Wlp-based verification engine

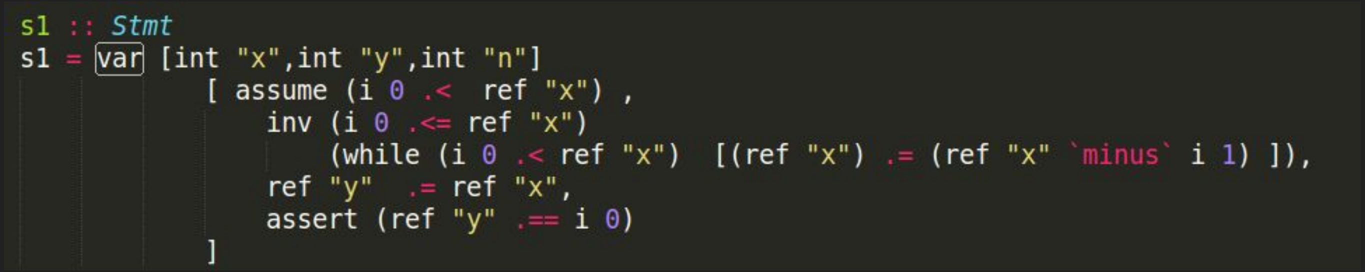
Ferdinand van Walree & Matthew Swart

Agenda

- Introduction
- Implementation overview
- Example/Questions

Introduction

- **Prover:** Z3
- **Language:** Haskell
- **Package:** Data.SBV



```haskell
s1 :: Stmt
s1 = var [int "x",int "y",int "n"]
            [ assume (i 0 .<  ref "x") ,
                inv (i 0 .<= ref "x")
                    (while (i 0 .< ref "x")  [(ref "x") .= (ref "x" `minus` i 1) ]),
                ref "y"  .= ref "x",
                assert (ref "y" .== i 0)
            ]
```

Implementation overview

- Base implementation
- Simultaneous assignments
- Array assignments
- Program call

Array assignments

GCL:

```
a[i] := 3  →  a := a(i repby 3)

a[i] := 3  →  ref “a” `withIndex` ref “i” .= 3
```

Array assignments - Collection

- Collect vars
- Collect refs
- Collect Programs

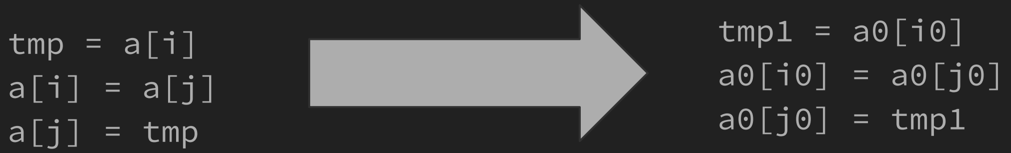
```
sem Expr
     | Repby lhs.allVars = S.union @expr1.allVars @expr2.allVars
        lhs.allRefs = S.union @expr1.allRefs @expr2.allRefs




Var [int “i”, int “j”, array “a”]
     [
          ref “a” `indexWith` ref “i” .= ref “a” `indexWith` ref “j”
     ]
```

Array assignments - Transforming

```
mkFreshExpr :: Int -> M.Map String String -> Expr -> (Int,Expr)
mkFreshExpr n varMap (Repby e1 e2) =
     let (n1,expr1) = mkFreshExpr n varMap e1
         (n2,expr2) = mkFreshExpr n1 varMap e2
     in (n2, Repby expr1 expr2)
```



```
tmp = a[i]
a[i] = a[j]
a[j] = tmp
```

```
tmp1 = a0[i0]
a0[i0] = a0[j0]
a0[j0] = tmp1
```

Array assignments - WLP

- Added case to wlp
- Added case to assign

```
wlp :: [Var] -> Stmt -> Expr -> IO ([Expr],Expr)
wlp vars (Assign (ArrayIndex (Name s) i) e2) q = do
     let pre = assign q (s, i) e2
     return $ ([],pre)
```

```
assign :: Expr -> (String, Expr) -> Expr -> Expr
assign (ArrayIndex (Name s) index) ref expr
     | s == fst ref && index == snd ref = expr
     | s == fst ref = ArrayIndex expr (assign index ref expr)
     | otherwise = ArrayIndex (Name s) (assign index ref expr)
```

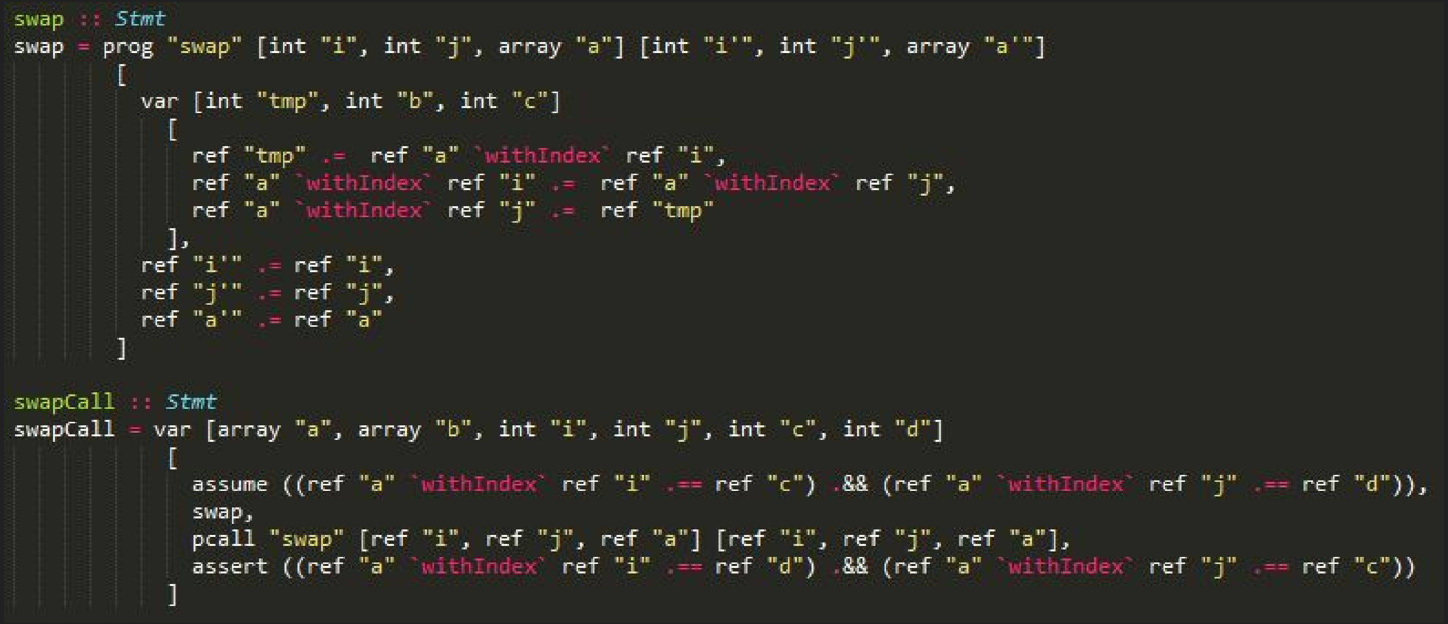
Array assignments - Z3 prover

- Collect refs → fill maps -> make predicate -> Prove predicate

```
mkPred :: M.Map String SInteger -> M.Map String (SArray Integer Integer) -> Expr -> Predicate
mkPred vars arr (Equal e1 e2)  = do
     p1 <- mkSymEq vars arr e1
     case p1 of
     Left sInt1 -> do
          Left sInt2 <- mkSymIntArr vars arr e2
          return $ sInt1 .== sInt2
     Right sArr1 -> do
          Right sArr2 <- mkSymIntArr vars arr e2
          return $ sArr1 .== sArr2

mkSymInt :: M.Map String SInteger -> M.Map String (SArray Integer Integer) -> Expr ->
Symbolic SInteger
mkSymInt vars arr (Repby (Name s) (Lit index)) = return $ readArray (fromJust $ M.lookup s
arr) index
```

Demo



```
swap :: Stmt
swap = prog "swap" [int "i", int "j", array "a"] [int "i'", int "j'", array "a'"]
         [
           var [int "tmp", int "b", int "c"]
             [
               ref "tmp" .=  ref "a" `withIndex` ref "i",
               ref "a" `withIndex` ref "i" .=  ref "a" `withIndex` ref "j",
               ref "a" `withIndex` ref "j" .=  ref "tmp"
             ],
           ref "i'" .= ref "i",
           ref "j'" .= ref "j",
           ref "a'" .= ref "a"
         ]

swapCall :: Stmt
swapCall = var [array "a", array "b", int "i", int "j", int "c", int "d"]
             [
               assume ((ref "a" `withIndex` ref "i" .== ref "c") .&& (ref "a" `withIndex` ref "j" .== ref "d")),
               swap,
               pcall "swap" [ref "i", ref "j", ref "a"] [ref "i", ref "j", ref "a"],
               assert ((ref "a" `withIndex` ref "i" .== ref "d") .&& (ref "a" `withIndex` ref "j" .== ref "c"))
             ]
```