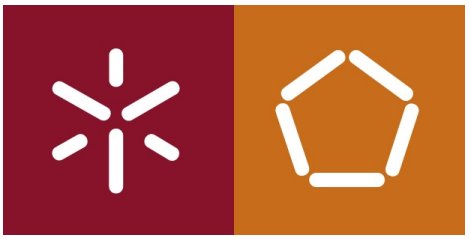

Universidade do Minho
Mestrado em Engenharia Informática
1ºano - 2º Semestre

Manutenção e Evolução de Software

Teste e propriedades de um processador de linguagem

A85635 - André Nunes
PG50495 - João Silva
PG47447 - Marco Sampaio

Junho, 2023

Conteúdo

Introdução

No âmbito da unidade curricular de **Manutenção e Evolução de Software** foi nos pedido que desenvolvesse-mos um front-end para uma linguagem à la BC do linux com o intuito de futuramente averiguar de que modo poderiamos gerar testes que verificassem algumas propriedades definidas e de que modo poderiamos otimizar a mesma linguagem. Assim sendo o projeto foi dividido em 2 partes. Na primeira fase do projeto foi desenvolvido um front-end para uma linguagem a la BC do linux, implementas algumas optimizações no código fonte (optimização de expressões aritméticas e lógicas) utilizando programação estratégica e a deteção e eliminação de code smells. Na segunda fase pretendeu-se utilizar testes baseados em propriedades para testar o processador de linguagem desenvolvido.

Linguagem

De modo desenvolver a linguagem à la BC do linux o grupo definiu primeiro uma gramática rudimentar de forma a criar um fio condutor para a produção da mesma.

```
Exp      = Add Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Mul Exp Exp
         | Less Exp Exp
         | Greater Exp Exp
         | Equals Exp Exp
         | GTE Exp Exp
         | LTE Exp Exp
         | And Exp Exp
         | Or Exp Exp
         | Not Exp
         | Const Int
         | Var String
         | Inc Exp
         | Dec Exp
         | Return Exp
         | Bool Bool

Item     = Decl String Exp
         | Arg Exp
         | Increment Exp
         | Decrement Exp
         | NestedIf If
         | OpenIf If
         | NestedWhile While
         | OpenWhile While
         | NestedLet Let
         | OpenLet Let
         | NestedFuncao Funcao
         | OpenFuncao Funcao
         | NestedReturn Exp
```

```
Funcao      = Funcao Name Args
            | DefFuncao Name Args Items

If          = If Exp Items
            | Else Exp Items Items

Let         = Let Items Exp

Name        = Name String

While       = While Exp Items

Items       = [Item]

Args        = [Item]
```

Como podemos ver, a nossa lingugem oferece a possíbilidade ao utilizador de declarar funcões, fazer operacoes lógicas e aritméticas, fazer ciclos while, ou frases condicionais bem como as típicas declarações. Com esta gramática definida o grupo começou então a desenvolver o parser que traduz a nossa linguagem para árvores de derivação as quais são depois usadas para implementar otimizações, identificar code-smells ou até mesmo fazer testes de propriedades.

Parser

Recorrendo ao conhecimento adquirido nesta unidade curricular acerca dos combinadores de parsing em Haskell o grupo produziu um front-end para a nossa linguagem que nos permitia transformar uma composição dessa mesma linguagem numa árvore de derivação e vice-versa. Como podemos ver o exemplo de código a seguir descrito,

```
def main(args){
    a=100;
    x=0;
    while(a>b){
        x++;
        if(!a>b){
            a--;
        };
    };

    let{
        a=a+b;
    }in c;

    func coco();
    return a;
}
```

é convertido para a seguinte árvore de derivação.

[(OpenFuncao (DefFuncao (Name "main") [Arg (Var "args")] [Decl "a"(Const 100),Decl "x"(Const 0),NestedWhile (While (Greater (Var "a") (Var "b")) [Increment (Var "x"),NestedIf (If (Not (Greater (Var "a") (Var "b"))) [Decrement (Var "a")])]),NestedLet (Let [Decl "a"(Add (Var "a") (Var "b"))] (Var "c")),NestedFuncao (Funcao (Name "coco") []),NestedReturn (Return (Var "a"))]),)]

A qual pode ser de novo convertida no código fonte.

Otimizações

Depois de desenvolvido o parser da nossa nova linguagem, começamos então por otimizar a mesma. Para isto o grupo optou por fazer otimização das expressões aritméticas, como a existência de elemento neutro e nulo das operações e otimização das expressões lógicas de modo a efetuar o contition coverage da nossa linguagem. Estas otimizações foram implementadas com recurso a **Zippers** e programação estratégica.

Os **Zippers** foram originalmente concebidos para representar uma árvore juntamente com uma subárvore que é o foco da atenção. Durante uma computação, o foco pode mover-se para a esquerda, para cima, para baixo ou para a direita dentro da árvore. A manipulação genérica de um **Zipper** é fornecida através de um conjunto de funções predefinidas que permitem o acesso a todos os nós da árvore para inspecção ou modificação.

As funções de programação estratégica da biblioteca **"Ztrategic"** permitem aplicar estas otimizações recorrendo a várias estratégias de travessias de àrvores. Posto isto o grupo desenvolveu as ditas otimizações e duas funcões que as aplicam com diferentes estratégias juntando estes dois ingredientes.

```
--full top down strategie
optWithZippersTD lista =
    let listaZipper = toZipper lista
        Just listaNova = applyTP (full_tdTP step) listaZipper
            where step = idTP `adhocTP` optItemz
        in
        fromZipper listaNova

--full bottom up strategie
optWithZippersBU lista =
    let listaZipper = toZipper lista
        Just listaNova = applyTP (full_buTP step) listaZipper
            where step = idTP `adhocTP` optItemz
        in
        fromZipper listaNova
```

Estas duas funções percorrem as árvores de derivação aplicando as otimizações definidas de forma automática. A primeira utiliza uma estratégia **"Full topdown"**, enquanto que a segunda utiliza uma estratégia **"Full BottomUp"**.

Tendo em conta a seguinte àrvore:

- treeTest = [(OpenFuncao (DefFuncao (Name "mht") [Arg (Const 15),Arg (Var "wbk"),Arg (Const 6)] [Arg (Add (Const 0) (Var "qhpzs"))]),)]

Podemos aplicar uma destas funções (uma vez que são equivalentes como vamos provar depois) obtendo o seguinte resultado:

- treeTest = [(OpenFuncao (DefFuncao (Name "mht") [Arg (Const 15),Arg (Var "wbk"),Arg (Const 6)] [Arg (Var "qhpzs")]),)]

Como podemos ver a soma de uma variável pela constante 0 foi otimizada para apenas a declaração dessa mesma variável, cobrindo assim a otimização do elemento neutro da soma.

Teste e Propriedades

De modo a desenvolver testes baseados em propriedades o grupo implementou 3 componentes para a geração de testes automáticos da linguagem desenvolvida. Estes componentes consitem num gerador de árvores sintáticamente corretas, um gerador de mutantes e por fim um conjunto de propriedades para várias componentes do processador da linguagem. Todas estas componentes foram desenvolvidas com o auxílio da biblioteca **"QuickCheck"** que é uma excelente biblioteca **"Haskell"** que permite expressar propriedades sobre programas e testá-las com valores gerados aleatoriamente.

Gerador de árvores

Relativamente à implementação do gerador de árvores de derivação da nossa linguagem o grupo implementou geradores de todos os tipos da nossa linguagem de acordo com algumas regras de semântica e sintaxe da nossa linguagem.

- As frases da nossa linguagem devem estar sempre declaradas dentro de uma função.
- Declarações de return nunca podem estar em condições de ciclos While ou de If-statements.
- Todas as árvores devem começar com um token "OpenFunção".
- Não se podem declarar funções dentro de funções.
- Argumentos de funções apenas podem ser variáveis, constantes ou chamadas de funções.
- Não se podem gerar divisões em que o divisor é a constante zero.
- Não se podem efetuar operações aritmeticas em expressões lógicas.

Tendo em conta este conjunto de regras básicas o grupo desenvolveu um gerador de árvores sintáticamente corretas o qual pode ser usado apenas compilando o ficheiro "generator.hs"e usando o comando "gen"que gera uma série de àrvores aleatórias que podem ser usadas postriormente numa bateria de testes.

Gerador de mutantes

No sentido de implementar o gerador de mutantes da nossa linguagem o grupo começou por definir um conjunto de mutantes prefefinidos e de seguida implementar a maquinaria necessária para os aplicar de forma automática e aleatória. Os mutantes definidos foram os seguintes:

- Transformar as adições em subtrações.
- Transformar as multiplicações em divisões.
- Trocar os elementos das subtrações.
- Trocar os elementos das divisões.
- Trocar os elementos da comparação lógica "menor".

Depois da definição destes mutantes o grupo definiu uma função que aplica um dos mutantes a uma árvore de derivação recorrendo a uma estratégia **"Full Top-Down"** e de seguida desenvolveu uma funcão que aplica aleatóriamente um dos mutantes a uma das árvores.

```
--Function that aplies one given mutant with full_tdTP strategie
mutant f lista =
    let listaZipper = toZipper lista
        Just listaNova = applyTP (full_tdTP step) listaZipper
            where step = idTP `adhocTP` f
    in
    fromZipper listaNova

--Function that aplies a random mutant with full_tdTP strategie
applyRandomMutant :: [(Item, String)] -> IO [(Item, String)]
applyRandomMutant tree = do
  randomIndex <- generate (choose (1, 5) :: Gen Int)
  case randomIndex of
    1 -> return (mutant m1 tree)
    2 -> return (mutant m2 tree)
    3 -> return (mutant m3 tree)
    4 -> return (mutant m4 tree)
    5 -> return (mutant m5 tree)
    _ -> error "Invalid mutant index"
```

Na primeira função recorremos aos Zippers e à biblioteca **"Ztrategic"** para aplicar os mutantes do mesmo modo que aplicamos as otimizações na primeira parte do projeto. Na segunda recorremos ao **QuickCheck** para gerar um número aleatório que dita qual o mutante a aplicar. Posto isto, este gerador de mutantes pode ser usado postriormente para criar uma bateria de testes que tente matar esses mesmos mutantes.

Propriedades

Após estas duas tarefas concluídas o nosso trabalho relativamente a testar o processador de linguagem tornou-se bastante mais simples uma vez que apenas temos que definir propriedades que o processador deve ou não cumprir. Posto isto o grupo definiu as seguintes propriedades:

- Testar se diferentes estratégias de otimização de expressões aritméticas são equivalentes (passed).
- Testar se uma expressão fica igual após otimizações e eliminação de smells (trivialmente falsa).
- Testar se uma expressão fica igual após eliminação de smells (trivialmente falsa).
- Testar se eliminação de smells e a optimização com a estratégia (full bottom up) e manual de expressões aritméticas são equivalentes (passed).
- Testar se eliminação de smells e a optimização de com a estratégia (full top down) e manual de expressões aritméticas são equivalentes (passed).
- Testar se a eliminação de smells e a optimização de expressões aritméticas são comutativas (passed).

Como podemos concluir pela implementação destes testes baseados em propriedades e os respetivos resultados o nosso processador de linguagem está completamente operacional e todos os componentes funcionam como era esperado.

Conclusões e trabalho futuro

Em suma, o projeto proposto foi desenvolvido com sucesso, cumprindo todos os objetivos propostos. O processador da linguagem e o respetivo front-end funcionam como esperado produzindo árvores de derivação de qualquer composição sintáticamente correta da nossa linguagem e fazendo a tradução das mesmas para a linguagem original. Foram otimizadas todas as operações aritméticas e lógicas recorrendo a Zippers e programação estratégica. O gerador de Mutantes consegue aplicar de forma aleatória um mutante da nossa lista predefenida a qualquer árvore de derivação. Finalmente todas as propriedades testadas foram validadas pelos testes automáticamente gerados do QuickCheck, o que garantiu a qualidade e operacionalidade de todos os componentes desenvolvidos neste projeto. Identificamos, possibilidades de melhorias e expansões futuras para o processador. Uma das funcionalidades que poderíamos considerar é o gerador garantir que as regras de definição/uso de nomes da linguagem são respeitadas e fazer o pretty printing do código após ter sido convertido de uma árvore de derivação.