Universidade do Minho

Mestrado Integrado em Engenharia Informática

Verificação Formal

Verificador de Programas

Group 3:
Mário Ferreira
José Fernandes

21 de Junho de 2017

Universidade do Minho
Escola de Engenharia

Conteúdo

1 Introdução

O objetivo deste trabalho passava por desenvolver um verificador de programas que utilizasse como ferramenta o Z3, para explorar o tratamento de um mecanismo de exceções.

Para isto, seria necessário desenvolver um parser para uma linguagem de programação simples, como exemplificado no enunciado. Utilizamos a linguagem Haskell para tal, utilizando a biblioteca *HaLeX* para desenvolver o parsing.

Uma vez que esta parte esteja concluída, é necessário gerar as condições de verificação e enviá-las para a ferramenta Z3 de forma a validar estas.

2 Abstract Syntax Tree

A nossa *Simple Language* suporta o tipo inteiro e booleano, assim como expressões inteiras. É composta pelo nome do programa, argumentos, pré-condição, um conjunto de declarações e instruções e uma terminação normal e excecional, para além do possível invariante no ciclo.

```
data SL = Program String [DeclProg] Boolean [Decl] [Inst] Boolean Boolean
        deriving Show

data Inv = Inv Boolean
         deriving (Show, Eq, Ord)
```

As declarações podem ser feitas atribuindo um valor (inteiro, char ou booleano), uma expressão ou pode apenas ser feita a declaração da variável. O programa pode receber argumentos também.

```
data Decl = Atr  TypeDecl String Expr
          | AtrT TypeDecl String
          | AtrS TypeDecl [Expr]
          deriving (Show , Eq , Ord)

data DeclProg = AtrProg TypeDecl String
          deriving (Show , Eq , Ord)

data TypeDecl = Int | Char | Bool
              deriving (Show , Eq , Ord)
```

Nas instruções é possível fazer atribuir novos valores a variáveis, ciclos *for* e *while*, para além do "statement" *IfThenElse* ou apenas *IfThen*.

A nossa árvore também contempleta *read*, *print* e *return*. Para além destes, finalmente o *try* e *catch*.

```
data Inst = Assign      String Expr
          | IfThenElse  Boolean [Inst] [Inst]
          | For         Decl Boolean Expr [Inv] [Inst]
          | While       Boolean Boolean [Inst]
          | Read        Expr
          | Print       Expr
          | Return      Expr
          | Try         [Inst] [Inst]
          | Throw
          deriving (Show , Eq , Ord)
```

Uma expressão pode ter qualquer operador aritmético (+, -, *, /).

```haskell
data Expr = Const Integer
          | Var   String
          | Add   Expr Expr
          | Mul   Expr Expr
          | Div   Expr Expr
          | Sub   Expr Expr
          | Same  Expr Expr
          deriving (Show , Eq , Ord)
```

Um booleano pode ter o valor True ou False, sendo que foram implementados os operadores lógicos essenciais (and, or, great, great or equal, less, less or equal, equal, different).

```haskell
data Boolean = Expr         Expr
             | Greater      Expr Expr
             | GreaterEqual Expr Expr
             | Less         Expr Expr
             | LessEqual    Expr Expr
             | And          Expr Boolean
             | Orl          Expr Boolean
             | Equal        Expr Expr
             | Different    Expr Expr
             | BoolConst    Bool
             | Implies      Expr Boolean
             | Not          Expr
             deriving (Show , Eq , Ord)
```

Uma vez que a biblioteca HaLeX apresentava limitações quanto à recursividade à esquerda, foi criado o Data Boolean2 para resolver este problema.

```haskell
data Boolean2 = Expr2        Expr
              | Greater2     Expr Expr
              | GreaterEqual2 Expr Expr
              | Less2        Expr Expr
              | LessEqual2   Expr Expr
              | And2         Boolean2 Boolean2
              | Orl2         Boolean2 Boolean2
              | Equal2       Expr Expr
              | Different2   Expr Expr
              | BoolConst2   Bool
              | Implies2     Boolean2 Boolean2
              | Not2         Boolean2
              deriving (Show , Eq , Ord)
```

3 Parsing

Para realizar o parsing dos programas em *Simple Language*, utilizamos os combinadores de parsing da biblioteca HaLeX. Note que não há problemas com espaços na nossa linguagem, uma vez que todos estes são filtrados.

É possível observar que esta biblioteca é intuitiva e fácil de entender. É possível ver a seguir que o parser, como esperado, tem a pré e ambas a pós condições, sendo que pelo meio terá as declarações e instruções.

```
parser = f <$> token' "pre" <*> pre <*> token' "program" <*> pString <*>
    symbol' '(' <*> declsProg <*> symbol' ')' <*> symbol' '{' <*> decls
          <*> spaces' <*> insts <*> symbol' '}' <*> token' "postn" <*> postn
               <*> token' "poste" <*> poste
     where f _ a _ b _ g _ _ c _ d _ _ e _ f = Program b g a c d e f
```

A seguir é possível observar o parsing das declarações, tanto dentro da função como dos argumentos da própria função. É flexível, sendo que a declaração pode ter o valor associado ou apenas pode ser criada a variável sem qualquer valor. Int, Char e Bool são os tipos permitidos.

```
decls = oneOrMore decl

decl =  f  <$> pTypeInt'  <*> pString <*> symbol' '=' <*> expr <*> symbol'
    ';'
    <|> g  <$> pTypeChar' <*> pString <*> symbol' '=' <*> expr <*> symbol'
        ';'
    <|> h  <$> pTypeBool' <*> pString <*> symbol' '=' <*> expr <*> symbol'
        ';'
    <|> i  <$> pTypeInt'  <*> pString <*> symbol' ';'
    <|> j  <$> pTypeChar' <*> pString <*> symbol' ';'
    <|> k  <$> pTypeBool' <*> pString <*> symbol' ';'
    <|> p1 <$> pTypeInt'  <*> expr <*> varios <*> symbol' ';'
    <|> p2 <$> pTypeChar' <*> expr <*> varios <*> symbol' ';'
    <|> p3 <$> pTypeBool' <*> expr <*> varios <*> symbol' ';'
     where f _ b _ c _ =  Atr Int b c
           g _ b _ c _ =  Atr Char b c
           h _ b _ c _ =  Atr Bool b c
           i _ b _     =  AtrT Int b
           j _ b _     =  AtrT Char b
           k _ b _     =  AtrT Bool b
           p1 _ b c _  =  AtrS Int ([b]++c)
           p2 _ b c _  =  AtrS Char ([b]++c)
           p3 _ b c _  =  AtrS Bool ([b]++c)

declsProg = oneOrMore declProg

declProg = i <$> pTypeInt'  <*> pString <*> symbol' ';'
    <|> j <$> pTypeChar' <*> pString <*> symbol' ';'
    <|> k <$> pTypeBool' <*> pString <*> symbol' ';'
     where i _ b _     =  AtrProg Int b
```

```
28          j _ b _     =  AtrProg Char b
29          k _ b _     =  AtrProg Bool b
```

O parsing das instruções, também é intuitivo. Por exemplo, sempre que há um *if*, haverá uma condição booleana e depois do token *then* haverá instruções (necessário pelo menos uma). O token *else* é flexível, sendo possível omiti-lo. Caso seja necessário, pode ser utilizado e conterá instruções, mais uma vez, sendo necessário pelo menos uma. Este procedimento é idêntico para todas, verificando por exemplo que o read apenas precisa do token *read*, a variável a ler e fecha com *;*.

```
1  insts = oneOrMore inst
2
3  inst =  f  <$> token' "print"  <*> expr <*> symbol' ';'
4       <|> g <$> pString <*> symbol' '=' <*> expr <*> symbol' ';'
5       <|> h <$> token' "if" <*> symbol' '(' <*> boolean <*> symbol' ')' <*>
            token' "then" <*> symbol' '{'
6           <*> insts <*> symbol' '}' <*> token' "else" <*> symbol' '{' <*>
                insts <*> symbol' '}'
7       <|> i <$> token' "if" <*> symbol' '(' <*> boolean <*> symbol' ')' <*>
            token' "then" <*> symbol' '{'
8           <*> insts <*> symbol' '}'
9       <|> j <$> token' "return" <*> expr <*> symbol' ';'
10      <|> k <$> token' "read" <*> expr <*> symbol' ';'
11      <|> l <$> token' "for" <*> symbol' '(' <*> decl <*> boolean <*> symbol'
             ';' <*> expr <*> symbol' ')'
12          <*> symbol' '{' <*> token' "inv" <*> invs <*> insts <*> symbol' '}'
13      <|> m <$> token' "while" <*> symbol' '(' <*> boolean <*> symbol' ')'
            <*> symbol' '{' <*> token' "inv"
14          <*> boolean <*> symbol' ';' <*> insts <*> symbol' '}'
15      <|> n <$> token' "try" <*> symbol' '{' <*> insts <*> symbol' '}' <*>
            token' "catch" <*> symbol' '{'
16          <*> insts <*> symbol' '}'
17      <|> o <$> token' "throw" <*> symbol' ';'
18    where f _ b _ =                   Print b
19          g a _ c _ =                 Assign a c
20          h _ _ a _ _ _ b _ _ _ c _ = IfThenElse a b c
21          i _ _ a _ _ _ b _ =         IfThenElse a b []
22          j _ a _ =                   Return a
23          k _ a _ =                   Read a
24          l _ _ a b _ c _ _ _ d e _ = For a b c d e
25          m _ _ a _ _ _ b _ c _ =     While a b c
26          n _ _ a _ _ _ b _ =         Try a b
27          o _ _ =                     Throw
```

Para além do invariante, que será um boolean, temos as expressões. Podem ser apenas uma expressão ou então uma operação aritmética (*, +, -, /) entre expressões. É possível fazer o assign de uma variável a outra expressão.

(Por exemplo x=1+2*3/4)

```
invs = zeroOrMore inv

inv = f <$> boolean <*> symbol' ';'
    where f a _ = Inv a

expr = id <$> expressao
    <|> f <$> expressao <*> symbol' '*' <*> expr
    <|> g <$> expressao <*> symbol' '+' <*> expr
    <|> h <$> expressao <*> symbol' '/' <*> expr
    <|> i <$> expressao <*> symbol' '-' <*> expr
    <|> j <$> expressao <*> symbol' '=' <*> expr
  where f l _ r = Mul l r
        g l _ r = Add l r
        h l _ r = Div l r
        i l _ r = Sub l r
        j l _ r = Same l r

expressao =  f <$> pString
         <|> g <$> pInt
  where f a = Var a
        g a = Const (read a :: Integer)
```

De seguida, temos o parsing dos boolean. Para os operadores de comparação (maior, menor, etc), é feita a comparação entre uma expressão e outra expressão (ou várias expressões). No caso do And e Or, a comparação é feita entre uma expressão e um boolean.

```
boolean =   (\a -> BoolConst True)       <$> token' "true"
        <|> (\a -> BoolConst False)      <$> token' "false"
        <|> (\a _ b -> Less a b)         <$> expr <*> symbol' '<' <*> expr
        <|> (\a _ b -> Greater a b)      <$> expr <*> symbol' '>' <*> expr
        <|> (\a _ b -> LessEqual a b)    <$> expr <*> token' "<=" <*> expr
        <|> (\a _ b -> GreaterEqual a b) <$> expr <*> token' ">=" <*> expr
        <|> (\a _ b -> Equal a b)        <$> expr <*> token' "==" <*> expr
        <|> (\a _ b -> Different a b)    <$> expr <*> token' "!=" <*> expr
        <|> (\a _ b -> And a b)          <$> expr <*> token' "&&" <*>
            boolean
        <|> (\a _ b -> Orl a b)          <$> expr <*> token' "||" <*>
            boolean
        <|> (\a _ b -> Implies a b)      <$> expr <*> token' "=>" <*>
            boolean
        <|> (\a -> Expr a)               <$> expr
        <|> (\a -> Not a)                <$> expr
```

4 Geração das VC's

Primeiro convertemos as expressões inteiras e booleanas com **exprZ** e **booleanZ** respetivamente.

```
exprZ (Var s) = mkFreshIntVar s

exprZ (Const c) = mkInteger c

exprZ (Add a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkAdd [c, d]
}

exprZ (Mul a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkMul [c, d]
}

exprZ (Div a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkDiv c d
}

exprZ (Sub a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkSub [c, d]
}

booleanZ (Expr2 e) = exprZ e

booleanZ (BoolConst2 e)
      | e = mkTrue
      | otherwise = mkFalse

booleanZ (Less2 a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkLt c d
}

booleanZ (Not2 a) = booleanZ a >>= mkNot


booleanZ (LessEqual2 a b) = do {
  c <- exprZ a;
  d <- exprZ b;
  mkLe c d
}

booleanZ (Greater2 a b) = do {
  c <- exprZ a;
```

```
52    d <- exprZ b;
53    mkGt c d
54  }
55
56  booleanZ (GreaterEqual2 a b) = do {
57    c <- exprZ a;
58    d <- exprZ b;
59    mkGe c d
60  }
61
62  booleanZ (Equal2 a b) = do {
63    c <- exprZ a;
64    d <- exprZ b;
65    mkEq c d
66  }
67
68  booleanZ (Different2 a b) = do {
69    c <- exprZ a;
70    d <- exprZ b;
71    mkEq c d >>= mkNot
72  }
73
74
75
76  booleanZ (And2 a b) = do {
77    c <- booleanZ a;
78    d <- booleanZ b;
79    mkAnd [c,d]
80  }
81
82  booleanZ (Orl2 a b) = do {
83    c <- booleanZ a;
84    d <- booleanZ b;
85    mkOr [c,d]
86  }
87
88  booleanZ (Implies2 a b) = do {
89    c <- booleanZ a;
90    d <- booleanZ b;
91    mkImplies c d
92  }
```

De seguida temos o código haskell que faz o resto do trabalho. Como sugerido pelo professor, a substituição na árvore é feita do nosso lado sendo que apenas é enviado para o Z3 no final.

```
1  gera_vc = sequence . vcgZ . vcg
2
3  vcgZ a = if a == [] then [] else map booleanZ a
4
5  vcg (Program a b c d e f g) = [(Implies2 (boolTobool2 c) (wp e (boolTobool2
       f) (boolTobool2 g) ))]
6                                ++ (vcaux e (boolTobool2 f) (boolTobool2 g))
7
```

```
8
9
10 vcaux [] q1 q2 = []
11 vcaux [Assign v i] q1 q2 = []
12 vcaux [IfThenElse b s1 s2] q1 q2 = (vcaux s1 q1 q2) ++ (vcaux s2 q1 q2)
13 vcaux [While b i s] q1 q2 = (([Implies2 (And2 (boolTobool2 i) (boolTobool2 b
       )) (wp s (boolTobool2 i) q2),
14                              Implies2 (And2 (boolTobool2 i) (Not2 (
                                   boolTobool2 b))) q1])) ++ (vcaux s (
                                   boolTobool2 i) q2)
15 vcaux [Try s1 s2] q1 q2 = (vcaux s1 q1 q2) ++ (vcaux s2 q1 q2)
16 vcaux [Throw] q1 q2 = []
17 vcaux (s1:sn) q1 q2 = (vcaux [s1] (wp sn q1 q2) q2) ++ (vcaux sn q1 q2)
18
19
20 wp [] q1 q2 = q1
21 wp [Assign x e] q1 q2 = aux1 q1 x e
22 wp [IfThenElse b s1 s2] q1 q2 = And2 (Implies2 (boolTobool2 b) (wp s1 q1 q2)
       ) (Implies2 (Not2 (boolTobool2 b)) (wp s2 q1 q2))
23 wp [While b i s] q1 q2 = (boolTobool2 i)
24 wp [Try s1 s2] q1 q2 = (wp s1 q1 (wp s2 q1 q2))
25 wp [Throw] q1 q2 = q2
26 wp (s1:sn) q1 q2 = (wp [s1] (wp sn q1 q2) q2)
27
28
29 aux1 (Greater2 a b) x e = Greater2 (aux2 a x e) (aux2 b x e)
30 aux1 (GreaterEqual2 a b) x e = GreaterEqual2 (aux2 a x e) (aux2 b x e)
31 aux1 (Less2 a b) x e = Less2 (aux2 a x e) (aux2 b x e)
32 aux1 (LessEqual2 a b) x e = LessEqual2 (aux2 a x e) (aux2 b x e)
33 aux1 (Equal2 a b) x e = Equal2 (aux2 a x e) (aux2 b x e)
34 aux1 (Different2 a b) x e = Different2 (aux2 a x e) (aux2 b x e)
35 aux1 (Not2 a) x e = Not2 (aux1 a x e)
36 aux1 (And2 a b) x e = And2 (aux1 a x e) (aux1 b x e)
37 aux1 (Orl2 a b) x e = Orl2 (aux1 a x e) (aux1 b x e)
38 aux1 (Implies2 a b) x e = Implies2 (aux1 a x e) (aux1 b x e)
39 aux1 a x e = a
40
41
42 aux2 (Var a) x e = if a == x then e
43                               else Var a
44 aux2 (Add a b) x e  = Add (aux2 a x e) (aux2 b x e)
45 aux2 (Mul a b) x e  = Mul (aux2 a x e) (aux2 b x e)
46 aux2 (Sub a b) x e  = Sub (aux2 a x e) (aux2 b x e)
47 aux2 (Div a b) x e  = Div (aux2 a x e) (aux2 b x e)
48 aux2 (Same a b) x e = Same (aux2 a x e) (aux2 b x e)
49 aux2 a x e = a
50
51
52 auxPrintVCs []     = []
53 auxPrintVCs (x:xs) = auxPrintVC x ++ "\n" ++ auxPrintVCs xs
54
55
56 auxPrintVC (Expr2 a) = auxPrintExpr a
57 auxPrintVC (Greater2 a b) = (auxPrintExpr a) ++ " > " ++ (auxPrintExpr b)
58 auxPrintVC (GreaterEqual2 a b) = (auxPrintExpr a) ++ " >= " ++ (auxPrintExpr
       b)
59 auxPrintVC (Less2 a b) = (auxPrintExpr a) ++ " < " ++ (auxPrintExpr b)
```

```
auxPrintVC (LessEqual2 a b) = (auxPrintExpr a) ++ " <= " ++ (auxPrintExpr b)
auxPrintVC (And2 a b) = (auxPrintVC a) ++ " && " ++ (auxPrintVC b)
auxPrintVC (Orl2 a b) = (auxPrintVC a) ++ " || " ++ (auxPrintVC b)
auxPrintVC (Equal2 a b) = (auxPrintExpr a) ++ " == " ++ (auxPrintExpr b)
auxPrintVC (Different2 a b) = (auxPrintExpr a) ++ " != " ++ (auxPrintExpr b)
auxPrintVC (BoolConst2 a) = show a
auxPrintVC (Implies2 a b) = (auxPrintVC a) ++ " ==> " ++ (auxPrintVC b)
auxPrintVC (Not2 a ) = "not (" ++ (auxPrintVC a) ++ ")"

auxPrintExpr (Const a) = show a
auxPrintExpr (Var a) = a
auxPrintExpr (Add a b) = (auxPrintExpr a) ++ " + " ++ (auxPrintExpr b)
auxPrintExpr (Mul a b) = (auxPrintExpr a) ++ " * " ++ (auxPrintExpr b)
auxPrintExpr (Div a b) = (auxPrintExpr a) ++ " / " ++ (auxPrintExpr b)
auxPrintExpr (Sub a b) = (auxPrintExpr a) ++ " - " ++ (auxPrintExpr b)
auxPrintExpr (Same a b) = (auxPrintExpr a) ++ " = " ++ (auxPrintExpr b)
```

Por fim, podemos validar as condições de verificação:

```
auxBP x = do
    vc <- gera_vc x
    mapM (\l -> reset >> assert l >> check) vc

main = do
    putStrLn $ auxPrintVCs (vcg sl2)
    final <- evalZ3 $ auxBP sl2
    mapM_ print final
```

5 Testes

5.1 Teste 1

```
pre d==10;

program a (int d;){
  int nada;
  while(d<12){
    inv d<12;
    d = d+1;
  }
}

postn d==12;
poste false;
```

```
d == 10 ==> d < 12
d < 12 && d < 12 ==> d + 1 < 12
d < 12 && not (d < 12) ==> d == 12

Sat
Sat
Sat
```

5.2 Teste 2

```
pre c > 12;

program a (int d;){
  int b;
  while(c!=10){
    inv c > 10;
    c = c-1;
  }
}

postn c==10;
poste false;
```

```
c > 12 ==> c > 10
c > 10 && c != 10 ==> c - 1 > 10
c > 10 && not (c != 10) ==> c == 10

Sat
Sat
Sat
```

6 Conclusão

Não foram realizados alguns pontos propostos no enunciado, tais como a utilização da interface e elementos como o mecanismo de exceções não estão a funcionar corretamente, pelo que haveria algum trabalho futuro a realizar.

No entanto, o objetivo principal foi atingido, uma vez que o parsing da linguagem simples está bem construído, assim como a geração e validação de condições de verificação deste.

Sendo assim, consideramos que o trabalho deveria estar superior, mesmo com a conjugação de trabalhos e exames, no entanto foi concluído alguns dos objetivos e que suportam um fácil melhoramento no futuro.