Checking Termination of Abstraction Functions

Bachelor's Thesis Project Description

Patrick Gruntz
Supervisor: Malte Schwerhoff

ETH Zürich

March 28, 2017

Motivation

In deductive software verification, the programs to verify are usually annotated with specifications (or contracts) in order to express what the program is supposed to do. A method specification, for example, usually consists of a precondition and a postcondition, which summarise the effects of an invocation of that method.
It is common practice to use abstraction functions in such specifications in order to abstract over implementational details. For example:

```
function sum(xs:Seq[Int]):Int
{
    |xs| == 0 ? 0 : xs[0]+sum(xs[1..])
}
```

This function can be used in the postcondition of an append method to state that the sum of all elements of the resulting sequence is the sum of the input sequence plus the newly appended value.

Such specifications, however, are only meaningful if all abstraction functions are well-defined[1], which, among other things, means that all functions terminate, and thus actually represent values. Proving termination of mutually recursive and heap-dependent abstraction functions is therefore an important task for an automated verifier.

The Viper verification infrastructure[2] developed at ETH Zurich is a framework where programs plus specifications written in a source language (for example Java) can be encoded as programs in the intermediate verification language Viper. The translated program can then be verified by two different verifiers. The key property of such a translation is that the source program is correct (with respect to its specifications) if its encoding in Viper verifies. Each verifier therefore has to check that user-provided abstraction functions are well-defined.

The termination checks of abstraction functions are currently not implemented at all. One goal therefore is to define proof obligations in Viper e.g. by following approaches used in other verifiers such as Dafny[3] or Chalice[4]. An implementation of the checks in each verifier separately wouldn't be a good option considering that the necessary efforts duplicate and that each implementation would have to be adapted if the rules for checking termination change. Since Viper is an expressive language, a Viper-to-Viper transformation can be implemented that encodes, for a given Viper program, the termination-related proof obligations as another Viper program. The abstraction functions in the original Viper program are then guaranteed to terminate if the generated Viper program verifies.

In order to show termination of an abstraction function, Viper can be used to prove that a given variant is monotonic and bounded. In the example above, the length of the sequence (i.e. |xs|) would be such a variant. The program would be extended by the following code, which then can be proven by Viper:

```
function sum(xs: Seq[Int]): Int
//decreases |xs|
{ |xs| == 0 ? 0 : xs[0] + sum(xs[1..]) }

method sum_termination_proof(xs: Seq[Int]) {
  if (|xs| == 0) {
    // no recursive call; function terminates
  } else {
    assert |xs[1..]| < |xs|
  }
}
```

Another core goal is to implement termination measures to show that the amount of heap locations accessible to the function decreases with every function call. Thus, they can be used to prove termination of heap-manipulating functions. For example:

```
field val: Int
field next: Ref
predicate list(x: Ref) {
  x != null ==> acc(x.val) && acc(x.next) && list(x.next)
}
function list_sum(x: Ref): Int
  requires list(x)
{
  unfolding list(x) in x == null ? 0 : x.val + list_sum(x.next)
}
```

In this example the termination of the function `list_sum` can be shown by proving that the amount of accessible locations in each function is getting reduced. The recursive call is only being able to access the list's tail starting at `x.next`, which is only a part of the whole accessible list. The amount of accessible locations on the heap will therefore be reduced with every function call. Since the definition of the predicate `list` implies that lists are finite and acyclic, the amount of accessible locations will decrease and the function will terminate.

One of the challenges that we expect to arise in such a transformation is to identify which references, fields and predicate instances are relevant for the generation of the Viper code.

Core Goals

- Implement a Viper-to-Viper transformation, such that the generated Viper program verifies that a given value decreases. This requires:
  - A decrease relation (strict partial order) $\ll$: $T \times T \rightarrow Boolean$ for all relevant Viper types T (following an approach taken by Dafny). The relation should be applicable for single expressions ($e1 \ll e2$) and tuples of expressions.
  - An implementation of a `decrease` annotation, such that the user can specify a variant i.e. an expression, that becomes strictly smaller each time a recursive function or method is called.
- Integrate heap/permission based termination measures into above relation.
- Provide a mechanism to ensure that the definitional axioms for a function are availabe only once the function is known to be well-defined

- Add a mechanism for translating error messages for the generated Viper program to meaningful, termination-related error messages.
- Evaluate the implementation with known termination problems to demonstrate performance and expressiveness of the chosen approach

Possible Extensions

- Integrate termination proofs for predicates
- Adapt the approach developed for functions and allow proving the termination of loops
- Extend the heap/permission-based measures to also cover examples using fractional permissions.
- Reduce annotation overhead by:
  - Designing and implementing heuristics for automatically inferring decreases clauses (as e.g. done in Dafny)
  - Building and analyzing the static call graph and detecting cycles

References

[1] A. Rudich and A. Darvas and Müller, P.: **Checking Well-Formedness of Pure-Method Specifications**, FM 2008.

[2] P. Müller and M. Schwerhoff and A. J. Summers: **Viper: A Verification Infrastructure for Permission-Based Reasoning**, VMCAI 2016.

[3] K. Rustan M. Leino: **Dafny: An Automatic Program Verifier for Functional Correctness**, 2010.

[4] K. R. M. Leino and P. Müller and J. Smans: **Verification of Concurrent Programs with Chalice**, Foundations of Security Analysis and Design V 2009.