Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Tool Support for Termination Proofs

Bachelor's Thesis - Description

Fabio Streun

November 12, 2018

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

Department of Computer Science, ETH Zürich

# Introduction

Program verification is used to check correctness of program code and can help detect programming errors. Proving termination of a program, i.e. showing that a program finishes in finite time, is an important but also difficult part of the verification process. Different kinds of verification approaches exist to prove termination of a program.

In deductive software verification, programs are usually annotated with specification (or contracts) to express what the program is supposed to do. A function's specification, for example, usually consists of a precondition and a postcondition. They summarize the expected program state before a function call and after, respectively. In order to reason about termination, it is common to specify a termination measure: an expression whose value is bounded from below and decreases in every loop iteration or recursive invocation.

Consider, for example, the following recursive function which calculates the sum of the first *n* natural numbers.

```
function sumFirst(n: Int): Int
  requires 0 <= n
  decreases n
{
  n == 0 ? 0 : n + sumFirst(n-1)
}
```

The *decreases* clause specifies the termination measure, here *n*. Assuming the precondition (specified with *requires*) holds and thanks to the well-ordering principle of natural numbers one can prove that this function terminates in finitely many steps. Every time *sumFirst* invokes itself recursively, *n* decreases (here exactly by one) until it reaches 0, in which case the recursion stops and the function terminates.

Viper, a verification infrastructure developed at the ETH Zurich [Vip18], offers the possibility to encode a source language with specifications (for example, Rust) into an intermediate verification language, also called Viper, which in turn can be verified by two verifiers.

In a previous project, support for termination proofs for recursive functions was implemented [Gru17]. A Viper to Viper transformation is used to encode the required checks in additional Viper code, also sometimes referred to as proof code. The output then can be normally verified. This approach allowed to extend the functionality of Viper without having to modify the verifiers.

The following sections show some of the remaining open problems regarding termination proofs in Viper.

### Mutually Recursive Functions

To reason about mutually recursive functions, the transformation uses an approach which explores static, "direct" execution paths and substitutes arguments on the way until it encounters a call to an already checked function. However the approach is unsound, as it does not account for all possible actual executions, and therefore can produce a wrong result. Consider the following example of the non-terminating functions $f$ and $g$:

```
function f(x: Int): Int
  requires 0 <= x
  decreases x
{ x == 0 ? g(x+1) : f(x-1) }

function g(x: Int): Int
  requires 0 <= x
  decreases x
{ x == 0 ? f(x+1) : g(x-1) }
```

Calling $f$ with $x = 1$ invokes $f$ recursively with $x = 0$, then $g$ with $x = 1$, then $g$ with $x = 0$ and then again $f$ with $x = 1$, which is the same input as we started with and shows non-termination of the function.

To prove termination of $f$, the current approach explores the following paths, with $a$ as an arbitrary integer: $f(a) \rightarrow f(a-1)$ if $a \neq 0$ on which the termination measure decreases. The path $f(a) \rightarrow g(a+1)$ 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 call, $g(a+1-1)$, is unsoundly considered harmless, since $g$ is separately checked for termination (which succeeds equally unsoundly).

In summary: the current approach works for self-recursive functions, but not for mutually recursive ones.

### Heap-Dependent Functions as Termination Measure

A heap-dependent function used in the decreases-clause could depend on the function currently checked for termination. This would create termination checks which mutually depend on each other, which is not trivially solved. Because the current approach does already not support functions that are mutually recursive through their bodies it also does not support functions that are mutually recursive through their contracts.

### Public Functions Used by User and Viper Back-End

To check if termination measures decrease and are bounded, the Viper-to-Viper transformation and the Viper back-ends use the functions *decreasing* and *bounded*, which can be axiomatised by the user. To make this user-provided axioms type-check, users are currently also expected to declare these functions. This expectation, as well as the required function signatures, is currently not apparent to users, which can be confusing and makes the whole setup brittle and error prone.

### Import Files Provided by Viper

For Viper's built-in types, such as integers and sequences, suitable declarations and axiomatisations of *decreasing* and *bounded* can be imported from Viper files that are part of Viper's code base. However, Viper's import feature currently expects either a relative or an absolute path to a file, which has two significant limitations: users need the sources of Viper to import such standard definitions, and the import paths depend on where the current and the imported files are located. Other programming languages, e.g. C, offer a dedicated, simplified way to include such standard library functions (*include "../my/file.h"* vs. *include <stdlib.h>*). The implementation of a similar alternative for importing files will improve the usability of Viper.

### Predicates

Besides functions also predicates, which are parameterized assertions, can be defined recursively [PB05]. In Viper, a predicate instance that has an infinite number of predicate instances folded within it is, by definition, equivalent to false. Using such a predicate as a precondition is semantically equivalent to requiring false. A user defining a predicate wrongly by mistake, such that the predicate only has non-terminating interpretations, could therefore cause methods to verify only due to their false precondition.

## Core Goals

The core goals of this thesis are divided into conceptual goals and implementation goals.

### Conceptual

- Develop and implement a sound, but still reasonably complete approach to verify termination of mutually recursive functions. Dafny, another deductive verifier, is sound but fails with simple mutually recursive functions, which seem easily verifiable (see Dafny Example on the next page).
- Support heap-dependent functions as termination measures.

### Implementation

- Use a Viper to Viper transformation, as in the previous project, to encode the additional required checks and avoid changes to the verifiers. Recently added features, which allow to add plugins [Sch17] and to extend the Viper AST with custom nodes [Mei18], could be used to achieve this goal in an way that is more elegant and easier to maintain than the previous ad-hoc solution.
- Implement an alternative way for users to import files provided by Viper.
- Design and implement a less error prone approach for declaring and using functions that are not part of the Viper core language, but which nevertheless are expected to have a particular signature, by components at different steps of the verification infrastructure.
- Improve termination-related error reporting to provide more helpful messages when verification fails.

## Extension Goals

- Extend the support for termination proofs to loops and methods.
- Support termination proofs also for predicates and warn the user of possible definitions which are equal to false.
- Implement termination checks for a Viper front-end (e.g. Python, Rust, Voila)
- Decrease annotation overhead by automatically trying to infer termination measures e.g. by simple syntactic heuristics.

## Dafny Example

Following simple example of mutually recursive functions fails to be verfied by Dafny (version: 2.2.0.10923).

```
function f(x: int): int
  requires 0 <= x
  decreases x
 {
   if x == 0 then 0 else g(x-1)
 }

function g(y: int): int
  requires 0 <= y
  decreases y
 {
   f(y)
 }
```

# Bibliography

[Gru17] Patrick Gruntz. Checking Termination of Abstraction Functions. Bachelor's thesis, ETH Zurich, 2017.

[Mei18] Severin Meier. Verification of Information Flow Security for Python Programs. Master's thesis, ETH Zurich, 2018.

[PB05] M. Parkinson and G. Bierman. Separation logic and abstraction. In POPL, pages 247-158, 2005.

[Sch17] Benjamin Schmid. Abstract Read Permission Support for an Automatic Python Verifier. Bachelor's thesis, ETH Zurich, 2017.

[Vip18] Viper. http://viper.ethz.ch, 2018. [Accessed 18.11.2018].