Towards better Function Axiomatization in a Symbolic-execution-based Verifier

Bachelor thesis project description
ETH Zürich

Mauro Bringolf
supervised by Dr. Malte Schwerhoff and Dr. Alexander J. Summers

April 12, 2019

1 Introduction

Software is inherently complex and difficult to get right, and software verification aims to provide a level of trust beyond what well-established techniques such as testing and code reviews can give. To this end, desired source code properties such as absence of memory leaks or functional correctness are expressed as formal specifications and then proven to be satisfied by the implementation. These proofs are typically automated by expressing the correspondence between source code and specification as an SMT problem for which sophisticated solvers exist.

Many verifiers take a layered approach by not directly translating problems down to the SMT level, but to an intermediate verification language such as Boogie [1] or Why [2] instead. For imperative languages, one particular challenge is handling heap manipulation. The heap creates complex dependencies between program statements and expressions which need to be taken into account when reasoning about them. One approach that has proven to be effective for the book-keeping of these heap dependencies are permissions. Roughly speaking, each statement or expression requires certain permissions to access a heap location. This enables modular reasoning, since statements without sufficient permission to some heap location cannot depend on it or change its value. In particular, a function's specification includes all permissions its body requires and thus a caller can reason about it without knowing its implementation.

The Viper infrastructure [3] provides an intermediate verification language which includes permissions natively. By encoding into Viper, front-end verifiers can use the notion of permissions to encode various verification techniques and program semantics while the translation to SMT is handled by Viper. This translation can be done by two independent back-ends: Silicon, based on symbolic execution, and Carbon, based on verification condition generation. Our work is concerned with Silicon's SMT encoding of Viper functions.

2 Reasoning about heap-dependent functions

In Viper, assertions express properties of the current program state, including the permissions available for heap accesses. Pure assertions such as `x.f > 0` constrain program values and accessibility predicates such as `acc(x.f)` represent permissions. This state is modified by the Viper operations `inhale` and `exhale`. Informally, `inhale A` assumes all properties expressed in `A` and adds all permissions denoted in `A`. Symmetrically, the operation `exhale A` checks that all properties in `A` are satisfied and all permissions in `A` are currently held, before removing the latter.

The function `f` in listing 1 depends only on the value of `x.g` as indicated by `f`'s precondition. The method `m` requires `f(x) == 0` and exclusive permissions to `x.g` and `y.g`. The semantics of accessibility predicates imply that in this case `x` and `y` cannot be aliases. As a result of this, Viper is able to frame the value `f(x)` across the assignment of `y.g` and successfully prove the assertion in line 10.

```
 1  field g : Int
 2
 3  function f(x: Ref) : Int
 4  requires acc(x.g)
 5  {
 6    ...
 7  }
 8
 9  method foo(x:Ref, y:Ref)
10  requires acc(x.g) && f(x) == 0 && acc(y.g)
11  {
12      y.g := 1
13      assert f(x) == 0 // Verifies due to framing
14  }
```

Listing 1: The value of `f(x)` is framed across the assignment `y.g := 1`.

In Silicon, exhaling an assertion builds up a snapshot of the relevant heap locations and their symbolic values. This snapshot is used when inhaling preconditions for a function application or predicate instance to capture its heap dependencies and ultimately enables Silicon to frame expressions.

2.1 Function application

A function application is translated by Silicon into an uninterpreted function application by symbolically evaluating the arguments and adding a snapshot as additional argument. In order to give meaning to the uninterpreted function, Silicon axiomatizes functions by mapping heap-dependent expressions in the body to parts of the snapshot. This mapping can then be added to the assumptions once and implicitly gives meaning to all symbolic function applications.

3 Snapshots in Silicon

3.1 Current representation

Inspired by previous work [4], Silicon uses binary trees to represent heap snapshots. As illustrated in figure 1, the tree structure is determined by the separating conjunctions from which assertions are built and each leaf represent either a heap value (from an accessibility predicate) or nothing (from a pure assertion). The main benefit of binary trees is that they have a small axiomatization in SMT-LIB and are easily created from assertions, since these are already of tree structure.

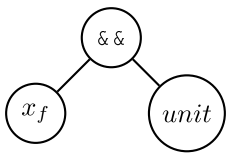

Figure 1: Silicon's binary tree representation of a snapshot for the assertion `acc(x.f) && x.f > 0` where x_f stands for the symbolic value of `x.f` in the current heap and *unit* for an empty snapshot.

3.2 Problems

There is a price to pay for the simplicity of binary trees when it comes to function axiomatization. As mentioned above, Silicon computes a mapping from heap accesses in the function body to corresponding snapshot components. The problem with using binary trees is that branching and the notorious problem of aliasing need to be taken into account by this algorithm. Consider the following assertion `acc(x.f) && acc(y.f) && b ? z == x : z == y`. If this is the precondition of a function and a field access `z.f` appears in the body, then its position in the snapshot depends on `b`. Thus Silicon needs to keep track of branch conditions and aliases which is a laborious process and results in complex function axiomatization algorithms.

4 Solution approach

The primary design goal is to simplify the function axiomatization algorithm and the resulting definitional axioms using a more powerful snapshot representation. In particular, such a representation should defer aliasing questions to the underlying SMT solver. To this end, snapshots could behave like a mapping from syntactic field accesses such as `x.f` to symbolic values of the corresponding field and an appropriate axiomatization of such a mapping structure on the SMT layer would then take care of aliasing questions.

Such a mapping structure describing a partial heap could solve some simple cases of the aliasing problems mentioned above. However, the general situation is more intricate and will have to be explored carefully. Some of the anticipated difficulties are illustrated in listing 2. Let us first consider the recursive instance `List(x.next)`. A predicate instance will always contain a finite (but potentially statically unbounded) number of recursive instances which naturally maps to structural heap snapshots. The partial heap corresponding to the recursive instance (as required when unfolding an instance for example) can be identified by matching the structure of the assertion with the snapshot. In contrast, it is not clear how snapshots represented as mapping structures could allow this step in a similarly effective manner. Of course, associating the complete snapshot of `List(x)` to `List(x.next)` would be sound but a coarse over-approximation and miss the main idea of partial heaps in the first place: local reasoning. A similar question arises for the recursive call `at(x.next, i-1)` in the function's body.

```
field val: Int
field next: Ref

predicate List(x: Ref) {
  acc(x.val) &&
  acc(x.next) &&
  (x.next != null ==> List(x.next))
}

function at(x: Ref, i: Int): Int
  requires List(x) && i >= 0
{
  unfolding List(x) in
    i == 0 || x.next == null ? x.val : at(x.next, i-1)
}
```

Listing 2: The predicate `List` defines the standard notion of a linked list. The function `at` returns the element at a certain position (for valid positions). Both are naturally defined with recursion.

So far we have not considered quantified permissions. For reasons of brevity, we mention only that they require a more powerful snapshot representation which Silicon already implements. The solution approach we propose will most likely reduce the differences between Silicon's treatment of quantified and non-quantified permissions and thus further reduce its complexity.

As a result, we expect a trade-off with respect to function axiomatization between the current structural snapshot representation which gives Silicon more work to do, and a semantic representation which offloads some of the effort to the underlying SMT solver. Additionally, the semantic representation might have a more costly axiomatization in SMT-LIB. To decide whether or not the new representation is favorable over the current one we will consider its impact on overall performance, algorithmic complexity and completeness of Silicon and Silicon's treatment of quantified and non-quantified permissions.

5 Core goals

5.1 Non-quantified permissions

- Investigate how a less structural and more semantic snapshot representation could look. Consider if quantified permission snapshots can be used as a starting point.
- Conceptually adapt Silicon's function axiomatization algorithm using the new representation and evaluate how this would affect its complexity
- Verify that the new representation allows inhale-exhale assertions in predicate bodies (a current, known limitation of Silicon induced directly by the structural snapshot representation)

5.2 Quantified permissions

- Benchmark the experimental case when non-quantified permission snapshots are disabled completely. This tests if Silicon's (from a logical perspective redundant) distinction between quantified and non-quantified permissions pays off performance-wise.
- Investigate if potential function axiomatization improvements from a new non-quantified permission snapshot representation can be obtained in a similar way for quantified permissions.
- Investigate how the combination of quantified and non-quantified permissions is affected

5.3 Implementation

- Based on conceptual results of previous sections, choose and implement a new snapshot representation and adapt affected operations (predicate folding and unfolding for example)
- Benchmark against current implementation

6 Extension goals

- Extend implementation to quantified permissions
- Adapt new snapshot design to magic wands, such that the currently existing unsoundness with snapshots for partial magic wands is avoided.
- Study and use ideas from Carbon's treatment of functions
- Describe the effect on Silicon's completeness (if any)

References

[1] M. Barnett et al, *Boogie: A modular reusable verifier for object-oriented programs*, FMCO. Vol. 4111. LNCS. Springer, 2005.

[2] F. Bobot et al, *Why3: Shepherd your herd of provers*, Boogie, 2011.

[3] P. Müller and M. Schwerhoff and A. J. Summers, *Viper: A Verification Infrastructure for Permission-Based Reasoning*, Verification, Model Checking, and Abstract Interpretation (VMCAI), 2016.

[4] J. Smans, B. Jacobs and F. Piessens, *Heap-dependent expressions in separation logic*, FMOODS/FORTE. Vol. 6117. LNCS. Springer, 2010.