



UNIVERSIDADE DO MINHO

VERIFICAÇÃO FORMAL

Verificador de Programas de uma Linguagem de Programação Simples

Desenvolvido por:

Isac Meira PG31577

Hugo Ribeiro PG32996

Docentes :

José Sousa Pinto

Maria João Frade

20 de Junho de 2017

Conteúdo

1	Introdução	2
2	Abstract Syntax Tree	3
3	Parsing dos programas da TinyL	6
4	Geração das Condições de Verificação	13
5	Apresentação das Condições de Verificação e sua Verificação	16
6	Conclusão	19

1 Introdução

No âmbito da componente prática da Unidade Curricular de Verificação Formal, fomos desafiados a desenvolver um verificador de programas de uma linguagem de programação simples, explorando o tratamento de um mecanismo de exceções.

Para tal, desenvolvemos um parser para uma linguagem de programação que intitulámos de “*TinyL*” utilizando uma biblioteca de combinadores de parsing para Haskell chamada de “*parsec*”.

Ao realizarmos o parsing do programa da “*TinyL*” carregamos a estrutura do mesmo para uma “*Abstract Syntax Tree*” para que futuramente a possamos percorrer para gerar as condições de verificação do programa e verificar as mesmas enviando-as para o “*SMTSolver*” **Z3**.

2 Abstract Syntax Tree

Dado que a “TinyL” só suporta o tipo inteiro e expressões inteiras e booleanas a nossa AST não contempla declarações, pelo que a sua estrutura é composta por uma pré-condição, um conjunto possivelmente vazio de instruções e uma pós-condição por terminação normal e uma pós-condição por terminação excecional.

Tanto a pré-condição como as pós-condições são expressões booleanas, não tendo nós implementado os construtores \forall , \exists nem *old*.

Listing 1: Tipo de dados da Tiny Language

```
1 data TinyL = TinyL BoolExpr [Stmt] BoolExpr BoolExpr
2   deriving (Show, Eq, Ord)
```

O tipo de dados “BoolExpr” representa as expressões booleanas, que poderão ser os valores booleanos **true** ou **false**, o operador unário da negação de uma expressão booleana, conjunções, disjunções ou implicações de expressões booleanas, ou ainda duas expressões inteiras relacionados por um operador relacional. Os operadores relacionais permitidos para as expressões booleanas são os operadores $>$, $<$, \geq , \Leftrightarrow e \cdot .

Listing 2: Tipo de dados das expressões booleanas e seus operadores

```
1 data BoolExpr = BoolConst Bool
2               | Not BoolExpr
3               | BoolBinary BoolBinOp BoolExpr BoolExpr
4               | RelationalBinary RelBinOp IntExpr IntExpr
5               deriving (Show, Eq, Ord)
6
7 data BoolBinOp = And
8               | Or
9               | Implies
10              deriving (Show, Eq, Ord)
11
```

```
12 data RelBinOp = GT
13               | LT
14               | GTE
15               | LTE
16               | EQ
17               | Diff
18               deriving (Show, Eq, Ord)
```

O tipo de dados das instruções, poderá ser a instrução de atribuição, uma instrução condicional com apenas um ramo e com dois ramos, uma instrução de repetição, a instrução *Try-Catch* e a instrução *Throws*.

Listing 3: Tipo de dados das instruções

```
1 type VarName = String
2
3 data Stmt = Attrib VarName IntExpr
4           | If BoolExpr [Stmt]
5           | TryCatch [Stmt] [Stmt]
6           | Throw
7           | IfThenElse BoolExpr [Stmt] [Stmt]
8           | Loop BoolExpr BoolExpr [Stmt]
9           deriving (Show, Eq, Ord)
```

Por fim, as expressões inteiras poderão ser apenas o nome de uma variável, um valor inteiro, o simétrico de um valor ou as operações típicas de $+$, $-$, \times , \div .

Listing 4: Tipo de dados das expressões inteiras

```
1 data IntExpr = Var String
2               | IntConst Integer
3               | Neg IntExpr
4               | IntBinary IntBinOp IntExpr IntExpr
5               deriving (Show, Eq, Ord)
6
7 data IntBinOp = Add
8               | Sub
9               | Mul
10              | Div
```

11 | `deriving (Show, Eq, Ord)`

3 Parsing dos programas da TinyL

O parsing dos programas escritos com a “TinyL” é realizado através do uso de combinadores de parsing monádicos da biblioteca *parsec*.

Ao realizarmos o parsing dos programas da linguagem iremos carregar os dados para *Abstract Syntax Tree* definida na secção anterior.

Inicialmente definimos uma lista em *Haskell* com os nomes das palavras reservadas da linguagem e uma outra lista com os operadores que a mesma utiliza, sendo as mesmas utilizadas para gerarmos uma definição da linguagem, como se mostra de seguida:

Listing 5: Geração da Definição da Linguagem

```
1 reservedNames' = ["pre", "inv", "posn", "pose", "if", "then",
2                   "else", "while", "true", "false",
3                   "throw", "try", "catch", "old"]
4
5 reservedOps = ["+", "*", "/", "=", "==", "!=", "&&", "||",
6               ">", "<", "<=", ">=", "!"]
7
8 tokens':: GenLanguageDef String st Identity
9 tokens' = LanguageDef {
10   commentStart = "{-",
11   commentEnd = "-}",
12   commentLine = "--",
13   nestedComments = True,
14   identStart = letter,
15   identLetter = alphaNum,
16   opStart = oneOf "+*/*=<>|<>",
17   opLetter = oneOf "&|",
18   reservedNames = reservedNames',
19   reservedOpNames = reservedOps,
20   caseSensitive = True
21 }
```

Como se pode ver, foi definido que a linguagem poderá ter comentários comentários escritos de maneira análoga aos comentários em *Haskell*, são permitidos comentários *nested*, os identificadores das variáveis terão que começar com uma letra minúscula e só poderão ser compostos por letras ou números. Os nomes das palavras reservadas da linguagem e os operadores reservados são os definidos nas listas e a linguagem é

case sensitive.

A partir desta definição da linguagem iremos criar um gerador de parsers de tokens que nos dará de bandeja um conjunto de métodos úteis ao parsing dos programas da “TinyL” que renomeámos para que os pudéssemos usar mais comodamente.

Listing 6: Criação do gerador de parsers de tokens e renomeação de funções

```
1  lexer = P.makeTokenParser tokens'
2
3  identifier' = P.identifier lexer
4  reserved' = P.reserved lexer
5  reservedOp' = P.reservedOp lexer
6  operator' = P.operator lexer
7  integer' = P.integer lexer
8  whiteSpace' = P.whiteSpace lexer
9  parens' = P.parens lexer
10 braces' = P.braces lexer
11 semi' = P.semi lexer
12 semiSep' = P.semiSep lexer
13 semiSep1' = P.semiSep1 lexer
14 symbol' = P.symbol lexer
```

Assim, partimos para a escrita do parsing da “TinyL” que é uma pré-condição, seguida de um conjunto de instruções, que por sua vez é seguida por uma pós-condição com terminação normal e uma pós condição para uma terminação excecional.

Listing 7: 1ªetapa do parsing

```
1  tinyL = do{
2      whiteSpace';
3      p <- pre;
4      s <- statments;
5      p1 <- posn;
6      p2 <- pose;
7
8      return $ TinyL p s p1 p2
9  }
```


Como os parsers de tokens só ignoram os espaços após os tokens temos que inicialmente livrar-nos dos caracteres de espaço iniciais.

Os combinadores de parsing *pre*, *posn* e *pose* mais não são que:

Listing 8: Combinadores pre

```
1 pre = do{
2   reserved' "pre";
3   a <- boolexp;
4   semi';
5
6   return a
7 }
8
9 posn = do{
10  reserved' "posn";
11  a <- boolexp;
12  semi';
13
14  return a
15 }
16
17 pose = do{
18  reserved' "pose";
19  a <- boolexp;
20  semi';
21
22  return a
23 }
```

E dão o parse da sua respetiva palavra reservada da “TinyL”, fazendo o parse de seguida usando o combinador boolexp e por fim dão parse ao “;”.

O combinador de parsing *statments* é zero ou mais vezes um *statment*, em que o combinador de parsing *statment* é por sua vez ou uma atribuição, ou uma instrução condicional com um ramo, ou uma instrução condicional com dois ramos, ou uma instrução *Try-Catch* ou uma instrução *Throw*, que lança uma exceção, ou uma instrução de ciclo.

Listing 9: Combinadores de parsing statments e statment

```
1 statements = many statement
2
3 statement = atrib
4             <|>
5             try condIf
6             <|>
7             condIfThenElse
8             <|>
9             trycatch
10            <|>
11            throw
12            <|>
13            loop
```

O combinador de parsing *boolexp* está definido utilizando uma tabela de operadores em que definimos a precedência dos operadores e se são associativos à esquerda ou à direita e a forma como realizamos o parsing dos mesmos como se mostra de seguida:

Listing 10: Combinador de parsing boolexp e tabela de precedência dos operadores booleanos

```
1 boolOperators = [ [Prefix (reservedOp' "!" >> return (Not))]
2                  , [Infix (reservedOp' "&&" >> return (BoolBinary And)) AssocLeft,
3                    Infix (reservedOp' "||" >> return (BoolBinary Or)) AssocLeft]
4                  ]
5
6 boolTerm = parens' boolexp
7           <|> (reserved' "true" >> return (BoolConst True))
8           <|> (reserved' "false" >> return (BoolConst False))
9           <|> relexp
10
11 boolexp = buildExpressionParser boolOperators boolTerm
```

A forma de realizar o parsing de expressões inteiras é análoga à forma da das expressões booleanas como se pode verificar pela sua definição:

Listing 11: Combinadores de Parsing das expressões inteiras

```
1 intOperators = [ [Prefix (reservedOp' "-" >> return (Neg))]  
2               , [Infix (reservedOp' "*" >> return (IntBinary Mul)) AssocLeft,  
3                 Infix (reservedOp' "/" >> return (IntBinary Div)) AssocLeft]  
4               , [Infix (reservedOp' "+" >> return (IntBinary Add)) AssocLeft,  
5                 Infix (reservedOp' "-" >> return (IntBinary Sub)) AssocLeft]  
6               ]  
7  
8 intTerm = parens' intexp  
9          <|> liftM Var identifier'  
10         <|> liftM IntConst integer'  
11  
12 intexp = buildExpressionParser intOperators intTerm
```

Os restantes combinadores que irão dar parse de um *statement* são também eles muito simples, sendo o seu código auto-explicativo do seu funcionamento.

Listing 12: Combinadores de parsing attrib

```
1 atrib = do {  
2   id <- identifier';  
3   reservedOp' "=";  
4   v <- intexp;  
5   semi';  
6  
7   return $ Attrib id v  
8 }  
9  
10 condIf = do{  
11   reserved' "if";  
12   b <- parens' boolexp;  
13   reserved' "then";  
14   s <- statments;  
15  
16   return $ If b s  
17 }  
18  
19 condIfThenElse = do{  
20   reserved' "if";  
21   b <- boolexp;  
22   reserved' "then";  
23   s1 <- statments;  
24   reserved' "else";  
25   s2 <- statments;  
26  
27   return $ IfThenElse b s1 s2
```

```
28 }
29
30 trycatch = do {
31   reserved' "try";
32   s1 <- braces' statments;
33   reserved' "catch";
34   s2 <- braces' statments;
35
36   return $ TryCatch s1 s2
37 }
38
39 throw = do {
40   reserved' "throw";
41   semi';
42   return Throw
43 }
44
45 loop = do{
46   reserved' "while";
47   b <- parens' boolexp;
48   symbol' "{";
49   l <- inv;
50   s <- statments;
51   symbol' "}";
52
53   return $ Loop b l s
54 }
55
56 inv = do{
57   reserved' "inv";
58   a <- boolexp;
59   semi';
60
61   return a
62 }
```

Por fim é definida uma função que dado o nome de um ficheiro que contenha um programa escrito na “TinyL” tenta realizar o seu parse para a *Abstract Syntax Tree* da linguagem e que em caso de erro retornará uma mensagem de erro com o detalhe da linha de código em que o parsing falhou.

Listing 13: Parsing de um ficheiro com um programa da “TinyL” para a AST do mesmo

```
1 parseFile :: String -> IO TinyL
2 parseFile f = do {
3   lang <- readFile f;
4   case parse tinyL "" lang of
5     Left e -> print e >> fail "Parse Error"
6     Right r -> return r
```

7 }

4 Geração das Condições de Verificação

A geração das condições de verificação é realizada da forma que se segue:

$$\text{vcg}(\{P\} \ C \ \{Q_n, Q_e\}) = \{ P \rightarrow \text{wp}(C, (Q_n, Q_e)) \} \cup \text{vcaux}(C, (Q_n, Q_e))$$

$$\begin{aligned} \text{vcaux}(\text{skip}, (Q_n, Q_e)) &= \emptyset \\ \text{vcaux}(\text{throw}, (Q_n, Q_e)) &= \emptyset \\ \text{vcaux}(x := e, (Q_n, Q_e)) &= \emptyset \\ \text{vcaux}(\text{if } b \text{ then } Ct, (Q_n, Q_e)) &= \text{vcaux}(Ct, (Q_n, Q_e)) \\ \text{vcaux}(\text{if } b \text{ then } Ct \text{ else } Cf, (Q_n, Q_e)) &= \text{vcaux}(Ct, (Q_n, Q_e)) \cup \text{vcaux}(Cf, (Q_n, Q_e)) \\ \text{vcaux}(\text{whilebdo}\{I\}C, (Q_n, Q_e)) &= \{(I \&\& b) \rightarrow \text{wp}(C, (I, Q_e)), (I \&\& !b) \rightarrow (Q_n, Q_e)\} \cup \\ &\quad \text{vcaux}(C, (I, Q_e)) \\ \text{vcaux}(\text{try}C1\text{catch}C2, (Q_n, Q_e)) &= \text{vcaux}(C1, (Q_n, Q_e)) \cup \text{vcaux}(C2, (Q_n, Q_e)) \\ \text{vcaux}(C1; C2, (Q_n, Q_e)) &= \text{vcaux}(C1, \text{wp}(C2, (Q_n, Q_e))) \cup \text{vcaux}(C2, (Q_n, Q_e)) \end{aligned}$$

$$\begin{aligned} \text{wp}(\text{skip}, (Q_n, Q_e)) &= Q_n \\ \text{wp}(\text{Throw}, (Q_n, Q_e)) &= Q_e \\ \text{wp}(x := e, (Q_n, Q_e)) &= Q_n[x \mapsto e] \\ \text{wp}(\text{if } b \text{ then } Ct, (Q_n, Q_e)) &= (b \rightarrow \text{wp}(Ct, (Q_n, Q_e))) \\ \text{wp}(\text{if } b \text{ then } Ct \text{ else } Cf, (Q_n, Q_e)) &= (b \rightarrow \text{wp}(Ct, (Q_n, Q_e))) \&\& (!b \rightarrow \text{wp}(Cf, (Q_n, Q_e))) \\ \text{wp}(\text{whilebdo}\{I\}C, (Q_n, Q_e)) &= I \\ \text{wp}(\text{try}C1\text{catch}C2, (Q_n, Q_e)) &= \text{wp}(C1, (Q_n, \text{wp}(C2, (Q_n, Q_e)))) \\ \text{wp}(C1; C2, (Q_n, Q_e)) &= \text{wp}(C1, (\text{wp}(C2, (Q_n, Q_e)), Q_e)) \end{aligned}$$

Desta forma, obteremos uma lista de expressões booleanas que teremos que verificar. Para verificarmos as propriedades com o SMTSolver *Z3* teremos então que transformar a lista de condições booleanas, numa lista de condições codificadas para o *Z3* as avaliar. A conversão é realizada da seguinte forma:

Listing 14: Conversão de lista de expressões booleanas para lista de condições a provar no monad do *Z3*

```
1 vcg2Z3 :: MonadZ3 z3 => Set.Set BoolExpr -> [z3 AST]
2 vcg2Z3 s | Set.null s = []
```

```

3         | otherwise = map boolExprToZ3 (Set.toList s)
4
5 vcs :: MonadZ3 z3 => TinyL -> z3 [AST]
6 vcs = sequence . vcg2Z3 . vcg

```

O passo mais importante da conversão é a chamada da função **boolExprToZ3** que irá varrer o tipo de dados das expressões booleanas e o tipo de dados das expressões inteiras que ocorre dentro dele e utilizar os construtores do Z3 para de forma recursiva criar a árvore de prova com que o *Z3* irá trabalhar.

Como a expressão booleana poderá ser construída a partir de expressões inteiras é também desenvolvida a conversão das expressões inteiras para o monad do Z3.

Listing 15: Conversões de uma boolExpr e uma intExpr para o Monad do Z3

```

1 boolExprToZ3 (BoolConst b) | b = mkTrue
2                               | otherwise = mkFalse
3
4 boolExprToZ3 (Not exp) = boolExprToZ3 exp >>= mkNot
5
6 boolExprToZ3 (BoolBinary op exp1 exp2) = do {
7     zexp1 <- boolExprToZ3 exp1;
8     zexp2 <- boolExprToZ3 exp2;
9
10    case op of
11      And -> mkAnd [zexp1, zexp2]
12      Or  -> mkOr [zexp1, zexp2]
13      _   -> mkImplies zexp1 zexp2
14  }
15
16 boolExprToZ3 (RelationalBinary op exp1 exp2) = do {
17     zexp1 <- intExprToZ3 exp1;
18     zexp2 <- intExprToZ3 exp2;
19
20    case op of
21      AST.GT -> mkGt zexp1 zexp2
22      AST.LT -> mkLt zexp1 zexp2
23      AST.GTE -> mkGe zexp1 zexp2
24      AST.LTE -> mkLe zexp1 zexp2
25      AST.EQ -> mkEq zexp1 zexp2
26      AST.Diff -> mkEq zexp1 zexp2 >>= mkNot
27  }
28
29 intExprToZ3 (Var s) = mkFreshIntVar s
30
31 intExprToZ3 (IntConst i) = mkInteger i
32

```

```
33 | intExprToZ3 (Neg exp) = intExprToZ3 exp >=> mkUnaryMinus
34 |
35 | intExprToZ3 (IntBinary op exp1 exp2) = do {
36 |     zexp1 <- intExprToZ3 exp1;
37 |     zexp2 <- intExprToZ3 exp2;
38 |     case op of
39 |         Add -> mkAdd [zexp1, zexp2]
40 |         Sub -> mkSub [zexp1, zexp2]
41 |         Mul -> mkMul [zexp1, zexp2]
42 |         Div -> mkDiv zexp1 zexp2
43 | }
```

Resta agora a verificação das condições de verificação que é realizada da seguinte forma:

Listing 16: Verificação das condições de verificação

```
1 | script :: TinyL -> Z3 [Result]
2 | script x = do
3 |     vc <- vcs x
4 |     mapM (\l -> reset >> assert l >> check) vc
```


5 Apresentação das Condições de Verificação e sua Verificação

Estando toda a maquinaria por detrás montada, resta agora interrogar o utilizador sobre qual o ficheiro com programa da *TinyL* pretende verificar e lançar-lhe para o ecrã as condições de verificação e a sua satisfação ou não.

As condições de verificação, tanto poderão dar *Sat* representando a sua satisfação bem como *Unsat* mostrando que não foi possível verificar as mesmas e *Undef* quando o solver do Z3 não conseguiu fornecer valorações que satisfaçam a condição, nem conseguiu prova que não existe valorações que demonstrem as propriedades.

Desenvolvemos um *pretty-print* para a apresentação das condições de verificação que se mostra de seguida:

Listing 17: Pretty-Print das condições de verificação

```
1  lstPrettyPrint :: [BoolExpr] -> String
2  lstPrettyPrint [] = ""
3  lstPrettyPrint (x:xs) = "\n" ++ prettyPrint x ++ "\n" ++ lstPrettyPrint xs
4
5  prettyPrint :: BoolExpr -> String
6  prettyPrint (BoolConst b) = show b
7  prettyPrint (Not exp) = "! (" ++ prettyPrint exp ++ " )"
8  prettyPrint (BoolBinary op exp1 exp2) = prettyPrint exp1 ++ ppBoolOp op ++
   prettyPrint exp2
9  prettyPrint (RelationalBinary op exp1 exp2) = ppIntExpr exp1 ++ ppRelOp op ++
   ppIntExpr exp2
10
11 ppIntExpr :: IntExpr -> String
12 ppIntExpr (Var s) = s
13 ppIntExpr (IntConst i) = show i
14 ppIntExpr (Neg exp) = "- " ++ ppIntExpr exp
15 ppIntExpr (IntBinary op exp1 exp2) = ppIntExpr exp1 ++ ppIntOp op ++ ppIntExpr exp2
16
17 ppBoolOp :: BoolBinOp -> String
18 ppBoolOp And = " && "
19 ppBoolOp Or = " || "
20 ppBoolOp Implies = " ==> "
21
22 ppIntOp :: IntBinOp -> String
23 ppIntOp Add = " + "
24 ppIntOp Sub = " - "
25 ppIntOp Mul = " * "
26 ppIntOp Div = " / "
27
```

```
28 ppRelOp :: RelBinOp -> String
29 ppRelOp AST.GT = " > "
30 ppRelOp AST.LT = " < "
31 ppRelOp AST.GTE = " >= "
32 ppRelOp AST.LTE = " <= "
33 ppRelOp AST.EQ = " == "
34 ppRelOp AST.Diff = " != "
```

Dado o nome do ficheiro ao programa por parte do utilizador é dado o *pretty-print* das condições de verificação geradas e apresenta-se o resultado da verificação dessas condições.

Listing 18: Função main da aplicação desenvolvida

```
1 main :: IO ()
2 main = do
3   putStrLn "Nome do ficheiro a dar parse:"
4   f <- getLine
5   tiny <- parseFile f
6   putStrLn $ lstPrettyPrint (Set.toList $ vcg tiny)
7
8   result <- evalZ3 $ script tiny
9   mapM_ print result
```

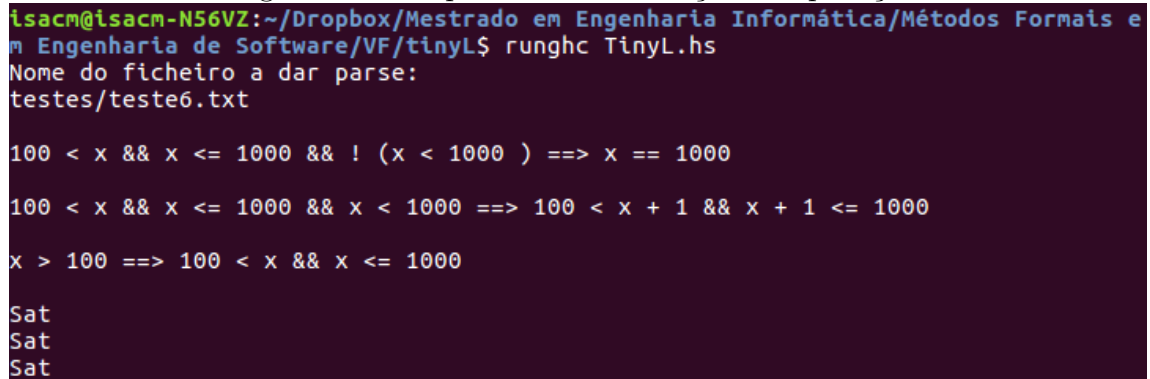
Resultando numa utilização em que dado por exemplo o seguinte programa escrito na “TinyL” de entrada a aplicação

Listing 19: exemplo de programa escrito na TinyL

```
1 pre x > 100;
2
3 while (x < 1000) {
4     inv 100 < x && x <= 1000;
5
6     x = x + 1;
7 }
8
9 posn x == 1000;
10 pose false;
```

Resultará o seguinte output.

Figura 1: Exemplo de uma utilização da aplicação.



```
isacm@isacm-N56VZ:~/Dropbox/Mestrado em Engenharia Informática/Métodos Formais e
m Engenharia de Software/VF/tinyL$ runghc TinyL.hs
Nome do ficheiro a dar parse:
testes/teste6.txt

100 < x && x <= 1000 && ! (x < 1000 ) ==> x == 1000

100 < x && x <= 1000 && x < 1000 ==> 100 < x + 1 && x + 1 <= 1000

x > 100 ==> 100 < x && x <= 1000

Sat
Sat
Sat
```

6 Conclusão

O grupo desenvolveu as metas propostas pelos docentes que conferem ao trabalho uma avaliação de no máximo 18 valores, pois não desenvolvemos uma GUI para a aplicação nem realizámos a utilização da interface *Why3* para a gestão da interface com diversas ferramentas de prova.

Contudo houveram algumas conceções no desenvolvimento da pré-condição, invariante e das pós-condições, pois desenvolvemos as mesmas apenas como expressões booleanas, não lhes sendo permitido a utilização de elementos como \forall , \exists ou operador *old*.

O projeto foi desenvolvido em *Haskell* permitindo assim possuir um código de fácil leitura e interpretação que permite a quem quer que o olhe perceber o que é feito e a forma como é realizado, juntando para além disso a vantagem de um código mais compacto do que seria utilizando as restantes opções que nos foram fornecidas para o desenvolvimento da mesma.

Desta feita, é nosso entender possuímos um trabalho que vai de acordo às exigências da Unidade Curricular e que funciona como um *proof-of-concept* sobre a forma como as condições de verificação devem ser realizadas e reforça a explicação do porquê da necessidade de pré-condições, invariantes e pós-condições quando realizamos na UC verificação de programas com ferramentas de verificação estática de código como o *frama-C*.