Desenrolando Funções Recursivas

```haskell
module Unrolling where
import Prelude hiding (foldr, map)
exampleList = [92, 55, 20, 57, 60, 43, 77, 81, 76, 83,
               5, 69, 84, 64, 52, 45, 45, 93, 6, 24, 9,
               43, 53, 18, 93, 40, 57, 28, 59, 62] :: [Int]
```

No artigo *Recursion Unrolling for Divide and Conquer Programs* de Rugina e Rinard (2000) temos um algoritmo para desenrolar chamadas recursivas até um certo fator de expansão, com o objetivo de melhorar o desempenho do código compilado, que potencialmente pode ser melhorado por outras etapas de otimização do compilador. O método no artigo leva em consideração uma linguagem imperativa.

Usaremos as idéias desse trabalho para pensar como desenrolar chamadas recursivas de uma linguagem funcional simples, com o objetivo de eliminar a recursão em si, garantindo a terminação da execução de uma chamada mesmo que seja abortá-la após atingir o fator de expansão.

Inlining

Tendo uma função recursiva f, é de se esperar que ela possua um caso base e um caso propriamente recursivo, onde a ramificação acontece por uma condicional ou casamento de padrão. Com duas cópias dessa função, f_1 e f_2, que executem a mesma computação da mesma forma, mudando apenas o nome da função e das chamadas recursivas, podemos fazer um passo de expansão. Ao transformarmos as chamadas de f_1 para f_1 em chamadas para f_2, e transformarmos as chamadas de f_2 para f_2 em chamadas para f_1, ou seja, criando funções **mutualmente** recursivas, podemos fazer o inlining de f_2 em f_1, e assim obtendo uma função recursiva com dois casos base em que o segundo é maior, e um caso recursivo que faz mais chamadas que f porém cada uma de tamanho menor.

Vamos criar um exemplo tendo em vista uma linguagem funcional. Para isso, utilizaremos um subconjunto de Haskell. Considere a função abaixo que constrói uma lista com os sucessores de uma lista de entrada:

```haskell
ex1 :: [Int] -> [Int]
ex1 ls = case ls of
           []     -> []
           (x:xs) -> (x + 1) : ex1 xs
```

Ela poderia ser escrita de outra forma, utilizando `foldr` e `map`, mas mesmo com uso dessas funções de ordem superior ainda haveria uma recursão equivalente acontecendo.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v ls = case ls of
```

```
                []      -> v
                (x:xs) -> f x (foldr f v xs)
```

```
ex1' :: [Int] -> [Int]
ex1' = foldr (\x xs -> (x+1):xs) []
```

```
map :: (a -> b) -> [a] -> [b]
map f = foldr (\x xs -> (f x):xs) []
```

```
ex1'' :: [Int] -> [Int]
ex1'' = map (+ 1)
```

Ou seja, escrever com essas funções não elimina o fato de que há chamadas recursivas acontecendo, por isso vamos considerar a forma com a recursão explícita. Seria até mesmo possível provar por indução estrutural sobre a lista de entrada que as três formas `ex1`, `ex1'` e `ex1''` são equivalentes.

Tendo em mãos agora uma cópia de `ex1`:

```
ex2 :: [Int] -> [Int]
ex2 ls = case ls of
           [] -> []
           (x:xs) -> (x+1) : ex2 xs
```

Vamos criar suas versões mutualmente recursivas:

```
ex1M :: [Int] -> [Int]
ex1M ls = case ls of
            [] -> []
            (x:xs) -> (x+1) : ex2M xs
```

```
ex2M :: [Int] -> [Int]
ex2M ls = case ls of
            [] -> []
            (x:xs) -> (x+1) : ex1M xs
```

E fazer o inlining de uma na outra:

```
ex1M' :: [Int] -> [Int]
ex1M' ls = case ls of
             [] -> []
             (x:xs) -> (x+1) :
               case xs of
                 [] -> []
                 (y:ys) -> (y+1) : ex1M' ys
```

Poderíamos repetir o processo fazendo o inlining de `ex1M'` com `ex1` para obter mais um grau de expansão:

```
ex1M'' :: [Int] -> [Int]
ex1M'' ls = case ls of
              [] -> []
```

```
             (x:xs) -> (x+1) :
               case xs of
                 [] -> []
                 (y:ys) -> (y+1) :
                   case ys of
                     [] -> []
                     (z:zs) -> (z+1) : ex1M'' zs
```

Supondo que nosso fator de expansão fosse exatamente 3, poderíamos encerrar o último case com um erro, já que a função seria abortada se chegasse ali.

```
ex1Er :: [Int] -> [Int]
ex1Er ls = case ls of
             [] -> []
             (x:xs) -> (x+1) :
               case xs of
                 [] -> []
                 (y:ys) -> (y+1) :
                   case ys of
                     [] -> []
                     (z:zs) -> error "provide more fuel"
```

A execução de `ex1Er` sobre listas de tamanho $0 \leq n \leq 2$ acontece exatamente como nas funções `ex1` e suas equivalentes, mas para qualquer lista de tamanho $n > 2$ lançamos um erro. Note que `ex1Er` **não** é recursiva.

Fusão de Condicionais/Casamentos

O exemplo acima é simples porque `ex1` é uma recursão com apenas uma chamada recursiva. Um exemplo com árvore ilustra uma situação com mais de uma chamada recursiva:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
                deriving (Eq, Show)

exampleTree = (Node 5 (Node 3 Empty Empty) (Node 7 Empty Empty)) :: Tree Int

sucTree :: Tree Int -> Tree Int
sucTree ts = case ts of
               Empty -> Empty
               Node x lt rt -> Node (x+1) (sucTree lt) (sucTree rt)

sucTree' :: Tree Int -> Tree Int
sucTree' ts = case ts of
               Empty -> Empty
               Node x lt rt -> Node (x+1)
                 (case lt of
                    Empty -> Empty
                    Node x lt1 rt1 -> error "provide more fuel")
```

```
                  (case rt of
                     Empty -> Empty
                     Node x lt2 rt2 -> error "provide more fuel")
```

Apesar de aumentar o código ainda mais, o método não se tornou mais complicado por isso e os `case of` não ficaram redundantes de nenhuma forma, todos eles testam condições distintas. Note que uma árvore poderia ser percorrida com `foldr` também.

E o que acontece se forçarmos um exemplo ruim de recursão, como a definição abaixo?

```
ex3 :: [Int] -> [Int]
ex3 ls = case ls of
           []  -> []
           [x] -> [x+1]
           xs  -> [(head ls) + 1] ++
                    ex3 (init . tail $ ls) ++
                    [(last ls) + 1]
```

Bem, ainda assim seria possível realizar o método sem grandes complicações.

```
ex3' :: [Int] -> [Int]
ex3' ls = case ls of
            []  -> []
            [x] -> [x+1]
            xs  -> [(head ls) + 1] ++
                     (case (init . tail $ ls) of
                        []   -> []
                        [x1] -> [x1 + 1]
                        xs1  -> [(head (init . tail $ ls)) + 1] ++
                          (case (init . tail $ (init . tail $ ls)) of
                             []   -> []
                             [x2] -> [x2 + 1]
                             xs2  -> error "provide more fuel") ++
                           [(last (init . tail $ ls))+1]) ++
                     [(last ls) + 1]
```

Como é possível criar sequências de comandos em linguagens imperativas, que potencialmente alteram o estado do programa a cada comando, o inlining dessas funções poderia gerar condicionais redundantes, visto que a sequência de comandos que são chamadas recursivas geraria uma sequência de dois condicionais que testam a mesma condição, cada um com seu caso base maior e seu bloco de chamadas recursivas. Para resolver isso usa-se a técnica de fusão de condicionais descrita no trabalho. Mas será que esse problema nos afeta quando vamos para o reino das linguagens funcionais? Nossos programas são expressões e as chamadas recursivas não estão em sequências de comandos. Ou será que é possível forçar um exemplo em que isso aconteça? Se o problema de fato não nos afeta, não é preciso pensar em fusão.

Bom, na verdade, afeta sim! Tome o exemplo abaixo (pensei nele enquanto tomava banho), que calcula a quantidade de números pares numa lista usando uma abordagem de divisão e conquista que usa um parâmetro para determinar se cairá no caso base ou não:

```
ex4 :: [Int] -> Int -> Int
ex4 ls n = case n of
             0 -> length $ filter even ls
             s -> ex4 (take (div n 2) ls) (div n 2) +
                  ex4 (drop (div n 2) ls) (div n 2)
```

Fazendo um passo de expansão dessa função teremos:

```
ex4' :: [Int] -> Int -> Int
ex4' ls n = case n of
              1 -> length $ filter even ls
              s -> (case (div n 2) of
                      0  -> length $ filter even (take (div n 2) ls)
                      s1 -> ex4' (take (div (div n 2) 2)
                                   (take (div n 2) ls)) (div (div n 2) 2) +
                            ex4' (drop (div (div n 2) 2)
                                   (take (div n 2) ls)) (div (div n 2) 2)
                   ) +
                   (case (div n 2) of
                      0  -> length $ filter even (drop (div n 2) ls)
                      s1 -> ex4' (take (div (div n 2) 2)
                                   (drop (div n 2) ls)) (div (div n 2) 2) +
                            ex4' (drop (div (div n 2) 2)
                                   (drop (div n 2) ls)) (div (div n 2) 2)
                   )
```

Os dois cases mais internos testam a mesma condição! Uma fusão seria produzir a seguinte função a partir de `ex4'`:

```
ex4'' :: [Int] -> Int -> Int
ex4'' ls n = case n of
              1 -> length $ filter even ls
              s -> case (div n 2) of
                      0  -> (length $ filter even (take (div n 2) ls)) +
                            (length $ filter even (drop (div n 2) ls))
                      s1 -> (ex4'' (take (div (div n 2) 2)
                                   (take (div n 2) ls)) (div (div n 2) 2) +
                            ex4'' (drop (div (div n 2) 2)
                                   (take (div n 2) ls)) (div (div n 2) 2)) +
                            (ex4'' (take (div (div n 2) 2)
                                   (drop (div n 2) ls)) (div (div n 2) 2) +
                            ex4'' (drop (div (div n 2) 2)
                                   (drop (div n 2) ls)) (div (div n 2) 2))
```

Apesar de parecer algo um pouco mais específico, será sim necessário cobrir fusão condicional. Note que eu usamos `case of` na condicional para evitar introduzir o `if then else`. Casamento de padrão também funciona sobre constantes então não há problema nisso.

Rerolling de Recursão

Como o objetivo do trabalho de Rugina e Rinard é apenas melhorar o desempenho eles ainda devem permitir que uma recursão muito grande ou até mesmo infinita seja executada, e assim quando o fator de expansão desejado é alcançado, eles fazem um rerolling para substituir o grande número de chamadas recursivas ao final por um número menor (para um subproblema maior, obviamente) e tornar o código mais simples. O nosso objetivo aqui é proibir totalmente que funções não terminantes sejam executadas mesmo que isso signifique abortár funções terminantes quando elas demorarem demais (e o problema da parada nos mostra que isso é inevitável). Então não há sentido em pensar em rerolling de recursão aqui, porque ao atingir o fator de expansão, qualquer chamada recursiva futura seria substituída com um erro, ou possivelmente com uma mônada. Já que não há mais chamadas recursivas, o monadification não seria mais uma abordagem tão complicada.

Conclusões Preliminares

Implementar o algoritmo de inlining e também o de fusão condicional! O artigo tem descrições “imperativas“ de como fazê-los. Mas tentarei fazer isso em Haskell. Precisarei definir uma linguagem simples para trabalhar direto com a AST, *parsing* nesse momento nem pensar.

Tentando Implementar

Vamos definir uma linguagem simples para tentar implementar o algoritmo de inlining. Não nos preocuparemos com parsing ou aspectos concretos da sintaxe. Os exemplos serão esboçados à mão como ASTs válidas, se necessário. Nossa linguagem é um λ-cálculo simples com duas novas construções: uma para definir uma expressão a um nome e uma para invocar a expressão definida por um nome. Isso tornará mais simples modelar a recursão, sem usar ponto fixo e combinadores.

```haskell
type VarName  = String
type FuncName = String
type Value    = Int

data Exp = Const Value
         | Var VarName
         | Abs VarName Exp
         | App Exp Exp
         | Def FuncName Exp
```

```haskell
         | Call FuncName
         deriving (Eq, Show)

exampleProgram :: Exp
exampleProgram = Def "A" (Abs "x" (App (Call "B") (App (Call "A") (Var "x"))))
```

O nosso algoritmo de inlining supõe que a AST de entrada começa com um `Def` e que possui ao menos uma chamada recursiva. Temos aqui uma versão que parece funcionar! É preciso sempre fazer inline de f_{m-1} em f para obter f_m para que o crescimento de código não seja super-exponencial. Executar `expandN` sobre o `exampleProgram` aumentará o numero de aplicações da função hipotética `B` deixando uma última chamada recursiva ao final. Podemos substituir essa última chamada recursiva com o erro de `provide more fuel` e a primeira parte do problema está resolvida!

```haskell
{-- Expande N vezes --}
expandN :: Exp -> Int -> Exp
expandN f 0 = f
expandN f 1 = expandOnce f
expandN f m = inline f1 f2
  where (f1, f2) = mutualize f (expandN f (m-1))

{-- expande 1 vez --}
expandOnce :: Exp -> Exp
expandOnce (Def n f) = inline f1 f2
   where
     (f1, f2) = mutualize (Def n f) (changeFuncName (Def n f) (n ++ "_"))

{-- Modifica o nome de uma função --}
changeFuncName :: Exp -> FuncName -> Exp
changeFuncName (Def n f) n' = Def n' (inlSubs f n n')
changeFuncName _ _ = error "not a named func"

{-- cria versões mutualmente rec de funcoes equivalentes --}
mutualize :: Exp -> Exp -> (Exp, Exp)
mutualize (Def n1 f1) (Def n2 f2) = ( (Def n1 (inlSubs f1 n1 n2)) ,
                                      (Def n2 (inlSubs f2 n2 n1)) )
mutualize _ _ = error "s n h"

{-- faz inlining do segundo no primeiro --}
inline :: Exp -> Exp -> Exp
inline (Def n1 f1) (Def n2 f2) = Def n1 (inline' (n1, f1) (n2, f2))
inline _ _ = error "funções nomeadas pls"

{-- substitui chamadas de fo pra ft --}
inlSubs :: Exp -> FuncName -> FuncName -> Exp
inlSubs (Call n) fo ft
  | n == fo    = Call ft
  | otherwise = Call n
inlSubs (Abs v e) fo ft   = Abs v (inlSubs e fo ft)
```

```haskell
inlSubs (App e1 e2) fo ft = App (inlSubs e1 fo ft)
                                (inlSubs e2 fo ft)
inlSubs (Def _ _) _ _     = error "should not happen"
inlSubs e _ _ = e

{-- faz o inlining de f2 em f1 --}
inline' :: (FuncName, Exp) -> (FuncName, Exp) -> Exp
inline' (_, (Def _ _)) _ = error "s n h"
inline' _ (_, (Def _ _)) = error "s n h"
inline' (n1, (Call n')) (n2, f2)
   | n' == n2  = f2
   | otherwise = Call n'
inline' (n1, (Abs v e1)) (n2, f2) = Abs v (inline' (n1, e1) (n2, f2))
inline' (n1, (App e1 e2)) (n2, f2) = App (inline' (n1, e1) (n2, f2))
                                         (inline' (n1, e2) (n2, f2))
inline' (n1, f1) (n2, f2) = f1
```

Precisamos agora realizar a fusão condicional.