Specification and Automated Reasoning for Datastructure Comprehensions

Bachelor's Thesis Project Description

Tierry Hörmann

Supervised by Alexander J. Summers and Arshavir Ter-Gabrielyan

1 Introduction

1.1 Program Verification

Program verification is the task of formally describing a program or parts of a program (e.g. with axiomatic semantics) and proving certain properties of the program with this formal representation. As an example we might consider a (correct) implementation of Quicksort in Haskell. It should then be possible to prove that the implementation indeed returns a sorted list which holds the same elements as the provided list. The process of translating the implementation into a mathematical model and proving that it really outputs a sorted version of the provided list is then called program verification.

While it is usually possible to do program verification by hand, this is only practical for small and simple programs. However specific tools can drastically improve the practicality of program verification. Such tools should be able to automatize obvious deductions and maybe even automatize model generation. Ideally it should be possible to write a program in an arbitrary programming language and state what properties the input and output should fulfill, and a tool can then possibly, with a few additional annotations, prove that the output indeed fulfills the provided properties when executing the program on valid inputs.

A current approach for automatizing program verification is to design an intermediate verification language with which a program can be formally modeled and afterwards automatically verified. The way in which a program is formally modeled in an intermediate verification language can have a huge influence on the success of verification.

1.2 Viper

Viper is a verification infrastructure designed and developed at ETH Zürich (viper.ethz.ch). It uses an intermediate verification language, also called Viper, which can automatically be verified with verifiers provided by the infrastructure. Viper supports modeling of stateful and even parallel computations with a permission based approach for controlling heap manipulations, in the sense that permission to a heap location must first be acquired before being able to access (by reading or writing) the location.

1.3 Datastructure Comprehensions

Realistic programs make extensive use of heap manipulations via datastructures. Hence to verify realistic programs, it is also necessary to provide support for dynamically sized datastructures, such that it becomes possible to prove certain properties of a datastructure, after it was modified in some (arbitrary) ways. For this we first need to find a way to describe the content of the datastrucutre. Recursive definitions in functional programming is one approach of doing this. For general programs a recursive approach however gives rise to at least three major problems:

1. Many datastructures cannot be formulated recursively in a practical manner.

2. Programs which do not traverse or modify the datastructure according to the recursive definition would be hard to verify.
3. Proofs by induction are often not automatic in practice.

Another approach is to describe the content of a datastructure via operations over the datastructure, such as the sum over all elements, the maximum value, the minimum value, etc. Such operations are called *datastructure comprehensions* and the goal would be to automatically prove the result of such a comprehension after an arbitrary operation was performed on the datastructure.

2 Work

To illustrate the probem we are trying to solve, consider a program, which uses some kind of datastructure with values on the heap, e.g. an array. Now let's say, we want to prove a condition on the maximal value (our comprehension) of the graph after the program was executed. Imagine we know that a certain node is contained in the graph and we increase the value of this node in the program. The maximal value of the graph might then have changed with this increase, without adding or removing an element from the graph. This illustrates, that since changes to the heap can be made, without touching the graph, we cannot simply prove our condition by looking at what elements were added to the graph or removed. Hence we need to extend our vision on certain instructions, which manipulate the heap. What we want to get out of this project, is effectively a feature for Viper, with which we can automatically prove such conditions.

Currently there is no feature in Viper, with which we can model datastrucutre comprehensions, at least not in a practical way. Consider the following example for an approach of a sum comprehension over a set of references in Viper:

```
function sum(set: Set[Ref]): Int
    // permissions
    requires forall n:Ref :: {n.val} n in set ==> acc(n.val, 1/2)
    // definition of the sum itself
    ensures forall n:Ref :: {n.val} n in set ==>
        result == sum(set setminus Set(n)) + n.val
    ensures |set| == 0 ==> result == 0
```

If we take a look at what we can prove with this comprehension, we will see, that it is only possible to prove how the sum changed directly after a single field update, if right before the update the sum was mentioned in a statement. So if we want to prove how the sum changes after multiple different field updates, the code would get flooded with assert statements, which is obviously neither practical, nor really automatic in any sense. Also the above specification only handles sets of references and pointwise updates of a specific field, so it is in addition not in any way general. For different use-cases this approach might not even work at all.

The goal of this project is therefore to provide general support for datastructure comprehensions in Viper, which should solve the above issues. This means, that we should end up with a way of describing a comprehension and then, more importantly, automatically prove how the comprehension changes when arbitrary changes were made to the program heap.

The problem for this lies mainly in the area of quantifiers in program verification, because if we don't want to impose an ordering for a traversal over a datastructure, we need to quantify over it to describe a comprehension. To illustrate this, imagine we have an array on the heap, where we try to describe a sum comprehension on. To describe such a comprehension with a definition, we might come accross the idea from functional programming to define the sum recursively as the first element plus the sum over the array without the first element, according to this pseudocode

```
function sum(a: Array): Int
    len(a) == 0
        ? 0
        : a[0] + sum(a[1...len(a)-1])
```

Now when trying to prove for example, that after the increase of a single element in the array, the sum increased by the same amount, we need to unroll this definition, until we encounter the increased element. Since in general, there is no way to know how often we want to unroll the definition, we won't be able to prove such basic properties.
When we however use quantifiers, similar to the example on sets above, we can achieve that the quantifier gets instantiated for the element we increased, such that we will be able to prove our desired property.

2.1 Core Goals

- First we will want to provide example implementations for certain comprehensions over specific datastructures. With this we hope to gain approaches on how to generalize the design further on. The following list are some example datastructures we want to consider for our specific implementations. Note that we do not handle datastructures such as sequences, since we are not really interested on datastructures which impose a order for traversal.
    - Set of References
    - Multiset of References
    - Array
    - Graph

  Some examples for comprehensions we may want to consider are:
    - Sum
    - Product
    - Minimum
    - Maximum
    - Set of values
    - Equality between the content of a datastructure and the content of an arbitrary multiset
- After creating specific example implementations, we need to test how the implementation performs. We do this by writing methods which take a datastructure of the above types and modify it with certain operations. We will then test whether it is possible to prove how the comprehensions have changed after execution of the method. The following list provides some examples of possible operations:
    - Swap of two values in the datastructure.
    - Adding / removing values to / from the datastructure
    - Modifying single values in an arbitrary fashion.
    - Method calls which take permission to heap locations managed by the datastructure.
    - Combinations of the above
- The next step is then to analyze the knowledge gained from implementing the example comprehensions and creating a general design for arbitrary comprehensions, which can then later on be implemented as a feature into Viper.

2.2 Extension Goals

- From our last core goal we get a general design for comprehensions. We then want to adapt this design in an implementation of a feature for Viper. For this it will be necessary to modify the verifier of Viper. The feature should then be tested in the same fashion as specified in the core goals.

- With such a feature as part of Viper, we would now like to again consider our original examples from the core goals, for which we already implemented specific sorts of comprehensions, and test our general approach on them, to get to a conclusion of how well our result performs in comparison to specific implementations.