Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Tool Support for Termination Proofs

Bachelor's Thesis

Fabio Streun

May 2019

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff
Department of Computer Science, ETH Zürich

Contents

Contents **1**

1 Introduction **1**
1.1 Introduction to Viper 1
1.1.1 Proving Termination 3
1.2 Chapter Overview 4

2 Proving Termination of Functions **5**
2.1 Variant 5
2.1.1 Proof Encoding 7
2.1.2 Limitations of the Variant Termination Proof Approach 11
2.1.3 Function Inlining Transformation 12
2.1.4 Unsoundness of Previous Work 13
2.1.5 Limitations of FIT 14
2.2 Transition Invariants 15
2.2.1 Definitions and Theorems 16
2.2.2 Notation for Transition Invariants 18
2.2.3 Proof Encoding in Viper 20
2.2.4 Advantage over Variant Termination Proof Approach . . 22
2.2.5 Mutually Recursive Functions 24

3 Termination Plugin **29**
3.1 Decreases Clause 29
3.2 Plugin Overview 30
3.3 Modifying PAST 32
3.4 Generating Decreases Nodes 32
3.5 Termination Proof Encoding 33
3.5.1 Proof Method Generation 33
3.5.2 Termination Check 34
3.5.3 Predicates 36

3.5.4 Implementation of Function Inlining Transformation . . 38
3.6 Errors Reporting 40
3.6.1 Viper Error Structure 40
3.6.2 Error Transformation 41
3.6.3 Termination Error 42
3.6.4 Further Errors 44
3.6.5 Improvements to Previous Implementation 45

4 Proving Termination of Methods **47**
4.1 Termination Check Adjustments 48
4.1.1 Fields 48
4.1.2 Predicates 49
4.2 FIT for Methods 52
4.3 Termination of Loop 52

5 Standard Import **55**
5.1 The Import Feature 55
5.2 Providing Files 56

6 Conclusion **59**
6.1 Future Work 60

A Appendix **61**
A.1 Variant Termination Proof 61

Bibliography **63**

Chapter 1

Introduction

Program verification is used to check correctness of program code and helps to detect programming errors. In deductive verification, additional mathematical statements, referred to as contracts or specifications, are used to express what a program is supposed to do. A verifier then checks if the statements are satisfied by the program.

Viper [9], a deductive verification infrastructure developed at ETH Zurich, offers the possibility to encode a source language such as Rust and Java with specifications into an intermediate language. A Viper back-end is then used to verify correctness of the program. One of the back-ends is based on symbolic execution, whereas the other one is based on verification condition generation. A diagram of the Viper infrastructure can be seen in figure 1.1.

1.1 Introduction to Viper

The Viper intermediate language is a simple imperative programming language. The following example shows a function that calculates the sum of the first n positive integers.

```
1 function sum(n: Int): Int
2   requires n >= 0
3   ensures result == n * (n+1)/2
4 {
5   n == 0 ? 0 : n + sum(n-1)
6 }
```

Listing 1.1: sum of the first n positive integers

Preconditions of a function are declared with the keyword `requires` and specify how a function is allowed to be invoked. In the example above, the function `sum` can only be invoked with a non-negative argument n. The postcondition,

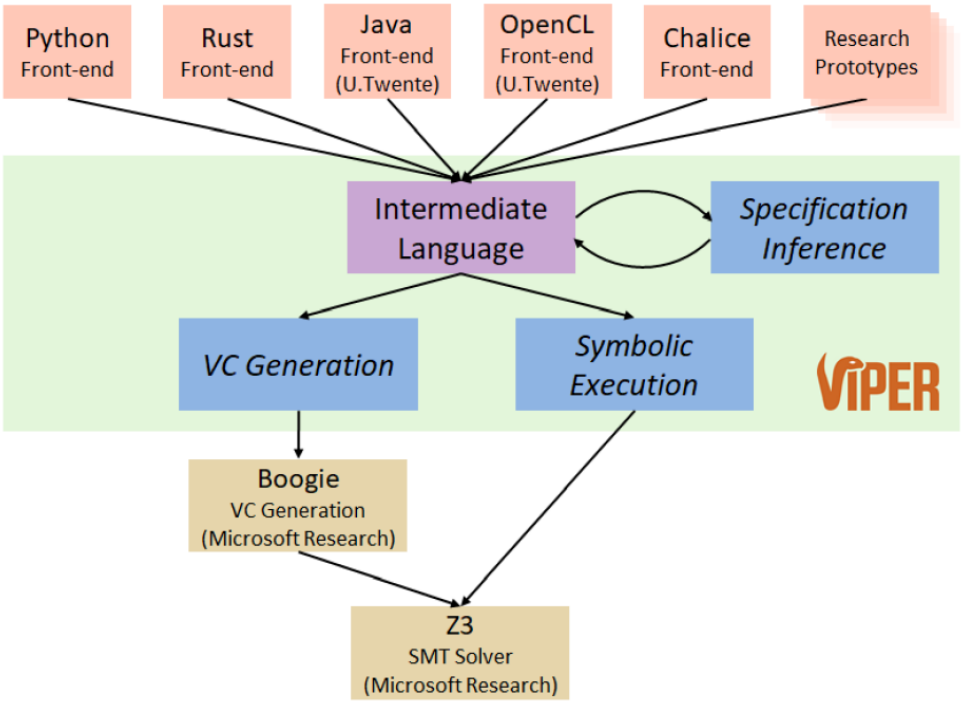


Figure 1.1: diagram of the Viper infrastructure with the front-ends and its dependencies.

declared with the keyword `ensures`, defines which guarantees are given (assuming the precondition is satisfied) after the function terminates. For the function `sum` the postcondition guarantees that the result is really the sum over the first n numbers (written in a non recursive mathematical expression).

To reason about a program heap, Viper uses a permission-based approach. A simple example of a method `increment` that increments the value of the field `ref.val` by the absolute value of `i`, is shown in listing 1.2. The precondition states that the method needs exclusive permission to the field, which is necessary to modify the field. This requires any caller of this method to also hold exclusive permission of the filed and transfer it to this method when invoking it. The postcondition states that the permission is transferred back to the caller. Additionally, in line 12 an assertion is added to the method, which verifies that the value of the field `ref.val` did not decrease. This is done by comparing the current value of the field with the value at the beginning of the method, denoted by `old(ref.val)`.

Viper distinguishes functions and methods because they have some important differences. Methods can be viewed as sequences of statements, which might read and modify fields, i.e. affect the program state. Functions, on the other hand, consist only of expressions, which might depend on the program state, i.e. read fields, but may never modify it. Thus, functions in Viper are side-

```
field val: Int

method increment(ref: Ref, i: Int)
    requires acc(ref.val)
    ensures acc(ref.val)
{
    if (i >= 0){
        ref.val := ref.val + i
    } else {
        ref.val := ref.val - i
    }
    assert ref.val >= old(ref.val)
}
```

Listing 1.2: simple method, which increments the value of a field by a particular value

effect free and can be used in assertions (e.g. precondition).

1.1.1 Proving Termination

Viper verifies partial correctness of a program. Function postconditions, for example, are proven to be satisfied by assuming the precondition and assuming termination of the function body. Because all function invocations are checked to satisfy the precondition, the first assumption is justified. However, Viper does not prove that a function terminates. Therefore, the second assumption can be wrong and lead to unsound results, as the following example shows.

```
  function bad(): Int
    ensures 0 == 1
{
    bad()
}
```

Listing 1.3: non-terminating function which causes unsound result

Because the precondition of the recursive call of `bad` is trivially satisfied, the postcondition of the invocation is assumed, hence the postcondition is unsoundly proven to be satisfied. By showing that the function actually does not terminate, such unsound verification can be prevented. In a previous project, a termination proof for functions was implemented for Viper [6]. It used the standard variant termination proof approach [2](described in section 2.1). A Viper-to-Viper transformation appended additional assertions to the Viper program, which check that a given value decreases at every recursive function call with respect to a well-founded order. This approach allowed to extend the functionality of Viper without having to modify the back-end verifiers.

However, the tool created unsound results, i.e. it unsoundly verified functions which do not terminate (further described in section 2.1.4). Therefore, the main goal for this Bachelor's thesis is to develop and implement a sound, but still reasonably complete approach to verify termination of recursive functions in Viper. Ideally an approach which is more complete than the one used in other verifiers, e.g. Dafny. Additionally, the implementation should use a Viper-to-Viper transformation, as in the previous project, to encode the additional required checks and avoid changes to the back-end verifiers. Also the Viper syntax must not be modified, such that no front-end is effected by the implementation.

A further goal of this thesis is to adapt the termination proof and extend the support to methods, which are another source of non-termination in Viper.

1.2 Chapter Overview

The thesis is divided into a conceptual part and an implementation part. The conceptual part is mainly chapter 2, in which two different termination proof approaches are presented. For both approaches, it is shown how they can be applied to Viper functions and how the termination proofs can be encoded into Viper. The implementation part consists of chapter 3, 4 and 5. In chapter 3 the implementation of the variant termination proof approach as a Viper plugin [12] is described. Chapter 4 describes the adjustments made to the termination proof implemented in the previous chapter, such that it is applicable to recursive methods. Chapter 5 presents the new import mechanism, which provides a new way for users to import files provided by Viper. In the last chapter a brief conclusion to the work is given and ideas for further improvements and projects regarding termination checks are presented.

Chapter 2

Proving Termination of Functions

Various approaches to prove termination of programs exist, such as the one presented 1993 by Floyd using well-founded orders [2] and the one presented 2004 by Podelski and Rybalchenko using Ramsey's theorem [10]. In section 2.1, we show how the approach by Floyd can be applied to Viper functions and propose an encoding of the termination proof into a Viper method, such that, if verified successfully, the method implies the termination proof. Additionally, we introduce a function transformation, which, applied prior to the proof, allows a more intuitive application of the approach in some cases. In section 2.2, the same is presented for the approach by Podelski and Rybalchenko. We show how it can be applied to Viper functions and propose an encoding of the termination proof into Viper methods, such that, if verified successfully, the methods imply the termination proof.

2.1 Variant

In general, to prove termination of a function f using the approach presented by Floyd, a *variant* is used, sometimes also referred to as measure [11] or termination measure [3], and a well-founded order over the type of the variant. It is then sufficient to show that for any invocation of f the variant becomes strictly smaller for all succeeding recursive invocations of f with respect to the defined well-founded order. Because the order is well-founded, no infinite decreasing sequence of the variant exists, hence no infinite sequence of recursive invocations of f exists and therefore the function f must terminate.

Directly Recursive Functions

A function f is considered *directly recursive* if it contains calls to f and calls to functions which do not invoke f directly or indirectly. Figure 2.1 shows the static call graph (CG) of two directly recursive functions.

Figure 2.1: CG of directly recursive functions

To prove termination of a directly recursive function f, it is sufficient to check that the variant decreases for every recursive invocation in f's body with respect to a well-founded order. Consider the function `sum` from the example in listing 1.1, again shown in listing 2.1.

```
1 function sum(n: Int): Int
2     requires n >= 0
3 {
4     n == 0 ? 0 : n + sum(n - 1)
5 }
```

Listing 2.1: sum of the first n positive integers

To prove termination of the function `sum`, the parameter n can be used as the variant. It is easily shown that n becomes strictly smaller in the one existing recursive invocation of `sum`. Because of `sum`'s precondition n can also not be smaller than 0. Thus, the variant decreases with respect to the well-founded order of positive integers and the function `sum` is proven to terminate.

Mutually Recursive Functions

Two functions are considered to be *mutually recursive* if both call each other, i.e. both contain potentially indirectly recursive calls to the other function, and thus, potentially recurse via the other function. If multiple functions recurse via each other, we have a set of mutually recursive functions. Such sets can be found by analyzing the static control flow graph of a program. Figure 2.2 shows the CG of mutually recursive functions. The functions f, g and h together build a set of mutually recursive functions.

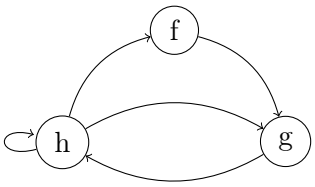


Figure 2.2: CG of mutually recursive functions

Considering a set of mutually recursive functions $F := \{f_1, f_2, ..., f_k\}$ with $k \geqslant 1$. To prove termination of the functions in F, a common approach is

to define a variant for each function, i.e. v_{f_j} for all $j \in [1, k]$, and one well-founded order for F. For all functions in F it has then to be shown that at each directly or indirectly recursive call, i.e. call to a function in F, the variant of the callee is strictly smaller than the variant of the caller with respect to the well-founded order. This is sufficient because the well-founded order implies that no infinite decreasing sequence of variants exists, hence also no infinite sequence of invocations of functions in F and therefore all functions in F terminate.

This approach is used in many other program verifiers, such as Dafny [3] and F* [13]. Hereinafter we refer to this approach as *variant termination proof*.

Listing 2.2 shows an example of two mutually recursive functions, which are equivalent to the identity function for non-negative integers.

```
1  function id1(x: Int): Int
2      requires x >= 0
3  {
4      x == 0 ? 0 : 1 + id2(x-1)
5  }
6
7  function id2(y: Int): Int
8      requires y >= 0
9  {
10     y == 0 ? 0 : 1 + id1(y-1)
11 }
```

Listing 2.2: example of two mutually recursive functions

To prove termination of the two functions in listing 2.2, the parameters x and y can be used as variants for the functions `id1` and `id2`, respectively, and the well-founded order over positive integers. For the function `id1` it has to be checked that at the call `id2(x-1)` the variant y is smaller than variant x with respect to the well-founded order. This succeeds because for this call $y = x - 1$, which is strictly smaller than x, and due to the precondition of `id1` it can be assumed $0 \leqslant x$. For the function `id2` the same check is done.

2.1.1 Proof Encoding

In this section we describe how the variant termination proof can be encoded into a Viper program, such that, if verified successfully, the program implies the termination proof. The approach is similar to the one used in the previous project [6]. However, soundness and completeness were improved. Further details to the changes are listed in section 6.1.

As already described, the variant termination proof approach requires a variant for each function. For the proof encoding, we expect the variant to be provided

by the user, as either a Viper expression or a tuple of Viper expressions[1]. Any variable used in an expression has to be a parameter of f and any call in an expression has to be pure, i.e. has to be a function call. Further, no recursive calls are allowed in the expressions, i.e. calls to functions which are mutually recursive to f.

To prove termination of a function f, it has to be shown that at each recursive call in f's body the variant of the callee is strictly smaller than the variant of the caller with respect to a well-founded order (hereinafter we refer to this check as *termination check*). To encode the termination proof of f, a semantically equivalent method, so-called proof method, is used. Each termination check for a recursive call is encoded as an assertion and prepended to the recursive call in the proof method's body such that the assertion only verifies if the termination check holds or the recursive call is not reachable.

To demonstrate the proof encoding, the function `sum` from listing 2.1 is used with the same variant as before, i.e. the parameter n. The proof method is shown in listing 2.3.

```
1  method sum_termination_proof(n: Int) returns (res: Int)
2      requires n >= 0
3  {
4      if (n == 0) {
5          res := 0
6      } else {
7          assert n-1 ≺ n  // termination check
8          res := n + sum(n-1)
9      }
10 }
```

Listing 2.3: proof method of the function `sum` (listing 2.1)

The termination check assertion in line 7 verifies successfully because $n - 1$ is smaller than n with respect to the well-founded order over positive integers. Note that the $\prec$ symbol is not a valid operator in Viper and is only used as an abstraction. We write $v_1 \prec v_2$ if v_1 precedes v_2 with respect to a well-founded order. In section 3.5.2 we present how $\prec$ can be encoded in Viper, which is not relevant for this chapter.

In this chapter we mainly use the well-founded order over positive integers, which is defined as follows for the Viper built-in type `Int`:

$$\forall i_1, i_2 : \texttt{Int}.\ v_1 \prec v_2 \iff v_1 < v_2 \wedge 0 \leqslant v_2$$

I.e. if an integer strictly decreases and is bounded by 0, it decreases with respect to a well-founded order.

[1] How a variant can be defined for a function is presented in section 3.1.

As mentioned, a variant can also be defined as a tuple of Viper expressions. A function for which a tuple as variant is useful to prove termination is the Ackermann function [1] (listing 2.4).

```
function Ack(m: Int, n: Int): Int
    requires m >= 0
    requires n >= 0
{
    m == 0 ? n + 1 :
    n == 0 ? Ack(m - 1, 1):
    Ack(m - 1, Ack(m, n - 1))
}
```

Listing 2.4: Ackermann function

To prove termination of `Ack` the tuple $[m, n]$ can be used as variant. In a termination check tuples are compared lexicographically. I.e. if the first element of the tuple, here m, decreases with respect to a well-founded order then the tuple is considered to decrease with respect to a well-founded order. The second element, here n, is only checked to decrease if the first element does not change. The third is only checked to decrease if the first and second do not change, etc.

The proof method of the function `Ack` is shown in listing 2.5.

```
method Ack_termination_proof(m: Int, n: Int)
    requires m >= 0
    requires n >= 0
{
    if (m == 0){
        // n + 1
    } else {
        if (n == 0){
            assert m-1 ≺ m || (m-1 == m && 1 ≺ n)
            // Ack(m - 1, 1),  [m-1, 1] ≺ [m, n]
        } else {
            assert m ≺ m || (m == m && n-1 ≺ n)
            // Ack(m, n - 1),  [m, n-1] ≺ [m, n]
            assert m-1 ≺ m || (m-1 == m && Ack(m, n-1) ≺ n)
            // Ack(m - 1, Ack(m, n - 1)),  [m-1, Ack(m, n-1)] ≺ [m, n]
        }
    }
}
```

Listing 2.5: proof method of the Ackerman function (listing 2.4)

All of the termination checks in 2.5 verify, which proofs termination of the function `Ack` in listing 2.4.

Because the result value of the proof method does not affect the result of the termination check, it can be omitted, which also keeps the proof method smaller. This was done in the proof method in listing 2.5 and also in further examples in this thesis.

Function Calls as Arguments

Consider the following listing showing the McCarthy 91 function [7]. The Mc-

```
1 function mc(n: Int): Int
2     ensures n <= 100 ==> result == 91
3     ensures n > 100 ==> result == n-10
4 {
5     (n>100) ? n-10 : mc(mc(n+11))
6 }
```

Listing 2.6: McCarthy 91 function

Carthy 91 function is a recursive function that evaluates to 91 for all arguments $n \leqslant 100$ and to $n-10$ for any other argument. By using $100-n$ as the variant one is able to prove its termination. The generated proof method can be seen in listing 2.7.

```
1  method mc_termination_proof(n: Int)
2  {
3      if(n > 100){
4          // n - 10
5      } else {
6          assert 100-(n+11) ≺ 100-n
7          // mc(n+11)
8
9          assert 100-mc(n+11) ≺ 100-n
10         // mc(mc(n+11))
11     }
12 }
```

Listing 2.7: proof method of the McCarthy 91 function (listing 2.6)

In the proof method, the assertion in line 6 checks that the variant decreases for the call `mc(n+11)`, which is nested inside another `mc` call. Afterwards it can be assumed that the function terminates for that call. The termination check in

line 9 verifies because the postcondition of the call `mc(n+11)` is assumed by the verifier, as usual, since its precondition is trivially satisfied. This could give the impression that in the proof a cyclic dependency exists because termination of the function is proven by assuming that the function terminates. But actually, termination of a function f is proven by showing that f terminates with any possible variant value v while assuming f terminates with any variant value $v' \prec v$. This is sound because it implies an induction scheme over the well-founded order of the variants.

Postcondition Check

As described above, recursive function calls can be assumed to terminate if the variant is smaller. This allows us to assume the postcondition of recursive function calls if needed. To ensure well-formedness of postconditions, they too have to be proven to terminate. Therefore, in an additional proof method, all postconditions are proven to terminate in the same way as a function body.

2.1.2 Limitations of the Variant Termination Proof Approach

While the variant termination proof is trivial to apply to a function if the variants for it is given, finding such a variant is not. Listing 2.8 shows an example of two mutually recursive functions `f` and `g`. The function `f` is terminating because it invokes itself via `g` always with a smaller x and x cannot become negative. Function `g` terminates for the same reason. Since x decreases in each recursive invocation of `f` it would be intuitive to choose it as f's variant. The same can also be said about y for `g`. However, the variant termination proof would not succeed because at the call `g(x+1)` in `f` the variant actually increases by 1.

```
1  function f(x: Int): Int
2      requires x >= 0
3      ensures result == 0
4  {
5      x == 0 ? 0 : g(x+1)
6  }
7
8  function g(y: Int): Int
9      requires y >= 2
10     ensures result == 0
11 {
12     f(y-2)
13 }
```

Listing 2.8: mutually recursive functions, which require an extended variant

A solution for this particular example would be to adjust and extend both variants to tuples containing an additional constant. E.g. the variants $[x, 0]$ and $[y - 2, 1]$ for the functions `f` and `g`, respectively, would work. Because such examples can become arbitrarily complex and finding suitably extended variants more difficult, we propose another approach, which in many cases allows to directly use the intuitively chosen variants.

2.1.3 Function Inlining Transformation

The general idea of the approach we suggest is to apply a transformation to the function, prior to the variant termination proof.

Definition 2.1 (Function Inlining Transformation (FIT))**.** *The FIT transforms a function by transitively inlining each indirectly recursive function call until a cycle is detected.*

The result of FIT applied to `f` (listing 2.8) is shown in listing 2.9.

```
1  function f(x: Int): Int
2      requires x >= 0
3      ensures result == 0
4  {
5      x == 0 ? 0 : f((x+1)-2)
6  }
```

Listing 2.9: result of FIT applied to function `f` (listing 2.8)

The transformed function now only contains the directly recursive call `f`(x+1)-2)) and thus, the more intuitive expression x can be used as the variant to prove termination. Function `g` can also be proven to terminate using the variant y after applying FIT to it. The proof method of the function f after applying FIT to it is shown in listing 2.10.

```
1  method f_termination_proof(x: Int)
2      requires x >= 0
3  {
4      if (x == 0){
5          // 0
6      } else {
7          assert (x+1)-2 ≺ x  // termination check
8          // f((x+1)-2)
9      }
10 }
```

Listing 2.10: proof method of function `f` (listing 2.9)

The termination check in line 7 verifies successfully, as expected.

Lemma 2.1 (Termination of FIT)**.** *The transformation of an arbitrary function with the FIT always terminates.*

Informal Proof. Because the number of functions is finite, only a finite number of calls can be inlined until a cycle is detected.

Lemma 2.2 (Semantical Preservation of FIT)**.** *The transformation of an arbitrary function with the FIT does not change its semantics.*

Informal Proof. Because inlining function calls does not change the semantical meaning of a function, FIT also does not.

Theorem 2.3 (Soundness of Termination Proof after FIT)**.** *A valid termination proof for a function transformed with FIT is also a valid termination proof for the original function.*

Informal Proof. Because of lemma 2.2 there is no function for which a non-terminating execution exists before, but not after, it is transformed with FIT.

2.1.4 Unsoundness of Previous Work

In the previous work [6] an approach was chosen which explores static, "direct" execution paths and substitutes arguments on the way, i.e. inlines function calls, until a cycle is detected. This part is similar to FIT, but not equal because it inlines any call and not just the recursive ones as FIT does. After the path exploration, only directly recursive calls were checked to decrease the variant, instead of also indirectly recursive calls, as is done in the variant termination proof approach. This causes the approach from the previous project to be unsound for mutually recursive functions. Listing 2.11 shows an example of non-terminating, mutually recursive functions.

```
1 function f(x: Int): Int
2     requires 0 <= x
3     decreases x
4 { x == 0 ? g(x+1) : f(x-1) }
5
6 function g(y: Int): Int
7     requires 0 <= y
8     decreases y
9 { y == 0 ? f(y+1) : g(y-1) }
```

Listing 2.11: non-terminating, mutually recursive functions

The non-termination can be shown by the following execution. Calling `f` with $x = 1$ invokes `f` with $x = 0$, then `g` with $y = 1$, then `g` with $y = 0$ and then again `f` with $x = 1$, which is the same input as we started with.

As an attempt to prove termination of `f`, the previous approach explores the following paths with a as an arbitrary integer: $f(a) \to f(a-1)$ if $a \neq 0$ on which the variant decreases. The path $f(a) \to g(a+1)$ if $a = 0$ is also checked but the recursive call $f(a+1+1)$ in $g(a+1)$ is considered infeasible since its guard $a = 0$ contradicts the first guard in combination with the precondition. The other indirectly recursive call, $g(a+1-1)$, is unsoundly considered harmless, since `g` is separately checked for termination (which succeeds equally unsoundly).

In summary: the previous approach works only for directly recursive functions, but not for mutually recursive functions.

2.1.5 Limitations of FIT

Even though with FIT some functions can be proven to terminate with a more intuitive variant, it has its limitations. Listing 2.12 shows an example for which FIT does not allow to use the more intuitive variants.

```
1 function f(x: Int): Int
2 {
3     x <= 0 ? 0 : g(x+1, x)
4 }
5
6 function g(y: Int, z: Int): Int
7 {
8     z <= 0 ? f(y-2) : g(y, z-1)
9 }
```

Listing 2.12: mutually recursive functions that demonstrate the limitations of FIT

Intuitively, x is a good variant for `f`, since it decreases in each recursive invocation of `f` and is bounded. For `g` the variant $[y, z]$ would be an intuitive choice, for the same reasons. However, the termination proof approach fails with those variants, even in combination with FIT. The result of FIT applied to `f` is shown in listing 2.13.

```
1 function f(x: Int): Int
2 {
3     x <= 0 ? 0 : x <= 0 ? f((x+1)-2) : g(x+1, x-1)
4 }
```

Listing 2.13: result of FIT applied to function `f` (listing 2.12)

In listing 2.13, the directly recursive call `f((x+1)-2)` is not reachable because of the conditions and therefore must not be checked. The indirectly recursive

call `g(x+1, x-1)` does not decrease the variant because $x+1 < x$ is false, hence the proof fails. The proof method generated for the function `f` after applying FIT to it and using the intuitively chosen variants, i.e. x and $[y, z]$ for the functions `f` and `g`, respectively, is shown in listing 2.14.

```
1  method f_termination_proof(x: Int)
2  {
3      if (x <= 0) {
4          // 0
5      } else {
6          if(x <= 0) {
7              assert (x+1)-2 < x  // termination check
8              // f((x+1)-2)
9          } else{
10             assert x+1 < x  // termination check
11             // g(x+1, x-1)
12         }
13     }
14 }
```

Listing 2.14: proof method of function `f` (listing 2.13)

The termination check in line 10 fails to verify, as expected. A possible solution for this example is to use the extended variants $[x, 0]$ and $[y-2, 1, z]$ for the functions `f` and `g`, respectively, for which the termination proof for both functions succeeds.

However, in the next section we present an approach that allows us to use the more intuitive variants to prove termination of `f` and `g`.

2.2 Transition Invariants

In this section we describe the termination proof approach proposed by Podelski and Rybalchenko [10] and show how it can be applied to Viper functions. Further, some examples will be presented to demonstrate its advantages over the variant termination proof approach. Last, the example from section 2.1.5 will be solved with more intuitive variants.

In the approach by Podelski and Rybalchenko *transition invariants* are used to proof termination. For a recursive function f in Viper, a transition invariant is a binary relation over f's parameter values, which describes how f's parameter values change transitively over recursive invocations of f. This is similar to a variant, which, if proven to be correct, describes a value which decreases in each recursive invocation with respect to a well-founded order, and due to the transitivity of a well-founded order, also decreases transitively over recursive

invocations. If f's transition invariant is *disjunctively well-founded*, i.e. it is a finite union of well-founded relations, then the function f terminates.[2]

Podelski and Rybalchenko presented this approach with an algorithm which finds the transition invariant automatically. However, this is not relevant in our case because in Viper, users are expected to provide specifications to a function, which would also include a function's transition invariant.

2.2.1 Definitions and Theorems

In this section, we show definitions for transition invariants and disjunctive well-foundedness, all within the context of Viper functions. The definitions correspond to the ones introduced by Podelski and Rybalchenko [10], but are adjusted to our use case.

We define a function $f = \langle W, R \rangle$, where

- W is a set of f's parameter values, which are allowed by f's precondition[3].
- R is a transition relation over recursive invocations of f, such that $R \subseteq W \times W$.

To illustrate the newly introduced definition, function `sum` from listing 1.1 is used (shown again in listing 2.15).

```
1 function sum(n: Int): Int
2     requires n >= 0
3 {
4     n == 0 ? 0 : n + sum(n - 1)
5 }
```

Listing 2.15: sum of the first n positive integers

The function `sum`$= \langle W_{sum}, R_{sum} \rangle$ is defined as follows:

$$W_{sum} = \{n \mid n \geqslant 0\}$$

$$R_{sum} = \{(n, \widetilde{n}) \mid n \neq 0 \wedge \widetilde{n} = n - 1\} \cap (W \times W)$$

W_{sum} is the set representation of the precondition of `sum`. R_{sum} describes the relation between the values of `sum`'s parameter, i.e. n, for which `sum` invokes itself recursively and the parameter values of the recursive invocation. The function `sum` only recursive invokes itself if $n \geqslant 0 \wedge n \neq 0$. The parameter's value in the recursive invocation (here denoted with $\widetilde{n}$) is then $\widetilde{n} = n - 1 \wedge \widetilde{n} \geqslant 0$.

[2] Further descriptions of the definitions used are shown in section 2.2.1.

[3] The value of a reference parameter includes the values of fields referenced by the parameter and accessible to the function f.

Because `sum` only contains directly recursive calls in its body, R_{sum} is defined by `sum`'s body. This is generally the case for directly recursive functions. The transition relation of a mutually recursive function f, however, can also depend on the body of functions, via which f recurses.

Using the transition relation, the transition invariant can formally be defined as follows:

Definition 2.2 (Transition Invariant)**.** *A transition invariant T for a function $f = \langle W, R\rangle$ is a superset of the transitive closure of the transition relation R. Formally, $R^+ \subseteq T$.*

E.g. the relations $\{(n, n') \mid 0 < n \wedge 0 \leqslant n'\}$, $\{(n, n') \mid n' < n\}$ and $\{(n, n') \mid true\}$ are all transition invariants for the function `sum` in listing 2.15 because they are all supersets of R^+_{sum}. The relation $\{(n, n') \mid n' = n - 1\}$ does not contain the transitive closure of R_{sum}, and therefore, is not a transition invariant for `sum`.

Any termination proof approach for recursive functions (known to us), basically tries for a function $f = \langle W, R\rangle$ to show that R is well-founded. Well-foundedness of R implies non-existence of an infinite sequence $s_0, s_1, \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$; hence, there cannot exists a infinite sequence of recursive invocations of f and f therefore terminates[4]. Because a transition invariant T for f is a superset of R^+, and thus, also a superset of R, i.e. $R \subseteq T$, it would be sufficient to show well-foundedness of T to prove termination of f. However, Podelski and Rybalchenko introduced the weaker property *disjunctively well-foundedness* for T, which also implies well-foundedness of R, and therefore termination of f [10].

Definition 2.3 (Disjunctive Well-Foundedness)**.** *A relation T is disjunctively well-founded if it is a finite union $T = T_1 \cup ... \cup T_n$ of well-founded relations T_i.*

Theorem 2.4 (Termination)**.** *The function f is terminating if there exists a disjunctively well-founded transition invariant for f.*

Consider function `sum` from listing 2.15. Because n decreases in each recursive invocation of `sum` and is never negative, the following is a valid transition invariant for `sum`:

$$T_{sum} := \{(n, n') \mid n' < n \wedge 0 \leqslant n\}$$

I.e. $R^+_{sum} \subseteq T_{sum}$ is satisfied.
Since $n' < n \wedge 0 \leqslant n$ holds for all $(n, n') \in T_{sum}$, there cannot exist a infinite sequence $n_0, n_1, \ldots$ of integers such that $(n_i, n_{i+1}) \in T_{sum}$ for every natural

[4] In the variant termination proof approach, this was done by showing that a variant decreases with respect to a well-founded order at each recursive invocation.

number $i \geqslant 0$, hence T_{sum} is well-founded. Therefore also disjunctively well-founded, and `sum` is proven to terminate.

As already mentioned, we expected the user to provide the transition invariants for the functions. Therefore, to prove termination of a function f by using the provided transition invariant T, the following two properties have to be shown:

- T is a transition invariant for f.
- T is disjunctively well-founded.

In the following sections, we describe how this can be done in Viper, i.e. in a Viper program which, if verified successfully, implies the two properties. The approach for the former is described in section 2.2.3, the approach for the latter in section 2.2.2.

2.2.2 Notation for Transition Invariants

In this section we introduce a notation for transition invariants by using the formal notation of T_{sum} (used in section 2.2.1), which is then adjusted over multiple steps, with the goals that the notation is short and simplifies the well-foundedness proof for the transition invariant.

As shown in section 2.2.1, the binary relation

$$T_{sum} := \{(n, n') \mid n' < n \wedge 0 \leqslant n\}$$

is a well-founded transition invariant for the function `sum` (listing 2.15).

Since the function definition already declares all parameters, a transition invariant can completely be defined by the assertion of the set definition. In the assertion, for a parameter x, x' represents the same parameter after arbitrary many recursive function invocations and we refer to it as *future* x. Because the function `sum` (listing 2.15) is defined with the parameter n, T_{sum} can be written as the following assertion:

$$T_{sum} : n' < n \wedge 0 \leqslant n$$

We refer to this form of notation for a binary relation as *assertion form*. Note that for a function f, the assertion form of transition invariant only contains variables which are either a parameter of f or are of the form p' and p is a parameter of f.

Proving disjunctive well-foundedness of an arbitrary binary relation is not simple and to encode such a proof into Viper might even be more difficult, if not impossible. Therefore, we suggest a notation which guarantees disjunctive well-foundedness of a transition invariant, so that this proof can be omitted. First, the definition of a transition invariant T for a function f is split into finitely many binary relations $T_1, T_2, ..., T_n$ such that $T := T_1 \cup T_2 \cup ... \cup T_n$.

Secondly, for each binary relation T_i it is required to define a variant v_i and to prepend it to the assertion, separated by a comma. A variant can be defined as described in section 2.1.1, e.g. as a Viper expression or a tuple of Viper expressions, whereas any variable used in the expressions has to be a parameter of f. Let w_i be the binary relation defined by the assertion form $v_i' \prec v_i$[5]. The relation w_i is implicitly conjoined to T_i, i.e. the effective transition invariant is defined as $T := (w_1 \cap T_1) \cup ... \cup (w_n \cap T_n)$. Because the binary relations w_i is well-founded, $w_i \cap T_i$ is well-founded. And because T is a finite union over well-founded relations, it is disjunctively well-founded. The transition invariant T_{sum} has therefore to be defined as follows:

$$T_{sum_1} : \ n, \ n' < n \wedge 0 \leqslant n$$

The effective transition invariant is then $\{(n, n') \mid n' \prec n \wedge n' < n \wedge 0 \leqslant n\}$. Because $n' \prec n$ implies $n' < n \wedge 0 \leqslant n$ (due to the definition of $\prec$ given in section 2.1.1), the effective transition invariant is equal to T_{sum}. Further, the following definition generates also equivalent effective transition invariant but is shorter.

$$T_{sum_1} : n, \ true$$

the following definition generates an equivalent effective transition invariant.

$$T_{sum_1} : n, \ true$$

For brevity it is also allowed to only define the variant of a relation, without any assertion:

$$T_{sum_1} : n$$

The assertion is then implicitly assumed to be true.

The following example shows the `sum` function from listing 2.16 with a possible encoding of the transition invariant T_{sum} in a Viper program.

```
1 function sum(n: Int): Int
2     requires n >= 0
3     t_inv n              // T_sum_1: n
4     t_inv n, n' < n      // T_sum_2: n, n' < n
5 {
6     n == 0 ? 0 : n + sum(n - 1)
7 }
```

Listing 2.16: function `sum` with transition invariant specifications

Here, the keyword `t_inv` is used to specify the transition invariant. Line 4 serves only for demonstration purpose and would not be necessary to define T_{sum}.

[5] We use $\prec$ as defined in section 2.1.1.

In summary, we presented a transition invariant notation, which can be of the same size as the variant notation in the variant termination proof approach. Additionally, the notation guarantees disjunctive well-foundedness of any user-provided transition invariant. Hence, there is no need to show disjunctive well-foundedness of the transition invariant and it only remains to encode the transition invariant correctness proof into Viper, which is shown in the next section.

2.2.3 Proof Encoding in Viper

In this section we describe how the transition invariant correctness proof can be encoded into a Viper program, which, if successfully verified, implies the transition invariant correctness proof. In this section, only directly recursive functions are considered. In a later section, we show an approach which also works for mutually recursive functions.

Podelski and Rybalchenko guaranteed correctness of their automatically generated transition invariants by using *inductive relations* [10]. The following definition corresponds to the *inductive relation* definition given by Podelski and Rybalchenko [10], but is adjusted to our use case.

Definition 2.4 (Inductive Relation)**.** *Given a function $f = \langle W, R \rangle$, a binary relation T on W is inductive if it contains the transition relation R and it is closed under the relational composition R. Formally, $R \cup (T \circ R) \subseteq T$*[6]

Technically, proving $R \cup T \circ R \subseteq T$ for some transition relation R and binary relation T is an inductive proof, which shows that the property $R^n \subseteq T$ holds for any $n > 0$. Hence, an inductive relation for a function f is also a transition invariant for f.

We also make use of inductive relations and prove correctness of a user-provided transition invariant T for a function f by showing that T is an inductive relation for f. This prove can also be encoded in a Viper program.

Let $f = \langle W, R \rangle$ be a directly recursive function and T the transition invariant provided for f. For simplicity, f only has one parameter p of type P. To prove correctness of T, first, in the *base case*, it is checked that $R \subseteq T$ holds. Secondly, in the *step case*, it is checked that $T \circ R \subseteq T$ holds.
Base Case: To be able to add assertions to the body of f, for the proof encoding a semantically equivalent method is used. In order to prove $R \subseteq T$, it is shown $\forall a\colon P.\, (p, a) \in R \implies (p, a) \in T$. Since R describes the relation between the values of the parameter of f and the values of the argument of recursive calls in f's body, it is sufficient to show that for each recursive call, here $f(a)$, $(p, a) \in T$ is satisfied. This check is encoded in an assertion, called *transition check*, which is prepended to the recursive call $f(a)$ and only verifies

successfully if $(p, a) \in T$. Let t be T in assertion form, then the transition check for $f(a)$ is $t[a/p']$.

To demonstrate the encoding of the base case, we use the function `sum` (listing 2.15) with the same transition invariant as before, i.e. the user-defined relation $T_{sum_1} : n$ and effective transition invariant $T_{sum} := \{(n, n') \mid n' \prec n\}$. Listing 2.17 shows the base case proof method of function `sum`, which contains the transition check.

```
1  method sum_bc(n: Int) returns (res: Int)
2      requires n >= 0
3  {
4      if (n == 0)
5      {
6          res := 0
7      }
8      else
9      {
10         // argument of the recursive call
11         var n_1: Int := n - 1
12
13         // transition check
14         assert n_1 ≺ n
15
16         // recursive call
17         res := n + sum(n_1)
18     }
19 }
```

Listing 2.17: base case proof method of `sum`

Line 14 in listing 2.17 shows the transition check for the recursive call in line 17, which, as expected, successfully verifies. For clarity, the argument of the recursive call is assigned to a new variable.

Step Case: As in the base case, a semantically equivalent method is used. In order to prove $T \circ R \subseteq T$ it is shown that $\forall p_0 \colon P. (p_0, p) \in T$ it is satisfied that $\forall a \colon P. (p, a) \in R \implies (p_0, a) \in T$. Therefore, at the beginning of the method, a new variable p_0 is introduced, for which it is assumed that $(p_0, p) \in T$. Similar to the base case, at each recursive call in f, here $f(a)$, it is shown that $(p_0, a) \in T$. Again, let t be T in the assertion form. As in the base case, a *transition check* is prepended to the recursive call $f(a)$, but with the assertion $t[p_0/p, a/p']$.

We again use the function `sum` (listing 2.15) and the same transition invariant as before to demonstrate the encoding of the step case proof. The result is shown in listing 2.18. Line 5 in listing 2.18 shows the declaration of the newly

```
1  method sum_sc(n: Int) returns (res:Int)
2      requires n >= 0
3  {
4      // new variable
5      var n_0: Int
6      // assume transition invariant
7      inhale n ≺ n_0
8
9      if (n == 0)
10     {
11         res := 0
12     } else {
13         // argument of the recursive call
14         var n_1: Int := n - 1
15
16         // transition check
17         assert n_1 ≺ n_0
18
19         // recursive call
20         res := n + sum(n_1)
21     }
22 }
```

Listing 2.18: step case proof method of `sum`

introduced variable. In line 7 the transition invariant is assumed for the new variable and the parameter, i.e. $(n_0, n) \in T_{sum}$. The transition check in line 17 verifies as expected.

The encoding of both parts, base case and step case, can be optimized in many ways. E.g. any statement which is not relevant to the proof, such as the result value of the proof methods, can be omitted without affecting the result of the method verification, i.e. the result of the proof.

2.2.4 Advantage over Variant Termination Proof Approach

In this section we briefly compare the transition invariant termination proof approach with the variant termination proof approach. Listing 2.19 shows an example[7] of a function for which it is difficult to prove termination using the variant termination proof approach because it is difficult to find a variant. In each recursive invocation of the functions, y decreases while x increases, until y is negative, then x decreases down to 0.

[7] The example in listing 2.19 is an adaption from Program 7 by William Gasarch [5].

```
function p7(x: Int, y: Int): Int
{
    x > 0 ? p7(x + y, y - 1) : x + y
}
```

Listing 2.19: p7 function

Even though x is bounded by 0, it cannot be used as the variant because it does not decrease if y is not negative. Also, y cannot be used as the variant because for any bound b the invocation `p7(1, b)` would decrease y below b for the following recursive call. We were also not able to find a variant which is a combination of x and y, including tuples, without changing the function's definition[8].

However, because it is known that either y decreases and is positive or x decreases and is positive, a disjunctively well-founded transition invariant can easily be found for $p7$:

$$T_{p7_1} : x,\ y > y'$$

$$T_{p7_2} : y$$

The effective transition invariant for `p7` is then defined as follows:

$$T_{p7} := \{((x, y), (x', y')) \mid x' \prec x \wedge y > y' \vee y' \prec y\}$$

If y is negative, T_{p7_1} is satisfied because x decreases and is bounded by 0, hence, x decreases with respect to a well-founded order. If y is non-negative, T_{p7_0} is satisfied because y decreases and is bounded by 0, hence, decreases with respect to a well-founded order. The assertion $y > y'$ in the relation T_{p7_1} is necessary to prove that T_{p7} is an inductive relation for `p7`. In particular, the step case would not succeed without the additional assertion.

The transition invariant correctness proof encoding for the function `p7` and the transition invariant T_{p7} is shown in listing 2.20 and listing 2.21. In this case, the result value of the proof methods is omitted, which as already described, does not affect the result of the proof.

Both proof methods (listing 2.20 and listing 2.21) verify as expected, and hence, prove correctness of the chosen transition invariant T_{p7}. Therefore `p7` is proven to terminate.

[8] Appendix A.1 shows an modified version of function `p7` for which we were able to find a variant termination proof

```
method p7_bc(x: Int, y: Int)
{
    if (x > 0){
        var x_1: Int := x + y
        var y_1: Int := y - 1
        // transition check
        assert x_1 ≺ x && y > y_1
            || y_1 ≺ y
        // p7(x + y, y - 1)
    } else { // x + y }
}
```

Listing 2.20: base case of the transition invariant correctness proof encoding for function p7 (listing 2.19)

```
method p7_sc(x: Int, y: Int)
{
    var x_0: Int     // new variables
    var y_0: Int     // new variables
    // assume transition invariant
    inhale x < x_0 && 0 <= x_0 && y_0 > y
        || y < y_0 && 0 <= y_0
    if (x > 0){
        // arguments for recursive call
        var x_1: Int := x + y
        var y_1: Int := y - 1

        // transition check
        assert x_1 ≺ x_0 && y_0 > y_1
            || y_1 ≺ y_0

        // p7(x + y, y - 1)
    } else {
        // x + y
    }
}
```

Listing 2.21: step case of the transition invariant correctness proof encoding for function p7 (listing 2.19)

2.2.5 Mutually Recursive Functions

For a directly recursive function $f = \langle W, R \rangle$, R is defined by the function's body. This fact is used to encode the transition invariant correctness proof for

a directly recursive function (shown in section 2.2.3), which places transition checks before each recursive call in f's body. If function f is not directly recursive, R can also depend on the body of f's mutually recursive functions. Therefore, the transition invariant correctness proof encoding cannot be directly applied to a mutually recursive function. We propose in this section a function transformation, which, applied in advance to a mutually recursive function, allows to use the proof encoding proposed for directly recursive functions. We demonstrate the transformation on an example and also informally argue its correctness.

Basically, for a mutually recursive function $f = \langle W, R \rangle$, the transformation creates a directly recursive representation of f, $f' = \langle W, R' \rangle$, such that its transition relation is a superset of the transition relation of f, i.e. $R \subseteq R'$. Therefore, for a transition invariant T, a transition invariant correctness proof for f', i.e. a proof that T is a transition invariant for f', implies that T is a transition invariant for f because, $R' \cup T \circ R' \subseteq T$ implies $R \cup T \circ R \subseteq T$.

For simplicity, we assume all functions to have one parameter.

Definition 2.5 (Advanced Function Inlining Transformation (AFIT))**.** *The AFIT traverses the body of a function f depth-first top-down and recursively applies the following to each indirectly recursive call:*
Let $g(p)$ be the indirectly recursive function call and T_g the transition invariant for the function g.

1. *If no inlining of a call to g preceded this one then inline $g(p)$.*
2. *If one inlining of a call $g(q)$ preceded the call $g(p)$ then a new variable q' is introduced, which represents g's parameter for any recursive invocation after $g(q)$ by assuming $(q, q') \in T_g$. The call $g(p)$ is replaced with the call $g(q')$, which is then inlined.*
3. *If two other inlining of calls to g preceded this one, then $g(p)$ is removed.*

Intuitively, in case 2 in the definition 2.5, a cycle of unknown length starting with a call $g(q)$ and ending with a call $g(p)$ is detected and replaced by a cycle starting at $g(q)$ and ending at an arbitrary, potential successor $g(q')$.

Because AFIT introduces new variables and requires assumption statements, in Viper, the result of AFIT has to be encoded in a method and not a function. However, this is irrelevant in our case since the result is only used to encode the transition correctness proof, which is done in a Viper method.

To demonstrate AFIT the example from listing 2.13 is used, for which the variant termination proof approach did not work with the intuitively chosen variants. The example is shown again in listing 2.22.

```
1 function f(x: Int): Int
2 {
3     x <= 0 ? 0 : g(x+1, x)
4 }
5
6 function g(y: Int, z: Int): Int
7 {
8     z <= 0 ? f(y-2) : g(y, z-1)
9 }
```

Listing 2.22: mutually recursive functions from listing 2.13

Because in `f`, x decreases in each recursive invocation of `f` down to 0, it can be used as its transition invariant:

$$T_{f_1} : x$$

The effective transition invariant for `f` is $T = T_{f_1}$. In `g`, the tuple $[y, z]$ decreases with respect to the well-founded lexicographical order in each recursive invocation of `g` and therefore can be used as its transition invariant:

$$T_{g_1} : [y, z]$$

The effective transition invariant for `g` is $T_g = T_{g_1}$.

To encode the transition invariant correctness proofs for the mutually recursive functions `f` and `g`, first AFIT has to be applied to both the functions. For brevity, we only show AFIT applied to `f`. The result of applying AFIT to the function `f` is shown in listing 2.23.

All the indirectly recursive function calls, that are encountered by AFIT are mentioned in the comments. Line 6 shows the first inlined function call. For clarity, instead of propagating `f`'s parameter, when inlining the call `g(x + 1, x)`, new variables are introduced to represent the arguments. In line 13 is the second function call to `g` on this path, which is replaced with the function call in line 20. In line 15 and 16 the variables `y_new` and `z_new`, respectively, are declared, which represent the parameters of all further invocations of `g` on this path. The assumption $((y, z), (y\_new, z\_new)) \in T_g$ is encoded as an inhale statement in the lines 18 and 19. In line 24 is the third encountered call to `g` on this path, which was removed by AFIT.

The method `f_afit` can now be used to encode the transition correctness proof for the function `f` as described in section 2.2.3.

Soundness of Advanced Function Inlining Transformation (AFIT)

In this subsection we show an informal proof for the soundness of AFIT, i.e. a proof that a valid transition invariant correctness proof for a function obtained

```
method f_afit(x: Int) returns (res: Int)
{
    if (x <= 0) {
        res := 0
    } else {
        // g(x + 1, x)  inlined
        // parameters of the inlined call
        var y: Int := x + 1
        var z: Int := x
        if (z <= 0) {
            res := f_afit(y - 2)
        } else {
            // g(y, z - 1)  replaced
            // new parameters
            var y_new: Int
            var z_new: Int
            // assume transition invariant of g
            inhale (y < y_new)
                    || (y == y_new && z < z_new)
            // g(new_y, new_z)  new and inlined
            if (z_new <= 0) {
                res := f_afit(y_new - 2)
            } else {
                // g(new_y, new_z-1)  removed
            }
        }
    }
}
```

Listing 2.23: AFIT applied to the function `f` in listing 2.22

by AFIT is also a valid proof for the original function.

Lemma 2.5 (Termination of AFIT)**.** *The transformation of an arbitrary function with AFIT always terminates.*

Informal Proof. Because the number of functions is finite and AFIT stops the inlining after encountering three calls to the same function, AFIT must terminate.

Lemma 2.6 (Soundness of Termination Proof after AFIT)**.** *A valid transition invariant correctness proof for a function obtained by AFIT, is also a valid proof for the original function.*

Informal Proof. Let $f = \langle W, R \rangle$ be an arbitrary function and $f' = \langle W, R' \rangle$ the result of AFIT applied to f. Consider AFIT to be applied over multiple iterations, such that in each iteration of the AFIT, one of the three operations defined in definition 2.5 is applied to an indirectly recursive function call. $f_i = \langle W, R_i \rangle$ represents f after i iterations of AFIT. Therefore, $f_0 = f$ and $f_n = f'$ for some $n \geqslant 0$, if AFIT terminates after n iterations when applied to f. We prove $R \subseteq R'$ by showing that, in each iteration of AFIT, the transition relation never is reduced, i.e. show $R_i \subseteq R_{i+1}$ for all $i \in [0, n-1]$. In each iteration, one of the three operations in definition 2.5 is applied to an indirectly recursive call $g(p)$. Let T_g be the transition invariant for $g = \langle W_g, R_g \rangle$. The numbering in definition 2.5 is used to distinguish the three possible cases.

- Case 1: Because inlining of a function call, in particular $g(p)$, does not change the semantics, it holds that $R_i = R_{i+1}$. Hence, $R_i \subseteq R_{i+1}$.
- Case 2: Let $g(q)$ be the inlined call preceding $g(p)$ and q' the new parameter, for which $(q, q') \in T_g$ is assumed. In the transition invariant correctness proof of g it is, independently from f, proven that $(q, p) \in T_g$ (base case). Therefore, the replacement of $g(p)$ with $g(q')$ and the inlining of $g(q')$ does not reduce the transition relation, i.e. $R_i \subseteq R_{i+1}$.
- Case 3: Let $g(q)$ and $g(q')$ be the inlined calls preceding $g(p)$ in this order, then it was already assumed that $(q, q') \in T_g$ (because of second operation in AFIT). In the transition invariant correctness proof of g it is, independently of f, proven that $(q, p) \in T_g$ (step case). Since q' also represents p, removing the function call $g(p)$ does not reduce the transition relation, i.e. $R_i \subseteq R_{i+1}$ holds.

We have shown $R_i \subseteq R_{i+1}$ holds for all $i \in [0, n-1]$. Hence, $R = R_0 \subseteq R_n = R'$. Assume we have proved transition correctness for f', i.e., we have shown, for the transition invariant T_f, that $R' \cup R' \circ T_f$ is satisfied. This implies $R \cup R \circ T \subseteq T_f$. Therefore, T_f is also a transition invariant for the function f.

It is important to emphasize that this is just a brief sketch of how the soundness of AFIT could be proven.

Chapter 3

Termination Plugin

As part of this thesis we implemented the variant termination proof approach for functions presented in section 2.1. As in the previous project, the termination proofs are encoded as additional Viper code, also referred to as proof code. This made it possible to develope a single tool that can be used with both of the Viper verifiers without having to modify them. Additionally, no changes were made to the Viper syntax, which could have impacted different Viper front-ends. The following sections in this chapter should give a brief overview of the implementation.

3.1 Decreases Clause

As described in section 2.1, the variant termination proof approach uses a variant to prove termination of a function. To be able to define such a variant a decreases clause is used. The decreases clause is regarded as an additional part of the function specification and can be used in two different ways:

1. A **decreases tuple** $[e_1, ..., e_n]$ defines the variant of the function to be the tuple $[e_1, ..., e_n]$ of expressions e_i.
2. **Decreases star** prevents the tool from doing any termination checks for the function.

If no decreases clause is defined for a function, the parameters are automatically chosen as the variant.

To avoid changes to the Viper syntax, the decreases clause has to be added to a function's specification in such a way that the parser will accept it. Therefore, the following encoding was chosen:

1. **Decreases tuple** $[e_1, ..., e_n]$ is encoded in a postcondition as a `decreases` function call with the expressions of the tuple as the arguments:

```
requires decreases(e_1,..., e_n)
```

2. **Decreases star** is encoded in a postcondition as a `decreasesStar` function call with no arguments:

```
requires decreasesStar()
```

Listing 3.1 shows the `sum` function from listing 1.1 with the variant defined in the decreases clause.

```
1 function sum(n: Int): Int
2     requires n >= 0
3     ensures result == n*(n+1)/2
4     ensures decreases(n)
5 {
6     n == 0 ? 0 : n + sum(n - 1)
7 }
```

Listing 3.1: `sum` function (listing 1.1) with a decreases clause

Because the decreases clause is part of a function's specification, only pre- and postconditions seemed to be suitable to place it. And since the termination proof is done by analyzing a function's body, it is rather a postcondition which has to be satisfied. By using the postcondition for the decreases clause it is also possible to place it at the end of the specification clauses, which is also done by other verifiers such as Dafny [3], although in a specially dedicated clause.

However, a decreases clause is not a real postcondition of a function, and therefore, it is removed from the program before it is actually verified (described in section 3.5).

3.2 Plugin Overview

For the implementation of the termination proof, the newly added plugin system of Viper [12] was used. When activated, a Viper plugin can (among other things) use hooks, i.e. callbacks, to modify the AST at specific stages during parsing and resolving, and this without changing any of Viper's core-code.

The diagram in figure 3.1 shows the pipeline of our plugin. The ellipses describe the expected form of the data going through the pipeline. The arrows are labelled with the processes of Viper and the boxes represent the phases of the termination plugin, which are further described in the following sections.

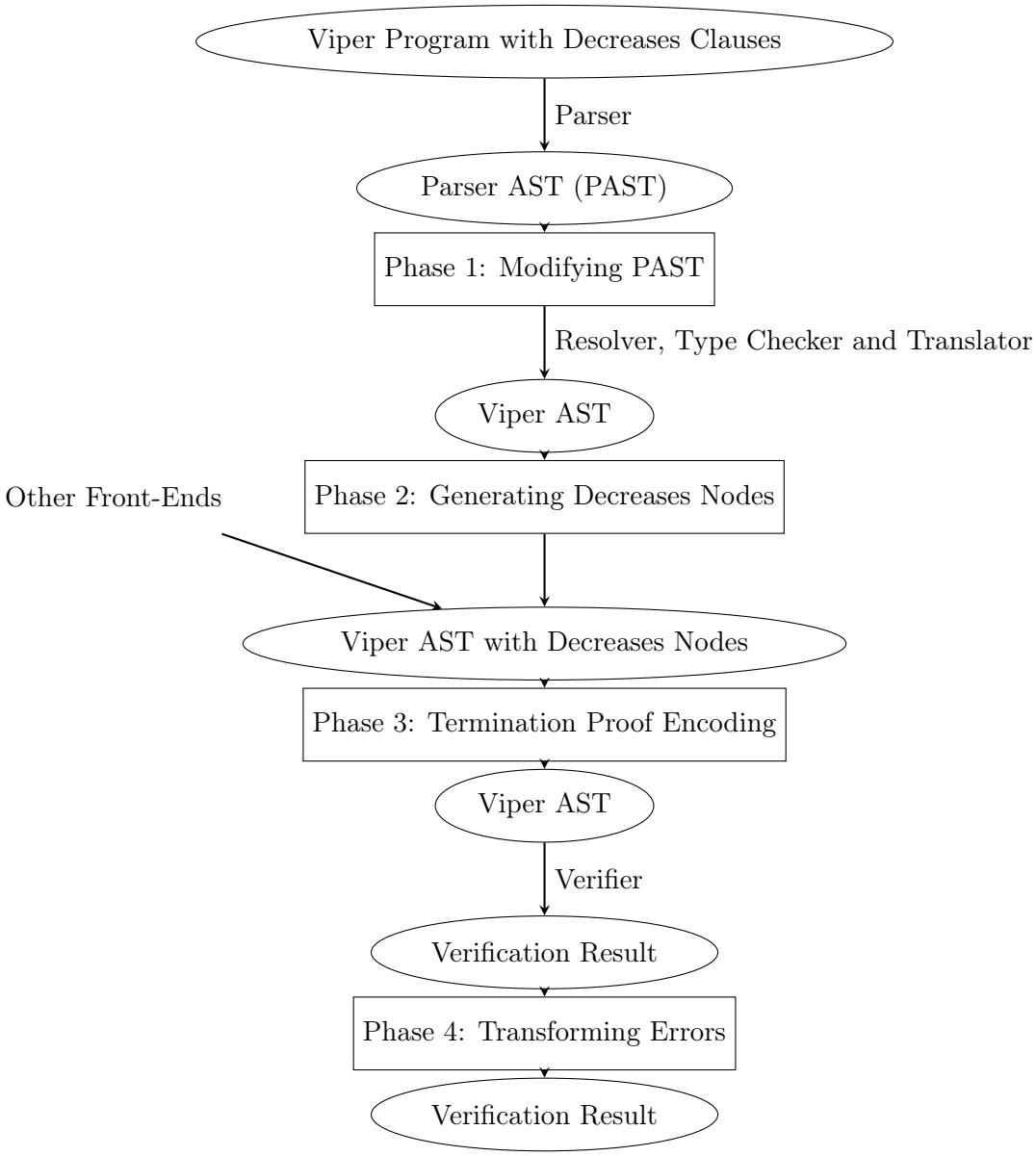


Figure 3.1: pipeline of the termination plugin

The plugin expects a Viper program, potentially containing decreases clauses as described in section 3.1, which is parsed by Viper into a so-called parse AST (PAST). In phase 1 (section 3.3), the PAST is then modified such that Viper can resolve and type check the decreases clauses and then translate the PAST into a Viper AST. In phase 2 (section 3.4), the plugin transforms the decreases clauses encoded as as functions calls into decreases clause nodes, which are custom AST nodes. Viper front-ends which directly generate the

Viper AST can use the decreases clause nodes to define a decreases clause for a function. The additional proof code for the termination proofs is generated in phase 3 (section 3.5). After one of the Viper verifiers provided its result for the program, termination-proof-related errors are transformed into more user-friendly messages in phase 4 (section 3.6.2).

3.3 Modifying PAST

Because decreases clauses are defined with calls to functions that are not defined in the program, the Viper resolver would report an error. Additionally, Viper does not allow overloading of functions, which would happen if two decreases clauses with different tuple sizes were defined in one Viper program.

To avoid these errors, in the first phase, the plugin modifies the PAST after the Viper program was parsed. Each `requires decreases(e_1,..., e_n)` is replaced with a `requires decreasesN(e_1,..., e_n)`, where `N` is the number n. This avoids overloading of the function `decreases`. Furthermore, all needed `decreasesN`, as well as the `decreasesStar` function, are declared as domain functions [9] returning a boolean. Appending the newly declared domain functions to the program then enables the resolver to proceed as normal. The type checker then firstly successfully verifies that the `decreasesN` and `decreasesStar` functions return a boolean, which is required for postcondition assertions. Secondly, all the arguments of the `decreasesN` functions, i.e. the expressions of the variant, are also type checked. This is crucial because all the expressions are later expected to be well-typed.

3.4 Generating Decreases Nodes

As described in section 3.1, in any Viper program, a decreases clause for a function is encoded in the function's postcondition as a function call. This is clearly not the best way to do it, but no changes to the Viper syntax had to be made for this encoding. However, for the AST it is possible to define custom nodes, so-called AST extensions [8]. Such nodes can be part of a Viper AST, just like any other regular AST node. Our plugin makes use of this possibility and offers custom AST nodes in Scala, which are used to represent a decreases clause.

1. `DecreasesTuple(tuple: Seq[Exp])` is the Scala class which represents a decreases tuple specification.
2. `DecreasesStar()`: is the Scala class which represents the decreases star specification.

Both nodes are subtypes of the abstract class `DecreasesExp` and are expected to be appended to the postcondition list of a function.

Viper front ends that directly generate Viper ASTs themselves must use a `DecreasesTuple` or `DecreasesStar` node to define a decreases clause for a function, instead of a function the call `decreases` or `decreasesStar`.

If the plugin already modified the PAST in order to avoid resolver errors, the `decreasesN(e_1,...,e_n)` and `decreasesStar()` function calls are replaced with `DecreasesTuple(Seq(e_1,...,e_n))` and `DecreasesStar()` nodes, respectively. This replacement takes place in phase 2 of the plugin, directly after Viper generated the AST.

Every AST reaching the next phase of the plugin, i.e. phase 3, is expected to potentially contain decreases clauses as `DecreasesExp` nodes.

3.5 Termination Proof Encoding

In section 2.1.1, we described how the variant termination proof for functions can be encoded in Viper. The described encoding is also used by the plugin. In this section, we present how, in phase 4 of the plugin, the proof methods for functions are generated. Additionally, we present how well-founded orders over Viper types are defined, which is necessary to encode the termination checks.

In general, the implementation is similar to the one in the previous project [6]. Any adjustments and improvements of the new implementation are listed in section 3.6.5.

3.5.1 Proof Method Generation

For each function that should be proven to termination, i.e. for each function which does not contain a `DecreasesStar` node in the postconditions, firstly, the variant is determined. If the postcondition contains a `DecreasesTuple(tuple)`, the tuple is used as the variant. Otherwise, the parameter list of the function is used. Secondly, a proof method, as described in section 2.1.1, is generated. Therefore, the plugin traverses the function's body in post-order and generates a semantically equivalent method. For each recursive call a termination check is encoded (further described in section 3.5.2) and prepended to the recursive call call in the method's body. As already mentioned, because the result value of the proof method is not relevant to the termination proof of the function, it can be omitted, which is done in the plugin.

Using the `sum` function with a defined decreases clause from listing 3.1 as example, listing 3.2 shows the proof method generated by the plugin. The encoding of the termination check in line 6 is further described in the following section (section 3.5.2).

```
method sum_termination_proof(n: Int)
    requires n >= 0
{
    if (n == 0) {
    } else {
        assert decreasing(n-1, n) && bounded(n)
    }
}
```

Listing 3.2: proof method generated by the plugin for the `sum` function (listing 3.1)

3.5.2 Termination Check

In order to encode termination checks, an encoding of the well-founded order over the variants has to be defined. As shown in the previous project [6], a well-founded order over a Viper type can be defined with the two boolean functions `decreasing` and `bounded`. The functions have the following form:
Let T be a Viper type.

$$\texttt{decreasing}\colon T \times T \rightarrow \mathit{Boolean}$$

$$\texttt{bounded}\colon T \rightarrow \mathit{Boolean}$$

For two Viper expressions e_1 and e_2 both of the same type it is defined that[1]

$$e_2 \prec e_1 \iff \texttt{decreasing}(e_2, e_1) \wedge \texttt{bounded}(e_1)$$

The `decreasing` function ensures that the expressions are in a descendant order and `bounded` ensures that the descendant order is bounded. As in the previous project [6], the two functions are implemented as domain functions (listing 3.3).

```
domain TerminationOrder[T] {
  function decreasing(arg1: T, arg2: T): Bool
  function bounded(arg: T): Bool
}
```

Listing 3.3: declaration of decreasing and bounded function (from `dec.vpr`)

Definitions for the two function are expected to be provided by the user in form of axioms. This allows the user to define well-founded orders for any Viper type and yields significant flexibility. Axioms, which define a well-founded order for Viper's built-in types, are offered by the plugin and can be imported using Viper's import mechanism (presented in chapter 5). Listing 3.4 shows the provided axioms which define a well-founded order over Viper's built-in type `Int`.

[1] We write $v_2 \prec v_1$ if v_1 precedes v_2 with respect to a well-founded order.

```
1  domain IntTerminationOrder {
2      axiom integer_ax_dec {
3          forall int1: Int, int2: Int :: {decreasing(int1, int2)}
4              int1 < int2 ==> decreasing(int1, int2)
5      }
6      axiom integer_ax_bound {
7          forall int: Int :: {bounded(int)}
8              int >= 0 ==> bounded(int)
9      }
10 }
```

Listing 3.4: Provided Int Axioms (from `int_decreases.vpr`)

By using the two functions `decreasing` and `bounded`, termination checks can be encoded as Viper assertions. Line 6 in listing 3.2 shows the encoding of the termination check for the recursive call `sum(n-1)` in `sum`'s body (listing 3.1).

Tuples

For tuples of Viper expressions the lexicographical well-founded order is used, which is defined as follows:
Let $t_1 = [v_1, v_2, ..., v_n]$ and $t_2 = [w_1, w_2, ..., w_n]$ for some $n \geqslant 0$ be two tuples then

$$t_1 \prec t_2 \iff \exists k \in [1, min(m, n)].\, v_k \prec w_k \wedge \forall i \in [1, k].\, v_i = w_i$$

or (equivalently)

$$t_1 \prec t_2 \iff v_1 \prec w_1 \vee (v_1 = w_2 \wedge (v_2 \prec w_2 \vee v_2 = w_2 \wedge (...)))$$

The definition is sufficient if the two tuples, which are compared, are of the same size and commonly typed. Because variants of different functions are sometimes compared in the termination checks, this is not guaranteed to be. Consider the following example in listing 3.5, which shows two mutually recursive functions with differently typed tules as variants. The definition of `decreasing` requires that each invocation of it uses two arguments of the same Viper type[2]. Therefore, the variants used in a termination check first have to be trimmed to the longest commonly typed prefix. This is also done in other deductive verifiers, such as Dafny [3].

Because the variants are of the same type until the third position, both are trimmed to length 2 and the effective variants used for the termination check are $[x, y]$ and $[m, n]$ for the function `f` and `g`, respectively. The proof method with the termination checks for both functions is shown in listing 3.6.

[2]Theoretically, it could be allowed to define a well-founded order over two different types. However, for proving termination this is rarely necessary and could probably also be avoided with an additional layer of abstraction.

```
1  function f(x: Int, y: Int, z: Int): Int
2      ensures decreases(x, y, z)
3  {
4      ... g(x, y - 1, y == z) ...
5  }
6
7  function g(m: Int, n: Int, b: Bool): Int
8      ensures decreases(m, n, b)
9  {
10     ... f(m - 1, n*n, n+n) ...
11 }
```

Listing 3.5: two mutually recursive functions with differently typed variants

```
1  method f_termination_proof(x: Int, y: Int, z: Int)
2  {
3      ...
4      assert (decreasing(x, x) && bounded(x))
5          || (x == x && (decreasing(y - 1, y) && bounded(y)))
6      ...
7  }
8
9  method g_termination_proof(m: Int, n: Int, b: Bool)
10 {
11     ...
12     assert (decreasing(m - 1, m) && bounded(m))
13     || (m - 1 == m && (decreasing(n*n, n)))
14     ...
15 }
```

Listing 3.6: termination checks for the functions `f` and `g` (listing 3.5)

3.5.3 Predicates

In Viper, describing recursive heap data structure is done with predicates. The following example (listing 3.7) shows a linked-list containing one integer field for each element and a method which calculates the sum over all elements in the list. Because of the precondition in line 9, the function `listSum` holds permission to all fields within the predicate instance `list(l)`, i.e. it holds permission to `l.elem`, `l.next` and if `l.next != null` it also holds permission to all fields within the predicate instance `list(l.next)`. To use the permissions to the fields and predicate instances within a particular predicate instance, the latter is necessary to unfold. By unfolding a predicate, its current instance is exchanged with its body. In the example above, after the unfold, the predicate

```
1  field elem: Int
2  field next: Ref
3  predicate list(this: Ref) {
4      acc(this.elem) && acc(this.next) &&
5      (this.next != null ==> list(this.next))
6  }
7
8  function listSum(l: Ref): Int
9      requires list(l)
10 {
11     unfolding list(l) in l.next != null ?
12         l.elem + listSum(l.next) : l.elem
13 }
```

Listing 3.7: list represented by a recursively defined predicate

instance `list(l.next)` is accessible, which is necessary to call `listSum(l.next)`.

In Viper, a predicate instance can only have a finite number of predicate instances folded within it, which implies that a predicate instance q that is folded within the predicate instance p has fewer predicate instances folded within it than p. As in the previous project, this property is used to prove termination of functions [6]. To this end, the *nested* relation is used, which for two predicate instances q, p is defined as follows:

$$nested(q, p) \iff \text{q is folded within p}$$

Using the property from above about the number of folded predicate instances folded within a given predicate instance, and the well-foundedness of non-negative integers, we can conclude the following for two predicate instances q, p:

$$q \prec p \iff nested(q, p)$$

I.e. *nested* is a well-founded relation. Hence, a predicate instance can be used as a variant for a function. For the function `listSum` in listing 3.7 the predicate instance `list(l)` could be used as variant to proof termination. Because `list(l.next)` is nested inside `list(l)` the variant decreases with respect to the well-founded order at the recursive call, which implies termination of `listSum`.

To encode the described termination proof into Viper some additional encodings have to be defined (most of it was already introduced in the previous project [6]). For the nested relation, the domain function `nested(q: T, p: T)` is used. Because predicate instances are not a first-class Viper type, it cannot be used as an argument for the `nested` function. Therefore, a representation of predicate instances is generated by the plugin. All predicate instance represen-

tations are of type `PredicateInstance` and are defined with help of a domain function, which is generated for each predicate.

For the predicate `list` in listing 3.7, the domain function `pred_list` is generated (listing 3.8).

```
1 domain PredicateInstance {
2     function pred_list(this: Ref): PredicateInstance
3 }
```

Listing 3.8: `PredicateInstance` domain with the domain function `pred_list`

A predicate instance representation of `list(l)` would be defined as follows:

```
var list_0: PredicateInstance := pred_list(l)
```

By definition, a predicate instance is nested inside another if it is folded within it. Therefore, at each unfold, predicate instances in the unfolded instance's body are added to the nested relation. This is done by first defining a representation of the unfolded predicate instance before the unfold. Second, after the unfold, the predicate's body is traversed and relevant expressions such as conditions are transformed into method statements. For each predicate instance in the predicate's body, a representation is defined and the nested relation is assumed.

At the termination check, the predicate instance of the callee's variant is checked to be nested inside the predicate instance of the caller's variant. The caller's predicate instance representation is defined at the beginning of the proof method and is referred to as initial predicate instance.

The generated proof method of the `listSum` function is shown, enhanced with comments, in listing 3.9.

3.5.4 Implementation of Function Inlining Transformation

In section 2.1.3 the Function Inlining Transformation (FIT) was presented and the advantages of using it in combination with the variant termination proof were shown. As part of this thesis, we implemented FIT as an additional feature for the termination plugin, which users can activate. The implementation of the proof method generation allows a simple integration of additional features, which the implementation of FIT makes use of. If FIT is activated the proof method generation, instead of placing a termination check directly at a recursive function call, the call is first recursively inlined until a cycle is detected.

Consider the example from listing 2.8, repeated in listing 3.10 with decreases clauses defined for both functions. Listing 3.11 shows the generated proof method of function `f` with FIT activated and enhanced with comments.

```
method listSum_termination_proof(l: Ref)
    requires list(l)
{
    // initial predicate instance
    var list_0: PredicateInstance := pred_list(l)
    // predicate to be unfolded
    var list_1: PredicateInstance := pred_list(l)
    unfold list(l)
    // predicate's body
    if(l.next != null){
        // predicate instance in the body of unfolded predicate
        var list_2: PredicateInstance := pred_list(l.next)
        inhale nested(list_2, list_1)
    }
    if (l.next != null){
        // predicate instance of callee's variant
        var list_3: PredicateInstance := pred_list(l.next)
        // termination check
        assert nested(list_3, list_0)
    }
    fold list(l)
}
```

Listing 3.9: Proof method of `listSum` function in listing 3.7

```
function f(x: Int): Int
    requires x >= 0
    ensures result == 0
    ensures decreases(x)
{
    x == 0 ? 0 : g(x+1)
}

function g(y: Int): Int
    requires y >= 2
    ensures result == 0
    ensures decreases(y)
{
    f(y-2)
}
```

Listing 3.10: mutually recursive functions (listing 2.8)

In line 7 the first inlined call and in line 8 the recursive call for which the

```
1     method f_termination_proof(x: Int)
2     requires x >= 0
3 {
4     if (x == 0){
5         // 0
6     } else {
7         // inlined g(x+1)
8         // f((x+1)-2)
9         assert decreasing((x+1)-2,x) && bounded(x)
10    }
11 }
```

Listing 3.11: proof method of `f` (listing 3.10) with FIT activated

termination check is created are shown as comments. The termination check in line 9 verifies as expected and proofs termination of function `f`.

3.6 Errors Reporting

In program verification, error messages should help the user to detect problems in the program code. Therefore, additional information about the error, e.g. position of occurrence or cause, are included in the messages. In this section we briefly describe how the Viper error system works and which kind of information are included in messages. Then we present how termination-proof-related errors are generated by the plugin and what information the user receives.

3.6.1 Viper Error Structure

In Viper, an error object always contains a message, an AST node which represents the position of the error, and a reason. A reason describes the cause of the error more precisely: it contains a message and an AST node, which contains the position of the reason. Listing 3.12 shows an example of an assertion for which Viper would issue an error.

```
1 var x: Int := -1
2 assert x != 0 && 0 < x && x == 1
```

Listing 3.12: assertion causing an error

The error message from Viper is shown in listing 3.13. In the first line is the message from the error, which describes what kind of error occurred. The second line is the message from the reason, which points out what caused the

```
Assert might fail.
Assertion 0 < x might not hold.
(2.1)
```

Listing 3.13: error message for example in listing 3.12

error. Because the short-circuiting evaluation of the assertion is taken into account, only the first expression which might not hold is considered to be the reason. The last line contains the position of the error.

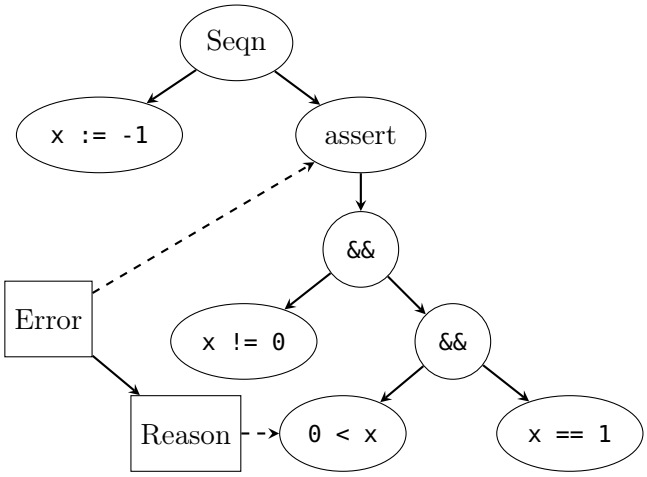


Figure 3.2: graph showing a simplified AST structure of the program in listing 3.12 and the error and reason connected to it.

3.6.2 Error Transformation

In section 3.5 we described how the plugin generates proof methods, which contain termination checks in form of assertions. Because the user is unaware of the added proof code, any error caused by it could be confusing. Consider the non-terminating function f and its corresponding proof method in listing 3.14. Function `f` does clearly not terminate, therefore, the termination check in the proof method fails. The error message for the failed assertion created by Viper looks like the following:

```
Assert might fail.
Assertion decreasing(i, i) might not hold.
(No Position)
```

Firstly, the error message does not mention the non-termination of the function `f`, which is an important information for the user. Secondly, the user did nowhere declare an assertion, hence the assertion error might be irritating.

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

Listing 3.14: non-terminating function and its proof method

The described problems are solved by transforming the termination-proof-related errors before the user sees them. This is done with the Viper transformation framework [4], which allows to add error transformers to AST nodes. If the verifier returns an error for an AST node which contains a transformer, the transformer can be invoked by the plugin. The same also works for error reasons: if the error contains a reason for an AST node which contains a transformer, then the transformer can be invoked by the plugin. The error transformers are added to the AST when the plugin creates the proof methods in phase 3, whereas the transformers are invoked in phase 4. In the next section, we describe the different error transformations used by the plugin.

3.6.3 Termination Error

An assertion error for a termination check is transformed to a termination error with the following message.

```
Function might not terminate.
```

The position of the function call for which the termination check was created is used as the position of the error. A user then knows which recursive call might cause non-termination of the function.

In the following, all the reason transformations for a termination check error done by the plugin are described. The reason message is always appended to the error message.

Variant not Decreasing

If a termination check fails because the variant does not decrease, the reason is transformed into a reason with the following message (related to listing 3.14).

```
Termination measure might not decrease.
Assertion (i, j)≺(i, j) might not hold.
```

The reason shows the variant of the caller and callee, hence, the user then exactly sees what was compared in the termination check.

Variant not Bounded

If a termination check fails because the variant is not bounded, the reason is transformed into a reason with the following message (related to listing 3.14).

```
Termination measure might not be bounded.
Assertion 0≺(i, j) might not hold.
```

Decreases Star

Functions declared with a decreases-star clause are not proven to terminate. If another function is recursive via this function and calls it, the following error will be reported at the function call.

```
Cannot prove termination,
if function declared with decreasesStar is called.
```

The error is only reported if the call is actually reachable. This is accomplished by placing a `assert false` at the position of the call in the proof method.

Error Messages with FIT

If the FIT transformation is activated, the reason messages are slightly adjusted to be more useful to the user. For an example we reuse the functions from listing 2.12, which are repeated in listing 3.15.

```
1  function f(x: Int): Int
2      ensures decreases(x)
3  {
4      x <= 0 ? 0 : g(x+1, x)
5  }
6
7  function g(y: Int, z: Int): Int
8      ensures decreases(y, z)
9  {
10     z <= 0 ? f(y-2) : g(y, z-1)
11 }
```

Listing 3.15: example from listing 2.12.

We have already shown that we cannot prove termination of `f` by using FIT and the variant termination proof. Listing 3.16 shows the proof method of the function `f`, which fails to verify as expected.

```
1  method f_termination_proof(x: Int)
2  {
3      if (x <= 0) {
4          // 0
5      } else {
6          // inlined g(x+1, x)
7          if(x <= 0) {
8              // f((x+1)-2)
9              assert decreasing((x+1)-2, x) && bounded(x)
10         } else{
11             // g(x+1, x-1)
12             assert decreasing(x+1, x) && bounded(x)
13         }
14     }
15 }
```

Listing 3.16: proof method of the function `f` (listing 3.15)

The termination check in line 12 fails because `f`'s variant does not decrease in the call `g(x+1, x-1)`. Since this call only exists in `f`'s body after FIT inlined the call `g(x+1, x)`, the user does not know for which call the termination check is done. Therefore, all the inlined calls and their position in the code, which together represent the path to the function call causing the error, are appended to the error reason message. The error and reason message for this particular example is the following:

```
Termination measure might not decrease.
Assertion (x+1)≺(x) might not hold.
Path: g(x+1, x)@4.18 -> g(x+1, x-1)@10.23.
```

Such a path will also be provided if the variant is not bounded or a star function is called recursively.

3.6.4 Further Errors

In this section, we present further errors issued by the plugin, which are not transformed from verification errors.

Declaration not Provided

The plugin requires the user to provide declarations of functions, e.g. `decreasing` and `bounded`, to create type safe termination checks. If a needed function declaration is not provided, the tool reports an error and aborts the verification.

E.g. the following error message would be shown if the `decreasing` function declaration was missing:

```
Function decreasing needed but not defined.
```

Multiple Decreases Clauses

If multiple decreases clauses are provided for one function, the plugin does not know which one to choose. Therefore, it reports an error at the position of the function definition and aborts the verification. E.g. if multiple decreases clauses are provided for a function `f`, the following error message will be reported:

```
Function f contains more than one decreases clause.
```

Recursion Via Decreases Clause

As described in section 2.1, the variant of a function is not allowed to contain a recursive (directly or indirectly) function call. If the a recursive call occurs in the variant, e.g. of function `f`, the tool reports the following error message and aborts the verification:

```
Function f recurses via its decreases clause.
```

If the function potentially recurses via other functions, e.g. `g` and `h`, a list of them is appended to the error message:

```
The cycle contains the function(s) g, h.
```

This message helps the user to detect the forbidden recursive calls in the decreases clause.

3.6.5 Improvements to Previous Implementation

As already mentioned, in a previous project, termination proofs for functions in Viper were already implemented. However, it used an unsound approach for mutually recursive functions. It was implemented into Viper's core code and affected Viper's syntax. Because the current implementation uses the variant termination proof approach, the soundness was improved. And since the implementation is a Viper plugin and does not affect Viper's syntax, it is easier to maintain as the previous one. Some other changes and improvements are described in the following paragraphs.

In section 3.5.2, the encoding of well-founded orders over Viper types was changed to allow more intuitive choices of variants. Additionally, the encoding of lexicographical well-founded order was changed to improve completeness.

This also affected the error transformation, which had to be adjusted to the termination checks.

If the user does not provide a decreases clause for a functions, the plugin automatically infers one, which reduced the annotation overhead. Further, heap-dependent functions calls are now allowed in the decreases clause, which improved expressiveness.

Chapter 4

Proving Termination of Methods

In Viper, methods, like functions, can be defined recursively and therefore cause non-termination. While in the previous project [6] only termination proofs for functions was implemented, we extended the support for termination proofs to methods. In this chapter we describe the implementation of the termination proof for methods in Viper. A variant termination proof for a method can be done in the same way as for functions: at each recursive call it is checked that a variant decreases with respect to a well-founded order. The decreases clause is again used to define a variant and the well-founded order is defined and encoded as for functions.

Reusing the sum function from listing 1.1, a semantical equivalent method can be seen in listing 4.1.

```
1  method sum(n: Int) returns (res: Int)
2      requires n >= 0
3      ensures res == n * (n+1) / 2
4      ensures decreases(n)
5  {
6      if (n == 0) {
7          res := 0
8      } else {
9          res := sum(n-1)
10         res := n + res
11     }
12 }
```

Listing 4.1: function sum (listing 1.1) as method

The decreases clause in line 4 defines the variant for the method to be n. The plugin places the termination check for the recursive call in line 9 directly in front of the call. This avoids the creation of an additional proof method and does not increase the code size by much.

```
...
} else {
    assert decreasing(n-1, n) && bounded(n)
    res := sum(n-1)
...
```

4.1 Termination Check Adjustments

Because methods contain statements, in contrast to functions, which only contain expressions, the program state may change within the method body. This has to be taken into account when proving termination of a method. The following sections describe necessary adjustments to the termination check for functions (section 3.5.2) so that it is also applicable for methods.

4.1.1 Fields

Because expressions can only read the value of a field, it can be assumed that in Viper, a function execution does not change the field value. Statements, on the other hand, can also modify a field value, e.g. by an assignment, hence a value of a field can change in a method execution. Consider the following method (listing 4.2), which calls itself recursively as long as the value of $r.val$ is non-negative. The method terminates because it decreases $r.val$ before each call and terminates if the field value is negative. Using the $r.val$ as the variant is therefore reasonable.

```
1  field val: Int
2
3  method m(r: Ref)
4      requires acc(r.val)
5      ensures decreases(r.val)
6  {
7      if (r.val >= 0)
8      {
9          r.val := r.val - 1
10         m(r)
11     }
12 }
```

Listing 4.2: Recursive Method with Field as Variant

Also here the termination check is added just before the recursive call: However, this assertion would fail because $r.val < r.val$ can not be satisfied. The problem is that the variant of the caller changed and the new value is used instead of the one from the beginning of the method. In our example the value decreased by one just before the termination check.

```
...
r.val := r.val - 1
assert decreasing(r.val, r.val) && bounded(r.val)
m(r)
...
```

The solution that we implemented is to use `old` expressions whenever the caller's variant is used. `old` expressions make it possible to refer to the value of a field at the beginning of the method, which is what we need. The termination check for listing 4.2 therefore has to be changed to the following:

```
...
r.val := r.val - 1
assert decreasing(r.val, old(r.val)) && bounded(old(r.val))
m(r)
...
```

Now the assertion successfully verifies.

4.1.2 Predicates

As described in section 3.5.3, predicates are used to specify recursive data structures and can also be used as variants to prove termination of functions. The same approach should also be used for proving termination of methods. However, as already shown in section 4.1.1, statements in a method body can change the program state, which causes new challenges for termination proofs. This also affects the use of predicates as variants.

Consider the example in listing 4.3. The method `append` traverses the predicate `list` recursively. When the end of the data structure is reached, i.e. when the `this.next` is null, a new element is appended to the list and the method calls itself recursively with the new element as its argument, which is the cause of the method's non-termination.

Listing 4.4 shows the proof method of `append` as generated by the plugin, enhanced with some comments. Suppose the predicate instance representation function `pred_list` were defined as described in section 3.5.3 (shown in listing 4.5). The termination check in line 34 would unsoundly succeeds even though the method does not terminate. The problem arises because the predicate instance representation function `pred_list` is defined as a domain function. In Viper, domain functions, unlike regular functions, are always heap-independent. Thus, `pred_list` defined as domain function were heap-independent and its result would only depend on the arguments. Therefore, the variables `list0` and `list3` were unsoundly considered to be equal, even though they represent possibly different predicate instances. And since the

```
1  field next: Ref
2
3  predicate list(this: Ref) {
4      acc(this.next) &&
5      (this.next != null ==> list(this.next))
6  }
7
8  method append(this: Ref)
9      requires list(this)
10     ensures  list(this)
11     ensures decreases(list(this))
12 {
13     unfold list(this)
14     if (this.next == null) {
15         var n: Ref
16
17         n := new(next)
18         n.next := null
19         this.next := n
20         fold list(n)
21     }
22     fold list(this)
23
24     unfold list(this)
25     append(this.next)
26     fold list(this)
27 }
```

Listing 4.3: non-nerminating method append

condition in line 27 is always true, `list4` and `list5` are considered to be equal and the termination check in line 34 would unsoundly succeed.

The solution implemented for this problem is to define the predicate instance representation functions as abstract functions instead of domain functions. For the given example listing 4.6 shows the `pred_list` function as it is defined by the plugin. The `wildcard` literal represents some unspecified positive amount of permission. This is required to obtain the predicate the instance `list(this)`. Because the field `this.next` in the proof method in listing 4.4 may be different when `list0` and `list3` are assigned in line 6 and line 25, respectively, they are not guaranteed to be equals. Thus, the termination check correctly fails.

```
method appendP(this: Ref)
    requires list(this)
    ensures  list(this)
{
    // initial predicate instance
    var list0: Predicate_Instance := pred_list(this)
    // unfolded predicate instance
    var list1: Predicate_Instance := pred_list(this)
    unfold list(this)
    if (this.next != null) {
        // nested predicate instance
        var list2: Predicate_Instance := pred_list(this.next)
        inhale nested(list2, list1)
    }

    if (this.next == null) {
        var n: Ref
        n := new(next)
        n.next := null
        this.next := n
        fold list(n)
    }
    fold list(this)
    // unfolded predicate instance
    var list3: Predicate_Instance := pred_list(this)
    unfold list(this)
    if (this.next != null) {
        // nested predicate instance
        var list4: Predicate_Instance := pred_list(this.next)
        inhale nested(list4, list3)
    }
    // predicate instance for termination check
    var list5: Predicate_Instance := pred_list(this.next)
    assert nested(list5, list0) // termination check
    append(this.next)
    fold list(this)
}
```

Listing 4.4: termination proof method for method append (listing 4.3)

```
domain PredicateInstance{
    function pred_list(x: Ref): PredicateInstance
}
```

Listing 4.5: wrong predicate instance representation function for predicate `list`

```
    domain Predicate_Instance{}

    function pred_list(this: Ref): Predicate_Instance
        requires acc(list(this), wildcard)
```

Listing 4.6: new predicate instance representation function for predicate `list`

4.2 FIT for Methods

The FIT transformation presented in section 2.1.3 has been shown to be advantageous for proving termination of functions. The same approach could theoretically also be applied to recursive methods.

However, unlike functions, methods are treated modularly in Viper. Which means that when a method is called, only the postcondition of it is assumed, independently of the methods implementation. Therefore, a user assumes that the verification of a method does not rely on the implementation of another method. By inlining recursive calls, what FIT would do, this assumption would be violated. Because of that we decided not to implement method termination proofs using FIT.

4.3 Termination of Loop

Loops are another potential source of non-termination in Viper. To prove termination of a loop, the same approach as the one applied to methods could be used. A termination check at the end of the loop could verify that the value of a variant at the end of the loop is smaller than the value at the beginning, with respect to a well-founded order. This would imply that the variant decreases in each iteration a finite number of times and therefore the loop must terminate.

In Viper, a loop inside a method can be created in two ways: with while loops or with goto statements. Because the goto statements create loops which are not trivially recognizable in an AST, Viper creates a control flow graph of the method. In the control flow graph each entry and exit of a loop is marked. This information would be crucial to place the termination check for a loop correctly. With the current Viper plugin system, however, is it not possible to access that control flow graph. Because of that, any implementation of termination proofs

for loops would also require changes to the Viper framework. Therefore, it was not implemented as part of this thesis.

Chapter 5

Standard Import

As mentioned in section 3.5.2, the termination plugin offers files containing axioms that define well-founded orders for all Viper built-in types and users should be able to use the axioms by importing the files into their programs. However, how such files are exactly provided by Viper and accessed by the user was not defined. Vipers normal import feature expects either an absolute or a relative path to a file, which has two significant limitations: users need the source of the termination plugin to import provided files, and the import path depends on where the plugin is located. Therefore, as part of this thesis we designed and implemented a new, additional import feature for Viper.

5.1 The Import Feature

In this section we introduce the syntax for the new import feature. Inspired by the C file inclusion feature, the following two import possibilities are now offered by Viper:

```
import "path/to/file"
```

and

```
import <path/to/file>
```

The first one, which was the already implemented one and hereinafter referred to as local import, does not change its semantics and continuous to be uses to include files with a relative or absolute path. The second one, hereinafter referred to as *standard import*, is used to import files provided by Viper. The file path enclosed in `<` and `>` is interpreted as a path to a file relative to the standard import directory. The provided files can either be directly inside that directory or in a sub-folder of it.

As an example, to import the axioms defining a well-founded order over the Viper built-in type `Int` when using the termination plugin, the following stan-

dard import can be used:

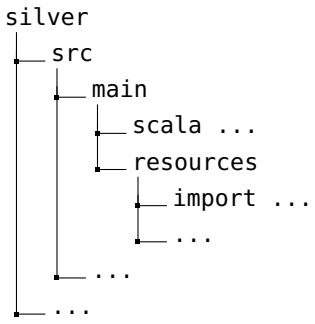
```
import <decreases/int_decreases.sil>
```

5.2 Providing Files

To provide files through the standard import mechanism, they must be placed onto the JVM classpath inside a directory named `import`. We show three different ways how this can be achieved.

In Viper

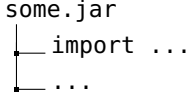
Viper developers can find the directory named `import` in the resources of the Silver project. The Silver project structure is shown in the following diagram:



By packaging the project with sbt everything, in the `recourses` directory is placed in the root of the JAR file, including the directory `import` containing the provided files.

As JAR

Files can also be provided by placing them in a directory named `import` inside JAR files which are added to the classpath of the Viper execution. The following shows a JAR structure containing a directory named `import`:



A JAR file can be added to the classpath of a JVM by using the `cp` flag:

```
java [...] -cp "/path/to/some.jar" [...]
```

As Directory

The easiest way for a Viper user to extend the standard import library with new files is by adding a directory, which contains a directory called `import`, to the classpath of the Viper execution. Any file inside that `import` directory can then be accessed through the standard import. The following shows a directory structure containing a directory named `import`:

```
directory
├── import ...
└── ...
```

A directory can be added to the calsspath of a JVM by using the `cp` flag:

```
java [...] -cp "/path/to/directory" [...]
```

Chapter 6

Conclusion

The main goal of this thesis was to develop and implement a sound, but still reasonable complete approach to verify termination of recursive functions in Viper. Therefore, we looked into two different termination proof approaches, the variant and the transition invariant termination proof approach, and showed how they can be applied to Viper functions. The variant termination proof approach, which is also used by other deductive verifiers, has the advantage that it can be applied modularly to functions, such that the proofs for each functions are kept small. However, we showed mutually recursive functions for which the modular approach does not allow an intuitive choice of the variant. For this we presented a transformation (FIT), which inlines recursive calls in the function's body, and allows in some cases a more intuitive choice of the variant to prove termination. With the transition invariant termination proof approach a less known approach was shown. The approach allows, on the one hand, to use specifications as simple as in the variant approach, but on the other hand, also allows to express more complex specifications.

We implemented the variant termination proof approach with the optional feature to use FIT. Additionally, because the implementation automatically uses the parameters as variants, if none are explicitly defined, the annotation overhead was reduced, which was another extension goal of this thesis. Further, the termination proof support was extended to recursive methods, which was another goal of this thesis. All termination proofs were implemented in a Viper plugin and use a Viper to Viper transformation to encode the proofs. The plugin can now be used by Viper users, which want to verify termination of functions or methods in their Viper programs. For functions, users can use the standard variant termination proof or the variant termination proof with FIT.

A further goal of the thesis was to implement a new import mechanism for the users to import files provided by Viper. This was accomplished by implement-

ing the standard import mechanism. Viper developers can now easily provide Viper programs, which are often required, to the users. This mechanism is also used by the termination plugin to provide pre-defined well-founded orders for built-in Viper types.

6.1 Future Work

In this section we present briefly possible future work.

Implementation of Transition Invariant Termination Proof

We presented in section 2.2 a proof encoding for the transition invariant termination proof approach. The encoding could, in a future project, be implemented and added to the termination plugin. This would make it possible to further explore the advantages or disadvantages of this rather less known approach in deductive program verification.

Furthermore, in this thesis, we only discussed the transition invariant termination proof approach applied to functions. A further goal would be to consider applying the approach also to recursive methods and loops.

Termination Proof Support for Loops

In section 4.3 we discussed the difficulties of implementing termination proof for loops in Viper. As already mentioned this would be another possible future project to extend the termination proof support in Viper.

Avoid Duplicate Errors

In section 2.1 we presented the encoding of the variant termination proof approach, which uses a proof method. Because the method contains several structures which are also contained in the function, such as conditions or unfolds, an error in such a structure would be issued twice. Once in the function and once in the proof method. Since the method is generated by the termination plugin the user is unaware of it and could be confused by the occurrence of a second error. A solution would be to filter all the errors from the proof method which are not termination proof related.

Termination Proof Support for a Viper Front-End

In a future project, the termination proof support could be extended to a Viper front-end, such as the Python front-end. The front-end could then directly use the designated AST nodes for the decreases clause, which were presented in section 3.4.

Appendix A

Appendix

A.1 Variant Termination Proof

p7 Function

As mentioned in section 2.2.4, we were not able to find a variant for a termination proof of the program `p7` without changing the function's definition. However, after extending `p7`'s signature with so-called starting values of x and y, and extending `p7`'s specification with upper and under bounds of x and y based on the starting values, we were able to find a variant termination proof. The following two listings show the modified function and its by the termination plugin generated proof method.

```
1  function p7(x: Int, y: Int, x0: Int, y0: Int, i: Int): Int
2      requires 0 <= i
3      requires y == y0 - i
4      requires y <= y0
5      requires x - x0 <= (y0 - y) * y0
6      requires x <= x0 + y0 * y0
7      requires 0 <= x ==> -2 * (x0 + y0 * y0) <= y
8  {
9      (x > 0 ? p7(x + y, y - 1, x0, y0, i + 1) : x + y)
10 }
```

```
1  method p7_termination_proof(x: Int, y: Int, x0: Int, y0: Int, i: Int)
2      requires 0 <= i
3      requires y == y0 - i
4      requires y <= y0
5      requires x - x0 <= (y0 - y) * y0
6      requires x <= x0 + y0 * y0
7      requires 0 <= x ==> -2 * (x0 + y0 * y0) <= y
8  {
9
10     if (x > 0) {
11       assert decreasing(y - 1 + 2 * (x0 + y0 * y0), old(y + 2 * (x0 + y0 * y0)))
12           && bounded(old(y + 2 * (x0 + y0 * y0)))
13     } else {
14     }
15 }
```

Bibliography

[1] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, Dec 1928.

[2] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[3] Richard L. Ford and K. Rustan M. Leino. Dafny Reference Manual. http://www.divms.uiowa.edu/~tinelli/classes/181/Papers/dafny-reference.pdf. Accessed: 2019-05-11.

[4] Simon Fritsche. A Framework for Bidirectional Program Transformations. Master's thesis, ETH Zurich, 2017.

[5] William Gasarch. *Proving Programs Terminate Using Well-Founded Orderings, Ramsey's Theorem, and Matrices*, volume 97, pages 147–200. 12 2015.

[6] Patrick Gruntz. Checking Termination of Abstraction Functions. Bachelor's thesis, ETH Zurich, 2017.

[7] Zohar Manna and John McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5:24, 10 1969.

[8] Severin Meier. Verification of Information Flow Security for Python Programs. Master's thesis, ETH Zurich, 2018.

[9] 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.

[10] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, pages 68–83, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] Benjamin Schmid. Abstract Read Permission Support for an Automatic Python Verifier. Bachelor's thesis, ETH Zurich, 2017.

[13] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

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):

Tool Support for Termination Proofs

Authored by (in block letters):

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

Name(s):	**First name(s):**
Streun	Fabio

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, 11. May 2019	F. Streun

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