WLP-based verification engine

Ferdinand van Walree
3874389
F.R.vanWalree@students.uu.nl

Matthew Swart
5597250
m.a.swart@students.uu.nl

March 27, 2016

1 Introduction

This document gives an overview of the implementation of a WLP-based verification engine. The engine itself is written in Haskell and we make use of the Data.SBV package for proving our WLP. We first give an overview of the implemented components followed by a more in-depth explanation of each component with examples. Our WLP-engine accepts programs written in GCL code. However, we do not do any parsing. Instead, we implemented GCL as an EDSL written in Haskell. To get an impression on how to write GCL in our code, see the following Hoare triple:

```
{b == 1} a := b - 1 {a == 0}
```

The corresponding GCL, that is accepted by our engine, is shown below:

```
plusExample :: Stmt
plusExample =
 var [int "a", int "b"]
    [
      assume (i 0 .== ref "b"),
      ref "a" .= ref "b" `plus` i 1,
      assert (ref "a" .== i 1)
    ]
```

All possible GCL syntactical constructs can be viewed in **GCL.hs**. **Examples.hs** contains a collection of verifiable GCL programs.

2 Implementation overview

This section consist of an overview of all the implemented components. We implemented the following features:

1. Base implementation

2. Simultaneous assignment
3. Array assignments
4. Program call
5. Loop reduction

In the next part of the document, we describe in more detail all of the components listed above. Each section is accompanied by a verified example that shows the implementation of that component.

3 Base implementation

The base implementation consists of the basic structure of our engine that is needed to prove basic GCL programs. The structure of the engine is divided in four components. See figure 1.

3.1 Collection

The syntax to represent GCL is defined in **Grammar.ag**. In **Collection.ag** we collect all the declared and referenced variables, and program definitions. We make a difference between the declaration of a variable and the reference of a variable. When declaring a variable, the type of the variable has to be explicitly stated. This is necessary so that we can differentiate between arrays and integers. We more than once want to know what variables are reference in a given expression, we therefore separately collect all references of these variables. It is possible to declare variables as a program parameter or as a local variable. Variables can be declared as: int "a" or array "a". Collection of program definitions is necessary for when we encounter a program call. Program call is explained in section 6.

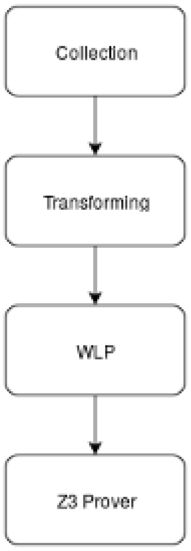

Figure 1: Structure

3.2 Transforming

For a given GCL program, we first have to do some transformations before we can calculate the WLP. The transformation is done in **Transformer.hs**. In this file we introduce fresh variables for whenever it is necessary. We use an integer starting at value 0 that we use to fold through all statements and expressions. Whenever new variables are declared, we append the current integer value to these variables. The integer value is then incremented. **Transformation.hs** also handles simultaneous assignments and program calls, but this is explained later in this document. **Transformation.hs** also contains a function to convert a given expression to prenex normal form. Namely, whenever we see a $\forall x$ or $\exists x$ in an inner expression, we use defined logical rules to convert this to prenex normal form.

3.3 Verification

In **verification.hs** we calculate the WLP. We expect every program to start with an assume. This is the given precondition. Generally we also expect every program to end with an assert, but this is not forced. We have a function called WLP that starts with the post condition True_ and then folds over all statements in a backwards manner. The function has a case for each type of statement. The implementation for each case is implemented as specified in the lecture notes. We do however have two special cases. We differentiate between assigning to a regular variable and to an array that is indexed. We explain more about this in section 5. We also differentiate between a while that is provided with an invariant and a while that does not have an invariant. If it does not, we calculate a fix point. This is explained in section 7. When a while is supplied with an invariant, we first prove that this is a valid invariant using the prover. The result of the prover is given after the WLP has been calculated. However, if it is not a valid invariant, then we do not continue with the invariant as the precondition, instead we assert that the condition of the while is false.

3.4 Prover

The last component of the engine is the prover. This is implemented in **Prover.hs**. We use the prover to verify whether the given precondition implies the WLP. The prover is implemented using the Data.SBV package. We can translate our expressions to Data.SBV defined expressions. Data.SBV lets us declare integer and array variables, universal and existential quantifiers, and lets us prove the validity of implications. Data.SBV itself binds to the Z3-solver, which does the actual verification. If we want to prove the implication of two given expressions, then we first declare all Data.SBV variables and insert these into Haskell defined Maps. Now we can traverse the expression tree and convert each expression to a Data.SBV expression. Ultimately this will be a Predicate (from Data.SBV), which we can prove the validity of. The prover either tells us that the implication is valid or it returns a counter-example. Whenever we want to prove an implication, we first convert the implication to prenex normal form and only then do we prove it. This is necessary, because Data.SBV initializes existential and universal quantifiers at the beginning of a predicate. To ensure that the predicate remains logically correct we have to transform the implication to prenex normal form.

3.5 Example

Here is a small correct example:

```
1 s1 :: Stmt
2 s1 = var [int "x",int "y",int "n"]
3        [ assume (i 0 .< ref "x") ,
4           inv (i 0 .<= ref "x")
5              (while (i 0 .< ref "x") [ref "x" .= ref "x" ‘minus‘ i 1]),
6           ref "y" .= ref "x",
7           assert (ref "y" .== i 0)
8        ]
```

You can try to verify this by writing: ”verify s1”, it will result into QED. However, when we modify the assert at line 7 to **(ref ”y” .== i 1)**, it results in:

```
Falsifiable. Counter-example:
  x0 = 1 :: Integer
```

4 Simultaneous assignment

In order to implement simultaneous assignments we added in **Grammar.ag** and **GCL.hs** the necessary syntax. Next we added in the **Transformer.hs** an extra case in the mkFreshStmt function. We declare new fresh variables for each variable that we are assigning to. We then first store the value of these variables in their corresponding new fresh variable. This allows us to now handle a simultaneous assignment as a normal sequential assignment.

As an example, consider the program below:

```
Var [int "a", int "b"]
    [
       ref "a" .= i 10,
       sim [ref "a",ref "b" ‘sim‘ [i 1, ref "a" ‘plus‘ i 3]
    ]
```

This will be transformed to the following code:

```
ref "a0" .= i 10
ref "a1" .= ref "a0"
ref "b1" := ref "b0"
ref "a0" := i 1
ref "b0" := ref "a1" + i 3
```

Variables are passed by value, which means that the assignment *ref "a0" := i 1* has no effect on the value of *ref "a1"* and therefore also not on the assignment *ref "b0" := ref "a1" + i 3*.

5 Array assignment

This section explains how we implemented array assignments in our engine. The first thing we did was modify the **grammer.ag** to add support for GCL arrays using Haskell. We added the constructor Repby to the Expression. This constructor might be a little bit misleading as it does not actually mean to replace a value by, instead it is simply used to index an array. For example to write the following statement: *a[i] := 1* you need to write this:

```
ref "a" ‘repby‘ ref "i" .= i 1
```

Our WLP function already handles the case when an array is assigned to, but not when a value is assigned to an indexed array. We added a case for this in WLP. However handling substitution for this case is somewhat complex. So we will not explain this in detail, but will explain by example. Given the following case:

```
WLP (a[i] := 0) (a[k] > a[j])
```

We can't directly substitute a[k] or a[j] with 0. Considering we do not know whether i equals j or k, we have to account for all possibilities. We therefore do the following substitution:

```
(i == k && i == j ==> 0 > 0) && (i == k && i != j ==> 0 > a[j]) && (i !=
    k && i == j ==> a[k] > 0) && (i != k && i != j ==> a[k] > a[j])
```

Proving We also have to ensure that our prover can handle arrays. As said before we already know, which variables are of type array. This allows us to create array quantifiers in Data.SBV. We can write and read values from array quantifiers. When we have an expression that contains an array that is indexed, we can read from that corresponding array quantifier using the given index (The index can also be an indexed array). Reading from the array will return us an integer, which is then used to construct the predicate.
Here is an example of the swap program that heavily makes use of array assignments. You can either verify this program or verify swapCall:

```
swap :: Stmt
swap = prog "swap" [int "i", int "j", array "a"] [array "a’"]
     [
       var [int "tmp", int "b", int "c"]
         [
            assume ((ref "a" `repby` ref "i" .== ref "b") .&& (ref "a"
                `repby` ref "j" .== ref "c")),
            ref "tmp" .= ref "a" `repby` ref "i",
            ref "a" `repby` ref "i" .= ref "a" `repby` ref "j",
            ref "a" `repby` ref "j" .= ref "tmp",
            assert ((ref "a" `repby` ref "i" .== ref "c") .&& (ref "a"
                `repby` ref "j" .== ref "b"))
          ],
        ref "a'" .= ref "a"
      ]
```

6 Program call

In this section we explain how we implemented the program call in the WLP Engine. In the **transformer.hs**, we added a case for program calls. Whenever we see a program call, we look up the definition of the program. The definition includes the argument parameters, result parameters and the body of the program. We declare variables for the parameters. The arguments of the program call are then assigned to the argument parameters. Next, the body of the program can be inserted and finally we can assign the result parameters to the return arguments of the program call. We have not made any other changes specifically for program call as we reuse existing WLP constructs. For a black-box call, we simply assume that the program contains an assert and assume and have thus not made any changes to our engine.

As of the time of writing this report, we have been unable to verify sort. Also we are unable to verify mindindCall, unless we also return all arguments of the program call. However, we are determined to fix this in the next couple of days.

```
minind :: Stmt
minind = prog "minind" [int "i", int "N", array "a"] [int "r'", int
    "i'", int "N'", array "a'"]
        [

            var [int "min", int "r", int "j"]
              [
              ref "j" .= ref "i",
              ref "r" .= ref "i",
              ref "min" .= ref "a" `repby` ref "r",
               inv (forall "x" (ref "j" .< ref "N" .&& ref "j" .<= ref
                   "i" .&& ref "j" .<= ref "r" .&&
                              (ref "j" .== ref "i" .==> ref "r" .== ref
                                  "i") .&&
                              (ref "j" .< ref "i" .==> ref "r" .< ref
                                  "i") .&&
                              ref "min" .== ref "a" `repby` ref "r" .&&
                             (ref "j" .<= ref "x" .&& ref "x" .< ref "i"
                                 .==> ref "a" `repby` ref "r" .<= ref
                                 "a" `repby` ref "x")))
                (while (ref "i" .< ref "N")
                  [
                    if_then_else (ref "a" `repby` ref "i" .< ref "min")
                      [
                        ref "min" .= ref "a" `repby` ref "i",
                        ref "r" .= ref "i"
                      ]
                      [
                        Skip
                      ],
                    ref "i" .= ref "i" `plus` i 1
                  ]),
              ref "r'" .= ref "r",
              ref "i'" .= ref "i",
              ref "a'" .= ref "a",
              ref "N'" .= ref "N"
              ]
  ]

minindCall :: Stmt
minindCall = var [int "i", int "N", array "a", int "r"] [
       assume (ref "i" .< ref "N"),
       minind,
       pcall "minind" [ref "i", ref "N", ref "a"] [ref "r", ref "i",
           ref "N", ref "a"],
       assert (forall "x" (ref "i" .<= ref "x" .&& ref "x" .< ref "N"
            .==>
                                (ref "a" `repby` ref "r" .<= ref "a"
                                    `repby` ref "x")))
    ]
```

7 Loop reduction

This section explains how the loop reduction is implemented. In **Verification.hs** we added an extra pattern match for the while without an invariant that tries to calculate the fix point of the While. Unfortunately, the loop reduction can not always find a fix point. In the case that we could not find a fix point we return the $\{\text{while}\}^0$. However, when we found a fix point it will return the fix point.

while G do S

$\{\text{while}\}^0 = \neg g \Rightarrow Q$

$\{\text{while}\}^1 = (\text{g} \wedge \text{wlp s } \{\text{while}\}^0) \vee (\neg g \wedge \text{Q})$

In the project description there are two examples of GCL code given, which we had to verify. See the below code of those two sample written in our WLP-engine:

```
loop1 :: Stmt
loop1 = var [int "i", int "N", array "a", int "s"] [
          assume ((i 0 .== ref "i") .&& (ref "s" .== i 0) .&& (i 0 .<=
              ref "N")),
          while (ref "i" .< ref "N") [
             assert ((i 0 .<= ref "i") .&& (ref "i" .< ref "N")),
             ref "s" .= ref "s" ‘plus‘ ref "a" ‘repby‘ ref "i",
             ref "i" .= ref "i" ‘plus‘ i 1
          ],
          assert True_
       ]

loop2 :: Stmt
loop2 = var [int "i", int "N", array "a", int "s", int "k"] [
          assume (i 0 .== ref "i" .&& i 0 .<= ref "N" .&& i 0 .<= ref
              "k" .&& ref "k" .< ref "N"),
          while (ref "i" .< ref "N") [
             ref "i" .= ref "i" ‘plus‘ i 1
          ],
          assert (i 0 .<= ref "k" .&& ref "k" .< ref "N"),
          ref "s" .= ref "a" ‘repby‘ ref "k",
          assert True_
       ]
```

Our WLP-engine could only verify loop1.