Wlp in Spin

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

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

March 26, 2016

1 Introduction

This document describes the approach on how we implemented the WLP engine. The engine is written in Haskell. In order to proof the Wlp we use the data.SBV package. We divided the document in 6 section and each of the section explains a particular part of the system. To get an impression on how to write GCL in our code, see the following code:

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

You need to write it like this in our engine:

```
1 plusExample :: Stmt
2 plusExample =
3  var [int "a", int "b"]
4     [
5       assume (i 1 .== ref "b"),
6       ref "a" .= ref "b" `minus` i 1,
7       assert (ref "a" .== i 0)
8     ]
```

To see all the possibilities see gcl.hs and examples.hs.

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

In the the next part of the document we describes in more detail all of the components listed above. Each of the sections will also contain an example that shows our best work of that particular component.

3 Base implementation

The first implementation is the base implementation. In this part of the engine we created the basic structure. The structure is divided into four components. See figure 1.

3.1 Collection

The syntax to represent GCL is defined in **Attribute.ag**. In **Collection.ag** we collect all the variables and programs calls. In the collection of the variables we divide vars and refs into separate lists. We do this to distinguish between arrays and integer variable. For example to define an integer variable you need to write: int "a". The program call is explained in chapter 6.

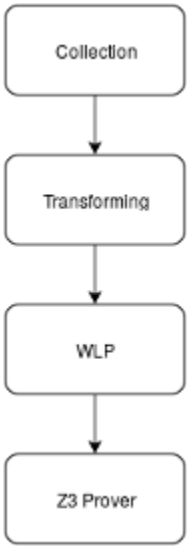

Figure 1: Structure

3.2 Transforming

The transforming is written in Transformer.hs. In this file we first introduce new fresh variables. We do this by folding over the Statements and Expression. In the folding we keep track of a number, which we add at the end of each variable. If we pattern match on Var, we will increase this number, so that all of the variable of the var will have 1 higher than the previous var type. However, in the forall we also do this. In the transforming we also added array and program calls, but this will be explained later in the document.

3.3 Verification

In **verification.hs** we calculate the WLP. We do this by recursive going over the wlp function. In order to calculate the WLP we use the lemma's of the lecture

notes. However, there are some special cases where we had to do more work. In the assign we had modified the Q.$_T$*hesecondcaseisinthewhile.Inthewhilethereare*2*specialcases.Ifthewhileisp*

3.4 Prover

The last part of the structure is the Prover. This is written in **Prover.hs**. In this file we proof the WLP, calculated in the verification. In order to proof the WLP we use the package Data.SBV. The first thing we do to proof is to store the of all the variables its value. This has the possibility to be an int or array. We need to distinguish between these two, so we use two maps to store them. Now that we retrieved all the possible variables, we will transform our calculate WLP to the correct

proveImpl :: Expr -¿ Expr -¿ IO SBV.ThmResult mkSymEq :: M.Map String SInteger -¿ M.Map String (SArray Integer Integer) -¿ Expr -¿ Symbolic (Either SInteger (SArray Integer Integer)) mkPred :: M.Map String SInteger -¿ M.Map String (SArray Integer Integer) -¿ Expr -¿ Predicate mkSymInt :: M.Map String SInteger -¿ M.Map String (SArray Integer Integer) -¿ Expr -¿ Symbolic SInteger mkInt :: M.Map String SInteger -¿ M.Map String (SArray Integer Integer) -¿ (SInteger -¿ SInteger -¿ SInteger) -¿ Expr -¿ Expr -¿ Symbolic SInteger

3.5 Example

To see what is possible see the following code:

```
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            ]
```

This proofs to QED. However when we modify the assert at line 7 to *(ref "y" .== i 0)*, this results in:

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

4 Simultaneous assignment

In this section we will explain how we implemented simultaneous assignments. We did this in the tranformer.hs.

```
1 a := 10
```

```
2 a,b := 1, a + 3
```

Transformed to:

```
1 a0 := 10
2 a1 := a0
3 b1 := b0
4 a0 := 1
5 b0 := a1 + 3
```

5 Array assignment

This section explains how we implemented array assignments in our system. 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. For example to use the following array **a[i] := 1** you need to write this:

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

Next we had to transform the array assigned to the correct type. See the lemma:
P $\Rightarrow$ Q[1/a[i]]
{P} a[i] := 1 {Q}

This is done in Verification.hs. In the function wlp we added an extra patten match at the Assign Stmt to support arrays. When the function is executed with that pattern match we will modify the Q, so that all element that consist of the same variable and index as the array will be replaced by the value of correct value. This means that we will replace the value of the array element with the right Expression.

Proof Now that we assigned the correct value in Q. This allows us to verify the correctness of the code. We do this in the Prover.hs. To distinguish between variables and arrays we added an extra map, which got as key the variable name and as value the index. We use this array map to get the correct element to get the SInteger of this array. To get the SInteger of an array we use the function readArray from the Data.SBV package to get the correct element. We now can do the same as a normal assignment, the only difference is that we use readarray instead of sInteger.

Example:

```
1 swap :: Stmt
2 swap = var ["a", "i", "j", "tmp", "c", "b"]
3            [assume ((ref "a" `repby` ref "i" .== ref "b") .&& (ref "a"
                 `repby` ref "j" .== ref "c")),
4            ref "tmp" .= ref "a" `repby` ref "i",
```

```
5               ref "a" `repby` ref "i" .= ref "a" `repby` ref "j",
6               ref "a" `repby` ref "j" .= ref "i",
7               assert ((ref "tmp" .== ref "b") .&& (ref "a" `repby` ref
                    "i" .== ref "c") .&& (ref "a" `repby` ref "j" .== ref
                    "b"))
8           ]
```

6 Program call

In this section we explain how we implemented the program call in the WLP Engine. In the transformer.hs we added an extra pattern match to transform the program call to the correct form. See lemma:

```
1  minind :: Stmt
2  minind = prog "minind" [int "i", int "N", array "a"] [int "r'", int
       "i'", int "N'", array "a'"]
3          [
4
5              var [int "min", int "r", int "j"]
6                [
7                ref "j" .= ref "i",
8                ref "r" .= ref "i",
9                ref "min" .= ref "a" `repby` ref "r",
10               inv (forall "x" (ref "j" .< ref "N" .&& ref "j" .<= ref
                     "i" .&& ref "j" .<= ref "r" .&&
11                              (ref "j" .== ref "i" .==> ref "r" .== ref
                                    "i") .&&
12                              (ref "j" .< ref "i" .==> ref "r" .< ref
                                    "i") .&&
13                              ref "min" .== ref "a" `repby` ref "r" .&&
14                              (ref "j" .<= ref "x" .&& ref "x" .< ref "i"
                                    .==> ref "a" `repby` ref "r" .<= ref
                                    "a" `repby` ref "x")))
15               (while (ref "i" .< ref "N")
16                 [
17                   if_then_else (ref "a" `repby` ref "i" .< ref "min")
18                     [
19                       ref "min" .= ref "a" `repby` ref "i",
20                       ref "r" .= ref "i"
21                     ]
22                     [
23                       Skip
24                     ],
25                   ref "i" .= ref "i" `plus` i 1
26                 ]),
27               ref "r'" .= ref "r",
28               ref "i'" .= ref "i",
29               ref "a'" .= ref "a",
```

```
30              ref "N'" .= ref "N"
31              ]
32  ]
33
34 callMinind :: Stmt
35 callMinind = var [int "i", int "N", array "a", int "r"] [
36        assume (ref "i" .< ref "N"),
37        minind,
38        pcall "minind" [ref "i", ref "N", ref "a"] [ref "r", ref "i",
              ref "N", ref "a"],
39        assert (forall "x" (ref "i" .<= ref "x" .&& ref "x" .< ref "N"
              .==>
40                                (ref "a" `repby` ref "r" .<= ref "a"
                                      `repby` ref "x")))
41    ]
```

7 Loop reduction

This explains how the loop reduction is implemented. In the Verification.hs we added an extra pattern match for a while without an invaraint, which tries to calculate a fixpoint of the While. This is done in the loopReduction function. If the function does not find a fixpoint, we return the w0 calculation. Otherwise we return the fixpoint.

while G do S
$\{\text{while}\}^0 = \neg g \Rightarrow Q$
$\{\text{while}\}^1 = (\text{g} \wedge \text{wlp s } \{\text{while}\}^0) \vee (\neg g \wedge \text{Q})$
For example we have this code:

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

Not sure if need to calculate a loop reduction here.

8 Relevance work

Relevant work..

9 Conclusion

Conclusion...