Manutenção e Evolução de Software - PicoC

Diogo Camacho Barbosa
Universidade do Minho
pg50326@uminho.pt

João Tiago Sousa
Universidade do Minho
pg50500@uminho.pt

I. Introdução

Foi-nos proposta a criação do PicoC, uma linguagem de programação de alto nível com uma sintaxe similar à da linguagem C, incluindo um parser, a gramática e a execução da mesma. Para além disso foi-nos pedido para implementar técnicas de programação estratégica para criar um módulo de testes para esta linguagem, capaz de gerar e correr testes sobre o PicoC.

No documento que se segue serão apresentados os detalhes do desenvolvimento e implementação do PicoC.

II. Linguagem

A. Gramática

Encontra-se abaixo a gramática do PicoC, definida em EBNF[1] (Extended-BNF):

```
PicoC: Inst+
Inst: Attrib | Print | While | ITE | Return
Attrib: Names "=" Exp ";"
Print: "print" "(" Exp ")"
While: "while" Exp "{" CBlock "}"
Return: "return" Exp
ITE: "if" "(" Exp ")" "then" "{" CBlock "}" else?
else: "else" "{" CBlock "}"
CBlock: Inst+

Exp: Exp5
Exp5: Exp4 "||" Exp5
    | Exp4
Exp4: Exp3 "&&" Exp4
    | Exp3
Exp3: Exp2 "==" Exp3
    | Exp2 token' "!=" Exp3
    | Exp2
Exp2: Exp1 "<" Exp2
     | Exp1 ">" Exp2
     | Exp1 ">=" Exp2
     | Exp1 "<=" Exp2
     | Exp1
Exp1: Exp0 "+" Exp1
     | Exp0 "-" Exp1
     | Exp0
Exp0: Factor '*' Exp0
     | Factor '/' Exp0
     | Factor
Factor: Int
      | Names
      | "TRUE"
      | "FALSE"
      | '-' Factor
      | '!' Factor
      | '(' Exp ')'

Int: /-?[0-9]+/
Names: /[a-z]\w+/
```

B. Descrição e Funcionalidades Suportadas

O PicoC é uma linguagem de programação que consiste numa série de instruções.

Uma instrução pode ser uma atribuição `Attrib`, um ciclo `While`, um condicional `if then else`, sendo o `else` opcional, um `return` ou um `print`.

- `Attrib`: faz uma atribuição de uma `exp` a uma variável
- `While`: executa o seu corpo (`CBlock`) n vezes, até que a sua condição (representada por uma `exp`) seja falsa
- `ITE`: avalia uma condição (representada por uma `exp`) e se for verdadeira executa o corpo do `then` e se for falsa, executa o corpo do `else` se existir, se não o programa simplesmente continua
- `Return`: devolve como output o resultado
- `Print`: é uma instrução utilizada para debug e instrumentalização do código

III. Parse e Unparse

A. Biblioteca de parsing

Durante o semestre foi desenvolvido uma biblioteca de parsing genérica em Haskell baseada em gramáticas do tipo EBNF. Este parser operadores para mapear o texto processado para estruturas de Haskell, por exemplo:

- <|> : alternativa de parsing
- <*> : sequência durante o parsing
- <$> : aplicação de uma função ao conteúdo que foi parsed

Segue-se um exemplo do parser do PicoC destes operadores a serem utilizados:

[1]Extended-BNF: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

```
pExp3 :: Parser Exp
pExp3 = f <$> pExp2 <*> token' "==" <*> pExp3
        <|> g <$> pExp2 <*> token' "!=" <*> pExp3
        <|> pExp2
  where
    f x _ = EQ x
    g x _ = DIFF x
```

A invocação de `pExp3 "a == 4"` retornaria o valor de `EQ (Var "a") (Const 4)`

B. *Parser do PicoC*

Foi construído um parser para a linguagem PicoC com base na biblioteca de parsing mencionada previamente. Este parser recebe como input uma String correspondente a um programa escrito em PicoC e devolve um PicoC:

```
picoC :: String -> PicoC
```

O parser abrange todas as instruções do PicoC mencionadas acima, transformando o texto processado em estruturas de dados em memória.

Foi prestada especial atenção para o parser das expressões que requer uma ordem explícita de parsing para manter a coerência e correção dos programas no que toca a expressões lógicas e numéricas. Por isso a precedência dos operadores em PicoC foi baseada nas precedências utilizadas na linguagem C[2].

C. *Unparser*

O Unparse é uma função que recebe um PicoC retorna a String correspondente. O instance Show do PicoC foi definido como:

```
instance Show PicoC where
    show = unparse

unparse :: PicoC -> String
```

Desta forma uma instância de PicoC pode ser imprimida diretamente num formato muito semelhante ao que estava antes de passar pelo parser.

IV. QuickCheck

A. *Geradores*

Foram desenvolvidos alguns geradores de código PicoC usando o `Gen` do QuickCheck. Foram gerados programas através da função `genPicoC :: Gen PicoC`, que gera aleatoriamente programas PicoC válidos e representativos da linguagem, porém nada garante a sua correção.

Ao gerar programas PicoC, foi considerado como representativo uma rácio de 90% de atribuições, 5% de ITE (`if then else`) e 5% de ciclos `While`. Para além disso as expressões geradas para cada uma das instruções são dirigidas por um rácio. Inicialmente é dada uma probabilidade inferior às constantes, e às operações aritméticas e lógicas é dada uma probabilidade maior, sendo que esta vai diminuindo ao longo do processo de geração. Desta forma o programa gera expressões variadas, com uma complexidade razoável, não demasiado grande.

B. *Propriedades*

Para desenvolver as propriedades de teste da linguagem, foi usada a biblioteca QuickCheck. Para testar a correção das funcionalidades do programa, foram criadas propriedades da linguagem PicoC.

C. *Propriadade de validade*

```
prop_valid :: PicoC -> Bool
prop_valid p = p == picoC (unparse p)
```

D. *Propriedade de idempotência do Refactor*

```
prop_refactor_idempotent :: PicoC -> Bool
prop_refactor_idempotent p = refactor p == (refactor
(refactor p))
```

E. *Propriedade de idempotência das Otimizações*

```
prop_optimization_idempotent :: PicoC -> Bool
prop_optimization_idempotent p = arithmeticOpt p ==
arithmeticOpt (arithmeticOpt p)
```

V. Refactoring

Foi desenvolvido código para efetuar refactor de programas PicoC. Este refactor tem como objetivo eliminar alguns *code smells* que aparecem frequentemente em linguagens de programação. Foram considerados os seguintes *code smells*:

A. *If's aninhados*

Este *code smell* acontece quando existe existem duas instruções `if` aninhadas em que o corpo do primeiro `if` é somente outra instrução do tipo `if`,e em que nenhuma delas contém um `else`. Neste caso as expressões de ambas as instruções `if` podem ser combinadas com o operador "&&" numa só instrução `if`.

Encontra-se abaixo um exemplo do refactor deste *code smell*.

```
//Antes:
if(a) then {
  if(b) then {
    a = 0;
  }
}
//Depois:
if(a && b) then{
  a = 0;
}
```

B. *Comparaçao Lógica redundante*

[2]C Operator Precedence: https://en.cppreference.com/w/c/language/operator_precedence

Este *code smell* observa-se quando existe uma expressão lógica que contém uma operação redundante. Segue-se um exemplo de um possível refactor deste *code smell*.

```
//Antes:
if(a && TRUE) then {
  a = 0;
}
//Depois:
if(a) then{
  a = 0;
}
```

C. Expressões Lógicas com Not

Este *code smell* consiste consiste na composição de uma instrução do tipo `if then else` em que a condição do `if` contém o operador lógico `Not`. Este tipo de instruções adicionam complexidade desnecessária ao programa e tornam o código elegível e mais difícil de compreender/manter. Este *code smell* pode ser eliminado através da remoção do operador `Not` e inversão do corpo do `then` e do `else`.

Segue-se um exemplo de um possível refactor deste *code smell*.

```
//Antes:
if(!(a>10)) then {
  a = 0;
} else {
  a = 1;
}
//Depois:
if(a>10) then {
  a = 1;
} else {
  a = 0;
}
```

VI. Suite de Testes

A. Execução de Programas

1) *Avaliação de Expressões:* Decidimos que todas as expressões resultam em inteiros, incluindo as lógicas. Para representar *boleanos* é assumido que 0 representa falso e qualquer outro número representa verdadeiro.

A avaliação é feita pela função:

```
eval :: Exp -> [(String, Int)] -> Int
```

2) *Evaluate:* Para executar os programas definimos a função

```
evaluate :: PicoC -> Inputs -> Int
evaluate (PicoC i) inp = ret
  where
    (c, Just ret) = runPicoC i inp
```

que recorrendo à função

```
runPicoC :: [Inst] -> Inputs -> (Inputs, Maybe Int)
```

executa e extrai o resultado do *return* da execução do programa.

B. Execução da Test Suite

Definimos 3 funções de modo a executar uma Test Suite.

```
runTestSuite :: PicoC -> [(Inputs, Int)] -> Bool
runTestSuite p l = and $ runTests p l

runTests :: PicoC -> [(Inputs, Int)] -> [Bool]
runTests _ [] = []
runTests p (h : t) = runTest p h : runTests p t

runTest :: PicoC -> (Inputs, Int) -> Bool
runTest p (i, result) = evaluate p i == result
```

C. Programas Escolhidos

1) *Programa 1:* Subtração entre 2 números positivos, -1 em caso de erro de input

```
if((a >= 0) && (b >= 0)) then {
  if(a>b) then {
    c = a - b;
  } else {
    c = b - a;
  }
} else {
  c = -1;
}
return c;
```

2) *Programa 2:* Conta arbitrária

```
if (a>10)then {
  a = a * 10;
} else {
  a = a - 10;
}
return a;
```

3) *Programa 3:* Maior de 3 números, resultado em m

```
if (a > b) then {
  if (a > c) then {
    m = a;
  } else {
    m = c;
  }
} else {
  if (b > c) then {
    m = b;
  } else {
    m = c;
  }
}
return m;
```

D. Mutações

Foi desenvolvido código que insere mudanças aleatórias no código. Estas mutações foram obtidas através da utilização da função `mutations`, que de acordo com uma função de mutação, percorre uma estrutura de dados (neste caso o PicoC) e devolve uma lista de inputs modificados.

O output de `mutations` é uma lista de possíveis combinações de código alterado, então é escolhido um dos elementos dessa lista de forma aleatória. Também existe a opção de fazer o mesmo, mas recorrendo a uma semente para um resultado pseudo-aleatório para garantir consistência e ajudar em testes futuros.

Estas mudanças introduzidas são, na verdade, uma introdução deliberada de falhas, que têm o objetivo de servir de casos de teste aleatório para o programa. Desta forma, podemos partir de um programa correto, inserir um erro aleatório e obter um programa com uma falha, que já não passa nos testes.

Segue-se a função utilizada para introduzir uma falha do código:

```
breakCode :: Inst -> Maybe Inst
breakCode (Attrib v n) = Just $ Attrib v (Neg n)
breakCode (While exp b) = Just $ While (Not exp) b
breakCode (ITE exp t e) = Just $ ITE (Not exp) t e
breakCode _ = Nothing
```

E. Instrumentação

Para a instrumentação dos programas foi usada a função `instrumentation`, que com o auxílio da função `instrumentationInst`, percorre o programa recursivamente e insere uma instrução `Print` antes de todas as instruções do programa de modo a imprimir quais as instruções que são executadas.

```
instrumentedTestSuite :: PicoC -> [(Inputs, Int)] ->
[Bool]
instrumentedTestSuite p i = runTests (instrumentation
p) i
```

Esta função executa uma Test Suite após a instrumentação do programa.

Devido à natureza *Lazy* do Haskell decidimos usar a função `runTests` ao invés da função `runTestSuite`, que finaliza a execução após o primeiro teste falhado. Isto garante que todos os testes são executados para que a localização de falhas seja possível.

Para executar os testes com mutações definimos a função `runMutationSuite` que a partir de uma seed cria mutação aleatória do programa input, e executa-o.

```
runMutationSuite :: PicoC -> [(Inputs, Int)] -> Int
-> IO [Bool]
```

VII. Localização de Falhas

Após as execuções dos 3 programas escolhidos com uma mutação aleatória, agregamos os resultados numa folha de cálculo, e aplicamos a técnica *Spectrum Based Fault Localization* com o coeficiente de *Jaccard*

	T1	T2	T3	Coef
if (((a > -1) && (b > -1))) then {	1	1	1	0.3333333333
if (!((a > b))) then {	1	1	0	0.5
c = (a - b);	1	1	0	0.5
}else {				
c = (b - a);	0	0	0	0
}				
}else {				
c = -1;	0	1	1	0
}				
return c;	1	1	1	0.3333333333
Fail	1	0	0	

Figura 1: *Spectrum based fault localization* do Programa 1

No programa 1 podemos concluir que o erro está na linha 2 ou 3. Analisando o programa original, descrito na Secção VI.C.1, verificamos que a mutação ocorreu na linha 2.

	T1	T2	T3	Coef
if ((a > 10)) then {	1	1	1	0.6666666667
a = (a * 10);	1	0	0	0
}				
else {				
a = -((a - 10));	0	1	1	1
}				
return a;	1	1	1	0.6666666667
Fail	0	1	1	

Figura 2: *Spectrum based fault localization* do Programa 2

No programa 2 podemos concluir que o erro está na linha 5, e após observar o programa original, descrito na Secção VI.C.2, isto é provado.

	T1	T2	T3	Coef
if (!((a > b))) then {	1	1	1	0.6666666667
if ((a > c)) then {	0	1	0	0.5
m = a;	0	0	0	0
} else {				
m = c;	0	1	0	0.5
}				
} else {				
if ((b > c)) then {	1	0	1	0.3333333333
m = b;	0	0	0	0
} else {				
m = c;	1	0	0	0
}				
}				
return m;	1	1	1	0.6666666667
Fail	0	1	1	

Figura 3: *Spectrum based fault localization* do Programa 3

No programa 3, assumindo que o erro não se encontra no `return`, podemos concluir que está na linha 1. Após observar o programa original, descrito na Secção VI.C.3, isto é provado.

VIII. Conclusão

No desenvolvimento do projeto PicoC, foram abordadas diversas etapas essenciais para a criação e validação de uma linguagem de programação de alto nível. A construção deste projeto envolveu a implementação de um parser e um unparser para a linguagem, a definição de geradores e propriedades de teste utilizando QuickCheck, bem como a criação de mecanismos de refactoring e otimização de código.

Em suma, através da criação do PicoC, foi possível averiguar a importância da utilização de técnicas de análise estática de código, como o refactor, também foi destacada a importância da realização de testes sobre os programas. Conclui-se também que este processo de testagem é mais simples e fidedigno quando é feito através de uma suite de testes bem definida, com geração de testes automática e complementada por técnicas de deteção de falhas como Spectrum Based Fault Localization.