing(i_1, i_2))$$
$$\forall i_1 : Integer_{\bullet}(0 \leq i_1 \Leftrightarrow bounded(i_1))$$

3.1.1 Generation of the proof methods

Consider the simple example shown in listing 3.1.

Listing 3.1: Simple recursive function

```
1 function fun(i:Int):Int
2    decreases i
3 {
4    i<=1 ? 0 : fun(i-1)
5 }
```

While traversing the function's body, conditional- and unfold-expressions will be transformed to statements and copied into the generated method body. For the condition (`i<=1`) in listing 3.1 a statement of the form `if(i<=1) then ... else ...` will be constructed. Therefore, it can be guaranteed that at every function call the constraints for the arguments and thus also for the termination checks are preserved. In our example checking if the argument in the recursive call decreases is only important if i is greater than 1, because only then the call `fun(i-1)` will be made.

Listing 3.2 shows the program supplemented with the necessary checks.

Listing 3.2: Termination proof of listing 3.2

```
1 method fun_termination_proof(i: Int)
2 {
3   if (i <= 1) {
4   } else {
```

```
5     assert bounded(i - 1)
6     assert decreasing(i - 1, i)
7   }
8 }
```

3.1.2 Function Arguments

Consider a more complex example in listing 3.3.

Listing 3.3: More complex function with several recursive calls

```
1 function fun(i:Int):Int
2    ensures result == 1
3    decreases i
4 {
5    i > 8 && (i > 4 || fun(i+3) != 1) ? fun(fun(i-1)) : 1
6 }
```

The second recursive call in the function `fun` in listing 3.3 (`fun(fun(i-1))`) has as its argument a recursive call itself. To ensure that the arguments are well-defined before the function call will be made, the checks for proofing that the arguments terminate are always done before checking if the called function terminates. Therefore, checks for the recursive call (`fun(fun(i-1))`) in listing 3.1 have the following form:

Listing 3.4: Checking termination of the second recursive call of listing 3.1

```
1  method fun_termination_proof(i : Int)
2  {
3     ...
4     if (i > 8 && (i > 4 || fun(i + 3) != 1))
5     {
6        //Checks for fun(i-1)
7        assert bounded(i - 1)
8        assert decreasing(i - 1, i)
9
10       //Checks for fun(fun(i-1))
11       assert bounded(fun(i - 1))
12       assert decreasing(fun(i - 1), i)
13    }
14 }
```

3.1.3 Short-Circuiting Evaluation

For the first recursive call in listing 3.3. (`fun(i+3)`) we have to understand short-circuiting evaluation. Viper supports short-circuiting evaluation, which means that a boolean expression is stopped being evaluated as soon

as its outcome is determined. For a disjunction (or a conjunction) this means that the second argument is evaluated only if the first argument does not suffice to determine the value of the expression. We see that the recursive call is inside of a disjunction (`i > 4 || fun(i+3)`). The first argument is always true, because the previous conjunction ensures that i ¿ 8. Therefore, the disjunction automatically evaluates to true independent of the outcome of `fun(i+3)`. This means that this function call never will be made and thus no checks are needed for this recursive call. The generated body is shown in listing 3.5.

Listing 3.5: Checking termination of the first recursive call of listing 3.1

```
1  method fun_termination_proof(i: Int)
2  {
3    if (i > 8) {
4      if (!(i > 4)) {
5        assert bounded(i + 3)
6        assert decreasing(i + 3, i)
7      }
8    }
9    ...
10 }
```

The verifier realizes that is impossible for any input to reach line 5 in listing 3.5., because the conditions `i>8` and `!(i>4` contradicts each other. Hence the assertion error on line 6 will no be reported.

3.2 Predicates

Let us now go back to the example in listing 1.3, where we proved termination by showing that the amount of accessible heap location decreases with every function call. The example of listing 1.3. is repeated in listing 3.6.

Listing 3.6: Example of listing 1.3: Implementation of adding elements on the heap together

```
1  field val: Int
2  field next: Ref
3
4  predicate list(ref: Ref) {
5     ref != null ==> acc(ref.val) && acc(ref.next) && list(ref.next)
6  }
7
8  function list_sum(x: Ref): Int
9     requires list(x)
10 {
11    unfolding list(x) in x == null ? 0 : x.val + list_sum(x.next)
```

```
12 }
```

To prove termination of heap-dependent functions (see function `list_sum` in listing 3.6) we give in section 2.2.3 a way of defining a decrease and bounded relation for predicates, which requires the nested relation. Like the decrease and bounded relation, the nested relation is implemented as an abstract function:

$$nested(arg1 : T, arg2 : T) \rightarrow Boolean$$

The arguments of this function are predicate instances. A predicate instance is uniquely identified by its name and its arguments. Since the nested relation is transitive and asymmetric the following axioms hold:

$$\forall p_1, p_2, p_3 \bullet (nested(p_1, p_2) \land nested(p_2, p_3) \Rightarrow nested(p_1, p_3))$$
$$\forall p_1, p_2 \bullet (nested(p_1, p_2) \Rightarrow \neg nested(p_2, p_1))$$

Since predicate instances as a type don't exist in Viper yet, we implemented a workaround to support the nested relation.

3.2.1 Representing Predicate Instances

In Viper you can define own types with the keyword `domain`. A domain can also have multiple generic type parameters (`domain exampleType[T]`) and it is also possible to define globally accessible functions inside a domain.

To represent predicate instances the transformation will generate for every predicate:

- A unique type depending of the predicate's name. This type will represent the predicate's name.
- A global function with the same arguments as the predicate instance and of return type `Loc[T]`. This function represents the arguments of a predicate instance and ensures that for equivalent input it will return the same output.

The idea is to represent a predicate instance with the type `Loc[T]`, which will be generated with the global functions representing the predicate's arguments. The generic type parameter `T` can then be instantiated with the type representing the predicate's name.

E.g. for listing 3.6. the domain `domain list_predicateName` will represent the predicate's name and the return value of the global function `loc_Ref(r:Ref):Loc[T]` will represent the arguments (see listing 3.7).

Listing 3.7: Representation of the predicates of listing 3.6

```
domain list_predName{}

domain Loc[T]{
   function loc_Ref(r: Ref) : Loc[T]
}
```

3.2.2 Assumptions about Nested Predicates

According to our definition of the decrease relation for predicate instances (chapter 2.2.3), two predicate instances are in the decrease relation iff they are in the nested-relation. Two predicate instances are per definition in the nested-relation exactly when one instance gets called inside of the body of the other one. Every time a function unfolds a predicate the transformation will check the predicate's body for possible other predicate instances and will add them to the nested relation. In Viper this will be a simple assumption of the nested function with the two variables representig the instances.

The generation of the nested assumptions are similar to the generation of termination checks: The body of a predicate will be traversed, special expressions such as conditions will be translated to statements and predicate instances inside of other predicate instances will be assumed to be in the nested-relation.

Listing 3.8: Termination checks of listing 3.6

```
method list_sum_termination_proof(r: Ref)
{
  unfold list(x)
  var List_r: Loc[List_PredName]
  var List_rnext: Loc[List_PredName]
  if (x != null) {
    List_r := loc_Ref(r)
    List_rnext := loc_Ref(r.next)
    inhale nested(List_rnext, List_r)
  }
  if (x==null){
  } else {
    assert bounded(List_rnext)
    assert decreasing(List_rnext, List_r)
  }
  fold list(x)
}
```

The termination checks for listing 3.6. are encoded in listing 3.8. The transformation will after the unfold of the predicate instance `list(x)` traverse

the predicate's body and will assume the predicate instance `list(x.next)` exactly when x is not the empty reference (line 6-10). The variable `list_r` will represent the predicate instance `list(r)` and variable `list_rnext` represents `list(r.next)`. After that the transformation will encounter the recursive call `list_sum(x.next)` and has to prove that the predicate instances `list(x)` and `list(x.next)` are in the decrease and bounded relation. Since these predicates are in the nested relation and also is in the bounded relation (like every predicate instance), both checks (line 13 and 14) will verify.

3.3 Implementation

In this section we explain in pseudo code how the generation of the termination code is implemented.

To generate the methods, which encode the termination checks, the procedure `addTerminationChecks` is invoke. It takes the abstract syntax tree of the program and return necessary generated methods, functions and domains as described in chapter 2 which are then included into the final AST.

Algorithm 1: addTerminationProof

```
1:  global variables
2:      nodes
3:      neededDomains
4:      neededDummyFuncs
5:  end global variables
6:  procedure AddTerminationChecks(ast)
7:      nodes ← ast.nodes
8:      neededMethods ← Seq()
9:      for all functions func in the nodes do
10:         if func.decClause ≠ decreases* then
11:             alreadyChecked ← Seq()
12:             mName ← uniqueNameGen( )
13:             mParams ← func.params
14:             mPrecond ← func.precond
15:             mBody ←generateStmts(func.body, func, alreadyChecked)
16:             m← Method(mName, mParams, mPrecond, mBody)
17:             put m into neededMethods
18:     return Seq( neededMethods, neededDomains, neededDummyFncs)
```

The procedure `addTerminationChecks` (see algorithm 1) goes through all existing functions in the AST and invokes for each function, which has not

a `decrease *` as their decresing clause, the procedure `generateStmts`. This procedure returns the termination related proof obligations for the function, which then are put into the body of a method. The method itself has the same arguments and precondition as the related function. The name of the method is generated by the function `uniqueNameGen`, which goes through the whole AST and returns a unique name.

Algorithm 2.1: generateStmts

```
1:  procedure generateStmts(expr, originalFunc, alreadyChecked)
2:      if expr matches e1 ? e2 : e3 then
3:          condChecks ←generateStmts(e1, originalFunc, alreadyChecked)
4:          ifChecks ←generateStmts(e2, originalFunc, alreadyChecked)
5:          elsChecks ←generateStmts(e3, originalFunc, alreadyChecked)
6:          condStmt ← If e1 then ifChecks else elsChecks
7:          return Seq(condChecks, condStmt)
8:      if expr matches unfolding acc(P(e1), e2) in e3 then
9:          unfoldStmt ← unfold acc(P(e1), e2)
10:         nestedRelation ←generateNested((P(e1))
11:         predChecks ←generateStmts((P(e1), originalFunc, alreadyChecked)
12:         accChecks ←generateStmts(e2, originalFunc, alreadyChecked)
13:         bodyChecks ←generateStmts(e3, originalFunc, alreadyChecked)
14:         foldStmt ← fold acc(P(e1), e2)
15:         return Seq(unfoldStmt, predChecks, accChecks, nestedRelation, body-
    Checks, foldStmt)
```

The procedure `generateStmts` (algorithm 2.1, 2.2 and 2.3) takes three arguments: an expression `expr`, a function `originalFunc` and a set of strings `alreadyChecked`. The procedure checks the expression `expr` for possible function calls to the function `originalFunc`. The procedure uses the set `alreadyChecked` to store functions, which already are checked for possible recursion. Hence it is assured that the procedure will terminate. The use of the global variable `neededDummyFuncs` i sexplained in chapter 4.3.

In the lines 2 to 15 of the algorithm 2.1 is shown how the checks are generated if the expression has the type of a conditional expression or of an unfold-expression. The procedure `generateNested` generates the domains which are needed for the nested relation, which will be stored in the global variable `neededDomains`. Furthermore the procedure will return the assumptions of the nested relations. The implementation of short circuit evaluation of boolean expression is shown in algorithm 2.2.

Algorithm 2.2: generateStmts

```
16:  if expr matches e1 op e2 then
17:      leftChecks ←generateStmts(e1, originalFunc, alreadyChecked)
18:      rightChecks ←generateStmts(e2, originalFunc, alreadyChecked)
19:      if op matches || then
20:          notLeft ← !e1
21:          ifNotLeft ← If notLeft then rightChecks
22:          return Seq(leftChecks, ifNotLeft)
23:      else if op matches && or ==> then
24:          ifLeft ← If e1 then rightChecks
25:          return Seq(leftChecks, ifLeft)
26:      else
27:          return Seq(leftChecks, rightChecks)
```

If the function consists of a function call, we have to check if it is a recursive call or not (see algorithm 2.3). For a recursive call we have to generate the checks to verify the decreasing and bounded relation. This is not necessary if there is a call to another function, in such a case the argument has to be passed to the callee and its body has to be checked for recursive calls. The procedure `replace` takes as arguments an expression and a map. It will replace in the received expression every node according to the mapping and return the new expression.

Algorithm 2.3: generateStmts

```
28:  if expr matches Func(funcName, args) then
29:      if funcName in alreadyChecked then return
30:      callee ← Function with funcName in nodes
31:      params ← callee.parameters
32:      funcBody ← callee.body
33:      decr ← callee.decreasingClause
34:      replaceMap← mapping of params to args
35:      argumentsChecks ←generateStmts(args, originalFunc, already-
         Checked)
36:      if funcName ≠ name of originalFunc then
37:          if decr ≠ decrease * then
38:              replacedBody ← replace(funcBody, replaceMap)
39:              calledFncChecks ←generateStmts(replacedBody, original-
                 Func, Seq(alreadyChecked, funcName))
40:              return argumentsChecks and calledFncChecks
41:          else
42:              Report error: func with dec. clause calls function with dec*
43:      else                                          ▷ Recursion detected
44:          smallerExpr ← replace(decr, replaceMap)
45:          biggerExpr ← decr
46:          boundedFunc ← bounded(smallerExpr)
47:          decreaseFunc ← decreasing(smallerExpr, biggerExpr)
48:          return Seq(argumentsChecks, boundedFunc, decreaseFunc)
49:  if expr matches Literal then
50:      return Seq()
```

Chapter 4

Definitional Axioms

During verification of functions two important axioms, so called *definitional axioms* [4] hold: one assumes that if the preconditions of a function instance holds then the instance is equal to its body. The second one assumes that if the preconditions of a function hold then also its postconditions hold. However the second axiom only holds when the function terminates.

Let's consider a simple example:

Listing 4.1: Simple non-terminating function

```
1 function fun(i: Int, b: Bool):Int
2   requires i>0
3   ensures result == 1
4 {
5   b ? 1 : fun(i+1,b)
6 }
```

The two definitional axioms for this function are:

$$\forall i \bullet (i > 0 \Rightarrow fun(i,b) = 1) \text{ and} \tag{4.1}$$
$$\forall i \bullet (i > 0 \Rightarrow fun(i,b) = b\ ?\ 1 : fun(i+1,b)) \tag{4.2}$$

The verifier has to verify that these axioms always hold. To prove the first axiom (4.1), we assume the left side ($i > 0$). Using (4.2) it follows that `fun(i,b) = b?1:fun(i+1,b)`.

- If $b = true$, then `fun(i,b)=1` holds and the first axiom is obtained.
- If $b = false$, then `fun(i,b)=fun(i+1,b)` holds. Since the function call of `fun(i+1)` ensures the return value 1, it follows that `fun(i,b)=1`.

The problem is that termination is not included in this chain of proof. If `b=false` this function will never end and although the precondition holds

the postcondition will never hold.

To conclude, the transformation has to ensure that the definitional axioms are only valid after the termination is being verified. We solve that, by implementing a solution, where the original functions are not used directly in the generated methods and thus their definitional axioms can not be used for verifying the methods.

This chapter illustrates when exactly this problem will occur in the generated Viper code and elaborates a solution which is independent of the verifiers.

4.1 Example

Lets consider the following example of a function `f` and its termination checks (Listing 4.2):

Listing 4.2: Unsound use of the definitional axioms in the termination checks

```
1  function f(i:Int):Int
2     requires 0 <= i
3     ensures false
4     decreases i
5  {
6      (0<i && f(0)>5) ? f(i) : 7
7  }
8
9  method f_termination_proof(i: Int)
10   requires 0 <= i
11 {
12   if (0 < i) {
13     assert bounded(0)
14     assert decreasing(0, i)
15   }
16   if (0 < i && f(0) > 5) {
17     assert bounded(0)
18     assert decreasing(i, i) //Verifies
19   } else {
20   }
21 }
```

In Listing 4.2 the function `f` calls itself (`f(i)`) exactly when `i>0`. Its trivial to see that there is an infinitely call chain and therefore the termination checks should fail. But the generated assertions on line 17 and 18 will verify.

The problem would be here that the function `f` has a postcondition (`false`)

and that there is a second recursive call (`f(0)`) inside of the condition on line 16: `0<i&&f(0)>5`. When the verifier checks this condition, it will verify the precondition of the recursive call. This will succeed, because due to the first part of the condition it holds that `0<i` and hence also `i-1`$\geq$`0`. Now the definitional axioms will imply that the postcondition must hold and therefore the method will assume the postcondition of `f(i-1)` which is `false`. Hence termination can be proven.

4.2 Implementation of the dummy functions

The solution we implemented to avoid the problems illustrated in 4.1 is to add simple *dummy-functions* for every function: a dummy function has no body and no postconditions but it has the preconditions of the original function. The idea is to use these dummy functions in the method's body, which checks the termination, instead of the original functions. The correctly added method for function `f` can be seen in Listing 4.3.

Since the dummy functions have no body and no postconditions, the method can't, inside the condition (line 10 in listing 4.3), assume the postcondition of `f`. The assumptions, which are given through the definitional axioms, are then "manually" assumed after the termination checks.

Listing 4.3: Proof method of Listing 4.1 with step-wise assuming the assumptions of the definitional axioms

```
1  method f_termination_proof(i: Int)
2    requires 0 <= i
3  {
4    if (0 < i) {
5      assert bounded(i)
6      assert decreasing(0, i)
7      inhale f_withoutBody(0) == (0 < 0 && f_withoutBody(0) > 5 ? 0 :
           f_withoutBody(0)) //body of f(0)
8      inhale false //postcondition of f(0)
9    }
10   if (0 < i && f_withoutBody(0) > 5) {
11   } else {
12     assert bounded(i)
13     assert decreasing(i, i) //Will now fail
14     inhale f_withoutBody(i) == (0 < i && f_withoutBody(0) > 5 ? 0 :
           f_withoutBody(i)) //body of f(i)
15     inhale false //postcondition of f(i)
16   }
17 }
18
19 function f_withoutBody(i: Int): Int
20   requires 0 <= i
```

Every time when discovering a recursive function call, a corresponding termination check will be added and after the checks the function's body as well as its postconditions will be assumed. This approach of gradually axiomatising uninterpreted "stub functions" was chosen because it is coupled with the goal to implement termination checks independent from the verifiers.

4.3 Implementation

The code from chapter 3.3 in algorithm has to be changed as follows. The

generateStmts part 3: function calls

```
28: if expr matches Func(funcName, args) then
29:     if funcName in alreadyChecked then return
30:     ...
36:     if funcName ≠ name of originalFunc then
37:         ...
43:     else                                          ▷ Recursion detected
44:         smallerExpr ← REPLACE(decr, replaceMap)
45:         biggerExpr ← decr
46:         smallerExpr ← INSERTDUMMYFNC(smallerExpr)
47:         biggerExpr ← INSERTDUMMYFNC(biggerExpr)
48:         boundedFunc ← bounded(smallerExpr)
49:         decreaseFunc ← decreasing(smallerExpr, biggerExpr)
50:         dummyFnc←DUMMYFUNC(originalFunc)
51:         fncBody←INSERTDUMMYFNC(originalFunc.body)
52:         assBody← Assume (dummyFnc == fncBody)
53:         postCond←INSERTDUMMYFNC(originalFunc.postConds)
54:         assPost← Assume (dummyFnc == postCond)
55:         return Seq(argumentsChecks, boundedFunc, decreaseFunc, assBody,
    assPost)
```

procedure `insertDummyFunc` will replace every function call inside of the argument with the corresponding dummy-function. Every newly created dummy-function will be added to the global variable `neededDummyFuncs`. The procedure `dummtFunc` will return for every given function its corresponding dummy-function.

Chapter 5

Error Message Transformations

For a user it is always helpful to see meaningful error messages as feedback for code which does not verify. The problem in our transformation is that we generate, transparent of the users (they are not aware of those newly introduced methods), additional methods with termination checks included. Verification errors occurring in these methods can therefore irritate the user.

Listing 5.1: Example of a Non-terminating Function

```
1  function fun(i:Int, j:Int):Int
2     decreases i, j
3  {
4      fun(i, j)
5  }
6
7  method fun_termination_proof(i: Int, j: Int)
8  {
9    assert bounded(i) && bounded(j)
10   assert decreasing(i, i) || i == i && decreasing(j, j)
11 }
```

Consider the function and its corresponding termination checks in listing 5.1. The function will terminate if the tuple `(i,j)` decreases at the recursive function call. Clearly, the function `fun` doesn't terminate and therefore the assertions in the method `fun_termination_proof` won't hold. In order to inform the user that this function won't terminate, we implemented an error transformation, which will transform such errors inside of the check methods to more understandable and more meaningful error messages.

In the following chapter we will cover the ideas behind the four different error transformations, when they will occur and what the error will be on the user's side.

Since our termination checks are of the form `assert decreasing(...)` or `assert bounded(...)` an assertion error will be reported by the verification back-end for each assertion that might not hold. One error in listing 5.1 would be: *Assertion might not hold (9)*, where the number between the parentheses represents the line where the error occurred. Since the user wrote only the code between the lines one and five, this error message doesn't make much sense to him. Our implemented error transformations will therefore convert the error to a more useful one as well as adapt the position.

5.1 Variant not Decreasing

Assertion errors related to the decrease relation (`assert decreasing ...`) will be transformed to a `TerminationFailed` error with the following error message (relating to listing 5.1):

> *Function fun might not terminate.*
> *Termination measure (i,j) might not (indirectly) decrease at fun(i, j) (1)*

The position, which will be visible to the user, will represent the function, which won't terminate. In listing 5.1. this would be function `fun` at line one. There will be an error for each recursion, where the variant wont decrease.

5.2 Variant not Bounded

If the value of the decreasing clause cannot be proven to be bounded, that is when the corresponding assertion `assert bounded`... fails, the error is also be transformed to a `TerminationFailed` error but now with another message. Since the definition of tuples demands that every expression in a tuple should be bounded, the following final error messages are reported to the user:

> *Function fun might not terminate.*
> *Decreases expression (i) of the decreasing clause (i,j) might not be bounded. (1)*

> *Function fun might not terminate*
> *Decreases expression (j) of the decreasing clause (i,j) might not be bounded. (1)*

Again there will be an error for each recursion, where an expression wont be bounded.

5.3 Decreases *

Functions with a 'decreases *' as their decreasing clause are not checked for termination and may or may not terminate. If another function (without decreases * as decreasing clause) calls such a function an error will be reported. Lets consider the following example in listing 5.2.

Listing 5.2: Function with decreases * as it decreasing clause

```
1  function foo(k:Int):Int
2     decreases *
3  {
4       foo(k+1)
5  }
6
7  function bar(i : Int) : Int
8     decreases i
9  {
10      foo(i-1)
11 }
```

The function bar calls the function foo. Due to prevention of termination checks for function foo, it is possible that bar won't terminate. The following error will be reported:

Function bar might not terminate.
The function calls foo, which might not terminate. (7)

In Viper the only possibility that an error will be reported, is when the verifier fails to verify a condition. Since in our example there is not a direct way to encode the condition "decreasing clause of the called function is not decreases *", we implemented a little trick to force the verifier to do it instead: the transformation will add the code "assert false" inside of the generated method as soon as it discovered a call to a function with a decreases *. This assertion will never be verified and therefore an assertion-failed error will be reported, which then again can be transformed to the error message described above.
The generated termination checks for the function bar are represented in listing 5.3.

Listing 5.3: Proof Method of function bar from Listing 5.2

```
1  method bar_termination_proof(b: Bool)
2  {
3    // Called Fnc (foo) does not terminate
4    assert false
5  }
```

The advantage of this choice of implementation is that the error will only be reported when it is also semantically possible, that a function call to a function with `decreases *` could be made. For example, if we modify the function `bar` in listing 5.2 to the function presented in listing 5.4, no error will be reported, because the condition `i > 0` will always be true and the function call `foo(i-1)` will never be made. This is more precise than just a syntactical approach.

Listing 5.4: Modification of listing 5.2 to prevent error report

```
1 ...
2
3 function bar(i : Int) : Int
4   requires i>0
5   decreases i
6 {
7    i>0 ? 3 : foo(i-1)
8 }
```

5.4 Decreasing Clause not defined

When a function does not have defined a decreasing clause, but there is at least one direct or indirect recursion of the function, the transformation does not know which values has to decrease and thus an error will be reported.

Listing 5.5: Function with no decreasing clause

```
1 function foo(k:Int):Int
2 {
3     foo(k-1)
4 }
```

In the example of listing 5.5 following error will be reported:

Function foo might not terminate.
foo is (indirectly) recursive but has no decreases clause. (1)

Again, the error will be triggered through an additional assertion which always will be false. The corresponding termination method is shown in listing 5.6.

Listing 5.6: Proof method of Function foo from listing 5.3

```
1 method foo_termination_proof(k: Int)
2 {
3   // Recursion but no decClause
4   assert false
5 }
```

Chapter 6

Evaluation

In the following chapter we will illustrate the impact of our implementation to prove termination for abstraction functions. We measured the time it needs to generate the additional termination checks, which are encoded in methods, and how long it does take for each verifier to verify these additional checks.

6.1 Testing

We did two different benchmarks:

- We analyzed the time which is needed for traversing the functions, detecting recursions and adding error transformations. This is done by stopping the process of the verification as soon as the generation of the termination checks ends and before the verifiers would start to verify the code.
- We measured the time needed for the verifiers Carbon and Silicon to verify the additional methods, in which the termination checks are encoded.

For the first benchmark 872 test files are tested. For the second are only 37 test files are used, which have relatively complex recursive functions encoded, such that the influence of the additional methods is more visible.

6.2 Outcome

6.2.1 Testing the additional time for the generation

As illustrated in chapter 2.1, the additional checks are generated shortly before they are passed to the verifier. The termination checks are generated

by traversing every function's body and looking for recursion. First we measured the time needed for the parser and typechecker without generating the termination checks and after that we measured the time now generating the checks. The outcome is represented in the graph figure 6.1. On the x-axis of the graph are the test files listed and on the y-Axis their corresponding time.

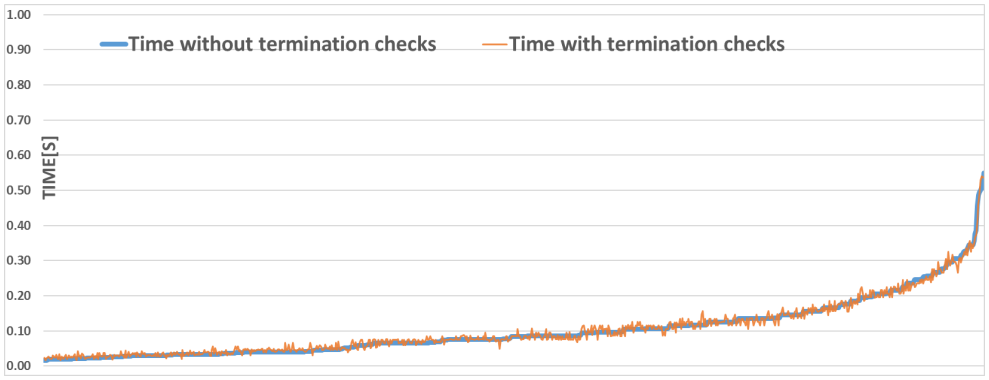

Figure 6.1: Comparison of time needed for generating termination checks

As you can see on the graph, there is practically no difference in time, hence the time needed for generating the additional termination checks is almost equal to zero. The average difference between these two times is 1 millisecond.

6.2.2 Testing the verification time for the generated termination checks

In the second comparison we measured the influence of the additional termination checks in both verifiers to examine how much more time they do need to additionally verify termination of abstraction functions. We therefore use mainly test files with recursive functions, such that termination checks will be generated and a difference in times is visible.

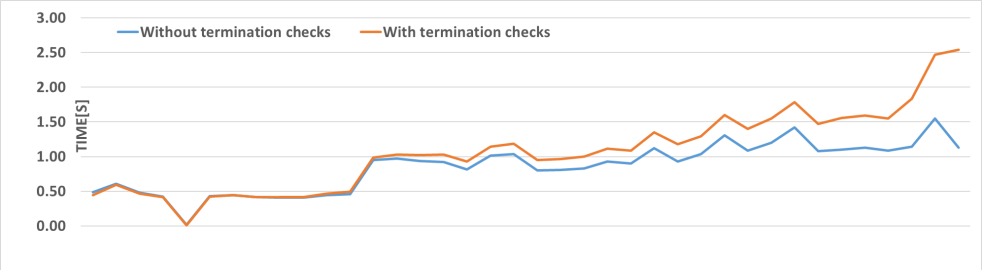

Figure 6.2: Comparison of time needed for the verifier Carbon

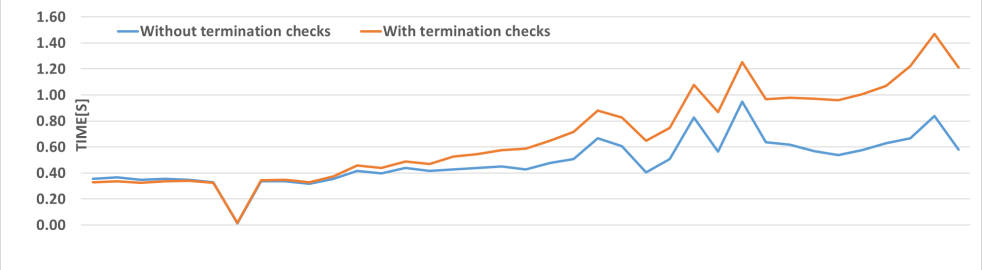

Figure 6.3: Comparison of time needed for the verifier Silicon

In the diagrams in the figures 6.2 and 6.3 you can the time needed for a verifier to verify a test file. (Again are test files in the x-axis and their corresponding time in the y-Axis). Silicon is in general the faster verifier than carbon. Silicon needs at most 1.4 second for a test file whereas Carbon needs 2.5 seconds. The testings show that in the verifiers the additional termination checks become more noticeable than in the tests in 6.2.1. Nevertheless the maximum amount of time needed for checking the termination related checks is 1.5 second more (0.6 in Silicon). It must be pointed out, that these test-files include only relatively complicated recursive functions and nothing more. Hence when we consider a more general example with also methods and domains, which don't influence the generated termination checks, the relative error should shrink.

6.3 Conclusion

We can conclude that the time needed for the generation of the additional methods can be ignored and that the generated time overhead generated by the verifiers is also small enough to be accepted in a normal verification.

Chapter 7

Future Work

In this chapter we will give an overview of some limitations of our implementation and how it can be improved in the future. Furthermore, we will illustrate different approaches to complement termination checks, so that not only termination of abstraction functions can be proven.

7.1 Limitations

During the implementation of adding termination checks for functions, we discovered several limitations, which currently could not be solved due to limitations of the Viper infrastructure. One of them is for example that error messages of function calls, which violate the function's precondition, will be reported several times.

7.1.1 Multiple precondition errors

In chapter 4 we discussed, how to ensure that the definitional axioms are available only when the function is known to terminate. Therefore, we generated several dummy-functions with the same precondition as their original function. If now such a function has a precondition, which does not hold for some function calls, the error will also be reported inside of the additional methods, which are generated to proof termination of functions. Lets consider the example of the function `f` in listing 7.1:

Listing 7.1: Termination proof with multiple precondition errors

```
1 function f(i:Int):Int
2   requires i>5
3   decreases i
4 {
5     i>0 ? f(i-1) : 1
6 }
```

```
7
8  function f_withoutBody(i: Int): Int
9    requires i > 5
10
11 method f_termination_proof(i: Int)
12   requires i > 5
13 {
14   if (i > 0) {
15     assert bounded(i-1)
16     assert decreasing(i - 1, i)
17     inhale f_withoutBody(i - 1) == (i - 1 > 0 ? f_withoutBody(i - 1 -
           1) : 1)
18   }
19 }
```

The following errors will be reported for the code in listing 7.1:

- *precondition of function f might not hold (5)*
- *precondition of function f_withoutBody might not hold (17)*
- *precondition of function f_withoutBody might not hold (17)*

Due to the additional methods also additional errors could be reported. Since these generated methods and functions are transparent to the user, it can be irritating for them.

One solution could be to force the framework to ignore errors inside of generated methods. Unfortunately the Viper framework does only support transformations of an error to another one, but you are not able to completely ignore an error reported by a verifier. Another way to prevent these additional errors would be to assume the method's preconditions of the original function in the respective situations. Currently this is also not possible because Viper doesn't allow to assume permissions. Even inhaling (special encoding in Viper to gain permission) is not possible, since it would potentially add permissions that are already being held.

7.1.2 Functions in the decreasing clause

A further limitation of the implementation is that it is not possible to prove termination with a function call as a variant. The reason is because the generated methods only use the dummy-functions inside their body and therefore is the function's body unknown to the method. But when the implementation wants to check if two function calls are in the decrease relation, it has to know their body.

Listing 7.2: Multiple precondition errors

```
1  function f(i:Int):Int
2     decreases ident(i)
3  {
4       i>1 ? f(i-1) : 1
5  }
6
7  function ident(i:Int):Int
8  {
9     i
10 }
11
12 function f_withoutBody(i: Int): Int
13
14 function ident_withoutBody(i: Int): Int
15
16 method f_termination_proof(i: Int)
17 {
18   if (i > 1) {
19     assert bounded(ident_withoutBody(i - 1))
20     assert decreasing(ident_withoutBody(i - 1), ident_withoutBody(i))
21     inhale f_withoutBody(i - 1) == (i - 1 > 1 ? f_withoutBody(i - 1 -
           1) : 1)
22   }
23 }
```

In listing 7.2 for example, the decreasing clause is equivalent to `decreases i`. The generated proof method however uses the function `ident_withoutBody` and due to the fact that this function has no body, the assertions in the lines 19 and 20 fail.

One solution would be to inhale the body of the function `ident`. This is possible because the definitional axiom, saying that an instance is equal to its body, only depends on the precondition of the function. Since the method's precondition is the same, we can use the implication following through the axiom. This would be indeed an improvement and more functions could be proven, but in cases where the functions such as `ident` are recursive the proof would fail again.

Listing 7.3: Multiple precondition errors

```
1  function f(i:Int):Int
2     decreases ident(i)
3  {
4       i>1 ? f(i-1) : 1
5  }
6
7  function ident(i:Int):Int
8  ensures i==0 ==> result == 0
```

```
9  ensures (i > 1) ==> (result == i)
10 {
11    i>1 ? i + ident(0) : 0
12 }
13
14 method f_termination_proof(i: Int)
15 {
16   if (i > 1) {
17         inhale ident_withoutBody(i) == (i>1 ? ident_withoutBody(0) : 0)
18     inhale ident_withoutBody(i - 1) == (i-1>1 ?
           ident_withoutBody(i-1) : 0)
19     assert bounded(ident_withoutBody(i - 1))
20     assert decreasing(ident_withoutBody(i - 1), ident_withoutBody(i))
21     inhale f_withoutBody(i - 1) == (i - 1 > 1 ? f_withoutBody(i - 1 -
           1) : 1)
22   }
23 }
```

In listing 7.3 is an example of this case. Although the bodies of the instances `ident(i)` and `ident(i-1)` is inhaled it is not possible to prove the assertions. If the method would be able to assume also the postconditions, the assertion can be verified. But assuming the postcondition depends whenever the function terminates.

A proper solution would be to create the static call graph, to detect the functions with no recursion and then to successively assume the postconditions and proof termination. In the example of listing 7.3 we would have to proof the termination of the function `ident` and then we were able to assume its postcondition, which would lead to a successful verification for function `f`.

7.2 Extensions

For further termination checks our implementation could be adapted or be the base for following extensions:

- Termination in this thesis is only discussed for functions. A further goal could be to implement termination checks also for recursive methods or for loops.
- In chapter two and three we assumed well-definedness of predicates. Compared to functions, predicates are not checked for termination. An extension could be to adapt the current implementation for proving termination of functions and try to prove termination of predicates as well.
- To improve the user-friendliness one could add an error, which reminds the user of a redundant decreasing clause. For example, there

can be an error when the user had defined a decreasing clause, but the transformation never detected a recursion. Without a recursion the function will terminate and a decreasing clause is not necessary. We didn't implement this feature in our solution, because from our point of view it is more a sort of a warning than an actual error and unfortunately warnings are currently not supported by the Viper infrastructure.

- A further improvement would be to infer a variant if there is no decreasing clause. To infer a variant, the body of the functions could be analyzed.

Bibliography

[1] Müller P., Schwerhoff M., and Summers A.J. "viper: A verification infrastructure for permission-based reasoning.". Jobstmann B., Leino K. (eds) Verification, Model Checking, and Abstract Interpretation, 2016.

[2] Rudich A., Darvas Á., and Müller P. "checking well-formedness of pure-method specifications.". Cuellar J., Maibaum T., Sere K. (eds) FM, 2008.

[3] Floyd R.W. "assigning meanings to programs.". Colburn T.R., Fetzer J.H., Rankin T.L. (eds) Program Verification., 1993.

[4] Heule S., Kassios I.T., Müller P., and Summers A.J. "verification condition generation for permission logics with abstract predicates and abstraction functions.". Castagna G. (ed) ECOOP, 2013.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Checking Termination of Abstraction Functions

Authored by (in block letters):

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

Name(s):	**First name(s):**
Gruntz	Patrick

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)**
Zürich, 1 September 2017	P. Gruntz

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