

Implementação de Linguagens Funcionais

Notas de Aula

Roberto da Silva Bigonha

19 de janeiro de 2017

Todos os direitos reservados
Proibida a cópia sem autorização do autor

ORGANIZAÇÃO DO CURSO

I. Compilação de Linguagens Funcionais

1. Introdução
2. Linguagem Miranda
3. λ -Calculus Puro
4. λ -Cálculo Enriquecido
5. Compilação de Miranda
6. Tipos Estruturados
7. Compilação de Pattern-Matching
8. Compilação do λ Enriquecido
9. Compilação de ZFs
10. Verificação Polimórfica de Tipos
11. Um Verificador de Tipos

II. Redução de Grafos

1. Representação do Programa
2. Seleção do Próximo Redex

3. Redução de Grafo de Expressões
4. Supercombinadores e λ -Lifting
5. Supercombinadores Recursivos
6. λ -Lifting Totalmente Lazy
7. Combinadores SK
8. Coleta de Lixo

III. Redução de Grafos Avançada

1. Máquina-G
2. Código-G
3. Implementação da Máquina-G
4. Otimização da Máquina-G
5. Otimização Chamada de Cauda
6. Análise de Strictness
7. Redução de Grafos Pragmática
8. Redução de Grafos Paralela

PARTE I

COMPILAÇÃO DE LINGUAGEM FUNCIONAIS

I. INTRODUÇÃO

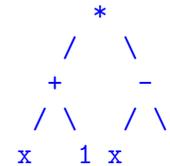
Tradução de Linguagens Funcionais de Alto Nível

- Linguagens funcionais mais importantes são muito parecidas semanticamente.
- Inicialmente, será mostrado como traduzir um programa escrito numa funcional de alto-nível para uma linguagem intermediária com uma sintaxe e semântica muito simples.
- A linguagem intermediária para a qual traduziremos nossos programas é a notação do λ -Calculus.
- O λ -Calculus é muito simples, mas suficientemente expressivo para permitir a tradução de qualquer linguagem funcional de alto nível.
- Será definido superconjunto enriquecido do λ -Calculus que para facilitar nosso processo de tradução.
- Será mostrado também a transformação do λ -Calculus enriquecido para o λ -Calculus puro.

Redução de Grafos

- Redução de grafos é uma técnica para implementar linguagens funcionais, especialmente λ -Calculus.
- Considere a função f (Miranda):

$$f\ x = (x + 1) * (x - 1)$$
- Podemos representar o corpo de f pelo grafo:

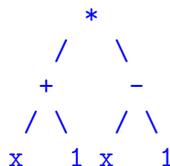


Exemplo de Redução de Grafo ...

- Suponha que desejamos avaliar $f\ 4$
- Podemos representar $f\ 4$ pelo grafo:

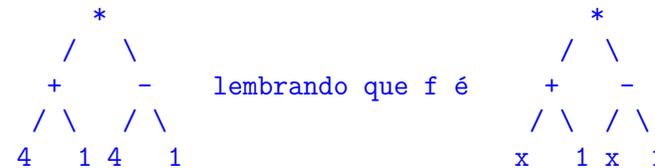


onde @ representa uma aplicação de função e o corpo de f é:



... Exemplo de Redução de Grafo

- Aplicando f a 4 teremos



- Agora podemos executar uma adição e uma subtração (em qualquer ordem), dando



- Finalmente executando a multiplicação, temos como resultado: 15

Execução de um Programa

Desse exemplo dado, vê-se que:

- Executar um programa funcional consiste em avaliar uma expressão.
- Um programa funcional tem como representação natural uma árvore, ou mais genericamente, um grafo.
- A avaliação se processa através de passos simples, chamados de reduções.
- Daí o termo redução de grafos.
- A avaliação pode ser feita em várias ordens, ou em paralelo, porque elas não podem interferir entre si.
- Uma avaliação está completa, quando não existem mais expressões reduzíveis.

Outros Pontos Importantes

- Como efetuar uma redução.
- O papel dos supercombinadores, que será um ponto chave no curso.
- Avaliação *lazy*, que é a *full laziness*.
- Administração de memória: Coleta de lixo.
- Redução de Grafos Avançada
 - Pode parecer que a redução de grafos é um modelo inerentemente menos eficiente que modelos convencionais de execução, pelo menos para máquinas von Neumann.
 - Entretanto, a redução de grafos pode ser compilada para uma forma interessante para ser executada em máquinas sequenciais.
 - Conceitos básicos da máquina-G e otimizações.
- Implementações paralelas de redução de grafos.

I. A LINGUAGEM MIRANDA

A Linguagem de Referência

- A linguagem de alto nível usada por nós será Miranda.
- Ela provê um conjunto rico e poderoso de construções.
- O nosso objetivo será mostrar como construções de Miranda podem ser traduzidas em cálculo lambda.
- Entre essas construções serão discutidos tipos estruturados de dados, *pattern-matching*, equações condicionais, e expressões ZF.
- Não serão discutidos outras construções como tipo abstrato de dados.

CARACTERÍSTICAS MIRANDA

- Tipos Fortes
- Puramente Funcional
- Funções Não-Estritas
- Funções de Ordem mais Alta
- Programa é chamado Script (coleção de equações)

```
> z = sq x / sq y
> x = a + b
> y = a - b
> a = 10
> b = 5
> z sistema responde com > 9
```

Tipos

1. Tipos Básicos

- num
- char
- bool

2. Tuplas

- Grupo de elementos de tipos variados
- employee = ("Jones", True, False, 39)
- Operador : (construtor)

```
a : b : c
```

3. Listas

Listas ...

- Grupo de elementos de mesmo tipo
- week_days = ["Mon", "Tue", "Wed", "Thur", "Frid"]
- Strings
 - são listas de caracteres
 - "Mon" é equivalente a ['M', 'o', 'n']
- Operador ++ (concatenação)


```
days = week_days ++ ["Sat", "Mon"]
```
- Operador : (prefixador)


```
0:[1, 2, 3] é equivalente a [0, 1, 2, 3]
```

... Listas

- Operador # (tamanho)


```
#days é 7
```
- Operador ! (indexação)


```
days !1 tem o valor "Tue"
```
- Operador -- (Subtração)


```
[1, 2, 3, 4, 5] -- [2, 4] tem valor [1, 3, 5]
```
- Operador .. (subrange)


```
fac n = product [1 .. n]
result = sum [1, 3 .. 100]
```

Equações Guardadas ...

- $\text{gcd } a \ b = \text{gcd } (a - b) \ b, a > b$
 $= \text{gcd } a \ (b - a), a < b$
 $= a, a = b$
- Os guardas são testados em ordem
- Último guarda pode ser otherwise
`f args = rhs1 , test1`
`= rhs2, test2`
`...`
`= rshn, otherwise`

... Equações Guardadas

- **Multiplicação $m \times n$:**
`mult m 0 = 0`
`mult m n = m + mult m (n - 1), n > 0`
- **Divisão m/n :**
`div m n = 0, m < n`
`= 1, m = n`
`= 1 + div (m-n) n, m > n`

Definições Locais ...

- **Raízes de equação:**
`qs a b c = error "Complex roots", delta < 0`
`= [-b / (2*a)], delta = 0`
`= [-b / (2*a) + radix / (2*a),`
`-b / (2*a) - radix / (2*a)], delta > 0`
where
`delta = b * b - 4 * a * c`
`radix = sqrt delta`

... Definições Locais

- **Pesquisa binária:**
`binsearch a x = bs a x 0 (#x - 1)`
where `bs a x i j = -1, i > j`
`= k, a = x!k`
`= bs a x i (k-1), a < x!k`
`= bs a x (k+1) j, a > x!k`
`k = (i + j) / 2`

Pattern Matching ...

- **Permite definir equações por suas alternativas**

- `sum [] = 0`
`sum (a:x) = a + sum x`
- `reverse [] = []`
`reverse (a:x) = reverse x ++ [a]`
- `no_dups x = x, #x < 2`
`no_dups (a:a:x) = no_dups (a:x)`
`no_dups (a:b:x) = a:no_dups(b : x), a ~ = b`
- `fst(a,b) = a`
`snd(a,b) = b`

... Pattern Matching ...

- **Pesquisa linear:**

```
linsearch1 a x = ls a x 0
where ls a [ ] n = -1
      ls a (b:x) n = n , a = b
                  = ls a x (n+1), a ~ = b
```

- **Pesquisa linear rápida:**

```
linsearch2 a x = n, n ~ = #x
              = -1, otherwise
where n = rs a (x ++ [a]) 0
      rs a (b:x) n = n , (a=b)
                  = rs a x (n+1) , otherwise
```

... Pattern Matching

- **Busca de valor em tabela:**

```
newtable = [ ]

enter k v t = (k,v):t

lookup t k = fst z, z ~ = [ ]
           = "no entry", otherwise
where z = [ v1 | (k1,v1) <- t; k1 = k]
```

Funções de Ordem mais Alta ...

- **Funções passadas como parâmetro ou retornadas como resultado**

- **Aplicações de função associam-se à esquerda**

$$f x y \equiv (f x) y$$

- **Currying**

$$\text{plus } x y = x + y$$

$$\text{plus } 3 = 3 + y$$

$$\Rightarrow$$

plus 3 é função que soma 3 ao seu argumento

... Funções de Ordem mais Alta

- `foldr op k [] = k`
`foldr op k (a:x) = op a (foldr op k x)`
- `sum = foldr (+) 0`
`sum [1,2,3] = foldr (+) 0 [1,2,3]`
`= (+)1(foldr(+) 0 [2,3])`
`= (+)1((+)2(foldr (+) 0 [3]))`
`= (+)1((+)2((+)3(foldr(+)0[])))`
`= (+)1((+)2((+)3 (0)))`
`= 6`
- `product = foldr (*) 1`
- `reverse = foldr postfix [] where postfix a x = x ++ [a]`

Tipos

- **Tipos fortes** \Rightarrow toda expressão tem 1 tipo, deduzido em tempo de compilação
- **Tipos primitivos**
`num`, `char` e `bool`
- **Tipo de listas de elementos de tipo T**
`[T]`
- **Tipo de tuplas com componentes de tipos T1, ..., Tn**
`(T1, ..., Tn)`

Declarações de Tipo

```
sq :: num -> num
fac :: num -> num
sum :: [num] -> num
queens :: num -> [ [num] ]
```

Tipos Polimórficos

```
id :: * -> *
reverse :: [*] -> [*]
fst :: (* , * *) -> *
snd :: (* , * *) -> * *
perms :: [*] -> [ [* ] ]
```

Lazy Evaluation

- Argumentos de funções não-estritas (non-strict) avaliados somente quando necessários

```
f True x y = x
f False x y = 0
```

Então

```
f ( a = 0 ) a ( 1 / a ) retorna sempre 0
```

Estruturas Infinitas

- ones = 1 : ones
fst (a:x) = a
⇒ fst (ones) tem valor 1
- nats = [1 ..]
- odds = [1,3..]
- fibs = f 0 1
where f a b = a : f b (a + b)

Expressões ZF (Zarmelo Franket) ...

- Lista dos quadrados de 1 .. 100

```
[n * n | n <- [1..100] ]
      gerador
```

- Lista de todas as permutações

```
perms [ ] = [ [ ] ]
perms x = [ a:y | a <- x; y <- perms (x -- [a]) ]
```

- Lista dos fatores de um número

```
factors n = [ i | i <- [1.. n div 2]; n mod i = 0 ]
      filtro
```

... Expressões ZF (Zarmelo Franket)

- Lista de divisores

```
divisors n = [ i | i <- [1.. n ]; n mod i = 0 ]
```

- String com n brancos

```
spaces n = [ ' ' | j <- [1 .. n] ]
```

- Maior divisor comum

```
gcd a b = max [ d | d <- divisors a ; b mod d = 0 ]
```

... Expressões ZF (Zarmelo Franket)

- **Predicado isprime**

```
isprime n = (divisors n) = [1,n] )
isprime n = n > 1 /\
  ([d | d <- [2 .. sqrt n]; n mod d = 0] = [ ] )
```

- **Crivo de Eratóstenes**

```
sieve [ ] = [ ]
sieve (a:x) =
  a : sieve[ b | b <- x; b mod a /= 0]
```

... Expressões ZF (Zarmelo Franket)

- **Pega n primeiros elementos de uma lista**

```
take 0 x = [ ]
take (n+1) [ ] = [ ]
take (n+1) (a : x) = a : take n x
```

- **Elimina n primeiros elementos de uma lista**

```
drop 0 x = x
drop (n+1) [ ] = [ ]
drop (n+1) (a : x) = drop n x
```

- **100 primeiros primos**

```
primes = take 100 (sieve [2..])
```

... Expressões ZF (Zarmelo Franket)

- **Quick sort of a list:**

```
sort [ ] = [ ]
sort (a : x) = sort[b | b <- x; b <= a] ++ [a]
  ++ sort[ b | b <- x ; b > a ]
```

... Expressões ZF (Zarmelo Franket)

- **Merge of two ordered lists of numbers or chars:**

```
merge [ ] y = y
merge x [ ] = x
merge (a:x) (b:y) = a : merge x (b:y) , a <= b
merge (a:x) (b:y) = b : merge (a:x) y , a > b
```

... Expressões ZF (Zarmelo Franket)

- Merge of two ordered lists of any type:

```
merge c [ ] y = y
merge c x [ ] = x
merge c (a:x) (b:y) = a : merge c x (b:y) , c a b
merge c (a:x) (b:y) = b : merge c (a:x) y , otherwise
```

... Expressões ZF (Zarmelo Franket)

- 8 rainhas
Cada rainha ocupa uma coluna, portanto basta armazenar as linhas

```
queens 0 = [ [ ] ]
queens n = [ q:b | q <- [0 ..7];
             b <- queens (n-1);
             safe q b ], n > 0

where
safe q b = and [~ checks q b i | i <- [0 .. #b-1] ]
checks q b i = (q = b!i) \ / (abs( q - b!i) = i + 1)
```

- `fst(queens 8) = primeira solução encontrada`

Implementação de Expressões ZF ...

- `map :: (* -> **) -> [*] -> [**]`
`map f x = [f a | a <- x]`

ou, sem usar expressão ZF:

```
map f [ ] = [ ]
map f (a:x ) = (f a) : map f x
```

... Implementação de Expressões ZF ...

- `filter :: (* -> bool) -> [*] -> [*]`
`filter p x = [a | a <- x; p a]`

ou, sem usar expressão ZF:

```
filter p [ ] = [ ]
filter p (a:x) = a : filter p x, p a
                = filter p x, ~ p a
```

... Implementação de Expressões ZF

• `concat :: [[*]] -> [*]`
`concat x = [a | y <- x; a <- y]`

ou, sem usar expressão ZF:

`concat [[]] = []`
`concat (a : x) = a ++ concat x`

Transformações de ZF

1. `[a | a <- x] ⇒ x`
2. `[f a | a <- x] ⇒ map f x`
3. `[e | a <- x ; p a ; ...] ⇒ [e | a <- filter p x ; ...]`
4. `[e | a <- x ; b <- y ; ...] ⇒`
`concat [[e | b <- y ; ...] | a <- x]`

Tipos Definidos Pelo Usuário ...

N.B.: Construtores têm primeira letra maiúscula

- tipo de árvores binárias com rótulos inteiros
`tree ::= Nilt | Node num tree tree`
- `t1 = Node 7 (Node 3 Nilt Nilt) (Node 4 Nilt Nilt)`
- `mirror :: tree -> tree`
`mirror Nilt = Nilt`
`mirror (Node a x y) =`
`Node a (mirror y) (mirror x)`

... Tipos Definidos Pelo Usuário

- **tipo de árvores binárias com rótulos:**
`tree * ::= Nilt | Node * (tree *) (tree *)`
`tree num`
`tree char`
`tree bool`
`tree (char -> char)`
- **enumeração:**
`color ::= Red | Orange | Yellow | Green`

Tipos Sinônimos

- São transparentes para o type checker

```
string == [char]
matrix == [ [num] ]
array * == [ [* ] ]
```

Tipos Abstratos

- `abstype stack *`

```
with empty :: stack *
    isempty :: stack * -> bool
    push :: * -> stack * -> stack *
    pop :: stack * -> stack *
    top :: stack * -> *

stack * == [* ]
empty = [ ]
isempty x = ( x = [ ] )
push a x = ( a : x )
pop ( a : x ) = x
top ( a : x ) = a
```

I. λ-CALCULUS PURO

λ-Calculus

- O λ-Calculus é uma linguagem simples, que será usada neste texto como ponte entre linguagens funcionais de alto nível e suas implementações em linguagens de baixo nível
- São características do λ-Calculus:
 - poucas construções
 - uma semântica simples, que possibilita uma facilidade de formalização do sistema
 - expressividade ou seja, por expressar todas funções computáveis, pode ser usada como linguagem destino para toda linguagem funcional

Sintaxe do λ -Calculus

- Usaremos o λ -Calculus na forma prefixada:
 $(+ 4 5)$
 $(+ (* 5 6) (* 8 3))$
- A execução de um programa funcional é a avaliação da expressão correspondente ao programa.
- A execução consiste em escolher expressões reduzíveis - *redexes* - e reduzi-las sucessivamente até que não existam mais *redexes*, e, assim, a expressão final é o resultado do programa.
- Utilizaremos o símbolo \rightarrow para indicar redução.
 $(+ (* 5 6) (* 8 3)) \rightarrow (+ 30 (* 8 3))$
 $(+ 30 (* 8 3)) \rightarrow (+ 30 24)$
 $(+ 30 24) \rightarrow 54$

Aplicação de Função

- A aplicação de função será denotada por justaposição: $f x$
- Todas as funções podem ser vistas tendo apenas um argumento
- Funções com mais de um argumento são construídas com a seguinte técnica:
 - No lugar de $(+(1, 2))$
 - Escreve-se $(+ 1 2)$
 - $(+ 1 2) \Leftrightarrow ((+ 1) 2)$
 onde $(+ 1)$ produz para uma função que soma 1 ao seu argumento
- A esse processo de transformação de funções de vários argumentos em funções com apenas um argumento dá-se o nome de *currying*, devido ao nome do pesquisador Curry.

O Uso de Parênteses

- Assim como na Matemática, podemos adotar convenções quanto ao uso de parênteses nas expressões lambda.
- Por exemplo:
 1. $((+ 1) 2)$ pode ser re-escrita como $(+ 1 2)$ ou ainda $+ 1 2$
 2. A expressão
 $*(+ a b) c$
 não pode perder parêntesis algum
 3. A expressão
 $((f ((+ 4) 3) (g x)))$
 pode ser re-escrita como
 $f (+ 4 3) (g x)$

Funções Primitivas e Constantes

- Na sua forma mais pura, o λ -Calculus não tem funções primitivas.
- Na prática, assumimos que existam funções como:
 - aritméticas ($+$, $-$, $/$, $*$)
 - comparação ($<$, $>$, $=$, \dots)
 - constantes numéricas (0 , 1 , 2 , \dots)
 - lógicas (AND, OR, NOT)
 - constantes lógicas (TRUE, FALSE)

... Funções Primitivas e Constantes

- constantes alfabéticas ('a', 'b', 'c', ...)
- função condicional IF:
IF TRUE $E_t E_f \rightarrow E_t$
IF FALSE $E_t E_f \rightarrow E_f$
- operadores de lista CONS, HEAD e TAIL, que obedecem às regras:
HEAD(CONS $a b$) $\rightarrow a$
TAIL(CONS $a b$) $\rightarrow b$
- constante NIL

Abstrações Lambda ...

- A abstração lambda é utilizada para criar novas funções
- A abstração lambda é uma expressão particular que denota uma função.
- Por exemplo, na expressão $\lambda x. + x 1$
 - λ representa uma função sem nome
 - x é o argumento dessa função
 - $.$ separa o corpo da função de seu parâmetro formal
 - $+ x 1$ é o corpo da função, o qual retorna o valor do seu argumento somado de 1

... Abstrações Lambda

- A abstração lambda sempre composta do λ , do parâmetro formal, do ponto e do corpo da função.
- As funções criadas como uma abstração lambda não tem nome.
- O corpo da abstração estende-se para a direita o máximo possível.
Exemplo:

$(\lambda x. + x 1) 4$

Resumo da Sintaxe do λ -Calculus

```

<exp> ::= <constante>      -- Constantes
      | <variável>         -- Variáveis
      | ( <exp> )
      | <exp> <exp>         -- Aplicações
      |  $\lambda$ <variável>. <exp> -- Abstrações

```

- Convenção:
Na escrita de expressões lambda serão usados:
 - minúsculas para variáveis,
 - maiúsculas para denotar expressões.
- As constantes são os inteiros, booleanos (por exemplo TRUE), os seus respectivos operadores e as funções primitivas já definidas.

A Semântica Operacional do λ -Calculus

- Até agora mostramos a sintaxe do cálculo lambda, mas para dignificar o nome de *cálculo*, devemos mostrar como calculá-lo.
- Há três regras de conversão, que descrevem como converter uma λ -expressão em outra:
 - α -conversão
 - β -conversão
 - η -conversão
- As vezes consideram-se aplicações de funções primitivas como sendo uma forma especial de conversão denominada δ -conversão.

Variáveis Livres e Ligadas ...

Sejam x e y variáveis e E e F expressões lambda.

- Ocorrência de variáveis livres
 - x ocorre livre em x
 - x ocorre livre em $(E F)$, se e somente se x ocorre livre em E ou x ocorre livre em F .
 - x ocorre livre em $\lambda y . E$, se e somente se x e y são diferentes, e x ocorre livre em E .
- Ocorrência de variáveis ligadas
 - x ocorre ligada em $(E F)$, se e somente se x ocorre ligada em E ou x ocorre ligada em F
 - x ocorre ligada em $\lambda y . E$ se e somente se x e y são a mesma variável, e x ocorre livre em E ; ou então x ocorre ligada em E .

... Variáveis Livres e Ligadas

- Nenhuma variável ocorre ligada numa expressão consistindo de uma única constante ou variável.
- Exemplos:

$$\lambda x . + ((\lambda y . + y z) 7) x$$

$$\lambda y . + x ((\lambda x . + x 1) y)$$

β -conversão

- β -conversão significa uma β -redução ou uma β -abstração
- β -redução é uma redução, onde o resultado da aplicação de uma λ -abstração a um argumento é uma instância do corpo da λ -abstração na qual as ocorrências livres do parâmetro formal da abstração são substituídas no corpo da abstração por cópias do argumento
- Por exemplo:

$$(\lambda x . + x 1)4 \rightarrow (+ 4 1)$$

β -conversão

- Então, exemplifica-se uma β -conversão da seguinte forma:

$$+ 4 1 \xrightarrow{\beta} (\lambda x. + x 1) 4$$

- $\xrightarrow{\beta}$ expressa a equivalência de duas expressões
- β -abstração é a operação inversa da β -redução, servindo para introduzir novas λ -abstrações.
- Por exemplo,

$$+ 4 1 \rightarrow (\lambda x. + x 1) 4$$

Exemplos de β -redução...

- $$(\lambda x. (\lambda y. - y x)) 4 5$$

$$\rightarrow (\lambda y. - y 4) 5$$

$$\rightarrow (- 5 4)$$

$$\rightarrow 1$$
- $$(\lambda f. f 3) (\lambda x. + x 1)$$

$$\rightarrow \lambda x. + x 1) 3$$

$$\rightarrow + 3 1$$

$$\rightarrow 4$$
- $$(\lambda x. (\lambda x. + (- x 1))x 3) 9$$

$$\rightarrow (\lambda x. + (- x 1))9 3$$

$$\rightarrow + (- 9 1) 3$$

$$\rightarrow + 8 3 \rightarrow 11$$

... Exemplos de β -redução

- $$(\lambda x. \lambda y. + x ((\lambda x. - x 3) y)) 5 6$$

$$\rightarrow (\lambda y. + 5 ((\lambda x. - x 3) y)) 6$$

$$\rightarrow + 5((\lambda x. - x 3) 6)$$

$$\rightarrow + 5 (- 6 3)$$

$$\rightarrow 8$$

Um Exemplo Maior

- Considere as seguintes definições:

$$\text{CONS} = (\lambda a. \lambda b. \lambda f. f a b)$$

$$\text{HEAD} = (\lambda c. c (\lambda a. \lambda b. a))$$

$$\text{TAIL} = (\lambda c. c (\lambda a. \lambda b. b))$$
- Note que $\text{HEAD} (\text{CONS } p \ q) = p$

$$\text{HEAD} (\text{CONS } p \ q)$$

$$\rightarrow (\lambda c. c (\lambda a. \lambda b. a)) (\text{CONS } p \ q)$$

$$\rightarrow \text{CONS } p \ q (\lambda a. \lambda b. a)$$

$$\rightarrow (\lambda a. \lambda b. \lambda f. f a b) p \ q (\lambda a. \lambda b. a)$$

$$\rightarrow (\lambda b. \lambda f. f p b) q (\lambda a. \lambda b. a)$$

$$\rightarrow (\lambda f. f p q) (\lambda a. \lambda b. a)$$

$$\rightarrow (\lambda a. \lambda b. a) p \ q \rightarrow (\lambda b. p) q \rightarrow p$$

α -conversão

- α -conversão é a troca do nome do parâmetro formal de qualquer λ -abstração, preservando-se a consistência da expressão.

- Por exemplo,

$$-(\lambda x. + x 1) \xrightarrow{\alpha} (\lambda y. + y 1)$$

$$-\lambda x. (\lambda x. + x 2)(- x 1) \xrightarrow{\alpha} \lambda y. (\lambda x. + x 2)(- y 1)$$

 η -conversão

- A definição de η -conversão é:

$$(\lambda x. F x) \xrightarrow{\eta} F$$

onde x não ocorre livre em F , e F denota uma função.

- A 1ª condição previne conversões falsas, como por exemplo:
 $(\lambda x. + x x)$ não se converte a $(+ x)$
- A 2ª condição previne conversões falsas envolvendo constantes, por exemplo:
 $(\lambda x. + 1 x)$ não se converte a $(+ 1)$
- A η -conversão permite expressar a equivalência de duas expressões que comportam exatamente da mesma forma quando aplicadas a um argumento.

Interconvertibilidade ...

- Se aplicações de determinadas conversões a duas expressões E_1 e E_2 levam a uma expressão comum, então estas expressões são ditas interconvertíveis.

- Por exemplo,

IF TRUE $((\lambda p. p) 3)$ e $(\lambda x. 3)$ são interconvertíveis

- Com efeito,

$$\text{IF TRUE } ((\lambda p. p) 3) \xrightarrow{\beta} \text{IF TRUE } 3 \xrightarrow{\eta}$$

$$(\lambda x. \text{IF TRUE } 3 x) \xrightarrow{\beta} (\lambda x. 3)$$

... Interconvertibilidade ...

- Em geral é tedioso encontrar se duas expressões são interconvertíveis, especialmente as funções.
- Um método conveniente para essa tarefa é aplicar a ambas expressões um argumento arbitrário, w , que não ocorra livre nem em E_1 e nem em E_2 .
- Por exemplo:

$$\text{IF TRUE } ((\lambda p. p) 3) w \rightarrow (\lambda p. p) 3 \rightarrow 3$$

e

$$(\lambda x. 3) w \rightarrow 3$$

... Interconvertibilidade

- Se $F_1 w \rightarrow E$ e $F_2 w \rightarrow E$
então

$$F_1 \leftrightarrow F_2$$

- Prova:

$$\begin{aligned} F_1 &\xrightarrow{\eta} (\lambda w. F_1 w) \\ &\leftrightarrow (\lambda w. E) \\ &\leftrightarrow (\lambda w. F_2 w) \\ &\xrightarrow{\eta} F_2 \end{aligned}$$

- Nem sempre expressões com o mesmo significado podem ser interconvertíveis.

Problema da Captura de Nomes ...

- Considere a função

$$\text{TWICE} = (\lambda f. \lambda x. f (f x))$$

- TWICE TWICE

$$= (\lambda f. \lambda x. f(f x)) \text{ TWICE}$$

$$\rightarrow (\lambda x. \text{TWICE} (\text{TWICE } x))$$

$$= (\lambda x. \text{TWICE} (\lambda f. \lambda x. f(fx))x))$$

$$\rightarrow ((\lambda x. \text{TWICE}(\lambda x. x(xx)))) \leftarrow \text{ERRADO!}$$

- Melhor fazer:

$$(\lambda x. \text{TWICE} (\lambda f. \lambda x. f (f x)) x))$$

$$\xrightarrow{\alpha} (\lambda x. \text{TWICE} (\lambda f. \lambda y. f (f y)) x))$$

$$\rightarrow (\lambda x. \text{TWICE} (\lambda y. x (x y))) \leftarrow \text{CERTO!}$$

... Problema da Captura de Nomes

- Onde se conclui:

1. β -redução somente é válida se variáveis livres do argumento não conflitarem com qualquer parâmetro formal no corpo da λ -abstração.

2. α -conversão pode ser usada para evitar (1).

A Notação $[M/x]$

- A notação $E[M/x]$ denota a expressão construída a partir de E pela substituição de toda ocorrência livre de x pela expressão M .

- Definição de $E[M/x]$:

1. $x[M/x] = M$

2. $c[M/x] = c$

onde c representa uma variável ou uma constante

3. $(E F)[M/x] = E[M/x] F[M/x]$

4. $(\lambda x. E)[M/x] = \lambda x. E$

A Notação $[M/x]$

• Definição de $E[M/x]$ (continuação):

5. Suponha que y seja uma variável distinta de x

$$(\lambda y . E)[M/x] = \begin{cases} \lambda y . E[M/x] & \text{se I} \\ \lambda z . (E[z/y])[M/x] & \text{se II} \end{cases}$$

onde:

- z é uma nova variável que não ocorre livre em E ou M
- **Condição I** = x não ocorre livre em E ou y não ocorre livre em M
- **Condição II** = x ocorre livre em E e y ocorre livre em M

Definição das α -, β - e η -conversões

• α -conversão:

se y não for livre em E , então

$$(\lambda x . E) \xrightarrow{\alpha} (\lambda y . E[y/x])$$

• β -conversão:

$$(\lambda x . E)M \xrightarrow{\beta} E[M/x]$$

• η -conversão:

se x não é livre em E e E denota uma função, então:

$$(\lambda x . E x) \xrightarrow{\eta} E$$

- Quando usadas da esquerda para a direita as regras β e η são chamadas reduções e podem ser denotadas por \rightarrow

Ordem de redução

- Quando a expressão não tem mais *redexes* para serem reduzidos, então a avaliação está completa e a expressão resultante é dita estar na forma normal.
- Quando a expressão possui mais de um *redex*, a redução pode seguir caminhos alternativos.
- Nem toda expressão tem uma forma normal, por exemplo, $(D D)$, onde $D = \lambda x . x x$
 $(\lambda x . x x)(\lambda x . x x) \rightarrow (\lambda x . x x)(\lambda x . x x)$
 Portanto, a avaliação dessa expressão resultaria em $(D D) \rightarrow (D D)$,
 que é uma situação correspondente a um *loop* infinito.

Ordem normal de redução

- Podem duas seqüências de redução distintas de uma mesma expressão levarem a formas normais diferentes?
- Dois teoremas respondem negativamente essa pergunta:
 Teorema de Church-Rosser I
 Teorema de Church-Rosser II

Teorema de Church-Rosser I

- Teorema de Church-Rosser I (CRT I)
 $E_1 \leftrightarrow E_2$ implica que $\exists E \mid E_1 \rightarrow E$ e $E_2 \rightarrow E$.
- Colorário do CRT I:
 Nenhuma expressão pode ser convertida a duas formas normais distintas, isto é, toda sequência de reduções que termina leva ao mesmo resultado.

Teorema de Church-Rosser II

- Church-Rosser Theorem II (CRT II)
 $E_1 \rightarrow E_2$ e E_2 estar na forma normal implicam que existe uma sequência, em ordem normal, de reduções de E_1 para E_2 .
- A ordem normal de redução especifica que o *redex* mais à esquerda e mais externo dever ser reduzido primeiro.
 - Princípio: Argumentos podem ser ignorados nos corpos das funções

Ordem ótima de redução

- Embora a ordem normal garanta encontrar uma forma normal (se existir), ela não garante necessariamente o menor número de reduções.
- O menor número de reduções depende da ordem utilizada.
- A melhor ordem depende também da estrutura em questão, por exemplo, redução de árvores, redução de grafos, redução de combinadores SK.
- Em Redução de Grafos a ordem normal é quase ótima.
- Em Reduções de Combinadores SK a ordem normal do grafo de reduções é ótima.

Funções recursivas ...

- Consideremos a seguinte definição recursiva da função fatorial:

$$\text{FAT} = (\lambda n. \text{IF } (= n 0) 1 (* n (\text{FAT } (- n 1))))$$
- A definição foi possível pelo fato de termos nomeado uma abstração lambda, e daí referirmos ao seu nome dentro do seu próprio corpo.
- Contudo, essa construção não é possível no cálculo lambda, pois as abstrações lambda são funções anônimas.

... Funções recursivas

- Vejamos como podemos contornar esse problema e expressar funções recursivas no cálculo lambda.
- Seja a definição de FAT enfocando somente a parte recursiva:

$$\text{FAT} = (\lambda n. \dots \text{FAT} \dots)$$
- Seja o resultado de uma β -abstração em FAT:

$$\text{FAT} = (\lambda f. (\lambda n. \dots f \dots)) \text{FAT}$$
- Assim, $\text{FAT} = H \text{FAT}$
 onde $H = (\lambda f. (\lambda n. \dots f \dots))$
- Note que a definição de H não é recursiva

A Noção de Ponto-Fixo

- Ponto-fixo de uma função é o argumento que quando a função é aplicada a ele, o resultado da aplicação é ele próprio. Por exemplo:
 - Os pontos-fixos de $f(x) = x^2 - 2$ são $x = -1$ e $x = 2$.
 - Os pontos-fixos de $\lambda x. * x x$ são 0 e 1.
- A equação $\text{FAT} = H \text{FAT}$ estabelece que FAT é um ponto-fixo de H.
- Agora, nosso problema passa a ser, encontrar um ponto fixo para a função H. Para isso, vamos depender somente da função H.
- Vamos inventar uma função Y que toma uma função H como argumento e retorna H aplicada ao seu ponto fixo como resultado.

$$Y H = H (Y H)$$

Chamamos a função Y de **combinador de ponto-fixo**.
- Basta definir Y, como abstração lambda sem usar recursão, e teremos resolvido nosso problema

CÁLCULO DE FAT 1

- $\text{FAT} = Y H$, onde Y e H não são recursivas e

$$H = \lambda \text{fat}. \lambda n. \text{IF} (= n 0) 1 (* n (\text{fat} (- n 1)))$$
- $\text{FAT } 1 = Y H 1 = H (Y H) 1$

$$= (\lambda \text{fat}. \lambda n. \text{IF} (= n 0) 1 (* n (\text{fat} (- n 1)))) (Y H) 1$$

$$\rightarrow (\lambda n. \text{IF} (= n 0) 1 (* n (Y H (- n 1)))) 1$$

$$\rightarrow \text{IF} (= 1 0) 1 (* 1 (Y H (- 1 1)))$$

$$\rightarrow * 1 (Y H 0)$$

$$\rightarrow * 1 (H (Y H) 0)$$

$$= * 1 ((\lambda \text{fat}. \lambda n. \text{IF} (= n 0) 1 (* n (\text{fat} (- n 1)))) (Y H) 0)$$

$$\rightarrow * 1 ((\lambda n. \text{IF} (= n 0) 1 (* n (Y H (- n 1)))) 0)$$

$$\rightarrow * 1 (\text{IF} (= 0 0) 1 (* 1 (Y H (- 0 1))))$$

$$\rightarrow * 1 1 \rightarrow 1$$

O Combinador Y

- Y pode ser definido da seguinte forma:

$$Y = (\lambda h. (\lambda x. (h(x x)))(\lambda x. (h(x x))))$$
- Assim temos uma definição não recursiva de Y e o problema está resolvido.
- Para verificar que Y tem a propriedade requerida:

$$Y H = (\lambda h. (\lambda x. (h(x x)))(\lambda x. (h(x x)))) H$$

$$\rightarrow (\lambda x. H(x x)) (\lambda x. H(x x))$$

$$\rightarrow H ((\lambda x. H(x x)) (\lambda x. H(x x)))$$

$$\rightarrow H (Y H)$$
- Portanto, $Y H = H (Y H)$, ou seja, Y H é um ponto-fixo de H.

A Semântica Denotacional do λ -Calculus

- A maneira denotacional de vermos uma função é como um conjunto fixo de pares ordenados (argumento, valor).
- Anteriormente, vimos que uma expressão pode ser avaliada via repetidas reduções.
- Reduções são meras transformações sintáticas sem referência ao significado das expressões.
- O significado das expressões foi dado por nossa intuição sobre funções abstratas, e assim chegamos a uma semântica operacional do λ -Calculus, onde as expressões são vistas como uma sequência de operações no tempo, que fornecem um valor para um dado argumento.
- O objetivo com a semântica denotacional do λ -Calculus é fornecer a ela um significado exato da idéia de função abstrata.

A função Eval

- Em semântica denotacional atribui-se um valor a todo objeto sintático da linguagem.
- A semântica da linguagem é uma função matemática Eval, que mapeia objetos sintáticos em valores.
- Para definir a semântica do λ -Calculus vamos definir a função Eval para cada objeto sintático da linguagem.
- Adotaremos a seguinte notação:
Eval[[Objeto sintático]] = denotação ou valor
- Os valores produzidos por Eval pertencem a uma coleção chamada domínio, o qual é uma estrutura um tanto complicada já que inclui todas as funções e valores que podem ser denotados por uma λ -expressão.

Definição de Eval...

- Seja ρ é o ambiente ou contexto no qual as variáveis estão inseridas.
- ρ é uma função que mapeia o nome das variáveis a seus valores.
- Eval[[k]] ρ = definido a seguir onde k é uma constante
- Eval[[x]] ρ = ρx onde x é uma variável.
- Eval[[$E_1 E_2$]] ρ = (Eval[[E_1]] ρ) (Eval[[E_2]] ρ), onde E_1 e E_2 são expressões

... Definição de Eval

- Eval[[$\lambda x. E$]] ρa = Eval[[E]] $\rho[x = a]$
onde:
 - E é uma expressão
 - $\rho[x = a]$ significa a função ρ estendida com a informação que a variável x está ligada ao valor a .
- Fato:
Eval[[<expr sem forma normal>]] = \perp
- \perp denota **não-terminação**

Semântica das Funções Primitivas e Constantes

- Seja, por exemplo, o objeto sintático "*" que intuitivamente significa a multiplicação.
Então $\text{Eval}[\ast] a b = a \times b$.
- É importante distinguir * e \times , onde o primeiro é um objeto sintático e o segundo é a operação matemática abstrata de multiplicação.
- No caso do operador + haveria um único símbolo para ambas as coisas

Semântica das Funções Primitivas e Constantes

- Uma definição mais completa de "*" seria:
 $\text{Eval}[\ast] a b = a \times b$, se $a \neq \perp$ e $a \neq 0$ e $b \neq \perp$
 $\text{Eval}[\ast] 0 b = 0$
 $\text{Eval}[\ast] a \perp = \perp$, se $a \neq 0$
 $\text{Eval}[\ast] \perp b = \perp$
- No caso de constantes teríamos a seguinte definição:
 $\text{Eval}[6] = 6$.
 Saliemos que, o primeiro "6" é um objeto sintático e o segundo "6" é um objeto matemático abstrato.
- Da mesma maneira podemos definir as demais funções primitivas e constantes.

Strictness e Laziness

- Uma função f é estrita se e somente se:
 $f \perp = \perp$
- Isto é, caso seja obrigatório ter o valor de um argumento para fazer a aplicação da função, então se o argumento não atingir a forma normal, o resultado da função também não atingirá a forma normal.
- Se uma função é não-estrita, dizemos que ela é *lazy*. Tecnicamente é um abuso de terminologia, pois avaliação *lazy* é apenas uma técnica para implementar semântica não-estrita.

Correção das Regras de Conversão

- Em [Stoy, 1981], é mostrado que as regras de conversão que expressam equivalência de expressões podem ser espelhadas no mundo denotacional, ou seja,
 se $E_1 \rightarrow E_2$ então $E_1 \leftrightarrow E_2$
 e daí
 $E_1 \rightarrow E_2$ implica em $\text{Eval}[E_1] = \text{Eval}[E_2]$.
- Porém o contrário já não é válido, ou seja, se duas expressões denotam o mesmo valor nem sempre elas são interconvertíveis.
- Por exemplo f_1 e f_2 não são interconvertíveis:
 $f_1 = (\lambda x . + x x)$
 $f_2 = (\lambda x . * x 2)$

I. λ -CALCULUS ENRIQUECIDO

Esquema de Tradução



λ -Calculus Enriquecido

- O λ -Calculus enriquecido é um superconjunto do λ -Calculus, assim todas construções do segundo permanecem inalteradas no primeiro.
- São acrescentadas quatro construções no λ -Calculus puro:
 1. expressões `let` e `letrec`
 2. abstrações lambda com *pattern-matching*
 3. o operador infix `□`
 4. expressões `case`

Sintaxe do λ -Calculus Enriquecido

```
<exp> ::= <constante> | <variável> | <exp> <exp> | ( <exp> )
      |  $\lambda$ <pattern>. <exp>
      | let <pattern> = <exp> in <exp>
      | letrec <pattern> = <exp>
          ...
          <pattern> = <exp>
      | in <exp>
      | <exp> □ <exp>
      | case <variável> of
          <pattern>  $\Rightarrow$  <exp>
          ...
          <pattern>  $\Rightarrow$  <exp>
<pattern> ::= <constante> | <variável> | ( <pattern> )
          | <construtor> <pattern> ... <pattern>
```

Expressões let simples ...

- Uma das principais construções em qualquer linguagem funcional é a definição
- Numa definição, um nome é ligado a um valor
- O λ -Calculus enriquecido provê esse mecanismo usando as expressões `let` e `letrec`
- Usamos o termo **expressões `let` simples** para diferenciar de expressões `let` com *pattern-matching*

... Expressões let simples

- A expressão `let` tem a seguinte sintaxe:

`let $v = B$ in E`

onde:

– v é uma variável

– B e E são λ -expressões enriquecidas, onde v está ligada a B no escopo de E mas não no escopo de B .

- Exemplo:

`let $x = 3$ in $(* x x)$`

$\rightarrow * 3 3$

$\rightarrow 9$

Expressões let simples ...

- Note que a expressão `let` é como outra qualquer, podendo inclusive ser aninhada:

`let $x = 3$ in (let $y = 4$ in $(* x y)$)`

\rightarrow `let $y = 4$ in $(* 3 y)$`

$\rightarrow * 3 4$

$\rightarrow 12$

- As expressões aninhadas também poderiam ser reescritas da seguinte forma, para o exemplo anterior:

`let $x = 3$`

`$y = 4$`

`in $(* x y)$`

Definição de `let` em λ -Calculus

- A equivalência de uma expressão `let` e o λ -Calculus puro pode ser definida da seguinte maneira:

$(\text{let } v = B \text{ in } E) \equiv ((\lambda v.E) B)$

Expressões letrec simples ...

- letrec é abreviação de let recursivamente.

- A sintaxe da expressão letrec simples é:

```
letrec  $v_1 = E_1$ 
       $v_2 = E_2$ 
      .....
       $v_n = E_n$ 
```

in E

onde:

- v_i são variáveis
- E_i e E são expressões lambda enriquecidas.

... Expressões letrec simples...

- Considere as expressões

```
let   $v_1 = E_1$ 
      $v_2 = E_2$ 
     .....
      $v_n = E_n$ 
in  $E$ 
```

```
letrec  $v_1 = E_1$ 
       $v_2 = E_2$ 
      .....
       $v_n = E_n$ 
in  $E$ 
```

- A diferença entre let e letrec é que:

- no caso do letrec, os v_i , para $1 \leq i \leq n$, estão no escopo de E , assim como nos dos E_j , para $1 \leq j \leq n$
- e no caso do let, cada v_i está apenas no escopo de E .

...Expressões letrec simples ...

- São possíveis as seguintes construções:

```
letrec fat =  $\lambda n. \text{IF}(=n 0) 1 (*n (\text{fat } (- n 1)))$ 
in fat 4
```

```
letrec f = ... f ... g ...
      g = f
in ...
```

- Semântica de letrec com uma única definição:

$$\text{letrec } v = B \text{ in } E \equiv (\text{let } v = Y (\lambda v. B) \text{ in } E)$$
...Expressões letrec simples

- O caso com múltiplas definições requer o uso de *pattern-matching* e é deixado para a frente.

- Apesar de ter escolhido o cálculo lambda como linguagem destino para a tradução, por razões de eficiência é conveniente introduzir expressões let e letrec simples na linguagem destino ao invés de traduzi-las para o cálculo lambda.

Operador \square

- Definição de \square :

$a \square b = a$, if $a \neq \perp$ and $a \neq \text{FAIL}$
 $\text{FAIL} \square b = b$
 $\perp \square b = \perp$

- A avaliação de $a \square b$ segue os passos:

- \square avalia seu primeiro argumento, o a
- se essa operação não terminar, $a \square b$ também não termina
- caso contrário e se o valor de a for diferente de FAIL, retorna-se esse valor
- mas se o valor de a for FAIL, retorna-se o valor de b

I. COMPILAÇÃO DE MIRANDA

Abordagens Para Tradução ...

- Abordagem I:

- Um programa em Miranda poderia ser traduzido através de sucessivas transformações de um programa em outro, onde cada transformação executa um passo de simplificação.
- O processo seria completado traduzindo o programa em Miranda simplificado para o λ -Calculus.
- A idéia é que as transformações iniciais façam o trabalho pesado, e que o passo final seja um pouco mais que uma transformação sintática.

... Abordagens Para Tradução

- Abordagem II:

- Simple transformação do programa Miranda numa versão enriquecida do cálculo lambda.
- A seguir transformar as construções extras do cálculo lambda enriquecido em λ -expressões puras.

Abordagem Escolhida

- A primeira abordagem pode parecer mais atraente que a segunda, pois não seria necessário definir uma nova linguagem, que é o λ -Calculus enriquecido.
- Porém, adotaremos a segunda abordagem pelos seguintes motivos:
 - Miranda não possui certos mecanismos, como:
 - * abstrações lambda
 - * capacidade de qualificar qualquer expressão com definições locais que são desejáveis numa transformação baseada em compiladores.

Outras Vantagens da Abordagem II

- Linguagens funcionais na sua maior parte são variações sintáticas uma das outras, com poucas modificações semânticas
- O segundo método seria aplicável a uma variada gama de linguagens funcionais além de Miranda.

Esquema de Tradução de Miranda



Tradução de Miranda

- Um programa Miranda consiste em um conjunto de definições juntamente com uma expressão a ser avaliada, que fornece o resultado do programa

Conjunto de definições

Expressão a ser avaliada

- Por exemplo,

$$\text{square } n = n * n$$

$$2 * (\text{square } 5)$$

Tradução de Miranda para λ -Calculus Enriquecido

- Informalmente poderíamos traduzir um programa Miranda para λ -Calculus enriquecido da seguinte forma:

```
let square =  $\lambda n. * n n$ 
in (* 2 (square 5))
```

Notação TE

- Vamos introduzir a seguinte notação para ajudar no processo de tradução.
- Seja TE a função que tem uma expressão Miranda como entrada e retorna uma expressão lambda como saída, isto é:

$$TE[\langle \text{expr Miranda} \rangle] \equiv \langle \lambda\text{-expressão} \rangle$$
- Utilizamos colchetes duplos [] para enfatizar que a expressão Miranda é um objeto sintático.
- Usamos \equiv para lembrar que a expressão lambda também é um objeto sintático.
- Chamamos TE de um esquema de tradução.

Notação TD

- Seja outro esquema de tradução TD, que traduz definições Miranda para definições cujo lado direito é uma λ -expressão.
- Por exemplo

$$TD[\text{square } n = n * n] \equiv \text{square} = \lambda n. * n n$$

Tradução de um Programa

- Para o programa


```
Definição 1
Definição 2
...
Definição n
```

 Expressão
- geramos a seguinte expressão lambda enriquecida:

```
letrec TD[Definição 1]
      TD[Definição 2]
      ...
      TD[Definição n]
in TE[Expressão]
```

Esquema de Tradução TE...

- **Constantes:**

$$TE[k] \equiv k,$$

onde k é uma constante ou função primitiva.

- **Variáveis:**

$$TE[v] \equiv v$$

onde v é uma variável incluindo funções definidas pelo usuário e construtores.

- **Aplicação de Funções:**

$$TE[E_1 E_2] \equiv TE[E_1]TE[E_2],$$

onde E_i são expressões.

...Esquema de Tradução TE

- Para os operadores infixos de Miranda a tradução fica da seguinte forma:

$$TE[E_1 <infixo> E_2] \equiv TE[<infixo>]TE[E_1]TE[E_2].$$

- Miranda também permite a criação de funções infixas definidas pelo usuário. Seus nomes devem ser prefixados com '\$'.

- Assim,

$$TE[E_1 $v E_2] \equiv TE[v]TE[E_1]TE[E_2]$$

- Existem ainda duas outras formas: as listas e expressões ZF

O Esquema de Tradução TD

- **Definição de variáveis:**

$$TD[v = E] \equiv v = TE[E]$$

onde v é uma variável e E é uma expressão.

- **Definição de funções simples:**

$$TD[f v_1 v_2 \dots v_n = E] \equiv \\ f = \lambda v_1 . \lambda v_2 \dots . \lambda v_n . TE[E]$$

Um Exemplo

- Considere o seguinte programa Miranda que calcula médias:

$$\text{media } a \ b = (a + b) / 2$$

$$\text{media } 2 \ (3 + 5)$$

- **Tradução:**

$$\text{letrec TD[media } a \ b = (a + b) / 2] \\ \text{in TE[media } 2 \ (3 + 5)]$$

ESQUEMA TD

1. TD[media $a\ b = (a + b) / 2$]

\equiv media = $\lambda a.\lambda b.$ TE[($a + b$) / 2]

\equiv media = $\lambda a.$ $\lambda b.$
TE[/]TE[($a + b$)]TE[2]

\equiv media = $\lambda a.$ $\lambda b.$
/ (TE[+]TE[a]TE[b]) 2

\equiv media = $\lambda a.$ $\lambda b.$ / (+ $a\ b$) 2

Esquema TE

1. TE[media 2 (3 + 5)]

\equiv TE[media]TE[2](TE[3 + 5])

\equiv media 2 (TE[+]TE[3]TE[5])

\equiv media 2 (+ 3 5)

2. Substituindo 2 e 3 em 1 temos:

letrec media = $\lambda a.\lambda b.$ / (+ $a\ b$) 2
in media 2 (+ 3 5)

3. Usando a semântica das expressões let para produzir λ -Calculus puro, temos:

((λ media. (media 2 (+ 3 5))) ($\lambda a.\lambda b.$ / (+ $a\ b$) 2))

I. TIPOS ESTRUTURADOS E PATTERN-MATCHING

Construtores de Tipos

- Seja a declaração em Miranda de um tipo estruturado:

tree ::= LEAF num | BRANCH tree tree,

onde o símbolo ::= denota declaração de tipo.

- A declaração é lida como: tree é

1. uma LEAF que contém um *num*, ou
2. um BRANCH que contém uma tree e outra tree

- LEAF e BRANCH são chamados construtores do tipo.

- num é o campo de LEAF e BRANCH tem dois campos do tipo tree.

- O número de campos associados com o construtor é chamado de aridade.

- Logo a aridade de LEAF é 1 e a de BRANCH é 2.

...Construtores de Tipos

- Construtores podem ser usados como funções para criação de valores, por exemplo,
 $a_1 = \text{BRANCH } (\text{LEAF } 1) (\text{LEAF } 2).$
- Construtores podem aparecer no lado esquerdo de uma equação, por exemplo,
 $\text{reflect } (\text{LEAF } n) = \text{LEAF } n$
 $\text{reflect } (\text{BRANCH } a_1 a_2) =$
 $\text{BRANCH } (\text{reflect } a_2) (\text{reflect } a_1)$
- Essa capacidade dos construtores aparecem do lado esquerdo implica na necessidade de existir *pattern-matching* para fazer a análise dos casos.
- Qual é o valor de $\text{reflect } a_1$?

Variáveis de Tipo

- Declarações de tipo podem conter variáveis de tipo.
- Árvore de folhas de qualquer tipo:
 $\text{tree } * ::= \text{LEAF } * \mid \text{BRANCH } (\text{tree } *) (\text{tree } *)$
- O símbolo $*$ é chamado variável de tipo genérica (ou esquemática).
- Assim, é válido o seguinte valor e seu tipo:
 $\text{BRANCH } (\text{LEAF } 'A') (\text{LEAF } 'B') :: \text{tree } \textit{char},$
 onde o símbolo $::$ é lido como "tem tipo".
- Não são válidas folhas com tipos diferentes, por exemplo,
 $\text{BRANCH } (\text{LEAF } 'A') (\text{LEAF } 1).$
- tree é um operador formador de tipos.

Casos Especiais...

- Definição do tipo *built-in* Lista:
 $\text{list } * ::= \text{NIL} \mid \text{CONS } * (\text{list } *)$
- Tradução de Listas de Miranda:
 $[]$ traduz-se para NIL
 $(x : xs)$ traduz-se para $(\text{CONS } x xs)$
 $[x, y, z]$ traduz-se para
 $(\text{CONS } x (\text{CONS } y (\text{CONS } z \text{NIL})))$

...Casos Especiais...

- Definição do tipo *built-in* Tuplas:
 $\text{pair } * ** ::= \text{PAIR } * **$
 $\text{triple } * ** *** ::= \text{TRIPLE } * ** ***$
 $\text{quadruple } * ** *** **** ::=$
 $\text{QUADRUPLE } * ** *** ****$
- Tradução de Tuplas de Miranda:
 (x, y) traduz para $(\text{PAIR } x y)$
 (x, y, z) traduz para $(\text{TRIPLE } x y z)$
 \dots
 $(*, **)$ traduz para $(\text{pair } * **)$
 $(*, **, ***)$ traduz para
 $(\text{triple } * ** ***)$

...Casos Especiais

- Note que a tupla pode conter elementos de tipos diferentes:
 $(3, 'a') :: \text{PAIR } num \text{ char}$
- Tradução de Tipos Enumerados
 - São tipos onde cada construtor tem aridade zero, por exemplo:
 - $\text{color} ::= \text{RED} \mid \text{GREEN} \mid \text{BLACK}$
 - $\text{bool} ::= \text{TRUE} \mid \text{FALSE}$

Tipos Estruturados Gerais...

- Forma genérica de um tipo estruturado:

$$T ::= c_1 T_{1,1} \dots T_{1,R_1} \mid \dots \mid c_n T_{n,1} \dots T_{n,R_n}$$

onde:

- c_i são construtores de aridade R_i
- $T_{i,j}$ são os tipos dos campos

...Tipos Estruturados Gerais

- Os tipos estruturados gerais podem ser vistos como uma união disjunta:

$$T = T_1 + T_2 + \dots + T_n,$$

onde T_i é visto como um produto cartesiano:

$$T_i = T_{i,1} \times T_{i,2} \times \dots \times T_{i,r_i}.$$

- Tipo estruturado pode ser visto como uma soma de produtos
- A soma tem mais de um construtor, e o produto tem exatamente um construtor.
- Exemplos de soma: tipos enumerados, listas, registros
- Exemplos de produto: as tuplas.

Pattern Matching...

- Um *Pattern* pode ser:

- constante k
- variável v
- *pattern* construtor

- *Pattern* construtor é da forma

$$(c \ p_1 \ \dots \ p_r),$$

onde

- c é um construtor de aridade r
- p_i são *patterns*

...Pattern Matching

- Lembremos da definição da função `reflect` :
 $\text{reflect (LEAF } n) = \text{LEAF } n$
 $\text{reflect (BRANCH } a_1 a_2) =$
 $\text{BRANCH (reflect } a_2) (\text{reflect } a_1)$
- Quando `reflect` é aplicado a um argumento, esse é avaliado para ver se existe um *matching* entre ele e um dos *patterns*:
 - (LEAF n)
 - (BRANCH $a_1 a_2$)

Características do *pattern-matching*...

- *Patterns* constantes:
 $\text{fat } 0 = 1$
 $\text{fat } n = n * \text{fat } (n - 1),$
- Sobreposição de *patterns*:
 - Ocorre quando um valor pode casar com dois *patterns* diferentes.
 - No exemplo acima, o valor 0 casa com os *patterns* 0 e n .

...Características do *pattern-matching*

- *patterns* aninhados:
 - Seja a seguinte função que determina o último elemento de uma lista:
 $\text{ultimo } (x : []) = x$
 $\text{ultimo } (x : xs) = \text{ultimo } xs.$
 - Aqui o *pattern* está aninhado em $(x : [])$.
- Conjunto de equações não exaustivas:
 - Se o argumento de entrada de `ultimo` for a lista vazia, ela não casará com nenhum *pattern*.

...Características do *pattern-matching*...

- Múltiplos argumentos:
 - Temos a função booleana *ou-exclusivo* com necessidade de aplicar um *pattern-matching* a vários argumentos:
 $\text{xor FALSE } y = y$
 $\text{xor TRUE FALSE} = \text{TRUE}$
 $\text{xor TRUE TRUE} = \text{FALSE}$

...Características do *pattern-matching*...

- **Equações condicionais:**
 - O uso de guardas permite controlar a seleção das alternativas.
 - A função `fat` poderia ser redefinida da seguinte maneira:


```
fat n = 1, n = 0
fat n = n * fat (n - 1)
```
 - Guarda interage com o *pattern-matching*:
 1. Executa *pattern-matching*
 2. Verifica as guardas das equações que casaram
 3. Se nenhuma guarda avalia em `TRUE`, continua *pattern*.

...Características do *pattern-matching*

- **Variáveis repetidas:**
 - Variáveis podem aparecer repetidas no lado esquerdo, implicando que seus valores devem ser iguais.
 - Usando esse recurso vamos definir uma função `noDups` que retira os elementos duplicados de uma lista:

$$\text{noDups } [] = []$$

$$\text{noDups } [x] = [x]$$

$$\text{noDups } (x: x: xs) = \text{noDups}(x: xs)$$

$$\text{noDups } (x: y: xs) = x: \text{noDups}(y: xs) .$$
 λ -abstrações com *pattern-matching*

- Dada uma definição de função do tipo:

$$f\ p = E$$
 onde p é um *pattern*, podemos traduzi-la:

$$\text{TD}[f\ p = E] \equiv f = \lambda \text{TE}[p]. \text{TE}[E]$$
- Por exemplo, seja a função `fst`, que retorna o primeiro elemento de um par: $\text{fst}(x, y) = x$
- Teremos a tradução da seguinte maneira:

$$\text{TD}[\text{fst}(x, y) = x] \equiv \text{fst} = \lambda(\text{PAIR } x\ y). x$$
- Novo tipo de abstração: a abstração lambda com *pattern-matching*.
- Veremos agora como transformar uma definição de função geral em Miranda para abstrações lambda com *pattern-matching*.
- Depois mostraremos qual o significado exato dessas abstrações.

Equações Múltiplas e Falha...

- Seja uma definição Miranda da forma:

$$f\ p_1 = E_1$$

$$f\ p_2 = E_2$$

$$\dots$$

$$f\ p_n = E_n$$
- A semântica que esperamos dessa definição é que tentemos a primeira equação, e somente se ela falhar tentemos a segunda e assim por diante.
- Como casamento (*matching*) pode falhar, introduz-se um novo valor primitivo `FAIL`, que é retornado quando um *pattern-matching* falha.

...Equações Múltiplas e Falha...

- Seja a seguinte função infix \square descrita pela seguinte equação semântica:

$$a \quad \square b = a, \text{ se } a \neq \perp \text{ e } a \neq \text{FAIL}$$

$$\text{FAIL} \quad \square b = b$$

$$\perp \quad \square b = \perp.$$

- É fácil verificar que \square possui propriedade associativa e que sua identidade é FAIL .
- Se todos *pattern-matches* falharem, será retornado ERROR , que é um valor especial cuja avaliação indica um erro.

...Equações Múltiplas e Falha ...

- Agora podemos traduzir a função f de Miranda para o λ -Calculus enriquecido:

$$f = \lambda x. (((\lambda p'_1. E'_1) x)$$

$$\square \dots$$

$$\square ((\lambda p'_n. E'_n) x)$$

$$\square \text{ERROR}),$$

onde:

- x é uma nova variável que não ocorre livre em lugar nenhum e serve de entrada para cada definição;
- ainda p'_i é o resultado de traduzir p_i
- e E'_i é o resultado de traduzir E_i .

...Equações Múltiplas e Falha

- Por exemplo, a função *head*,

$$\text{head} (x : xs) = x,$$

seria traduzida para

$$\text{head} = \lambda k. (((\lambda (\text{CONS } x \text{ } xs) . x) k) \square \text{ERROR}).$$

- Se *head* fosse aplicada a NIL, então o resultado seria ERROR

Funções com Argumentos Múltiplos

- Para traduzir funções com múltiplos argumentos usa-se o esquema:

$$f \ p_1 \ p_2 \ \dots \ p_m = E \text{ onde } p_i \text{ são } \textit{patterns}$$

$$f = \lambda v_1 . \dots \lambda v_m .$$

$$(((\lambda p'_1 . \dots \lambda p'_m . E') v_1 \dots v_m) \square \text{ERROR})$$

onde

- $v_1 \dots v_m$ são novas variáveis que não ocorrem livres em E ;
- p'_i é o resultado de traduzir de p_i ;
- E' é o resultado de traduzir E

- Se o primeiro *pattern-matching* falhar deve ser aplicada a regra de redução:

$$\text{FAIL } E \rightarrow \text{FAIL}$$

para que toda expressão falhe.

Equações Condicionais...

- Equações condicionais podem ser traduzidas com o seguinte esquema:

TR[rhs] que traduz o lado direito da definição.

TR [A₁, G₁ = A₂, G₂... = A_n, G_n] ≡

(IF TE[G₁]TE[A₁]

(IF TE[G₂]TE[A₂]

... ..

(IF TE[G_n]TE[A_n] FAIL) ...))

onde

- A_i é a expressão
- G_i são expressões booleanas

...Equações Condicionais

- Por exemplo,

$\text{gcd } a \ b = \text{gcd } (a - b) \ b, \ a > b$

$= \text{gcd } a \ (b - a), \ a < b$

$= a, \ a = b$

é traduzido para

(IF (a > b) (gcd (a - b) b)

(IF (a < b) gcd a (b - a)

(IF (a = b) a FAIL)))

Dificuldades com Variáveis Repetidas...

- Expressões condicionais podem ser usadas para eliminar repetição:

– Definição I:

nasty x x True = 1

nasty x y z = 2

– Definição II:

nasty x y True = 1, x = y

nasty x y z = 2

...Dificuldades com Variáveis Repetidas

- Compare os resultados das definições acima na aplicação:

nasty bottom 3 False

- Qual seria a ordem para se efetuar as comparações dos argumentos?

– Seja a definição

multi p q q p = 1

multi p q r s = 2

– Considere a aplicação

multi bottom 2 3 4

Cláusulas *where*

- Miranda permite que o dado direito de uma definição seja qualificado com uma cláusula *where*. Por exemplo,

$$\begin{aligned} \text{gcd } a \ b &= \text{gcd diff } b && , a > b \\ &= \text{gcd } a \ (- \text{diff}) && , a < b \\ &= a && , a = b \\ \text{where diff} &= a - b . \end{aligned}$$

que traduz para,

$$\begin{aligned} \text{gcd } a \ b &= \\ &\text{letrec TD}[\text{diff} = a - b] \\ &\text{in (IF TE}[(a > b)]\text{TE}[(\text{gcd (diff) } b) \\ &\quad (\text{IF TE}[a < b]\text{TE}[\text{gcd } a \ (- \text{diff})]] \\ &\quad (\text{IF TE}[a = b]\text{TE}[a]\text{FAIL}))) \end{aligned}$$

Patterns no Lado Esquerdo

- Seja a seguinte definição Miranda:

$$\begin{aligned} \text{somapar } w &= x + y \\ \text{where } (x, y) &= w \end{aligned}$$

onde (x, y) é um *pattern* no lado esquerdo da definição da cláusula *where*.

- A tradução para o λ -Calculus enriquecido seria:

$$\begin{aligned} \text{somapar} &= \lambda w . \\ &\text{letrec (PAIR } x \ y) = w \\ &\text{in } (+ \ x \ y) \end{aligned}$$

- Podemos generalizar o seguinte esquema de tradução:

$$\text{TD}[p = R] \equiv \text{TE}[p] = \text{TR}[R]$$

onde p é um *pattern* e R é um lado direito de uma aplicação

O Esquema de Tradução TR

- $\text{TR } [A_1, G_1 \dots = A_n, G_n \text{ where } D_1 \dots D_m]$
 $\equiv \text{letrec TD}[D_1] \dots \text{TD}[D_m]$
 $\text{in (IF TE}[G_1]\text{TE}[A_1] \dots (\text{IF TE}[G_n]\text{TE}[A_n]\text{FAIL}) \dots)$

onde

- A_i é uma expressão
- G_i é uma expressão com valor booleano
- D_i é um definição
- Se G_n for TRUE ou estiver ausente, então a expressão IF final deve ser substituída por $\text{TE}[A_n]$

O Esquema de Tradução TE

- $\text{TE}[:] \equiv \text{CONS}$
- $\text{TE}[[]] \equiv \text{NIL}$
- $\text{TE}[[E_1, \dots, E_n]] \equiv$
 $\text{CONS TE}[E_1]\text{TE}[[E_2, \dots, E_n]]$
- $\text{TE}[(E_1, E_2)] \equiv \text{PAIR TE}[E_1]\text{TE}[E_2]$
- $\text{TE}[(E_1, E_2, E_3)] \equiv$
 $\text{TRIPLE TE}[E_1]\text{TE}[E_2]\text{TE}[E_3]$

e assim por diante

$$\begin{aligned} \text{TE}[\text{True}] &\equiv \text{TRUE} \\ \text{TE}[\text{False}] &\equiv \text{FALSE} \\ \text{TE}[k] &\equiv k \\ \text{TE}[v] &\equiv v \end{aligned}$$

O Esquema de Tradução TE

- $TE[E_1 E_2] \equiv TE[E_1] TE[E_2]$
- $TE[E_1 \text{ infix } E_2] \equiv TE[\text{infix}] TE[E_1] TE[E_2]$
- $TE[E_1 \$v E_2] \equiv TE[v] TE[E_1] TE[E_2]$

onde:

- k é uma constante ou operador *built-in*
- v e v_i são variáveis
- E e E_i são expressões
- *infix* é um operador infix

O Esquema de Tradução TD

- $TD[p = R] \equiv TE[p] = TR[R]$
- $TD [f p_{1,1} \dots p_{1,m} = R_1 \dots f p_{n,1} \dots p_{n,m} = R_n] \equiv$
 $f = (\lambda v_1 \dots \lambda v_m \cdot$
 $((\lambda TE[p_{1,1}] \dots \lambda TE[p_{1,m}] \cdot$
 $TR[R_1]) v_1 \dots v_m)$
 $\square \dots$
 $\square ((\lambda TE[p_{n,1}] \dots \lambda TE[p_{n,m}] \cdot$
 $TR[R_n]) v_1 \dots v_m)$
 $\square \text{ERROR}))$

onde:

- f é uma variável e v_i é uma variável não livre em quaisquer R_j
- $p_{i,j}$ é um *pattern* e R e R_i são expressões do lado direito

Patterns do Tipo Variável e Constante...

- Se um *pattern* p é uma variável v então a λ -abstração com *pattern-matching*,
 $(\lambda p. E)$,
 é uma simples λ -abstração
 $(\lambda v. E)$
- A semântica dos *patterns* do tipo constante é dada por:
 $\text{Eval}[\lambda k. E] a$, onde k é uma constante
- O argumento a pode ser tal que:
 1. a é o mesmo que k , ou,
 2. a é \perp , ou,
 3. a é alguma outra coisa.

...Patterns do Tipo Variável e Constante

- Assim temos,
 1. $\text{Eval}[\lambda k. E] a = \text{Eval}[E]$, se $k = \text{Eval}[a]$
 2. $\text{Eval}[\lambda k. E] a = \perp$, se $a = \perp$
 3. $\text{Eval}[\lambda k. E] a = \text{FAIL}$, caso contrário
- Exemplos:
 - $(\lambda 1. + 3 4) 1 \mapsto + 3 4$
 - $(\lambda 1. + 3 4) 2 \mapsto \text{FAIL}$

A Semântica dos *patterns* Construtores de Soma...

- Sejam o *pattern* construtor de soma da forma

$$(s \ p_1 \ \dots \ p_r).$$

- Suas regras semânticas são:

$$\text{Eval}[\lambda(s \ p_1 \ \dots \ p_r). \ E] (s \ a_1 \ \dots \ a_r) =$$

$$\text{Eval}[\lambda \ p_1 \dots \ \lambda \ p_r. \ E] a_1 \ \dots \ a_r$$

$$\text{Eval}[\lambda(s \ p_1 \ \dots \ p_r). \ E] (s' \ a_1 \ \dots \ a_r) =$$

FAIL se $s \neq s'$

$$\text{Eval}[\lambda(s \ p_1 \ \dots \ p_r). \ E] \perp = \perp$$

... A Semântica dos *patterns* Construtores de Soma

- Exemplos:

$$- (\lambda(\text{BRANCH } t_1 \ t_2). \text{BRANCH } t_2 \ t_1)(\text{LEAF } 0) \mapsto \text{FAIL}$$

$$- (\lambda(\text{BRANCH } t_1 \ t_2). \text{BRANCH } t_2 \ t_1) (\text{BRANCH}(\text{LEAF } 0)(\text{LEAF } 1))$$

$$\mapsto (\lambda t_1. \lambda t_2. \text{BRANCH } t_2 \ t_1)(\text{LEAF } 0)(\text{LEAF } 1)$$

$$\mapsto (\lambda t_2. \text{BRANCH } t_2 (\text{LEAF } 0))(\text{LEAF } 1)$$

$$\mapsto (\text{BRANCH } (\text{LEAF } 1)(\text{LEAF } 0))$$

Avaliação Lazy de Patterns de Produto

- Seja a seguinte função Miranda:

$$\text{parzero } (x, y) = 0$$

Qual seria o valor de retorno de `parzero` \perp ?

- A resposta depende da semântica adotada para a avaliação do *pattern*.

- A resposta poderia ser:

- $\text{Eval}[\text{parzero}] \perp = 0$, se a semântica fosse a de um *lazy product-matching*

- $\text{Eval}[\text{parzero}] \perp = \perp$, se fosse um *product-matching* estrito.

- A opção de se trabalhar o *lazy product-matching*, possibilita manipular corretamente funções que tem como entrada dados infinitos, e também possibilita implementar corretamente recursão mútua.

Semântica do *Lazy Product-Matching*...

- $\text{Eval}[\lambda(t \ p_1 \ \dots \ p_r). \ E] a =$

$$\text{Eval}[\lambda \ p_1 \dots \ \lambda \ p_r. \ E]$$

$$(\text{SEL-t-1 } a) \ \dots \ (\text{SEL-t-r } a)$$

- SEL-t-i é uma função primitiva que seleciona o *i*-ésimo campo de um objeto estruturado com o construtor *t*:

$$\text{SEL-t-i } (t \ a_1 \ \dots \ a_i \ \dots \ a_r) = a_i$$

$$\text{SEL-t-i } \perp = \perp.$$

...Semântica do *Lazy Product-Matching*

• Exemplo:

parzero $(x, y) = 0$
 parzero = $\lambda(\text{PAIR } x \ y) . 0$

Eval[parzero] \perp
 = Eval[$\lambda(\text{PAIR } x \ y) . 0$] \perp
 = Eval[$\lambda x . \lambda y . 0$]
 (SEL-PAIR-1 \perp)(SEL-PAIR-2 \perp)
 = Eval[$\lambda y . 0$](SEL-PAIR-2 \perp)
 = 0

Vantagens da Avaliação Lazy...

• Seja a definição de firsts:

firsts [] = (0,0)
 firsts (x:xs) = combine x (firsts xs)
 combine x (od,ev) = (x,ev), odd x
 = (od,x), even x

• Avaliação estrita:

firsts [1..]
 → combine 1 (firsts[2..])
 → combine 1 (combine 2 (firsts[3..]))
 → ...

...Vantagens da Avaliação Lazy

• Avaliação Lazy:

firsts [1..]
 → combine 1 (firsts[2..])
 → (1,SEL-PAIR-2(firsts[2..]))
 → (1,SEL-PAIR-2
 (combine 2(firsts[3..])))
 → (1,SEL-PAIR-2
 (SEL-PAIR-1(firsts[3..]),2))
 → (1,2)

Avaliação Lazy e Não-Terminação...

• Considere as possíveis definições do produto:

- $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$
 O bottom do domínio é (\perp, \perp)
- $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\} \cup \{\perp\}$
 O bottom do domínio é \perp

...Avaliação Lazy e Não-Terminação

- Avaliação lazy permite esconder as diferenças entre (\perp, \perp) e \perp .
- $\text{Eval}[\lambda(\text{PAIR } p_1 \ p_2) . E] \perp$
 $\rightarrow \text{Eval}[\lambda p_1 . \lambda p_2 . E]$
 $(\text{SEL-PAIR-1 } \perp) (\text{SEL-PAIR-2 } \perp)$
 $\rightarrow \text{Eval}[\lambda p_1 . \lambda p_2 . E] \perp \perp$
- $\text{Eval}[\lambda(\text{PAIR } p_1 \ p_2) . E] (\text{PAIR } \perp \ \perp)$
 $\rightarrow \text{Eval}[\lambda p_1 . \lambda p_2 . E]$
 $(\text{SEL-PAIR-1 } (\text{PAIR } \perp \ \perp))$
 $(\text{SEL-PAIR-2 } (\text{PAIR } \perp \ \perp))$
 $\rightarrow \text{Eval}[\lambda p_1 . \lambda p_2 . E] \perp \perp$

Expressões-case...

- As transformações na seção anterior produz código ineficiente, pois é necessário testar o FAIL a cada momento que é tentada uma equação da definição da função.
- Muitas vezes, um único teste poderia servir para selecionar a equação apropriada.
- Expressões-case é uma forma simples de *pattern-matching*, que tem a propriedade de necessitar de um único teste.

...Expressões-case

- Pode-se traduzir a definição de reflect
 $\text{reflect } (\text{LEAF } n) = \text{LEAF } n$
 $\text{reflect } (\text{BRANCH } t_1 \ t_2) =$
 $\text{BRANCH } (\text{reflect } t_2) (\text{reflect } t_1)$
 para a seguinte forma:
 $\text{reflect} =$
 $\lambda t . \text{case } t \text{ of}$
 $\text{LEAF } n \Rightarrow \text{LEAF } n$
 $\text{BRANCH } t_1 \ t_2 \Rightarrow$
 $\text{BRANCH } (\text{reflect } t_2) (\text{reflect } t_1)$

Expressões-case...

- O ponto importante sobre as expressões-case é que os *patterns* não são aninhados e cobrem todos os construtores do tipo. Isso as torna simples de implementar.
- A forma geral de uma expressão-case é:
 $\text{case } v \text{ of}$
 $c_1 \ v_{1,1} \ \dots \ v_{1,r_1} \Rightarrow E_1$
 \dots
 $c_n \ v_{n,1} \ \dots \ v_{n,r_n} \Rightarrow E_n$
 onde v é uma variável,
 E_i são expressões,
 $v_{i,j}$ são variáveis distintas,
 c_i é uma família completa de construtores

...Expressões-case

- Formalmente a construção é equivalente à:

$$((\lambda (c_1 v_{1,1} \dots v_{1,r_1}). E_1) v)$$

$$\square \dots \dots$$

$$\square ((\lambda (c_n v_{n,1} \dots v_{n,r_n}). E_n) v)$$

- Intuitivamente, expressões-case correspondem a várias expressões aninhadas, IF... THEN... ELSE.

I. COMPILAÇÃO DO PATTERN MATCHING

Execução de Patterns Múltiplos ...

- Considere a função:

```
mappairs f [ ] ys = [ ]
```

```
mappairs f (x:xs) [ ] = [ ]
```

```
mappairs f (x:xs) (y:ys) =
  f x y : mappairs f xs ys
```

- O pattern-matching de

```
mappairs (+) [1,2] [3,4]
```

segue os seguintes passos:

1. inicia-se com a 1ª equação
2. tenta o casamento de (+) com f, que tem sucesso
3. tenta casamento de [1,2] com [], que falha

... Execução de Patterns Múltiplos

- Dada a função:

```
mappairs f [ ] ys = [ ]
```

```
mappairs f (x:xs) [ ] = [ ]
```

```
mappairs f (x:xs) (y:ys) =
  f x y : mappairs f xs ys
```

- O pattern-matching de

```
mappairs (+) [1,2] [3,4]
```

continua com os seguintes passos:

4. repete-se o processo com a 2ª equação
5. O casamento de (+) com f e de [1,2] com (x:xs) têm sucesso, mas o de [3,4] com [] falha
6. Então faz-se o pattern-matching com a 3ª equação, que apropriadamente liga f, x, xs, y e ys

Uma Melhor Solução

- O ideal é tentar o casamento uma única vez por parâmetro

- Uma solução é compilar `mappairs` para:

```
mappairs = λf.λxs'.λys'.
  case xs' of
    NIL ⇒ NIL
    CONS x xs ⇒
      case ys' of
        NIL ⇒ NIL
        CONS y ys ⇒ CONS(f x y)(mappairs f xs ys)
```

- O algoritmo que converte expressões como a do `mappairs` em expressões com a acima é chamado de **Compilador de Pattern-Matching**

Otimização de Patterns Aninhados

- Considere a função:

```
nodups [ ] = [ ]
nodups [x] = [x]
nodups (y:x:xs)
  = nodups (x:xs), y = x
  = y : nodups (x:xs), otherwise
```

... Otimização de Patterns Aninhados

- O pattern-matching seria mais eficiente em:

```
nodups = λxs''.
  case xs'' of
    NIL ⇒ NIL
    CONS x' xs' ⇒
      case xs' of
        NIL ⇒ CONS x' NIL
        CONS x xs ⇒ IF (= x' x)
                      (nodups (CONS x xs))
                      (CONS x' (nodups(CONS(x xs))))
```

O Compilador de Pattern-Matching

- A função Miranda

```
f p1,1 ... p1,n = E1
...
f pm,1 ... pm,n = Em
```

pode ser traduzida para:

```
f = λu1. ... λun.
  ((λp'1,1. ... λp'1,n. E'1) u1 ... un)
  □ .....
  □ ((λp'm,1. ... λp'm,n. E'm) u1 ... un)
  □ ERROR
```

... O Compilador de Pattern-Matching

- O objetivo é transformar uma expressão da forma

$((\lambda p_{1,1} \dots \lambda p_{1,n}. E_1) u_1 \dots u_n)$

 $((\lambda p_{m,1} \dots \lambda p_{m,n}. E_m) u_1 \dots u_n)$
 E

em expressão equivalente que use expressões-caseao invés de λ -abstrações com *pattern-matching*.

A Função match

- Função de transformação que:
 - recebe as várias partes da expressão de entrada $(p_{i,j}, E_i, u_j)$
 - produz a expressão transformada.
- Função de transformação que:
 - recebe as várias partes da expressão de entrada $(p_{i,j}, E_i, u_j)$
 - produz a expressão transformada.

A Função match

- Uma chamada para *match* toma três argumentos:

- uma lista de variáveis;
- uma lista de equações;
- uma expressão *default*.

Onde cada equação é um par, consistindo em :

- uma lista de *patterns* representando o lado esquerdo da equação;
- uma expressão representando o lado direito da equação.

- Uma chamada tem a seguinte forma:

$match [u_1, \dots, u_n]$
 $[([p_{1,1}, \dots, p_{1,n}], E_1)$
 $\dots \dots \dots$
 $([p_{m,1}, \dots, p_{m,n}], E_m)]$
 E

Exemplo Modelo do Uso de match

- Considere a definição:

$demo\ f\ []\ ys = A\ f\ ys$
 $demo\ f\ x:xs\ [] = B\ f\ x\ xs$
 $demo\ f\ (x:xs)\ (y:ys) = C\ f\ x\ xs\ y\ ys$

- *demo* pode ser transformada em:

$demo = \lambda u_1 . \lambda u_2 . \lambda u_3 .$
 $((\lambda f . \lambda NIL . \lambda ys . A\ f\ ys)\ u_1\ u_2\ u_3)$
 $((\lambda f . \lambda (CONS\ x\ xs) . \lambda NIL . B\ f\ x\ xs)\ u_1\ u_2\ u_3)$
 $((\lambda f . \lambda (CONS\ x\ xs) . \lambda (CONS\ y\ ys) .$
 $C\ f\ x\ xs\ y\ ys)\ u_1\ u_2\ u_3)$
 ERROR

... Exemplo Modelo do Uso de match

• Dada a expressão

$$\text{demo} = \lambda u_1 . \lambda u_2 . \lambda u_3 .$$

$$((\lambda f . \lambda \text{NIL} . \lambda \text{ys} . A f \text{ys}) u_1 u_2 u_3)$$

$$\square ((\lambda f . \lambda (\text{CONS } x \text{ xs}) . \lambda \text{NIL} . B f x \text{xs}) u_1 u_2 u_3)$$

$$\square ((\lambda f . \lambda (\text{CONS } x \text{ xs}) . \lambda (\text{CONS } y \text{ ys}) . \\ C f x \text{xs} y \text{ys}) u_1 u_2 u_3)$$

$$\square \text{ERROR}$$

• Com match, demo transforma-se em:

$$\text{demo} = \lambda u_1 . \lambda u_2 . \lambda u_3 . \text{match } [u_1, u_2, u_3]$$

$$[([f, \text{NIL}, \text{ys}], (A f \text{ys})),$$

$$([f, \text{CONS } x \text{xs}, \text{NIL}], (B f x \text{xs})),$$

$$([f, \text{CONS } x \text{xs}, \text{CONS } y \text{ys}], (C f x \text{xs} y \text{ys}))]$$

$$\text{ERROR}$$
A Regra da Variável ...• Se toda equação começa com um *pattern* do tipo variável, a chamada de match terá a seguinte forma:
$$\text{match } (u : us)$$

$$[((v_1 : ps_1), E_1), \dots (v_m : ps_m), E_m] \\ E$$

• Pode-se ainda reduzir para a seguinte chamada de match:

$$\text{match } us$$

$$[(ps_1, E_1[u/v_1]), \dots (ps_m), E_m[u/v_m])] \\ E$$

onde $E[M/x]$ significa E com x substituído por M .

... A Regra da Variável ...• Considere a chamada de match, onde f é uma variável:
$$\text{match}$$

$$[u_1, u_2, u_3]$$

$$[([f, \text{NIL}, \text{ys}], (A f \text{ys})),$$

$$([f, \text{CONS } x \text{xs}, \text{NIL}], (B f x \text{xs})),$$

$$([f, \text{CONS } x \text{xs}, \text{CONS } y \text{ys}],$$

$$(C f x \text{xs} y \text{ys}))$$

$$]$$

$$\text{ERROR}$$
... A Regra da Variável

• A chamada reduz-se a:

$$\text{match}$$

$$[u_2, u_3]$$

$$[([\text{NIL}, \text{ys}], (A u_1 \text{ys})),$$

$$([\text{CONS } x \text{xs}, \text{NIL}], (B u_1 x \text{xs})),$$

$$([\text{CONS } x \text{xs}, \text{CONS } y \text{ys}],$$

$$(C u_1 x \text{xs} y \text{ys}))$$

$$]$$

$$\text{ERROR}$$

A Regra do Construtor

- Os construtores são de um tipo com a seguinte forma: c_1, \dots, c_k
- As equações podem ser agrupadas em qs_1, \dots, qs_k , onde um grupo de equações qs_i começa com o construtor c_i
- Se existe um construtor c_i que não começa em nenhuma equação então faça qs_i vazia.
- Então, a chamada do `match` terá a seguinte forma:
 $match (u:us) (qs_1 ++ \dots ++ qs_k) E$
 onde $++$ é a concatenação de listas e qs_i tem a forma:
 $[((c_i ps'_{i,1}) : ps_{i,1}), E_{i,1}) ,$
 \dots
 $((c_i ps'_{i,m_i}) : ps_{i,m_i}), E_{i,m_i})]$

... A Regra do Construtor ...

- A chamada do `match`:
 $match (u : us) (qs_1 ++ \dots ++ qs_k) E,$
 onde qs_i tem a forma:
 $[((c_i ps'_{i,1}) : ps_{i,1}), E_{i,1}) ,$
 \dots
 $((c_i ps'_{i,m_i}) : ps_{i,m_i}), E_{i,m_i})]$
- Pode ser reduzida para a seguinte expressão-case:
 $case u of$
 $c_1 us'_1 \Rightarrow match(us'_1 ++ us) qs'_1 E$
 $\dots \dots \dots$
 $c_k us'_k \Rightarrow match(us'_k ++ us) qs'_k E$

... A Regra do Construtor ...

- Na expressão-case:
 $case u of$
 $c_1 us'_1 \Rightarrow match(us'_1 ++ us) qs'_1 E$
 \dots
 $c_k us'_k \Rightarrow match(us'_k ++ us) qs'_k E$
- qs'_i tem a forma:
 $[((ps'_{i,1} ++ ps_{i,1}), E_{i,1}),$
 \dots
 $((ps'_{i,m_i} ++ ps_{i,m_i}), E_{i,m_i})]$
- E us'_i é uma lista de novas variáveis contendo uma variável para cada campo em c_i .

... A Regra do Construtor ...

- Seja a chamada de `match`:
 $match [u_2, u_3]$
 $[([NIL,ys], (A u_1 ys)),$
 $([CONS x xs, NIL], (B u_1 x xs)),$
 $([CONS x xs, CONS y ys],$
 $(C u_1 x xs y ys))] ERROR$
- Que foi obtida pelas seguintes substituições:
 - qs_2 por: $[([CONS x xs, NIL], (B u_1 x xs)), ([CONS x xs, CONS y ys], \dots)]$
 - c_2 por $CONS$
 - $ps'_{2,1}$ por $[x, xs]$
 - $ps_{2,1}$ por NIL

... A Regra do Construtor

- $E_{2,1}$ por $(B\ u_1\ x\ xs)$
- $ps'_{2,2}$ por $[x\ xs]$
- $ps_{2,2}$ por $[CONS\ y\ ys]$
- $E_{2,2}$ por $(C\ u_1\ x\ xs\ y\ ys)$
- qs'_2 por $[(x\ xs,\ NIL), (B\ u_1\ x\ xs),$
 $(x\ xs,\ CONS\ y\ ys), (C\ u_1\ x\ xs\ y\ ys)]$
- A lista de novas variáveis us'_2 é $[u_4, u_5]$

A Regra do Vazio

- Depois de aplicações repetidas, poderá ocorrer chamada de *match* da forma:

$$\text{match } [] \\ [([], E_1), \dots ([], E_m)] \\ E$$
- que reduz-se para

$$E_1 \square E_2 \square \dots \square E_m \square E.$$
- Se pudermos garantir que E_1, E_2, \dots, E_m nunca produzirão FAIL, *match* acima reduz-se a:
 - E_1 , se $m > 0$
 - E , se $m = 0$

Um exemplo

- Com as regras dadas pode-se verificar que a seguinte definição:

```
mappairs f [ ] ys = [ ]
mappairs f (x:xs) [ ] = [ ]
mappairs f (x:xs) (y:ys) = f x y : mappairs f xs ys
```

- é equivalente a:

```
mappairs = λu1. λu2. λu3.
  case u2 of
  NIL ⇒ NIL
  CONS u4 u5 ⇒
    case u3 of
    NIL ⇒ NIL
    CONS u6 u7 ⇒ CONS (u1 u4 u6) (mappairs u1 u5 u7)
```

A Regra da Mistura...

- Há situações em que nem todas as equações começam com uma variável e nem todas as equações começam com um construtor.
- Neste caso, chamada de *match* é da forma:

$$\text{match } us\ qs\ E,$$
 onde a lista de equações qs pode ser particionada em k listas, qs_1, \dots, qs_k , tais que

$$qs_1 ++ \dots ++ qs_k = qs.$$
- A partição deve ser tal que cada qs_i
 - ou tem todas as equações que começam com variável,
 - ou todas as equações que começam com um construtor.
- Assim, a chamada do *match* pode ser reduzida para:

$$\text{match } us\ qs_1\ (\text{match } us\ qs_2\ (\dots (\text{match } us\ qs_k\ E)\dots))$$

...A Regra da Mistura ...

- Como exemplo, a seguinte definição Miranda:

$$\begin{aligned} demo' f [] ys &= A f ys \\ demo' f xs [] &= B f xs \\ demo' f (x : xs) (y : ys) &= C f x xs y ys \end{aligned}$$

- pode ser transformada em

$$demo' = \lambda u_1. \lambda u_2. \lambda u_3. E$$

onde E está definido a seguir

...A Regra da Mistura

- $demo' = \lambda u_1. \lambda u_2. \lambda u_3.$

```
case u2 of
  NIL => (A u1 u3)
  CONS u4 u5 =>
    case u3 of
      NIL => (B u1 u2)
      CONS u6 u7 =>
        case u2 of
          NIL => ERROR
          CONS u4 u5 =>
            case u3 of
              NIL => ERROR
              CONS u6 u7 => (C u1 u4 u5 u6 u7)
```

Compilação de Patterns ...

- Estrutura para representar patterns:

```
pattern ::= Var variable
         | CON constructor [ pattern ]
variable == [ char ]
constructor == [ char ]
```

- Por exemplo, $(x : xs)$ é representado por $(CON "CONS" [VAR "x" , VAR "xs"])$

... Compilação de Patterns

- Funções auxiliares:

- Retorna a aridade:

```
arity :: constructor -> num
```

- Retorna todos os construtores do tipo do argumento:

```
constructors :: constructor ->
[constructor]
```

Por exemplo,

- $(arity "NIL")$ retorna 0
- $(arity "CONS")$ retorna 2
- $(constructors "NIL")$ retorna ["NIL", "CONS"]

Representação de Expressões...

- `expression ::= CASE variable [clause]`
`| FATBAR expression expression`
`| ...`
`clause ::= CLAUSE constructor`
`[variable] expression`

- **Por exemplo, a expressão case:**

```
case xs of
  NIL      => E1
  CONS y ys => E2
```

- **seria representada por:**

```
CASE "xs"
  [ CLAUSE "NIL" [] E1' ,
    CLAUSE "CONS" ["y" , "ys" ] E2' ]
```

...Representação de Expressões

- **A expressão $E1 \square E2$ seria representada por:**

```
FATBAR E1' E2'
```

- **Função de substituição:**

```
subst :: expression -> variable ->
       variable -> expression
```

- **Se E representa (F x y) então**

```
subst E "_u" "x") representa (F _u y)
```

Tratamento de Equações...

- **Uma equação é uma lista de patterns e uma expressão:**

```
equation == ( [pattern] , expression)
```

- **Funções isVar e isCON determinam se o argumento é variável ou construtor:**

```
isVar :: equation -> bool
isVar (VAR v : ps, e) = True
isVar (CON c ps' : ps, e) = False
```

```
isCON :: equation -> bool
isCON q = ~ (isVAR q)
```

```
getCON :: equation -> constructor
getCON (CON c ps' : ps, e) = c
```

...Tratamento de Equações

- **Função makeVar gera novas variáveis:**

```
makeVar :: num -> variable
makeVar k = "_u" ++ show k
```

Função partition

- Função partition é usada na implementação de regra da mistura:

```
partition :: (* -> **) -> [*] -> [[*]]
partition f [ ] = [ ]
partition f [x] = [ [x] ]
partition f ( x : x' : xs)
  = tack x (partition f (x':xs)),
      f x = f x'
  = [x] : partition f (x' : xs),
      otherwise

tack x xss = (x : hd xss) : tl xss
```

Função foldr

- Função foldr:

```
foldr f a [x1, x2, ..., xn] =
  f x1 (f x2 (f x3 (...(f xn a)...))

foldr :: (*->** -> **)->** ->[*] -> **
foldr f a [ ] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Função match...

- Considere uma chamada de match da forma:

```
mappairs = λu1. λu2. λu3. match [u1, u2, u3]
  [ ([f, NIL, ys], E1 ),
    ([f, CONS x xs, NIL], E2 ),
    ([f, CONS x xs, CONS y ys], E3)
  ]
ERROR
```

...Função match...

- Então, temos:

```
match 3
  ["_u1", "_u2", "_u3"]
  [([VAR "f" ,
    CON "NIL" [],
    VAR "ys"], E1) ,
   ([VAR "f" ,
    CON "CONS" [VAR "x", VAR "xs"],
    CON "NIL" [] ], E2) ,
   ([VAR "f" ,
    CON "CONS" [VAR "x", VAR "xs"],
    CON "CONS" [VAR "y", VAR "ys"]],
    E3)]
  ERROR
```

...Função match...• **Função match:**

```

match::num->[variable] -> [equation]->
      expression -> expression
match k [ ] qs def
  = foldr FATBAR def [e | ([],e) <- qs]
match k (u:us) qs def
  = foldr (matchVarCon k (u:us)) def
        (partition isVar qs)

```

...Função match...• **Função matchVarCon recebe uma lista de equações começando somente com variáveis ou então somente com construtores:**

```

matchVarCon k us qs def
  = matchVar k us qs def, IsVar (hd qs)
  = matchCon k us qs def, isCon (hd qs)

```

• **A função matchVar implementa a regra da variável:**

```

matchVar k (u:us) qs def
  = match k us [(ps, subst e u v) |
                (VAR v : ps, e) <- qs] def

```

...Função match• **As funções matchCon e matchClause implementam a regra do construtor:**

```

matchCon k (u:us) qs def
  = CASE u [matchClause c k (u:us)
           (choose c qs) def | c <- cs]
where cs = constructors (getCon (hd qs))

matchClause c k (u:us) qs def = CLAUSE c us' (
  match(k' + k)(us'++ us)[(ps'++ ps, e)|(CON c ps':ps,e)<-qs]
  def)
where k' = arity c
      us' = [makeVar (i+k) | i <- [1..k']]

choose c qs = [q | q <- qs; getCon q = c]

```

Expressões-case...• **O ponto importante sobre as expressões-case é que os patterns não são aninhados e cobrem todos os construtores do tipo.**• **Isso as torna simples de implementar**• **A forma geral de uma expressão-case é:**

```

case v of
  c1 v1,1 ... v1,r1 ⇒ E1
  ...
  cn vn,1 ... vn,rn ⇒ En

```

onde v é uma variável E_i são expressões $v_{i,j}$ são variáveis distintas c_i são famílias completas de construtores

Expressões-case...

- Formalmente a construção é equivalente a:

$$((\lambda(c_1 v_{1,1} \cdots v_{1,r_1}). E_1)v) \square \cdots$$

$$\square((\lambda(c_n v_{n,1} \cdots v_{n,r_n}). E_n)v)$$

- Intuitivamente expressões-case correspondem a várias expressões IF ... THEN ... ELSE aninhadas

Expressões-case com Cláusula Default...

- Seja a seguinte função definida como:

$$func [] [] = A$$

$$func xs ys = B xs ys$$

- Essa função é transformada pelo compilador em:

$$func = \lambda xs. \lambda ys. case xs of$$

$$NIL \Rightarrow case ys of$$

$$NIL \Rightarrow A$$

$$CONS y' ys' \Rightarrow B xs ys$$

$$CONS x' y' \Rightarrow B xs ys$$
...Expressões-case com Cláusula Default...

- A expressão $(B xs ys)$ aparece duas vezes na definição.
- Se $(B xs ys)$ fosse uma expressão muito grande o tamanho do programa compilado poderia crescer ao fazer sua substituição.
- Resolve-se esse problema modificando as regras dadas, para que os lados direitos de definições nunca apareçam duplicados no processo de compilação.

...Expressões-case com Cláusula Default...

- Em vez da redução da seguinte chamada de *match*:

$$match (u : us)(qs_1 ++ \dots ++ qs_k) E$$

- ser transformada para

$$case u of$$

$$c_1 us'_1 \Rightarrow match (us'_1 ++ us) qs'_1 E$$

$$\dots \quad \dots \quad \dots$$

$$c_k us'_k \Rightarrow match (us'_k ++ us) qs'_k E$$

- transformaremos a chamada para:

$$case u of$$

$$c_1 us'_1 \Rightarrow match (us'_1 ++ us) qs'_1 FAIL$$

$$\dots \quad \dots$$

$$c_k us'_k \Rightarrow match (us'_k ++ us) qs'_k FAIL \square E$$

...Expressões-case com Cláusula *Default*

- O leitor pode transformar a definição de *func* e verificar que $(B\ x\ s\ y\ s)$ aparece somente uma vez e que a nova definição é equivalente à anterior.

Expressões com \square e FAIL...

- Pode-se eliminar certas ocorrências de \square e FAIL .
- Se FAIL nunca será retornado por uma expressão, então é possível eliminar uma ocorrência de \square .
- Suponha que uma expressão E retorne FAIL . Então é necessário uma das condições abaixo:
 - FAIL é mencionado explicitamente em E ,
 - E contém uma abstração lambda com *pattern-matching*, cuja aplicação pode falhar,
 - FAIL é o valor de variável livre de E .

...Expressões com \square e FAIL...

- Se o compilador de *pattern-matching* for aplicado ao programa nenhuma abstração lambda com *pattern* permanecerá, logo a segunda condição nunca ocorrerá.
- Se o programador não puder escrever FAIL explicitamente no programa a terceira condição nunca ocorrerá.
- Então, os FAIL foram introduzidos explicitamente pelo compilador. Possivelmente em dois lugares:
 - na tradução de equações condicionais,
 - na versão otimizada do compilador de *pattern-matching* mostrado na seção anterior.
- No primeiro caso é direta a remoção de \square e FAIL.

...Expressões com \square e FAIL

- Já no segundo caso parece ser inevitável. Há entretanto uma classe restrita de definições de função que sempre podem ser compiladas sem o uso de \square e FAIL
- Regras para Transformar \square e FAIL:
 - se FAIL não pode ocorrer em E_1 ,
 $E_1 \square E_2 \equiv E_1$.
 - se FAIL certamente ocorre do lado direito ou esquerdo,
 $E \square \text{FAIL} \equiv E$ ou $\text{FAIL} \square E \equiv E$
 - se E_1 nem E_2 não puderem retornar FAIL, então:
 $(IF\ E_1\ E_2\ E_3) \square E \equiv IF\ E_1\ E_2\ (E_3 \square E)$.

Eliminação do \square e FAIL...

- A regra do vazio produz expressão da forma:
 $E_1 \square \dots \square E_m E$
- Se uma E_i (lado direito da expressão original) não tinha uma cláusula "otherwise", então ela foi traduzida para:
 $\text{IF } G_1 A_1 (\text{IF } \dots (\text{IF } G_g A_g \text{ FAIL}) \dots)$
- Se havia um *otherwise* no final, então E_i teria sido traduzida para:
 $\text{IF } G_1 A_1 (\text{IF } \dots (\text{IF } G_{g-1} A_{g-1} A_g) \dots)$
- G_i e A_i não podem ser iguais a FAIL, pois são versões transformadas das expressões escritas pelo programador.

...Eliminação do \square e FAIL

- Se E_i é da primeira forma, i.e., puder retornar FAIL, então podemos usar a terceira regra da seção anterior, seguida da segunda regra para transformar:
 $(\text{IF } G_1 A_1 (\text{IF } \dots (\text{IF } G_g A_g \text{ FAIL}) \dots)) \square E$
 $\equiv (\text{IF } G_1 A_1 (\text{IF } \dots (\text{IF } G_g A_g E) \dots))$
- Se E_i é da segunda forma usamos a primeira regra da seção anterior e retornamos o próprio E_i .

Definições Uniformes...

- Há dois motivos para estudar essa classe restrita de definições:
 1. as definições uniformes evitam certos problemas sobre analisar definições de função que envolvem *pattern-matching*.
 2. as definições uniformes são fáceis de compilar e garantem que certos tipos de código ineficientes serão evitados.

...Definições Uniformes...

- Definição: um conjunto de equações é dito uniforme, se uma das três condições forem válidas:
 - ou, todas as equações começam com um *pattern* variável, e aplicando a regra da variável tem-se um novo conjunto de equações que também é uniforme,
 - ou, todas as equações começam com um *pattern* construtor, e aplicando a regra do construtor tem-se novos conjuntos de equações que também é uniforme,
 - ou, todas as equações tem uma lista vazia de *patterns*, tal que a regra do vazio se aplica e existe no máximo uma equação no conjunto.

...Definições Uniformes...

- Ou seja, um conjunto de equações é uniforme, se ele puder ser compilado sem usar a regra da mistura e se a regra do vazio for aplicada somente a conjuntos contendo zero ou uma equação.
- Teorema: Se uma definição é uniforme, a troca da ordem das equações não muda o significado da definição.
- Teorema: Se o lado esquerdo de uma definição é tal que a ordem das equações não importa, então a definição é uniforme.

...Definições Uniformes

- Assim as definições uniformes ficam fáceis de implementar, já que não acarretam problemas mencionados anteriormente como:
 - as expressões resultantes podem examinar algumas variáveis mais de uma vez,
 - o compilador deve usar uma regra de construtor modificada para evitar duplicar o lado direito de equações.
 - as expressões resultantes podem conter \square e FAIL .

I. COMPILAÇÃO DO λ -CALCULUS ENRIQUECIDO λ -abstrações com Pattern-Matching

- Uma abstração lambda com *pattern-matching* tem a forma:
 $(\lambda p. E)$
 onde p é um pattern
- Se o *pattern* p for uma variável, não há nada a fazer, pois não existe nenhum *pattern-matching* envolvido.
- Os casos restantes são quando o *pattern* p for:
 - uma constante
 - um *pattern* construtor de produtos
 - um *pattern* construtor de somas.
 Nesses casos, tem-se um casamento do argumento com o parâmetro formal e o binding dos identificadores contidos no pattern com os elementos correspondentes do argumento

Patterns do Tipo Constante (k)

- A semântica de $(\lambda k. E)$, onde k é uma constante:

1. $\text{Eval}[\lambda k. E]a = \text{Eval}[E]$, se $k = \text{Eval}[a]$
2. $\text{Eval}[\lambda k. E]a = \perp$, se $\text{Eval}[a] = \perp$
3. $\text{Eval}[\lambda k. E]a = \text{FAIL}$, caso contrário

- Define-se a seguinte transformação:

$$(\lambda k. E) \equiv (\lambda v. \text{IF } (= k v) E' \text{ FAIL})$$

Patterns do Tipo Constante (k)

- Por exemplo,

$$\begin{aligned} \text{flip } 0 &= 1 \\ \text{flip } 1 &= 0 \end{aligned}$$

- Pode ser transformado para:

$$\begin{aligned} \text{flip} &= \lambda x. (((\lambda 0. 1) x) \\ &\quad \square ((\lambda 1. 0) x) \\ &\quad \square \text{ERROR}) \end{aligned}$$

- Aplicando a transformação acima tem-se:

$$\begin{aligned} \text{flip} &= \lambda x. (((\lambda v. \text{IF } (= 0 v) 1 \text{ FAIL}) x) \\ &\quad \square ((\lambda v. \text{IF } (= 1 v) 0 \text{ FAIL}) x) \\ &\quad \square \text{ERROR}) \end{aligned}$$

Patterns Construtores de Produto ...

- Um pattern do tipo

$$(\lambda (t p_1 \dots p_r). E) \equiv (\text{Unpack-Prod-}t (\lambda p_1. \dots \lambda p_r. E))$$

onde:

- t é o construtor de produto (tupla)
- $\text{Unpack-Prod-}t f a = f (\text{Sel-}t\text{-}1 a) \dots (\text{Sel-}t\text{-}r a)$
- $\text{Sel-}t\text{-}i a = \text{seleciona o } i\text{-ésimo componente da tupla } a \text{ identificada pelo construtor } t$

- Por exemplo,

$$\begin{aligned} \text{somapar} &= \lambda (\text{PAIR } x y). + x y \\ \text{somapar} &= \text{Unpack-Prod-PAIR } (\lambda x. \lambda y. + x y) \end{aligned}$$

...Patterns Construtores de Produto

- Então, dado

$$\text{somapar} = \text{Unpack-Prod-PAIR } (\lambda x. \lambda y. + x y)$$

- Avalia-se $\text{somapar}(\text{PAIR } 3 \ 4)$ da seguinte forma:

$$\begin{aligned} \text{somapar}(\text{PAIR } 3 \ 4) &= \\ \text{Unpack-Prod-PAIR}(\lambda x. \lambda y. + x y)(\text{PAIR } 3 \ 4) & \\ \rightarrow (\lambda x. \lambda y. + x y) (\text{Sel-PAIR-1}(\text{PAIR } 3 \ 4))(\text{Sel-PAIR-2}(\text{PAIR } 3 \ 4)) & \\ \rightarrow (\lambda y. + (\text{Sel-PAIR-1}(\text{PAIR } 3 \ 4)) y) (\text{Sel-PAIR-2}(\text{PAIR } 3 \ 4)) & \\ \rightarrow + (\text{Sel-PAIR-1}(\text{PAIR } 3 \ 4)) (\text{Sel-PAIR-2}(\text{PAIR } 3 \ 4)) & \\ \rightarrow + 3 (\text{Sel-PAIR-2}(\text{PAIR } 3 \ 4)) & \\ \rightarrow + 3 \ 4 & \\ \rightarrow 7 & \end{aligned}$$

Patterns Construtores de Soma ...

- Um *pattern* do tipo

$$(\lambda (s p_1 \dots p_{r_s}). E) \equiv (\text{Unpack-Sum-}s (\lambda p_s. \dots \lambda p_{r_s}. E))$$

onde:

- s é um construtor de soma de aridade r_s
- Unpack-Sum- s é da seguinte forma:
 Unpack-Sum- $s f (s a_1 \dots a_{r_s}) = f a_1 \dots a_{r_s}$
 Unpack-Sum- $s f (s' a_1 \dots a_{r_s}) = \text{FAIL}$, se $s \neq s'$
 Unpack-Sum- $s f \perp = \perp$

...Patterns Construtores de Soma...

- Por exemplo,

espelho(FOLHA n) = FOLHA n
 espelho(RAMO $t_1 t_2$) = RAMO (espelho t_2) (espelho t_1)

é traduzido para

espelho = $\lambda t. ($
 ((λ (FOLHA n). FOLHA n) t)
 \square ((λ (RAMO $t_1 t_2$). RAMO (espelho t_2)(espelho t_1)) t)
 \square ERROR)

...Patterns Construtores de Soma

- Assim, a expressão

espelho = $\lambda t. ($
 ((λ (FOLHA n). FOLHA n) t)
 \square ((λ (RAMO $t_1 t_2$). RAMO (espelho t_2)(espelho t_1)) t)
 \square ERROR)

- Transforma-se em:

espelho = $\lambda t. ($
 (Unpack-Sum-FOLHA ($\lambda n. \text{FOLHA } n$) t)
 \square (Unpack-Sum-RAMO (
 $\lambda t_1. \lambda t_2. \text{RAMO (espelho } t_2$)(espelho t_1)) t)
 \square ERROR)

Reduzindo o Número de Funções Primitivas

- Para cada construtor associamos um número de funções.
- No processo de compilação, isso implicaria na criação de uma função Unpack-Sum- s (ou Pack-sum- s , Unpack-Prod- t , Pack-Prod- t) associada a cada construtor de um tipo.
- Num sistema de tipos consistidos é necessário apenas distinguir objetos somente de outros objetos do mesmo tipo. Isso implica que podemos criar funções genéricas que servirão para os diversos tipos.
- Generalização a função Unpack-Sum- x (similarmente para as outras):
 Unpack-Sum- $d-r_s$

onde:

- d é um inteiro para identificar um construtor do tipo em questão
- r_s é a aridade do mesmo construtor.

Reduzindo o Número de Funções ...

- Para o tipo lista,
 - NIL é trocado por Pack-Sum-1-0
 - CONS é trocado por Pack-Sum-2-2
 - Unpack-Sum-NIL é trocado por Unpack-Sum-1-0
 - Unpack-Sum-CONS é trocado por Unpack-Sum-2-2
- Para o tipo árvore temos:
 - FOLHA é trocado por Pack-Sum-1-1
 - RAMO é trocado por Pack-Sum-2-2
 -
- Note que Pack-Sum-2-2 aparece duas vezes, diminuindo portanto o número de funções a serem implementadas.

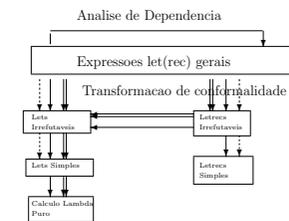
Transformando let e letrec...

- O objetivo é transformar sucessivamente construções $let(rec)$ que envolvem *pattern-matching* em cálculo lambda puro.
- A necessidade de *pattern-matching* decorre do fato de *patterns* arbitrários ocorrem no lado esquerdo de uma definição
- Assim, há a necessidade de verificar se o lado direito conforma-se com o lado esquerdo
- Entretanto, há uma classe de *patterns* que não sofrem desse problema: os *patterns* irrefutáveis.
- *Pattern* irrefutável é uma variável ou um *pattern* produto da forma $(t p_1 \dots p_r)$, onde p_i , para $1 \leq i \leq r$, são *patterns* irrefutáveis.

... Transformando let e letrec

- Seja $let (CONS \ x \ xs) = B \ in \ E$
 - É necessário verificar se B conforma-se com $(CONS \ x \ xs)$, caso contrário um erro deve ocorrer
 - Essa verificação acarreta um custo e deve ser evitada
 - Existem *patterns* que não necessitam dessa verificação de conformalidade. São os *patterns* irrefutáveis

Mapa das Transformações



Resumo das Transformações *lete* *letrec*

- Sejam a seguinte terminologia para tratar expressões *lete* *letrec*:
 - Uma expressão *let(rec)* simples é aquela na qual o lado esquerdo de cada definição é uma variável.
 - Uma expressão *let(rec)* irrefutável é aquela na qual o lado esquerdo de cada definição é um *pattern* irrefutável.
 - Uma expressão *let(rec)* geral é aquela na qual o lado esquerdo de cada definição é um *pattern* arbitrário.
- Há, para implementação, dois caminhos:
 1. A linguagem destino será o cálculo lambda puro, que será mais simples de ser implementado,
 2. A linguagem destino será o cálculo lambda incrementado com *let(rec)* simples. O programa resultante é mais complicado, porém mais eficiente.

Transformação de *let*

- *let* simples em λ -Calculus puro:

$$\text{let } v = B \text{ in } E \equiv (\lambda v. E) B$$

Por exemplo,

$$\text{let } x = 4 \text{ in } (+ x 3) \equiv (\lambda x. + x 3) 4$$

Transformação de *let* Irrefutável em *let* Simples

- Se p de $\text{let } p = B \text{ in } E$ for variável, a expressão-*let* já é simples
- Se p for *pattern* de produto, a expressão será da forma:

$$\text{let } (t p_1 \dots p_r) = B \text{ in } E$$
- que pode ser transformada para:

$$\text{let } v = B \text{ in } (\text{let } p_1 = \text{Sel-t-1 } v$$

$$\dots$$

$$p_r = \text{Sel-t-r } v$$

$$\text{in } E)$$
- Por exemplo,

$$\text{let}(\text{PAIR } x \ y) = B \text{ in } E$$

$$\equiv \text{let } v = B \text{ in } (\text{let } x = \text{Sel-PAIR-1 } v$$

$$y = \text{Sel-PAIR-2 } v$$

$$\text{in } E)$$

Transformação de *letrecs* irrefutáveis em *letrecs* simples ...

- A expressão

$$\text{letrec } (t p_1 \dots p_r) = B$$
 outras definições
 in E
- É equivalente a:

$$\text{letrec } v = B$$

$$p_1 = \text{Sel-t-1 } v$$

$$\dots$$

$$p_r = \text{Sel-t-r } v$$
 outras definições
 in E
- A presença *outras definições* deve-se ao fato de o *letrec* poder conter múltiplas definições, e essa transformação deveria ser aplicada a cada uma delas separadamente.

...letrecs irrefutáveis em lets simples

- Pode-se transformar *letrecs* com mais de uma definição

$$\text{letrec } \underbrace{p_1 = B_1}_{\text{def 1}} \dots \underbrace{p_n = B_n}_{\text{def n}} \text{ in } E$$
- Em *letrecs* com uma única definição:

$$\text{letrec } (t p_1 \dots p_n) = (t B_1 \dots B_n) \text{ in } E$$
 onde t é um construtor de produtos (p.ex. tuplas) de aridade n .
- Note que a transformação mantém o *letrec* irrefutável.
- Agora pode-se transformar um *letrec* em *letda* seguinte forma:

$$\text{letrec } p = B \text{ in } E$$

$$\equiv \text{let } p = Y(\lambda p. B) \text{ in } E$$
 onde p é um *pattern* irrefutável.

letrecs irrefutáveis em lets simples ...

- Por exemplo, a transformação

$$\text{letrec } p = B \text{ in } E \equiv \text{let } p = Y(\lambda p. B) \text{ in } E$$
- Aplicada a

$$\begin{aligned} \text{letrec } x &= \text{CONS } 1 \ y \\ y &= \text{CONS } 2 \ x \\ &\text{in } x \end{aligned}$$
- Produz $\text{letrec } (\text{PAIR } x \ y) = (\text{PAIR } (\text{CONS } 1 \ y) (\text{CONS } 2 \ x)) \text{ in } x$
- E finalmente

$$\text{let } (\text{PAIR } x \ y) = Y(\lambda (\text{PAIR } x \ y). \text{PAIR}(\text{CONS } 1 \ y) (\text{CONS } 2 \ x)) \text{ in } x$$
- Assim, trabalharemos com *lets* e *letrecs* com uma única definição.

let(rec)s gerais em let(rec)s Irrefutáveis

- *Let(rec)s* gerais tem a necessidade de teste de conformalidade em tempo de execução para verificar se existe possibilidade de ocorrer *pattern-matching*.
- O método vale tanto para expressões *let* ou *letrec*
- Dada uma definição, $p = B$, onde p é um *pattern* irrefutável, usa-se a transformação

$$p = B \equiv (t v_1 \dots v_n) = ((\lambda p. (t v_1 \dots v_n)) B) \square \text{ ERROR}$$
 onde t é um construtor de produtos, de aridade n .
- A definição resultante tem um *pattern* irrefutável do lado esquerdo.
- Essa transformação, chamada de **transformação de conformalidade**, deve ser aplicada a cada definição de um *let(rec)*.

let(rec)s gerais em let(rec)s Irrefutáveis ...

- As variáveis $v_1 \dots v_n$ são simplesmente as variáveis que aparecem no *pattern* p .
- Esse conjunto de variáveis do *pattern* p chamamos de $Var(p)$:
 - se p for uma variável v , $Var(p) = \{v\}$
 - se p for uma constante k , $Var(p) = \{\}$
 - se p for um *pattern* $cp_1 \dots p_r$, então

$$Var(p) = Var(p_1) \cup \dots \cup Var(p_r)$$

$let(rec)s$ gerais em $let(rec)s$ Irrefutáveis ...

- Para compilar esse *pattern-matching* com o compilador da seção anterior, será necessária a seguinte modificação na transformação de conformalidade:

$$p = B \equiv$$

$$(t v_1 \dots v_n) =$$

$$let v = B$$

$$in ((\lambda p. (t v_1 \dots v_n)) v) \quad \square \text{ ERROR}$$

onde

- $v_1, \dots, v_n = var(p)$
- t é um construtor de produtos de aridade n
- v é uma nova variável distinta de todos v_i

Análise de Dependência

- Compilador gera *letrecs* sempre, mesmo quando não é necessário. Deve-se então trocar *letrecs* por *lets* sempre que possível, reduzindo *letrecs* ao mínimo necessário.
- Essas transformações são chamadas análise de dependência.
- Segue agora o algoritmo para análise de dependência.

Análise de Dependência

- O seguinte exemplo serve como ilustração:

$$letrec \ a = \dots$$

$$b = \dots a \dots$$

$$c = \dots h \dots b \dots d \dots$$

$$d = \dots c \dots$$

$$f = \dots g \dots h \dots a \dots$$

$$g = \dots f \dots$$

$$h = \dots g \dots$$

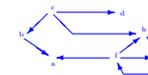
$$in \ \dots$$

- O exemplo é um *letrecs* simples, mas o algoritmo requer poucas modificações para lidar com *let(rec)s* gerais.

Análise de Dependência ...

O algoritmo se divide em 4 partes, cada uma aplicada separadamente em cada *letrec*:

- Para cada *letrec* construa um grafo dirigido, no qual os nós são as variáveis ligadas pelo *letrec*.
 - Existe um arco da variável x para a variável y , se y ocorre livre na definição de x , isto é, a definição de x depende diretamente de y .
 - Esse grafo de grafo de dependência:

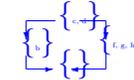


Análise de Dependência ...

2. • Duas variáveis x e y são mutuamente recursivas se existe um caminho (direto ou não) de x para y e de y para x .
- Isso é, descubra estes componentes fortemente conectados do grafo [Aho, et. al. 1974, 1983a], [Dijkstra, 1976].
 - No nosso grafo de dependência, os componentes são:
 $\{c, d\}$, $\{f, g, h\}$, $\{b\}$, $\{a\}$.

Análise de Dependência ...

3. • Ordene os componentes fortemente conectados na ordem de dependência.
- Faça uma contração no grafo original removendo todos os arcos que ligam elementos dentro de um mesmo componente fortemente conectado.
 - O grafo resultante é certamente acíclico.



...Análise de Dependência ...

- Agora faça uma ordenação topológica [Aho, et. al., 1983b], para colocar os nós em ordem de dependência.
- Um resultado possível para o sort topológico:
 $\{c, d\}$, $\{b\}$, $\{f, g, h\}$, $\{a\}$.
- Isso implica que a definição de $\{a\}$ deve encobrir a definição de $\{f, g, h\}$ que deve encobrir a definição de $\{b\}$ que por sua vez deve encobrir a definição de $\{c, d\}$.

Análise de Dependência ...

4. • Geram-se os let e *letrecs*. Para o exemplo abaixo:

letrec $a = \dots$

$b = \dots a \dots$

$c = \dots h \dots b \dots d \dots$

$d = \dots c \dots$

$f = \dots g \dots h \dots a \dots$

$g = \dots f \dots$

$h = \dots g \dots$

in \dots

- Gera-se uma expressão-letou *letrec* para grupo de definição na ordem topológica

$\{c, d\}$, $\{f, g, h\}$, $\{b\}$, $\{a\}$

...Análise de Dependência ...

- O nosso exemplo resulta na expressão:

```
let a = ...
in let b = a...
    in letrec f = ... g ... h ... a ...
        g = ... f ...
        h = ... g ...
    in letrec c = ...h ...b ...d
        d = ... c ...
    in ...
```

Transformando Expressões-case

- O esquema de tradução para o *pattern* da seção anterior faz uso de expressões-case.

- A expressão-case é da forma:

case v of

$$c_1 v_{1,1} \dots v_{1,r_1} \Rightarrow E_1$$

...

$$c_n v_{n,1} \dots v_{n,r_n} \Rightarrow E_n$$

onde

- c_i é a família completa de construtores de um tipo estruturado
- v é uma variável,
- E_i são expressões.

- Há duas possibilidades a considerar, dependendo se os construtores são do tipo soma ou do tipo produto.

Expressões-case com Tipo Produto...

- A forma geral de expressões-case para tipos produtos é:

case v of

$$t v_1 \dots v_r \Rightarrow E_1$$

- Esse é um caso degenerado, pois não há necessidade de testar v , podendo fazer o *lazy product-matching*:

case v of

$$t v_1 \dots v_r \Rightarrow E_1$$

$$\equiv \text{Unpack-Prod-t } (\lambda v_1 \dots \lambda v_r. E_1) v$$
...Expressões-case com Tipo Produto

- Por exemplo, uma função Miranda:

$$\text{somapar } (x, y) = x + y$$

- É traduzida para o cálculo lambda enriquecido:

$$\text{somapar} = \lambda w.$$

$$(\text{case } w \text{ of } (\text{PAIR } x \ y) \Rightarrow (+ \ x \ y))$$

- Aplicando a transformação acima teremos:

$$\text{somapar} = \lambda w.$$

$$(\text{Unpack-Prod-PAIR } (\lambda x. \lambda y. + \ x \ y)) w$$

- Por fim fazendo uma η -redução teremos:

$$\text{somapar} = (\text{Unpack-Prod-PAIR}(\lambda x. \lambda y. + \ x \ y))$$

Expressões-casecom Tipo Soma...

- A expressão geral será da forma

case v of

$s_1 v_{1,1} \dots v_{1,r_1} \Rightarrow E_1$

...

$s_n v_{n,1} \dots v_{n,r_n} \Rightarrow E_n$

onde s_1, \dots, s_n são os construtores de um tipo soma T .

- Pode-se transformar essa expressão-case em:

\equiv Case-T v

$(\text{Unpack-Sum-}s_1(\lambda v_{1,1} \dots \lambda v_{1,r_1}. E_1) v)$

...

$(\text{Unpack-Sum-}s_n(\lambda v_{n,1} \dots \lambda v_{n,r_n}. E_n) v)$

...Expressões-casecom Tipo Soma ...

- A função Case-T (a qual existe uma para cada tipo T) seleciona um de seus n argumentos dependendo do construtor usado para construir seu primeiro argumento.

Case-T $(s_i a_1 \dots a_{r_i}) b_1 \dots b_i \dots b_n = b_i$

Case-T $\perp b_1 \dots b_i \dots b_n = \perp$

...Expressões-casecom Tipo Soma ...

- Por exemplo,

espelho = $\lambda t. \text{ case } t \text{ of}$

FOLHA $n \Rightarrow$ FOLHA n

RAMO $t_1 t_2 \Rightarrow$

RAMO (espelho t_2) (espelho t_1)

\equiv

espelho = $\lambda t. \text{ Case-TREE}$

t

$(\text{Unpack-Sum-FOLHA } (\lambda n. \text{ FOLHA } n) t)$

$(\text{Unpack-Sum-RAMO}$

$(\lambda t_1. \lambda t_2.$

RAMO(espelho t_2) (espelho t_1)) $t)$

Usando let ao Invés de UNPACK

- Expressões letsimples podem ser implementadas mais eficientemente que abstrações lambda, por isto é melhor usá-las sempre que possível.

- No caso do tipo produto, tem-se a transformação:

case v of

$t v_1 \dots v_r \Rightarrow E_1$

\equiv

let $v_1 = \text{Sel-t-1 } v$

...

$v_r = \text{Sel-t-r } v$

in E_1

...Usando let ao Invés de UNPACK

- Por exemplo, *somapar* seria transformado em:

$somapar = \lambda w.$

$let\ x = Sel-PAIR-1\ w$

$y = Sel-PAIR-2\ w$

$in\ (+\ x\ y)$

Usando let ao Invés de UNPACK ...

- No caso do tipo soma, usaremos a seguinte transformação:

$$\begin{aligned}
 & case\ v\ of \\
 & \quad s_1\ v_{1,1}\ \dots\ v_{1,r_1} \Rightarrow E_1 \\
 & \quad \dots \\
 & \quad s_n\ v_{n,1}\ \dots\ v_{n,r_n} \Rightarrow E_n \\
 \equiv & Case-T\ v\ (let\ v_{1,1} = Sel-sum-s_1-1\ v \\
 & \quad \dots \\
 & \quad v_{1,r_1} = Sel-sum-s_1-r_1\ v\ in\ E_1) \\
 & \quad \dots \\
 & \quad (let\ v_{n,1} = Sel-sum-s_n-1\ v \\
 & \quad \quad \dots \\
 & \quad \quad v_{n,r_n} = Sel-sum-s_n-r_n\ v\ in\ E_n)
 \end{aligned}$$

- A função seletora $Sel-sum-s--i$ seleciona o i -ésimo componente de um objeto construído com o construtor soma s .

Reduzindo Número de Funções, \square e FAIL...

- As idéias anteriores sobre a redução de funções *built-in*, também podem ser aplicadas às funções *case*.
- Especificamente, o $Case-T$ pode ser trocado por $case - n$, onde n é o número de construtores do tipo T .
- Similarmente, $Sel-sum-s-i$ pode ser trocado por $Sel-sum-r-i$, onde r é a aridade de s .

...Reduzindo Número de Funções, \square e FAIL

- O operador \square pode ser transformado na função *Fatbar*:

$$E_1 \square E_2 \equiv Fatbar\ E_1\ E_2,$$

onde

$$Fatbar\ a\ b = a\ if\ a \neq FAIL\ and\ a \neq \perp$$

$$Fatbar\ FAIL\ b = b$$

$$Fatbar\ \perp\ b = \perp$$

- Mesmo não eliminando todas ocorrências de \square e FAIL do programa, elas ainda podem ser compiladas eficientemente como veremos na seção sobre a máquina-G.

COMPILAÇÃO DE COMPREENSÃO DE LISTAS

Compreensão de Listas...

- **Compreensão de listas permite expressar listas através de uma notação análoga a conjuntos na teoria de conjuntos de Zermelo-Frankel.**
- **Um exemplo de compreensão de conjunto seria:**
 $B = \{sq\ x \mid x \in A \ \& \ odd\ x\}.$
Em Miranda o correspondente é:
 $ys = [sq\ x \mid x \leftarrow xs; \ odd\ x].$
Ex.: se $xs = [1, 2, 3, 4, 5]$, então $ys = [1, 9, 25].$

...Compreensão de Listas

- **A forma geral de uma expressão ZF é:**
 $[< \text{expr} > \mid < \text{qualif} > \ \dots \ < \text{qualif} >],$
onde um $< \text{qualif} >$ é um gerador como $x \leftarrow xs$ ou um filtro como $odd\ x$.
- **No caso de um gerador $p \leftarrow L$ ter em p um *pattern* refutável, os elementos de L que não casam com o *pattern* não serão considerados.**
- **Por exemplo, a função *sozinhos* que toma uma lista de listas e retorna uma lista com os elementos das listas de tamanho 1:**
 $sozinhos\ xs = [x \mid [x] \leftarrow xs],$
e $sozinhos\ [[1, 2], [5], [], [2]] = [5, 2].$

Semântica da compreensão de listas...

- **Em Miranda, tem-se a forma abreviada:**
 $v_1, \dots, v_n \leftarrow L$
que é equivalente a:
 $v_1, \leftarrow L; \dots; v_n \leftarrow L$
onde v_i são variáveis.

...Semântica da compreensão de listas

- As 5 regras de redução que descrevem o comportamento da compreensão de listas:

- 1) $[E \mid v \leftarrow []; Q] \rightarrow []$
- 2) $[E \mid v \leftarrow E' : L'; Q] \rightarrow [E \mid Q][E'/v] ++ [E \mid v \leftarrow L'; Q]$
- 3) $[E \mid \text{False}; Q] \rightarrow []$
- 4) $[E \mid \text{True}; Q] \rightarrow [E \mid Q]$
- 5) $[E \mid] \rightarrow [E]$

onde $[E \mid Q][E'/v] ++ [E \mid v]$ significa $[E \mid Q]$ com todas as ocorrências livres de v , trocadas por E' .

Tradução de Compreensão de Listas...

- A tradução requer uma nova função, flatmap:

$$\text{flatmap } f [] = []$$

$$\text{flatmap } f (x : xs) = f x ++ (\text{flatmap } f xs)$$

- O esquema de tradução TE:

$$\text{TE}[[E \mid v \leftarrow L; Q]] \equiv$$

$$\text{flatmap } (\lambda v. \text{TE}[[E \mid Q]]) \text{TE}[[L]]$$

$$\text{TE}[[E \mid B; Q]] \equiv$$

$$\text{IF } \text{TE}[[B]] \text{TE}[[E \mid Q]] \text{NIL}$$

$$\text{TE}[[E \mid]] \equiv \text{CONS } \text{TE}[[E]] \text{NIL}$$

onde v é uma variável, E uma expr

B é uma expr que retorna um booleano

L é uma expr que retorna uma lista

Q é uma sequência de zero ou mais qualifs

...Tradução de Compreensão de Listas

- Por exemplo,

$$\text{TE}[[\text{sq } x \mid x \leftarrow xs; \text{odd } x]]$$

$$\equiv \text{flatmap}(\lambda x. \text{TE}[[\text{sq } x \mid \text{odd } x]]) xs$$

$$\equiv \text{flatmap}(\lambda x. \text{IF}(\text{odd } x) \text{TE}[[\text{sq } x \mid]]) \text{NIL}) xs$$

$$\equiv \text{flatmap}(\lambda x. \text{IF}(\text{odd } x)(\text{CONS } (\text{sq } x) \text{NIL}) \text{NIL}) xs$$

Transformações para Melhorar Eficiência...

- Primeiro passo para melhorar a eficiência:

$$\text{flatmap } (\lambda v. E) L$$

pode ser substituído por

$$\text{letrec } h = \lambda us. \text{case } us \text{ of}$$

$$\text{NIL} \Rightarrow \text{NIL}$$

$$\text{CONS } v \text{ us}' \Rightarrow \text{APPEND } E (h \text{ us}')$$

$$\text{in } (h L)$$

- O segundo passo, derivado aplicando métodos de transformação de programa, elimina as ocorrências de APPEND:

$$\text{APPEND } \text{TE}[[E \mid Q]](h \text{ us}').$$

- A expressão acima é traduzida para:

$$\text{TQ}[[E \mid Q] ++ L] \equiv \text{APPEND } \text{TE}[[E \mid Q]] \text{TE}[[L]].$$

...Transformações para Melhorar Eficiência...

- As novas regras de tradução são:

$$\text{TE}[[E \mid Q]] \equiv \text{TQ}[[E \mid Q] ++ []]$$

$$\text{TQ}[[E \mid v \leftarrow L_1; Q] ++ L_2] \equiv$$

$$\text{letrec } h = \lambda us. \text{ case us of}$$

$$\text{NIL} \Rightarrow \text{TE}[L_2]$$

$$\text{CONS } v \text{ us}' \Rightarrow \text{TQ}[[E \mid Q] ++ (h \text{ us}')]$$

$$\text{in } (h \text{ TE}[L_1])$$

$$\text{TQ}[[E \mid B; Q] ++ L] \equiv$$

$$\text{IF TE}[B] \text{TQ}[[E \mid Q] ++ L] \text{TE}[L]$$

$$\text{TQ}[[E \mid] ++ L] \equiv \text{CONS TE}[E] \text{TE}[L]$$
...Transformações para Melhorar Eficiência

- Por exemplo,

$$\text{TE}[\text{sq } x \mid x \leftarrow xs; \text{ odd } x]$$

$$\equiv \text{letrec } h = \lambda us. \text{ case us of}$$

$$\text{NIL} \Rightarrow \text{NIL}$$

$$\text{CONS } x \text{ us}' \Rightarrow$$

$$\text{IF (odd } x) (\text{CONS (sq } x) (h \text{ us}')) (h \text{ us}')$$

$$\text{in } (h \text{ xs})$$
Pattern-Matching em Compreensões de Listas

- Lado esquerdo de um gerador pode ser um *pattern*.
- As regras de reduções anteriores para os geradores são modificadas para:

$$1') [E \mid p \leftarrow []; Q] \rightarrow []$$

$$2') [E \mid p \leftarrow E' : L'; Q] \rightarrow$$

$$((\lambda p. [E \mid Q]) E') \square [] ++ [E \mid v \leftarrow L'; Q]$$

- Para o esquema de tradução a única regra que contém um gerador é a primeira, que pode ser modificada para:

$$\text{TE}[[E \mid p \leftarrow L; Q]]$$

$$\equiv \text{flatmap}$$

$$(\lambda u. ((\lambda TE[p]. \text{TE}[[E \mid Q]]) u) \square \text{NIL})) \text{TE}[L]$$

onde u é uma variável que não ocorre livre em p , E ou Q

Pattern-Matching em Compreensões de Listas ...

- Note que a subexpressão

$$((\lambda TE[p]. \text{TE}[[E \mid Q]]) u) \square \text{NIL})$$

já está na forma esperada pelo compilador de *pattern-matching*.

Pattern-Matching em Compreensões de Listas ...

- No esquema ótimo de tradução, a regra de tradução:

$$\text{TQ}[[E \mid v \leftarrow L_1; Q] ++ L_2] \equiv$$

$$\text{letrec } h = \lambda us. \text{ case us of}$$

$$\text{NIL} \Rightarrow \text{TE}[L_2]$$

$$\text{CONS } v \text{ us}' \Rightarrow \text{TQ}[[E \mid Q] ++ (h \text{ us}')]]$$

$$\text{in } (h \text{ TE}[L_1])$$

- deve ser substituída por

$$\text{TQ}[[E \mid p \leftarrow L_1; Q] ++ L_2] \equiv$$

$$\text{letrec } h = \lambda us. \text{ case us of}$$

$$\text{NIL} \Rightarrow \text{TE}[L_2]$$

$$\text{CONS } u \text{ us}' \Rightarrow (((\lambda \text{TE}[p]. \text{TQ}[[E \mid Q] ++ (h \text{ us}')]]) u) \square (h \text{ us}'))$$

$$\text{in } (h \text{ TE}[L_1])$$

I. VERIFICAÇÃO POLIMÓRFICA DE TIPOS

I. UM VERIFICADOR DE TIPOS

PARTE II

REDUÇÃO DE GRAFOS

II. REPRESENTAÇÃO DO PROGRAMA

Árvores de sintaxe abstrata

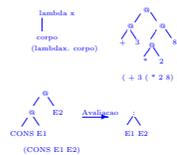
- Em implementações de redução de grafos a expressão a ser avaliada é mantida na máquina na forma de árvore sintática.
- As folhas da árvore são valores constantes, nomes de variáveis ou funções built-in.
- A aplicação de uma função f a um argumento x é representada por:



- O sinal '@' é chamado *tag* do nodo e nesse caso o *tag* indica que o nodo é uma aplicação.

Árvores de sintaxe abstrata

- Por exemplo,



O grafo...

- O processo de redução executa transformações sucessivas na árvore sintática. Durante esse processo a árvore se torna um grafo. Por exemplo,



- Cada nodo do grafo será representado por uma célula de memória.
- A célula contém:
 - um *tag* que indica o tipo da célula (aplicação, número, operador primitivo, abstrações lambda, célula CONS, etc.)
 - outros dois ou mais campos, dependendo da implementação



... O grafo...

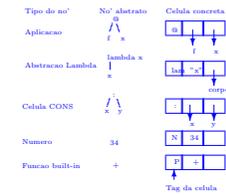
- Há duas famílias de *tags*:
 - Os *tags* de estrutura para dados. Ex.: CONS e NIL têm *tags* de estrutura diferentes
 - Os *tags* de sistema objetos do sistema. Ex.: nodos de aplicação, abstrações lambda, operadores *built-in*
- Dada uma célula da forma:



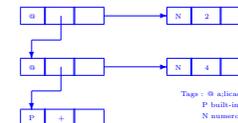
- Um campo pode ser um ponteiro para outra célula ou pode ser um dado atômico

...O Grafo

- Possíveis representações:



- A Árvore Concreta para (+ 4 2) :



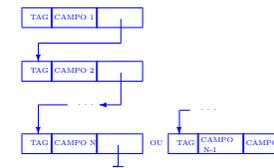
Representando Dados Estruturados...

- Um dado estruturado é construído por uma função construtora. Ex.: CONS, NIL
- Dado estruturado é um agregado de valores com um *tag* para distingui-lo de outros construtores do mesmo tipo. Um *tag* é um inteiro entre 1 e o número de construtores do tipo
- Se a implementação suporta células de tamanho variável, pode-se implementar essas estruturas diretamente como:



Representando Dados Estruturados...

- Se a implementação suporta apenas células de tamanho fixo, a estrutura deve ser implementada como uma coleção de células encadeadas



Outros Usos Para Células de Tamanho Variável

- Células de tamanho variável permitem uma representação bem mais elegante aos dados estruturados
- Elas são úteis para conter outros objetos como:
 - *arrays*
 - inteiros de precisão arbitrária
 - blocos de código compilado
 - aplicações múltiplas, por exemplo, pode-se representar $(f\ a\ b)$ como uma única célula com 3 campos contendo f , a e b
- Infelizmente, células de tamanho variável carregam um custo de implementação

Tempo de Compilação × Tempo de Execução...

- Em linguagens funcionais tipadas estaticamente, são requeridos *tags* distintos o suficiente:
 - para identificar objetos de sistema univocamente
 - distinguir alguns objetos de dados de um certo tipo, uns dos outros. Por exemplo, distinguir uma célula *CONS* de uma *NIL*)
- Em linguagens fortemente tipadas em tempo de execução, cada operador primitivo verifica o tipo de seus argumentos antes de proceder a chamada
- Isto requer que cada tipo de dado seja distinguível de todos os outros usados no programa
- Normalmente linguagens fortemente tipadas têm um conjunto fixo de tipos e não permite ao usuário a criação de outros tipos
- Nesse caso, um *tag* de tamanho fixo será suficiente

...Tempo de Compilação × Tempo de Execução

- Num sistema consistido em tempo de compilação é desejável carregar informação de tipo em tempo de execução para ajudar na depuração do sistema
- Isso é problemático em linguagens que permitem ao programador introduzir novos tipos porque não há limite ao número de tipos que devam ser distinguíveis
- Nesse caso, um mecanismo de escape para tipos definidos pelo usuário é fazer com que o primeiro campo da célula, representando o objeto, carregue uma identificação única de tipo

Objetos *Boxed* e *Unboxed*...

- Objetos de dados que podem ser completamente descritos por um único campo são chamados *unboxed*, enquanto aqueles que são representados por uma ou mais células são chamados *boxed*
- Candidatos típicos para uma representação *unboxed* são inteiros, booleanos, caracteres, e operadores *built-in* (que podem ser identificados por um inteiro ou ponteiro para código).
- Por exemplo,



- Num sistema *boxed*, o *tag* da célula já determina quais campos são ponteiros e quais não são.

...Objetos *Boxed* e *Unboxed*

- Já num sistema *unboxed*, qualquer campo que possa conter um ponteiro, pode também conter um objeto *unboxed*
- O campo de uma célula de aplicação, ora é um ponteiro, ora um objeto *unboxed*.
- Assim, todos os campos terão que ter um *bit* extra para distinguir ponteiros de objetos *unboxed*

Objetos *Boxed* e *Unboxed* ...

- Problemas: em sistemas checados em tempo de execução, os objetos *unboxed* não tem *tag*, pois *tags* são associados à células
- A saída é reduzir o número de *bits* do dado e incorporar um *tag* na célula
- Campos não-ponteiros teriam a seguinte forma:

```
+---+-----+      +---+-----+
| 1 | pointer |      | 0 + value |
+---+-----+      +-----+
```

- Mesmo para sistema checados em tempo de compilação é vital que funções *built-in* distingüam entre operandos avaliados e operandos não-avaliados
- Isso é fácil, pois se o operando é um ponteiro então o *tag* da célula apontada indicará se o operando foi avaliado ou não

- Se o operando não for um ponteiro, então será um objeto *unboxed* que não necessita ser avaliado

Ponteiros com *tag*...

- Algumas implementações também colocam um *tag* nos campos de ponteiros.
- Por exemplo, as máquinas de redução do SKIM e de NORMA fazem uso dessa técnica, embora elas usem o *tag* de maneiras diferentes.
- NORMA tem o *tag* de ponteiro como um *cache* para o *tag* da célula para a qual ele aponta. Assim, se o *tag* de ponteiro é válido, ele contém o *tag* da célula para a qual ele aponta.
- No SKIM, não existe nenhum *tag* nas células. Só existem *tag* nos campos.

...Ponteiros com *tag*

- Isso tem a vantagem de que uma célula consiste em dois campos idênticos, ao invés de dois campos idênticos mais um *tag*, o que permite um projeto de *hardware* mais uniforme para SKIM.
- Contudo, isso significa que uma célula não pode mudar seu *tag*
- Por exemplo, uma célula de aplicação deve permanecer uma célula de aplicação, porque seria impossível mudar os *tags* de todos os ponteiros para a célula, de uma só vez

Gerenciamento de Memória e *Garbage Collection*

- À medida que a redução se processa, precisaremos construir novos pedaços do grafo
- Por isso, é necessário alocar novas células
- Além disso, é preciso também desalocar células ou melhor, descartar ponteiros para células
- Células com mais de um ponteiro apontando para elas só poderão ser reusadas quando não houver mais nenhum ponteiro apontando para ela
- Células desse tipo são chamadas células de lixo e todas implementações de linguagens funcionais incluem um coletor de lixo (*garbage collector*) para identificar e reciclar essas células

II. SELEÇÃO DO PRÓXIMO REDEX

Avaliação Estrita

- Quando o grafo de um programa funcional for carregado num computador, um avaliador deve ser chamado para reduzir o grafo para a forma normal.
- O avaliador executa reduções sucessivas no grafo, envolvendo duas tarefas distintas:
 1. selecionar o próximo *redex* para reduzir,
 2. reduzir o *redex* selecionado.
- Em linguagens convencionais, normalmente na passagem de parâmetros, avaliamos os argumentos antes deles serem passados. A esse tipo de chamada dá-se o nome de chamada por valor.

Avaliação *lazy*

- Uma outra solução seria o avaliador atrasar a avaliação dos argumentos até que eles sejam realmente requeridos.
- A esse tipo de chamada dá-se o nome de chamada por necessidade ou call by need.
- O caso a favor da avaliação *lazy*:
 - A avaliação *lazy* adiciona expressividade à linguagem no que tange a construção e manipulação de estruturas de dados infinitas e *streams*.
- O caso contra a avaliação *lazy*:
 - O preço da avaliação *lazy* é o tempo de execução.
 - Implementações mais rápidas serão possíveis, quando os argumentos para funções puderem ser avaliados antes da função ser aplicada.

Ordem normal de Redução...

- Avaliação *lazy* tem dois ingredientes:
 1. argumentos de funções devem ser avaliados somente quando seu valor é requerido e não quando a função é aplicada.
 2. argumentos devem ser avaliados no máximo, somente uma vez.
- Na ordem normal de redução o *redex* a ser reduzido é o mais a esquerda e o mais externo.
- Dada uma aplicação de uma função a um argumento, o *redex* mais externo é a própria aplicação de função, que será então reduzida primeiramente, antes do argumento.
- A ordem normal de redução implementa diretamente o primeiro ingrediente para avaliação *lazy*

...Ordem normal de Redução

- Numa ordem aplicativa de redução, o argumento é reduzido a uma expressão lambda antes da redução da expressão que carrega aquele argumento.
- Essa ordem implementa a semântica estrita (não *lazy*).

Construtores de Dados, Entrada e Saída...

- Suponha que o resultado do nosso programa seja uma lista infinita.
- Essa lista deve ser impressa à medida que seja gerada para evitar loop infinito.
- O programa não deve avaliar toda sua entrada antes de produzir uma saída.
- Entrada e saída têm como característica o efeito colateral em linguagens imperativas.

...Construtores de Dados, Entrada e Saída

- Em sistemas funcionais, deve-se adotar uma abordagem diferente. Uma solução aceitável é ter um programa funcional como uma função que mapeia dados de entrada em dados de saída.
- Os dados de entrada normalmente são apresentados ao programa como uma lista infinita de caracteres, originada de um dispositivo de entrada.
- Os dados de saída são o resultado de aplicar o programa à lista de entrada e obter algum tipo estruturado de dado enviado para um dispositivo de saída.

Mecanismo de I/O...

- Para poder imprimir a estrutura de dados, assim que ela seja gerada, a avaliação de um programa funcional é dirigida pela necessidade de imprimir o resultado.
- O avaliador será chamado do programa impressor.
- O programa impressor chama o avaliador e olha a raiz do resultado (raiz do grafo avaliado). Se for um número (ou booleano, ou caracter) ele será impresso e a avaliação estará completa.
- Contudo, se a raiz for um construtor, o programa impressor chama o avaliador para avaliar os componentes e imprimir os resultados à medida que eles apareçam.

...Mecanismo de I/O

- Por exemplo, se um programa funcional quando avaliado, resultasse num número ou numa célula CONS, poder-se-ia escrever o seguinte pseudo-código para o programa impressor:

```

procedure print(E)
  begin E' := Evaluate(E)
        if (IsNumber(E')) then output(E')
        else begin print(HEAD(E'))
                  print(TAIL(E'))
        end
  end
end

```

- Para extrair caracteres da lista de entrada, o programa precisará avaliar a lista, elemento por elemento.
- Como no caso do programa impressor, a primeira avaliação não deve forçar a avaliação da lista inteira.

Formas Normais

- A conclusão é que ao avaliar uma expressão cujo resultado é uma célula CONS não se deve avaliar sua cabeça e sua cauda.
- Isso significa que deve-se parar a redução mesmo que ainda possam existir alguns *redexes* no grafo.
- Nenhum desses *redexes* serão reduzidos por uma ordem normal de redução até que a expressão inteira tenha sido avaliada para uma célula CONS.
- Isto ocorre porque sempre haverá um *redex top-level* o qual será selecionado pela ordem normal.
- *top-level redex* que não tem variáveis livres.
- Assim, é suficiente que a redução em ordem normal ocorra, mas pare quando não exista nenhum *redex top-level*, mesmo que existam *redexes* interiores.

Forma Normal com Cabeça Fraca

- Uma expressão lambda está na forma normal com a cabeça fraca (WHNF) se e somente for da forma:

$$F E_1 E_2 \dots E_n$$

onde :

- $n \geq 0$
- F é uma variável ou objeto de dado ou F é uma abstração lambda ou função *built-in*
- e $(F E_1 \dots E_m)$ não é um *redex* p/ $m \leq n$.
- Uma expressão não tem *redex top-level* se e somente se estiver na forma normal com a cabeça fraca.

...Forma Normal com Cabeça Fraca...

- Por exemplo, as seguintes expressões estão na forma normal com cabeça fraca (WHNF):
 - 3
 - uma célula CONS
 - $+ (- 4 3)$ o *top-level* $+$ não tem argumentos suficientes
 - $(\lambda x. + 5 1)$ não aplicado a nada.
- Os dois exemplos abaixo estão na forma normal com cabeça fraca, mas não na forma normal, porque eles contêm *redexes* mais internos.
 - $+ (- 4 3)$ o *top-level*
 - $(\lambda x. + 3 1)$

...Forma Normal com Cabeça Fraca

- O processo de redução segue o esquema:



- Esse processo de redução é um ingrediente fundamental para avaliação *lazy*, já que reduzindo para forma normal arrisca-se em fazer reduções desnecessárias.

Redução de *top-level* é mais Fácil

- O resultado de um programa funcional nunca tem variáveis livres.
- Já que só se reduz *redexes top-level* então os argumentos do *redex* também não terão variáveis livres.
- Isso significa que o problema da captura anteriormente sobre cálculo lambda nunca ocorrerá nestas implementações.

Forma Normal de Cabeça...

- A forma normal de cabeça normalmente é confundida com a forma normal de cabeça fraca.
- Embora, na maioria dos propósitos, a forma normal de cabeça é a mesma que a forma normal de cabeça fraca, elas são formas distintas.
- Uma expressão lambda está na forma normal de cabeça (HNF) se e somente se ela for da forma:

$$\lambda x_1. \dots \lambda x_n. (v M_1 \dots M_m)$$

onde

- $n, m \geq 0$
- v é uma variável (x_i), um objeto de dado ou uma função *built-in*
- $(v M_1 \dots M_p)$ não é um *redex* para $p \leq m$.

...Forma Normal de Cabeça

- A diferença entre a HNF e WHNF é significativa somente quando o resultado é uma expressão lambda, já que para os outros casos elas são idênticas.
- O problema de reduzir para HNF envolve em reduzir expressões mais internas com variáveis livres, voltando assim com o problema da captura de nome.

Avaliando Argumentos de Funções *built-in*...

- Há funções que são estritas nos seus argumentos, já que elas precisam deles avaliados para poder executar
- São exemplos as funções $+$ e HEAD
- Quando o avaliador percebe que o *redex top-level* é uma aplicação que avalia seus argumentos, deve-se verificar se os argumentos já estão na WHNF.
- Não estando, chama-se o avaliador para reduzir os argumentos à WHNF antes de se proceder com a aplicação da função

...Avaliando Argumentos de Funções *built-in*

- Por exemplo, na expressão,

$$(IF (NOT TRUE) f g) h$$

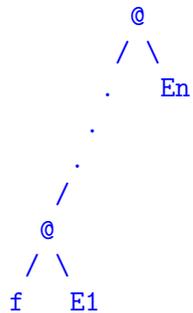
- seleciona-se o *redex* $(IF (NOT TRUE) f g)$ para redução
- IF deve avaliar seu primeiro argumento (somente)
- É verificado que ele não está na WHNF.
- Então o avaliador se chama recursivamente para avaliar $(NOT TRUE)$, que retorna $FALSE$.
- Agora sim, a avaliação de IF pode proceder.

Como Encontrar o Próximo *redex top-level*...

- As expressões só pode ser da seguinte forma:

$$f E_1 \dots E_n$$

cujo grafo é:



onde:

- f é um objeto de dado, ou uma função *built-in* ou uma abstração lambda (mas não uma aplicação)
- podem existir zero ou mais argumentos E_i , que podem ser expressões arbitrárias.

...Como Encontrar o Próximo *redex top-level* ...

- As várias possibilidades para f são:

- Um objeto de dado tal como um número ou uma célula CONS, e nesse caso a expressão já está na WHNF. Note que o número de argumentos deve ser zero e isso será garantido num sistema com checagem de tipos,
- Uma função *built-in* com k argumentos.
Se $k \leq n$, $(f E_1 \dots E_k)$ é o *redex* mais externo que a ordem normal irá selecionar.
Se $k > n$ então a expressão já está na WHNF,
- Uma abstração lambda:
Se ela tem um argumento disponível ($n \geq 1$), o *redex* que deve-se reduzir será $(F E_1)$. Se não existem argumentos ($n = 0$), então a expressão está na WHNF.

...Como Encontrar o Próximo *redex top-level* ...

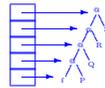
- De acordo com a sintaxe de uma expressão existe uma outra possibilidade para f , que seria um nome de variável. Contudo, nesse caso a variável deverá ocorrer livre na expressão, o que seria um erro
- Alguns avaliadores insistem que uma expressão sempre reduza a objetos de dados no final
- Neste caso, expressão lambda com menos argumentos que o necessário seria tratado como um erro.
- Para encontrar f basta caminhar para baixo e para esquerda no grafo a partir da raiz
- Essa cadeia de arestas à esquerda é chamada de espinha (Spine) da expressão

...Como Encontrar o Próximo *redex top-level* ...

- O fato de percorrer a espinha para baixo, é chamado de descer (Unwind) na espinha
- As vértebras da espinha são os nós de aplicação, encontrados durante a descida
- Os argumentos das aplicações são as costelas, e o extremo inferior esquerdo é o cóccix da espinha
- Então para encontrar o próximo *redex* para reduzir, tem-se que descer na espinha até encontrar uma função e baseando-se nessa função, volta-se na espinha para encontrar a raiz do *redex*

A Pilha da Espinha...

- **Problema:** como descer na espinha e voltar na espinha?
- Quando se desce na espinha, passa-se pelos argumentos necessários à redução da função (*built-in* ou abstração lambda) encontrada no cóccix.
- Isso sugere que devemos manter uma pilha de ponteiros para as vértebras, tal como as figura abaixo:



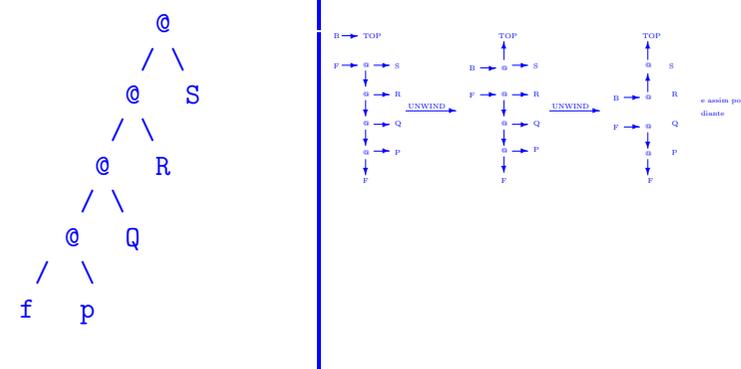
- Agora os argumentos estão todos disponíveis e o número de argumentos é dado pela profundidade da pilha.
- As próprias vértebras são acessíveis a partir da pilha, o que será muito prático na redução.

...A Pilha da Espinha

- Quando se avalia recursivamente os argumentos é necessário uma nova pilha. Felizmente:
 1. a pilha existente não se modifica até que a avaliação termine,
 2. a nova pilha pode ser descartada quando a avaliação termina.
- A nova pilha pode ser construída diretamente no topo da pilha original, tendo o cuidado de restituir corretamente a pilha original ao final da avaliação.
- Alternativamente pode-se usar uma pilha separada, chamada de *dump* para esse propósito.

Inversão de Ponteiros...

- Inversão de ponteiros é uma técnica que permite caminhar na espinha sem ter que construir uma pilha separadamente.
- Especificamente, tem-se dois ponteiros: *F* (apontando do ponto em que se está para frente) e *B* (apontando do ponto em que estamos para trás).
- À medida que se desce a espinha, o ponteiro *F* assume o valor do seu posterior e o ponteiro *B* assume o antigo valor de *F*.
- Depois será colocado um ponteiro antes de *B*, um ponteiro invertido para apontar o caminho de volta na espinha.



- Por exemplo, seja a seguinte espinha e sua representação

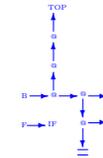
- Note, que a partir de *B* nós conseguimos caminhar de volta na espinha e a partir de *F* nós conseguimos descer na espinha.

...Inversão de Ponteiros

- Quando se chega ao cóccix, F apontará para a célula que representa a função f , e B apontará para o caminho inverso que possibilita achar as vértebras e os argumentos para a função.
- À medida que se volta na espinha, verifica-se ter chegado ao topo quando B for igual à TOP .

Argumentos e Inversão de Ponteiros

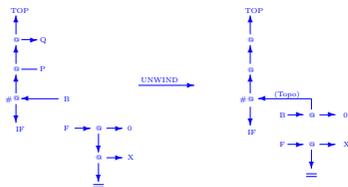
- Considere a expressão $(IF (= x 0) P Q)$.
- Quando se desce na espinha até encontrar o IF , o grafo fica da seguinte maneira:



- Agora deve-se avaliar o primeiro argumento do IF .
- Para isso tem-se que descer a espinha correspondente ao argumento.
- Infelizmente não se pode inicializar B com TOP , pois senão perder-se-ia o caminho inverso na espinha original.

Argumentos e Inversão de Ponteiros

- Assim, deve-se marcar a espinha original antes de passar F e B para o argumento.
- Quando se volta na espinha sabe-se que se alcançou o topo quando encontra-se o ponto marcado na espinha original.
- Depois de marcar o ponto de volta espinha original, teremos:



onde $\#$ é a marca na vértebra onde estava o B originalmente. $\#$ serve para marcar o topo da espinha do argumento.

Pilhas × Inversão de Ponteiros...

- A pilha é significativamente mais rápida que a inversão de ponteiros.
- Isso porque a pilha dá acesso instantâneo aos argumentos e às vertebrae sem ter que percorrer uma cadeia de ponteiros.
- Além disso, numa máquina paralela há maior *overhead* no acesso ao *heap* (global), do que à pilha (local);
- Inversão de ponteiros usa pouca memória. Necessário apenas um *bit* em cada célula para controlar a avaliação de argumentos para funções *built-in*.

...Pilhas × Inversão de Ponteiros

- Já a pilha, além de requerer espaço extra, dificulta o dimensionamento do espaço para ela em implementações em *hardware* já que a pilha pode crescer indefinidamente;
- A pilha oferece possibilidade para melhoria de performance como veremos nos capítulos sobre otimização da máquina-G e otimizações das chamadas de cauda generalizadas.
- Na implementação com inversão de ponteiro, o estado da avaliação é completamente salvo com os ponteiros F e B .
- Isso é útil numa máquina paralela onde uma avaliação pode ser suspensa e seu estado necessitar ser salvo de alguma maneira.

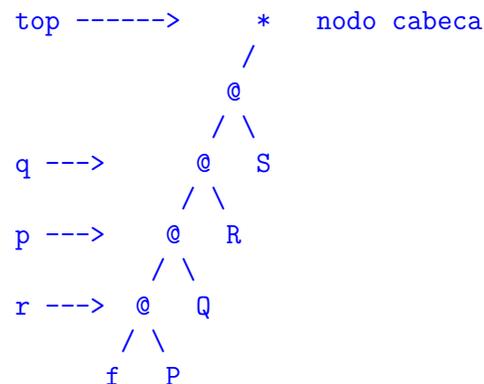
Espinha Duplamente Encadeada Eficiente...

- **Nodos da Espinha:**

```
struct Node {int code; Node* link; Node *right;}
typedef struct {struct Node *tip; struct Node * root;} Redex
```
- **Campo link:**
 - Da Cabeça: tem endereço da primeira vértebra
 - Demais nodos: ou exclusivo dos endereços vizinhos.
 - Por exemplo: $p \rightarrow \text{link} = q \oplus r$

...Espinha Duplamente Encadeada Eficiente

- ?????



- $A \oplus A = 0$, $A \oplus B = B \oplus A$, $A \oplus 0 = 0 \oplus A = A$

Determinação do Próximo Redex

- **Determinação do Redex:**

```

Redex getredex(struct Node * top){
    struct Node *p, *q, *r;
    Redex rdx; int n;
    q = top; p = top->link;
    while ( !istip(p) ){ -- procura tip
        r = q ^ p->link; q = p; p = r;
    }
    rdx.tip = p; n = nargs(p);
    while (n--){ -- procura root
        r = p ^ q->link; p = q; q = r;
    }
    rdx.root = p; return rdx;
}
  
```

II. REDUÇÃO DO GRAFO DE EXPRESSÕES

Redução uma Aplicação Lambda ...

- Executar uma redução é fazer uma transformação local no grafo, até que a forma final seja encontrada
- O cóccix da espinha pode ser uma abstração lambda, função *built-in* ou um objeto de dado
- Suponha que o *redex* consista em uma abstração lambda aplicada a um argumento.
- Então deve-se aplicar a regra de β -redução ao grafo.
- Este processo será chamado de instanciação do corpo lambda.
- Por exemplo

$(\lambda x. \text{NOT } x) \text{ TRUE} \rightarrow \text{NOT TRUE}$



...Redução uma Aplicação Lambda

- Deve-se considerar aqui três aspectos de implementação:
 1. o argumento pode ser volumoso e/ou conter *redexes*. Nesse caso, deve-se substituir o parâmetro formal por ponteiros para o argumento
 2. o *redex* pode ser compartilhado. Nesse caso, deve-se re-escrever a raiz do *redex* com o resultado
 3. a abstração lambda pode ser compartilhada. Nesse caso, deve-se construir uma nova instância do corpo lambda, ao invés de substituir diretamente no corpo original

Substituição com Ponteiros para Argumentos

- A solução de substituição de ponteiros para os argumentos implicará em compartilhamento.
- Poderão haver vários ponteiros para a mesma expressão e é por isso que a árvore se torna um grafo.
- Por exemplo

$(\lambda x. \text{AND } x x) (\text{NOT TRUE})$
 $\rightarrow \text{AND } (\text{NOT TRUE}) (\text{NOT TRUE})$



Re-Escrita da Raiz do *redex*

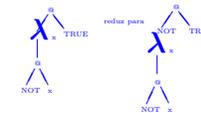
- Para explorar o compartilhamento adequadamente, deve-se garantir que quando uma expressão é reduzida, modifica-se o grafo para refletir o resultado.
- Isso garante que expressões compartilhadas sejam reduzidas uma só vez.
- Assim, seguindo o exemplo anterior,



- Note que os nodos NOT e TRUE são destacados da parte do grafo em que se está tratando.
- Os nodos NOT e TRUE não poderão ser reciclados pois podem estar compartilhados com outros nodos.

Construção de Nova Instância do Corpo

- A palavra instância implica em criar uma nova cópia do corpo lambda ao invés de atualizar o corpo original com as substituições.
- Isso é necessário, porque a abstração lambda pode ser aplicada várias vezes e o corpo original pode ser reutilizado.
- Por exemplo,



Função instantiate...

- Pode-se descrever uma operação de instanciação por uma função recursiva:

$$\text{instantiate}(\textit{Body}, \textit{Var}, \textit{Value})$$
 que copia *Body*, substituindo ocorrências livres de *Var* por *Value*
- $\text{instantiate}(\textit{Body}, \textit{Var}, \textit{Value})$ constrói $\textit{Body}[\textit{Value}/\textit{Var}]$
- Funções Auxiliares:
 - $\text{IsAp}(B)$ testa se B é um nodo de aplicação
 - $\text{GetFun}(B)$ retorna a função de um nodo de aplicação
 - $\text{GetArg}(B)$ retorna o argumento de um nodo de aplicação
 - $\text{MakeAp}(F, A)$ constrói um novo nodo de aplicação
 - $\text{IsVar}(B)$ testa se B é um nodo de variável

... Função instantiate

- Funções Auxiliares:
 - $\text{IsLam}(B)$ testa se B é um nodo de abstração lambda,
 - $\text{GetVar}(B)$ retorna o parâmetro formal de uma abstração,
 - $\text{GetBody}(B)$ retorna o corpo de um nodo de abstração,
 - $\text{MakeLam}(V, B)$ constrói um novo nodo de abstração lambda.
- A estratégia de redução de grafos *lazy* é conseguida com a avaliação na ordem normal para a forma normal com cabeça fraca, juntamente com a substituição de ponteiros e a atualização da raiz do *redex* com o resultado.

Código C para instantiate

```

inst (expression *Body, *Var, *Value){
  if (IsAp (Body)) {
    return (MakeAp (
      inst(GetFun(Body),Var,Value),inst(GetArg(Body),Var,Value));
  }
  if (IsVar (Body)) {
    if (Body == Var) return (Value); else return (Body);
  }
  if (IsLam (Body)) {
    if (GetVar (Body) == Var) return (Body);
    else return (MakeLam (GetVar (Body),
      inst(GetBody(Body),Var,Value)));
  }
  return(Body); /* Entao e' uma cte ou built-in */
}

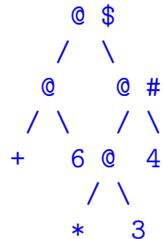
```

Redução da Aplicação...

- Quando aplica-se uma função *built-in*, pressupõe-se que seu número e tipo de argumentos estejam corretos.
- Os argumentos devem ser avaliados antes da chamada, podendo ser feito através de chamadas recursivas ao avaliador.
- Aí sim, a função *built-in* pode ser executada e o resultado escrito sobre a raiz do *redex*.

...Redução da Aplicação

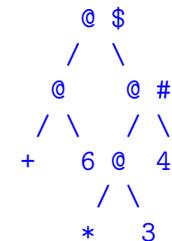
- Por exemplo, considere a avaliação da expressão $(+ 6 (* 3 4))$ cujo grafo é:



- Seleciona-se o nó \$ para redução.

Redução da Aplicação...

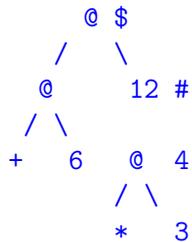
- Assim, dada a expressão



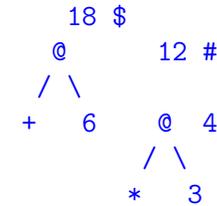
- Verifica-se que a função + precisa avaliar seus argumentos.
- O primeiro argumento é 6 e já está na WHNF.
- O segundo argumento começa a partir do nó # que precisa ser avaliado.

...Redução da Aplicação...

- O avaliador é chamado recursivamente.
- Tem-se agora que avaliar os argumentos da função *.
- Os argumentos de * já estão na WHNF. Procedemos a redução e o resultado é escrito sobre o nodo #.
- O grafo resultante será:

Reduzindo a Aplicação

- Os fragmentos presentes no grafo original precisam ser recuperados pelo coletor de lixo.
- Agora os argumentos de + estão na WHNF, e pode-se proceder a redução:

O Algoritmo de Redução Completo

REPEAT

1. Desça (*Unwind*) a espinha até que algo diferente de um nodo de aplicação seja encontrado.
2. Examine objetos encontrados no cóccix da espinha, que podem ser:
 - Um objeto de dados ⇒ Reduza o Objeto de Dados
 - Uma função *built-in* ⇒ Reduza a Função *Built-in*
 - Uma abstração lambda ⇒ Reduza a Abstração Lambda

UNTIL FALSE

... O Algoritmo de Redução Completo

- Reduza o Objeto de Dados:
Se o objeto não estiver aplicado a nada, a expressão está na WHNF, e aí pare. Senão ocorreu um ERRO
- Reduza a Função *built-in*:
Se o número de argumentos da função for insuficiente a expressão está na WHNF, e aí pare. Senão avalie os argumentos, execute essa função e escreva o resultado sobre a raiz do *redex*
- Reduza a Abstração Lambda:
Se não existirem argumentos a expressão está na WHNF, e aí pare. Senão, instancie o corpo da abstração lambda substituindo os parâmetros formais por ponteiros para os argumentos e escreva o resultado sobre a raiz do *redex*.

Nodos de Indireção...

- Na operação final de atualizar a raiz do *redex* com o resultado existe um perigo.
- Suponha que já se instanciou o corpo da abstração e está na hora de atualizar a raiz do *redex*.
- A maneira mais óbvia de se fazer isso seria copiar a célula raiz do resultado no topo da célula raiz do *redex*.
- Observe que:
 - O resultado pode não ter uma célula raiz para a cópia.
 - Por exemplo, $(\lambda x. 4) 5$, numa implementação *unboxed* resultado (4) não ocupa uma célula própria.
 - Existe uma certa ineficiência, pois a célula raiz do resultado é construída, copiada sobre a raiz do *redex* e descartada, pois não haverá nenhum ponteiro para ela.

...Nodos de indireção

- Seria mais eficiente construir a célula da raiz do resultado diretamente no topo da célula da raiz do *redex*.
- Contudo, na seguinte redução:

$$(\lambda x. x) (f 6),$$
 a célula da raiz do resultado não é uma célula que será construída de novo.
- Então não se pode construir essa célula em cima da raiz do *redex*.
- Abstrações lambda, nas quais o corpo consiste em constantes *unboxed* de uma única variável, constituem um caso especial.

Atualizações com Objetos *unboxed*

- Para atualizar a raiz do *redex* com um objeto *unboxed* é necessário introduzir um novo tipo de célula: a célula de indireção
- Essa célula de indireção tem um *tag(IND)* e um único campo que contém o conteúdo da célula
- Para se atualizar uma célula de aplicação com um objeto *unboxed*, escreve-se sobre a célula de aplicação uma célula de indireção cujo conteúdo é o objeto *unboxed*
- Por exemplo,



onde ∇ indica um nodo de indireção.

- Numa implementação *boxed*, isso não seria necessário, pois o objeto teria seu nodo próprio, que poderia ser copiado sobre a raiz do *redex*

Quando Corpo é uma Única Variável...

- Considere como exemplo a expressão $((\lambda x. x) (f 6))$



- Existem duas maneiras para atualizarmos a raiz do *redex*
- Maneira I: Copiar a célula da raiz do resultado sobre a raiz do *redex*. Assim teríamos o seguinte resultado:



- Apesar de o resultado estar correto, há uma duplicação da aplicação de $(f 6)$ com a existência do nodo $\#$
- Isso gera complicações se o nodo $\#$ for compartilhado, pois poderia ser avaliado duas vezes, gerando ineficiência, perdendo *laziness*, além do gasto de espaço com o nodo $\#$

...Quando Corpo é uma Única Variável

- Maneira II: Uma outra alternativa, seria o uso do nodo de indireção
- Assim, poderíamos escrever sobre o nodo \$ (raiz do *redex*) um nodo de indireção para o nodo #. O resultado da redução seria:



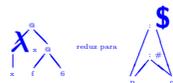
- O problema com a introdução de nodos de indireção é que eles podem aparecer em qualquer ponto do grafo e aí a máquina de redução deve conter testes para nodos de indireção em vários lugares
- Além do mais, cadeias longas de nodos de indireção podem "emperrar" a máquina
- Em geral, qualquer função, cujo resultado pode não ser uma célula construída durante a redução, gera o dilema de como atualizar a raiz do *redex*

Avaliando Antes de Atualizar...

- Uma solução para os principais problemas das duas alternativas anteriores é avaliar o resultado antes de atualizar a raiz do *redex*.
- Note que está-se interessado em reduzir o nodo \$ para a WHNF.
- Depois deseja-se reduzir o resultado ((f 6)) para a WHNF.
- Assim, pode-se reduzir de forma segura o nodo # para a WHNF antes de escrever o resultado sobre o no \$.
- Observe que se o resultado da redução do nodo # for, por exemplo, uma indireção para uma célula CONS(seja ela *), então poder-se-ia escrever sobre o nodo \$ uma célula com indireção para o nodo * (e não para o nodo #).
- Assim, nunca haveria mais de um nodo de indireção numa cadeia.

...Avaliando Antes de Atualizar

- Uma vez que uma expressão está na forma normal sua raiz não será re-escrita novamente, porque ela não será selecionada novamente como a raiz do *redex*
- A observação acima diz que é seguro copiar o nodo # uma vez que ele está na WHNF, desde que ele nunca será re-escrito
- Efetuando essa cópia continuamos a gastar espaço, porém não faremos reduções duplicadas
- Por exemplo, se (f 6) avaliasse para uma célula CONS, a cópia nos daria:



Sumário: Indireção × Cópia

- Quando a raiz do resultado é construída durante a redução e é suficientemente pequena, ela deve ser construída diretamente no topo da raiz do *redex*, ao invés de ser alocada em algum lugar e depois copiada e descartada.
- Se a raiz do resultado não foi construída durante a redução, então pode-se reescrever a raiz do *redex* ou com uma cópia da raiz do resultado ou com uma indireção para o resultado.
- Os casos cobertos pelo segundo ponto são:
 - funções (*built-in* ou abstrações lambda) que retornem resultados *unboxed*;
 - abstrações lambda cujo corpo consiste de uma única variável;
 - funções de projeção *built-in*, por exemplo: HEAD, TAIL, IF.

...Indireção × Cópia ...

- Nos casos cobertos pelo segundo item, o resultado deve ser avaliado para WHNF antes de re-escrever a raiz do *redex*.
- Sendo assim, não é perdido compartilhamento e o número de reduções é o mesmo.
- Existem os seguintes argumentos a favor do uso de indireção.
- Não há outra alternativa se o resultado é um *unboxed*.
- Indireção não usa menos células que a cópia, quando a redução se procede. Porém elas podem ser logo recicladas pelo coletor de lixo. Já o espaço alocado com a duplicação da cópia não pode ser recuperado.
- Não há problema se a raiz do resultado é maior que a raiz do *redex*.
- Cadeias de nodos de indireção podem ser evitadas.

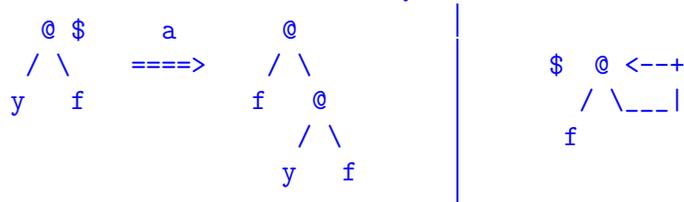
...Indireção × Cópia

- Existe facilidade para incorporar funções memo.
- Funções memo são aquelas que lembram a quais argumentos elas foram aplicadas juntamente com seus respectivos resultados.
- Daí, quando funções memos são aplicadas àqueles argumentos novamente, o resultado é entregue diretamente.
- A facilidade vem do fato de que se tivéssemos trabalhando com cópias dos nodos, então argumentos idênticos pareceriam diferentes à função memo.
- Existe um único argumento, porém persuasivo, contra o uso de indireção: a máquina de redução deve fazer testes contínuos para verificar a presença de nodos de indireção e de-referenciá-los à medida que eles apareçam.
- E isso adiciona um grande número de testes potencialmente lentos.

Implementando Y...

- Recordando, a regra de redução para Y é:
 $Y f \rightarrow f (Y f)$

- Existem duas maneiras de implementar:



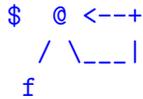
- O primeiro caminho é mais direto, porém o segundo é mais interessante.

...Implementando Y...

- Para verificar que o segundo caminho é correto, basta verificar a regra de redução para Y .
- No lado direito da regra, f é aplicada a $(Y f)$.
- Veja que o *redex* original também era $(Y f)$, então f pode ser aplicada a *redex* original.
- Outra maneira de ver a correção é:
 $Y f \rightarrow f (Y f)$
 $\rightarrow f (f (Y f))$
 \dots
 $\rightarrow f (f (f (...)))$
que é exatamente o que sugere o grafo.

...Implementando Y

- O grafo passou a ser cíclico.



- Grafos cíclicos economizam bastante o uso de memória.
- Grafos cíclicos dão representação finita para objetos infinitos, tais como funções recursivas e estruturas de dados infinitas.
- A principal desvantagem do uso da representação cíclica de Y é que a presença de ciclos inabilita o uso do coletor de lixo simples com contagem de referências.

II. SUPERCOMBINADORES E LAMBDA LIFTING

Supercombinadores e Lambda-Lifting...

- A operação de instanciar o corpo de uma abstração lambda é essencial, portanto deve-se fazê-la de forma eficiente:
 - Introduzem-se abstrações lambda especiais, os supercombinadores
 - Define-se o processo de transformação (*lambda-lifting*) das abstrações comuns em supercombinadores

...Supercombinadores e Lambda-Lifting

- A operação *instantiate* vista no capítulo anterior é ineficiente pelos seguintes motivos:
 1. em cada nodo do corpo, *instantiate* tem que fazer uma análise de caso no *tag* do nodo
 2. em cada nodo de variável, *instantiate* deve testar se o nodo é o parâmetro formal. Um teste similar tem que ser feito em cada nodo lambda
 3. novas instâncias de subexpressões, que não contenham nenhuma ocorrência livre do parâmetro formal, serão construídas quando puderem ser compartilhadas de maneira segura e benéfica

Redução mais Eficiente

- Uma alternativa mais eficiente é a compilação, onde se associa a cada corpo lambda uma sequência fixa de instruções que construirão uma instância do corpo lambda
- Essa sequência de instruções pode ser construída pelo compilador
- E pode conter implicitamente conhecimento sobre a forma do corpo e onde os parâmetros formais ocorrem
- Assim, pode-se esperar que o código compilado execute mais rápido que a operação de *instantiate*
- Infelizmente, nem toda abstração lambda pode ser compilada dessa maneira. Seja o exemplo,

$$(\lambda x. (\lambda y. - y x))$$
- Quando se aplica a abstração λx a um argumento, 3 por exemplo, criamos uma nova instância $(\lambda y. - y 3)$

...Redução mais Eficiente

- Cada aplicação da abstração λx cria uma nova aplicação λy , sendo inviável uma sequência fixa de instruções para cada λy diferente
- O problema acontece porque x ocorre livre em λy
- Caso as abstrações lambda não tenham variáveis livres, não existirá problema para a compilação para uma sequência fixa de código
- Uma solução é parametrizar o código nos valores das variáveis livres
- Essa abordagem levou à máquina SECD, cujo código acessa um ambiente que contém os valores das variáveis livres
- A abordagem redução de grafos, que não requer a adição de um ambiente para nosso modelo
- A idéia é fazer um *lambda-lifting*, isto é, transformar abstrações lambda quaisquer em abstrações equivalentes, que não contenham variáveis livres

O Problema das Variáveis Livres...

- Será introduzida uma nova forma de β -abstração, a qual substitui os argumentos de uma só vez
- Por exemplo, a expressão abaixo, que normalmente seria reduzida em dois passos,

$$(\lambda x. \lambda y. - y x) 3 4 \rightarrow \lambda y. - y 3 \rightarrow - 4 3$$
- seria reduzida diretamente para

$$\rightarrow - 4 3$$

...O Problema das Variáveis Livres

- Observe o seguinte em relação à redução multi-argumentos:
 1. Há um ganho considerável fazendo as reduções simultaneamente, pois não haverá necessidade de alocar espaço no *heap* para estruturas intermediárias
 2. Não existe o problema da ocorrência de x livre em λy
 3. Nada é perdido com a redução simultânea
 4. O resultado intermediário que seria a abstração λy , não seria usado, pois nada seria feito até que ela, λy , fosse reduzida

Supercombinadores

- Um supercombinador, $\$S$, de aridade n é uma abstração lambda da forma:

$$\lambda x_1. \dots \lambda x_n. E,$$

onde E não é uma abstração lambda tal que

1. $\$S$ não tem variáveis livres,
 2. qualquer abstração lambda em E é um supercombinador,
 3. $n \geq 0$, ou seja, não é necessário a existência de lambdas
- Um *redex* de supercombinador consiste na aplicação de um supercombinador a n argumentos
 - Uma redução de supercombinador troca um *redex* de supercombinador por uma instância do corpo do supercombinador
 - Os argumentos do supercombinador substituem as ocorrências livres dos respectivos parâmetros formais

Aplicação da Definição de Supercombinadores

- São exemplos de supercombinadores:

3	$\lambda x. x$	$\lambda x. \lambda y. - y x$
$+ 2 5 (+ 2 5)$	$\lambda x. x 1$	$\lambda f. f (\lambda x. + x x)$
	$\lambda x. + x x$	

- Enquanto não são supercombinadores:

- $\lambda x. y$
y ocorre livre
- $\lambda y. - y x$
x ocorre livre
- $\lambda f. f (\lambda x. f x 2)$
a abstração λx não é um supercombinador, pois nela f ocorre livre

Supercombinadores e sua Aridade...

- Supercombinadores de aridade não-zero são importantes pois eles serão nossa unidade de compilação
- Já que eles não tem variáveis livres, podem ser compilados para uma sequência fixa de código
- Nenhuma abstração lambda no corpo do supercombinador tem variáveis livres
- Por isso, ela não precisa ser copiada quando se instancia o corpo do supercombinador

...Supercombinadores e sua Aridade

- Um supercombinador é como uma função em linguagem imperativa que tem vários parâmetros e não usam variáveis globais e nem provocam efeitos colaterais.
- Os supercombinadores sem parâmetros são apenas expressões constantes
- Eles são chamados geralmente de formas aplicativas constantes (CAFs)
- São exemplos de CAFs: $1, + 1 2, + 2$

Combinadores

- Um combinador é uma expressão lambda que não contém ocorrências de variáveis livres.
- Um combinador é uma função pura no sentido que o valor da aplicação de um combinador a alguns argumentos depende somente dos valores do argumento e não de alguma variável livre.
- Algumas expressões lambda são combinadores e alguns combinadores são supercombinadores.

Compilação Baseada em Supercombinadores

- Por questões de clareza os supercombinadores receberão nomes.
- São nomes arbitrários, pois as abstrações lambda são anônimas.
- Por convenção eles começarão com \$.

Por exemplo:

$$\$XY = \lambda x. \lambda y. - y x$$

ou ainda

$$\$XY x y = - y x$$

Compilação Baseada em Supercombinadores...

- A estratégia é transformar a expressão lambda que se quer compilar em:
 1. um conjunto de definições de supercombinadores, mais
 2. uma expressão a ser avaliada
- Então, tal qual em Miranda, tem-se:

Definições de supercombinadores

... ..
... ..

Expressão a ser avaliada

...Compilação Baseada em Supercombinadores ...

- Por exemplo, pode-se representar a expressão:
($\lambda x. \lambda y. - y x$) 3 4

como

$$\underline{\$XY x y = - y x}$$

$$\$XY 3 4$$

- Um ponto importante é que uma redução de supercombinador só ocorre quando todos argumentos estão presentes.
- Por exemplo, ($\$XY 3$) não é um *redex* de supercombinador e, por isso, não será reduzido.

...Compilação Baseada em Supercombinadores

- Pode-se ver as definições de supercombinador como um conjunto de regras de re-escrita.
- Uma redução consiste em re-escrever uma expressão que casa com o lado esquerdo de uma regra com uma instância do lado direito correspondente.

Transformação de Lambdas em Supercombinadores

- Seja o seguinte programa,
 $(\lambda x. (\lambda y. + y x) x) 4$
 onde nenhuma das abstrações lambda é um supercombinador
- Considere a abstração lambda mais interna $(\lambda y. + y x)$
- Ela tem a variável x livre e por isso não é um supercombinador.
- Uma transformação simples (β -abstração) fará da abstração um supercombinador, fazendo cada variável livre um parâmetro extra (isso pode ser chamado de abstrair a variável livre)
- Assim, $(\lambda y. + y x)$ poderia ser transformada em
 $(\lambda x. \lambda y. + y x) x$ que é um supercombinador
- Para ficar mais claro, pode-se fazer uma α -conversão em λx , tal qual, $(\lambda w. \lambda y. + y w) x$

...Transformação de Lambdas em Supercombinadores ...

- Dada a expressão $(\lambda w. \lambda y. + y w) x$
- Fazendo a transformação no programa original tem-se:
 $(\lambda x. (\lambda w. \lambda y. + y w) x x) 4$
- Pode-se nomear o supercombinador mais interno como sendo $\$Y$:
 $\$X w y = + y w$

 $(\lambda x. \$Y x x) 4$
- Pode-se também nomear o supercombinador para λx :
 $\$Y w y = + y w$
 $\$X x = \$Y x x$

 $\$X 4$
- As reduções do supercombinador: $\$X 4 \rightarrow \$Y 4 4 \rightarrow + 4 4 \rightarrow 8$

...Transformação de Lambdas em Supercombinadores ...

- Revendo o algoritmo agora temos:
 WHILE existir abstrações lambda
 1. escolha qualquer abstração lambda que não tenha nenhuma outra abstração lambda interna ao seu corpo
 2. retire todas as variáveis livres da abstração escolhida e coloque-as como parâmetro extra
 3. dê um nome arbitrário à abstração lambda
 4. substitua a ocorrência da abstração lambda pelo nome dado aplicado às variáveis livres
 5. compile a abstração lambda e associe um nome ao código compilado
 END

...Transformação de Lambdas em Supercombinadores

- Quando se completa o algoritmo, chega-se a um programa da forma:

definições de supercombinadores ...

E

- Como a expressão E não deve conter variável livre, pode-se transformá-la numa CAF, tendo:

definições de supercombinadores ...

$\$PROG = E$

$\$PROG$

- Assim todo o programa é composto de supercombinadores.

Eliminando Parâmetros Redundantes...

- Considere o supercombinador:

$\lambda x. \lambda y. - y x$

- Uma aplicação cega do algoritmo anterior resulta:

$\$Y x y = - y x$

$\lambda x. \$Y x$

- Agora tratando a abstração λx , teremos:

$\$Y x y = - y x$

$\$X x = \$Y x$

$\$X$

...Eliminando Parâmetros Redundantes

- Pode-se simplificar $\$X$ para $\$X = \Y , e tem-se:

$\$Y x y = - y x$

$\$Y$

- Há duas otimizações a considerar:

1. remover parâmetros redundantes das definições por η -redução,
2. onde o item anterior produzir definições redundantes, remova-as

Ordenação de Parâmetros ...

- O objetivo é remover os parâmetros redundantes de supercombinadores e, se possível, eliminar sua própria definição.
- Deve-se ordenar as variáveis livres de tal forma que as de λ s mais internos fiquem por último nas lista de parâmetros do supercombinador.
- Isso sugere associar um nível léxico a cada abstração lambda.
- O nível léxico de uma abstração será definido como um número a mais que o número de abstrações que envolvem aquela abstração.

...Ordenação de Parâmetros

- O nível da abstração mais externa será 1.
- O nível de uma variável é o nível da abstração que a liga.
- O nível de uma constante é zero.
- Para maximizar as chances de poder aplicar η -reduções, ordenam-se os parâmetros extras na ordem crescente de seus níveis léxicos.

...Ordenação de Parâmetros...

- Seja a seguinte expressão:
 $(\dots (\lambda x. \lambda z. + y (* x z)) \dots)$
- Transformando a abstração λz , pode-se ter as diferentes expressões:

$$\frac{\$S x y z = + y (* x z)}{(\dots (\lambda x. \$S x y) \dots)} \quad \Bigg| \quad \text{ou} \quad \Bigg| \quad \frac{\$S y x z = + y (* x z)}{(\dots (\lambda x. \$S y x) \dots)}$$
- x e y são livres, então não faz diferença a ordem em que eles são colocados no supercombinador.

...Ordenação de Parâmetros

- Considerando a segunda possibilidade, i.e.,

$$\$S y x z = + y (* x z)$$

$$(\dots (\lambda x. \$S y x) \dots)$$

- E fazendo um *lifting* na abstração λx , tem-se:

$$\$S y x z = + y (* x z)$$

$$\$T y x = \$S y x$$

$$(\dots (\$T y) \dots)$$

Implementação de Supercombinadores: FORMA I

- Pode-se manter o corpo do supercombinador como uma árvore e instanciá-lo usando uma função similar à *instantiate*.
- Essa forma daria um redutor semelhante ao redutor lambda do último capítulo.
- Todos os mecanismos descritos nos capítulos anteriores, mostrando como encontrar o próximo *redex*, como fazer uma redução, nodos de indireção, a pilha e outros assuntos, ainda são válidos.
- A única mudança será requerida na implementação de *instantiate*.
- *instantiate* terá simplificações por um lado, pois todas abstrações lambda são supercombinadores, que por não ter variáveis livres não precisam ser copiadas.
- Por outro lado, há complicações, pois terá que substituir várias variáveis de uma vez;

Implementação de Supercombinadores: FORMA II

- Pode-se manter o corpo do supercombinador como uma árvore, mas mantido num espaço de memória contínuo.
- Agora a instanciação pode ser feita de maneira mais eficiente com movimentação de blocos.
- Essa ideia foi usada por Keller na redução de grafos distribuída.

Implementação de Supercombinadores: FORMA III

- Pode-se compilar o corpo para uma sequência linear de instruções que criará uma instância da abstração, quando executada.
- Essa será nossa abordagem descrita para a frente.

II. SUPERCOMBINADORES RECURSIVOS

Supercombinadores Recursivos

- Uma maneira de manipular definições recursivas seria o uso da função Y
- Isso é ineficiente pois:
 - não existe razão pela qual supercombinadores não devem ser explicitamente recursivos já que eles têm nome
 - por exemplo, para

$$F\ x = G\ (F\ (-\ x\ 1))\ 0$$
 seria necessário uma definição adicional:

$$F = Y\ F_1$$

$$F_1\ F\ x = G\ (F\ (-\ x\ 1))\ 0$$
 - no caso de recursão mútua sem letrec, o uso de Y acarretaria ter que manter uma tupla à parte para agrupar as definições
- A seguir, apresentam-se técnicas para obter um conjunto de supercombinadores recursivos sem o uso de Y .

Notação

- Não se quer que a recursão seja transformada em aplicações do combinador Y .
- Supõe-se também que o programa será traduzido para notação lambda incrementada com *lets* e *letrecs* simples.
- As seguintes notações são equivalentes:

$$\begin{array}{l}
 \$S_1 \ x \ y = B_1 \\
 \$S_2 \ f = B_2 \\
 \dots \\
 \text{-----} \\
 E
 \end{array}
 \quad \left| \begin{array}{l}
 \text{é} \\
 \text{equivalente} \\
 \text{a}
 \end{array} \right.
 \quad \begin{array}{l}
 \text{letrec } \$S_1 = \lambda x. \lambda y. B_1 \\
 \quad \quad \quad \$S_2 = \lambda f. B_2 \\
 \dots \\
 \text{in } E
 \end{array}$$

Lets e Letrecs nos Corpos de Supercombinadores

- Suponha que se queira uma descrição textual do seguinte grafo:



- Uma alternativa seria nomear os nodos (a, b e c) de cima para baixo e construir a seguinte expressão letrec:

```

letrec  a = c b
        b = c 3
        c = f b
in      a

```

Lets e Letrecs nos Corpos de Supercombinadores

- Permitindo *letrecs* no corpo de um supercombinador, permite-se que esse corpo ser um grafo cíclico.
- E aplicar tal supercombinador implica na construção de uma nova instância desse grafo.
- De maneira similar, permite-se que os corpos de supercombinadores tenham *lets*, sendo eles descritos como grafos acíclicos.

Lambda-Lifting na Presença de letrecs

- Em particular, o *lambda-lifting* só é aplicável por enquanto a expressões lambda e não a *letrecs*.
- As variáveis ligadas em um letrec serão instanciadas quando a abstração lambda que a envolve for aplicada a um argumento.
- O nível léxico de um letrec é igual ao da expressão lambda imediatamente envolvente.
- Se não houver nenhuma abstração lambda envolvendo o letrec, então o nível léxico é zero.
- Então um letrec nessa condição é um supercombinador.

Lambda-Lifting na Presença de letrecs

- Por exemplo,

```
letrec x = CONS 1 x
in x
```

- é equivalente a

```
$X = CONS 1 $X
```

```
$X
```

Geração de Corpos do Tipo Grafo

- Até agora nenhum de nossos supercombinadores teve um corpo do tipo grafo.
- Isso ocorre quando há letrec no corpo do supercombinador.
- Considere o programa:

```
let inf = λv. (letrec vs = CONS v vs in vs)
in inf 4
```

- que é equivalente a:

```
$inf v = letrec vs = CONS v vs in vs
$Prog = $inf 4
```

```
$Prog
```

- Note que o corpo do supercombinador, sendo do tipo grafo, preserva a representação cíclica (finita) de uma estrutura de dados

Um Exemplo...

- Considere o programa em Miranda para somar os primeiros 100 inteiros.

```
sumints m = sum (count 1)
where count n = [ ], n > m
              = n : count(n + 1), otherwise
```

```
sum [ ] = 0
sum (n : ns) = n + sum ns
```

```
sumints 100
```

... Um Exemplo...

- Traduzindo para o cálculo lambda enriquecido:

```
letrec
  sumints = λm.letrec count = λn.IF(> n m) NIL
                                (CONS n (count (+ n 1)))
in sum(count 1)
```

```
sum = λns.IF(= ns NIL) 0
      (+ (HEAD ns)(sum(TAIL ns)))
```

```
in sumints 100
```

...Um Exemplo ...

- Efetuando um *lambda-lifting* na função `count`:

```
$count count m n = IF (> n m) NIL (CONS n (count (+ n 1)))
```

```
letrec
```

```
  sumints =  $\lambda$  m.letrec count = $count count m
            in sum(count 1)
```

```
  sum =  $\lambda$  ns.IF(= ns NIL) 0 (+ (HEAD ns) (sum(TAIL ns)))
```

```
in sumints 100
```

...Um Exemplo

- Agora um *lambda-lifting* em `sumints` e `sum`:

```
$count count m n = IF(> n m) NIL (CONS n (count (+ n 1)))
```

```
$sum ns = IF(= ns NIL) 0 (+ (HEAD ns) ($sum (TAIL ns)))
```

```
$sumints m = letrec count = $count count m
              in $sum (count 1)
```

```
$Prog = sumints 100
```

```
$Prog
```

Abordagens Alternativas...

- Johnson [Johnson, 1985] descreve um algoritmo para construir corpos de supercombinadores do tipo grafo para estruturas de dados, mas não para funções.
- Para descrever essa técnica, seja um programa com uma função recursiva f contendo uma variável livre v :
 $(\dots \text{letrec } f = \lambda x. (\dots f \dots v \dots) \text{ in } (\dots f \dots) \dots)$
- Gera-se um supercombinador recursivo abstraindo as variáveis livres (somente v no caso) e não abstraindo f propriamente dita.
- Assim, todos os usos de f são substituídos por $\$f v$, inclusive no próprio corpo de $\$f$. Tem-se:

```
 $\$f v x = \dots (\$f v) \dots v \dots$ 
```

```
 $(\dots (\dots (\$f v) \dots) \dots)$ 
```

...Abordagens Alternativas ...

- Para ilustrar o método, lembre-se do exemplo anterior, que pode ser compilado para:

```
$count m n = IF(> n m) NIL (CONS n($count m (+ n 1)))
```

```
letrec
```

```
  sumints =  $\lambda$  m.sum($count m 1)
```

```
  sum =  $\lambda$  ns.IF(= ns NIL) 0 (+ (HEAD ns) (sum(TAIL ns)))
```

```
in sumints 100
```

...Abordagens Alternativas ...

- Agora `umints` e `sum` são supercombinadores, e portanto o *lambda-lifting* é direto:

```
$count m n = IF(> n m) NIL (CONS n ($count m (+ n 1)))
```

```
$sum ns = IF(= ns NIL) 0 (+ (HEAD ns)($sum (TAILns)))
```

```
$sumints m = $sum ($count m 1)
```

```
$Prog = $sumints 100
```

```
$Prog
```

...Abordagens Alternativas

- Note que nenhum supercombinador tem corpo do tipo grafo.
- Toda recursão é tratada diretamente pela recursão dos supercombinadores.
- Contudo, estruturas de dados cíclicas devem ser tratadas de maneira diferente e requerem supercombinadores com corpos do tipo grafo.
- A grande vantagem da abordagem é que a recursão explícita do supercombinador pode ser compilada de maneira mais eficiente.
- Por outro lado, o supercombinador `$count` tem um nodo de aplicação extra:


```
( $count m )
```
- Por isso, na instanciação desse nodo será consumido mais espaço.

Reduções em Tempo de Compilação

- Aplica-se um processo análogo a expansão de código numa chamada de procedimento, em compiladores convencionais.

- Por exemplo,

```
$F x y = + ($G y) x
```

```
$G p = * p p
```

- poderia ser transformado em:

```
$F x y = + (* y y) x
```

- No caso de `y` ser uma expressão mais elaborada pode-se preservar o compartilhamento com

```
$G E → let p = E in * p p
```

- Outra técnica vinda da tecnologia convencional de compiladores é a redução de expressões constantes.

Por exemplo, `$Hx = + x(* 3 4)` transforma-se em `$Hx = + x 12`

Eliminação de Subexpressões Comuns

- Às vezes, por questões de clareza, o programador escreve:


```
* ($F x) ($F x)
```

- Para promover o compartilhamento podemos substituir a expressão acima por

```
let fx = $F x
```

```
in * fx fx
```

- Essa identificação pode ser feita com um algoritmo de *hashing*.

Eliminando *lets* Redundantes

- Lados direitos da definição de *lets* que são variáveis podem ser eliminados
- Por exemplo, seja `let x = y in E`
- Esse `let` pode ser eliminado substituindo as ocorrências de `x` por `y` em `E`
- Quando uma variável definida num `let` é usada somente uma vez no corpo, pode-se removê-la
- Por exemplo,

$$\begin{array}{l} \text{let xsq} = * x x \\ \text{ysq} = * y y \\ \text{in } + \text{ xsq ysq} \end{array} \Rightarrow \quad + (* x x) (* y y)$$
- Isso pode ser feito simplesmente acumulando informação do número de ocorrências textuais distintas de cada variável do corpo do `let`

II. LAMBDA-LIFTING TOTALMENTE LAZY

Preguiça Total...

- Uma implementação direta do procedimento de instanciação corre o risco de construir múltiplas instâncias da mesma expressão, ao invés de compartilhar uma única cópia
- Esse compartilhamento é válido desde que não ocorra variáveis livres dentro da expressão
- Sabe-se das consequências em termos de espaço e tempo no caso de não compartilhamento
- Para solucionar a questão, o compilador talvez pudesse que marcar essas expressões que fossem compartilhadas
- Mas, acontece que muitas vezes expressões que precisam ser compartilhadas são criadas dinamicamente

...Preguiça Total...

- No exemplo,

$$\begin{array}{l} \text{letrec } f = g \ 4 \\ \quad \quad g = \lambda x. \lambda y. + y \ (\text{sqrt } x) \\ \text{in } + (f \ 1) (f \ 2) \end{array}$$
- A cada vez que λy que ocorre dentro de `g` for aplicada, é criada uma nova instância de `(sqrt 4)`
- Nesse caso, `(sqrt 4)` sempre poderia ser compartilhada

...Preguiça Total

- Esse problema também acontece com supercombinadores.
- Por exemplo,

```
$G x y = + y (sqrt x)
$F = $G 4
$Prog = + ($F 1) ($F 2)
```

```
$Prog
```

- Para uma redução normal pode-se perceber que $(\text{sqrt } 4)$ é avaliada duas vezes
- O objetivo é que toda expressão seja avaliada no máximo somente uma vez, depois que suas variáveis tenham sido ligadas
- Isso é o que se chama de *full laziness*

Expressões Livres

- *Laziness* pode ser perdida quando se instanciar demais uma expressão lambda
- As subexpressões do corpo de uma expressão lambda, que não contém ocorrência (livre) do parâmetro formal não devem ser instanciadas
- Isso porque para qualquer instância da abstração lambda, aquelas subexpressões serão sempre as mesmas e poderão ser compartilhadas
- Definição: Uma subexpressão lambda E de uma abstração lambda L é livre em L se todas as variáveis em E são livres em L

Expressões Livres Máximas

- Definição: A expressão livre máxima (ELM) de L é uma expressão livre de L que não seja uma subexpressão própria de outra subexpressão livre de L
- Definição: E é uma subexpressão própria de F se e somente se E é uma subexpressão de F e $E \neq F$
- Por exemplo, temos as expressões livres máximas grifadas:


```
 $\lambda x. \text{sqrt } x$ 
 $\lambda x. x (\text{sqrt } 4)$ 
 $\lambda y. \lambda x. + x (* y y)$ 
 $\lambda y. \lambda x. + (* y y) x$ 
 $\lambda x. (\lambda x. x) x$ 
```
- No último exemplo, $(\lambda x. x)$ é livre apesar da colisão de nomes

...Expressões Livres Máximas

- Desse modo, para conseguir *full laziness* não se deve instanciar expressões livres máximas, nas β -reduções
- Ao invés, deve-se substituir a única instância compartilhada no corpo da abstração lambda, por um ponteiro
- Para prover um lambda redutor com *full-laziness*, deve-se ser capaz de identificar expressões livres maximais dinamicamente
- Deve-se modificar o algoritmo de *lambda-lifting* de tal forma que o programa resultante seja *fully lazy*

Modificando o Algoritmo de *lambda-lifting*...

- A idéia central (e suficiente) da modificação é abstrair as expressões livres máximas ao invés das variáveis livres.
- No exemplo anterior, a função g tem a seguinte abstração lambda:

$$\lambda x. \lambda y. + y (\text{sqrt } x)$$
- Quando se fizer o *lambda-lifting* na abstração λy , em vez de abstrair o $\text{tt } x$ que ocorre livre, deve-se abstrair $(\text{sqrt } x)$, que é uma expressão livre máxima, gerando o supercombinador:

$$\$G_1 \text{ sqrtx } y = + y \text{ sqrtx}$$
 onde o nome sqrtx é arbitrário.
- Agora substitui-se a abstração λy pelo supercombinador $\$G_1$ aplicado à subexpressão, dando: $\lambda x. \$G_1 (\text{sqrt } x)$

...Modificando o Algoritmo de *lambda-lifting*...

- Completando a compilação, tem-se:

$$\begin{aligned} \$G_1 \text{ sqrtx } y &= + y \text{ sqrtx} \\ \$G x &= \$G_1 (\text{sqrt } x) \\ \$F &= \$G 4 \\ \$Prog &= + (\$F 1) (\$F 2) \end{aligned}$$

$$\$Prog$$

- Chega-se ao final com um supercombinador a mais, pois perde-se a oportunidade de uma η -redução quando se torna $(\text{sqrt } x)$ um parâmetro extra.
- Em compensação a execução será *fully-lazy*.

...Modificando o Algoritmo de *lambda-lifting*

- Uma ligeira otimização pode partir de que uma ELM não tem nenhuma variável livre (então é uma CAF).
- Aí, em vez de abstraí-la como um parâmetro extra, pode-se simplesmente dar-lhe um nome e transformá-la num supercombinador.

Lambda-lifting Totalmente lazy e letrecs...

- Considere o seguinte programa:

$$\begin{aligned} \text{let } f &= \lambda x. \text{letrec } \text{fac} = \lambda n. (\dots) \text{ in } + x (\text{fac } 1000) \\ \text{in } &+ (f 3) (f 4) \end{aligned}$$

- O algoritmo do capítulo anterior compilaria:

$$\begin{aligned} \$\text{fac } \text{fac } n &= (\dots) \\ \$f x &= \text{letrec } \text{fac} = \$\text{fac } \text{fac } \text{ in } + x (\text{fac } 1000) \end{aligned}$$

$$+ (\$f 3) (\$f 4)$$

- A função fac é definida localmente no corpo de f e $(\text{fac } 1000)$ não pode ser abstraída do corpo de f como uma expressão livre.
- Assim, $(\text{fac } 1000)$ será recomputada e perder-se-á *full laziness*.

...Lambda-lifting Totalmente lazy e letrecs...

- A solução é reconhecer que `fac` não depende de `x` e aí colocar `fac` para fora, gerando:

```
letrec fac = λn.(...)
in let f = λx.+ x (fac 1000)
   in + (f 3) (f 4)
```

- Agora sim, tem-se um programa *fully-lazy*:

```
$fac n    = (...)
$fac1000 = $fac 1000
$f x     = + x $fac1000
$Prog    = + ($f 3) ($f 4)
```

```
$Prog
```

...Lambda-lifting Totalmente lazy e letrecs...

- Esse exemplo mostra a utilidade de transformar uma expressão constante máxima num supercombinador (`fac1000`)
- A estratégia divide-se em duas fases:
 1. tornar definições `let` (e `letrec`) o mais externas possível,
 2. executar o *lambda-lifting totalmente lazy*
- Quanto longe um `letrec` pode ser lançado?
- O valor de uma variável ligada num `letrec` geralmente dependerá de valores de certas variáveis livres.

...Lambda-lifting Totalmente lazy e letrecs

- Chama-se de o conjunto de variáveis livres de `x`, aquele conjunto de variáveis livres das quais `x` depende.
- Uma vez que se conhece esse conjunto, pode-se lançar a definição de `x` até a próxima (primeira de dentro para fora) abstração `lambda` envolvente, que liga uma das variáveis do conjunto.
- Esse passo traz a vantagem de que definições que não tem nenhuma variável livre sejam transformadas diretamente em supercombinadores.

Um Exemplo...

- O programa Miranda M

```
sumInts n      = fold (+) 0 (count 1 n)
count n m     = [], n > m
count n m     = n : count(n + 1) m
fold op base [] = base
fold op base (x : xs) = fold op (op base x) xs
```

```
sumInts 100
```

- Transforma-se em M1:

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NIL CONS n (count (n + 1) m)
      fold = λop.λbase.λxs.
              IF(= xs NIL) base(fold op (op base(HEAD xs))(TAIL xs))
in sumInts 100
```

...Um Exemplo...

• O programa M1

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NILCONS n (count(n + 1) m))
      fold = λop.λbase.λxs. IF(= xs NIL) base
              (fold op (op base(HEAD xs)) (TAIL xs))
in sumInts 100
```

• Transforma-se em M2:

```
$R1 p q base xs = IF(= xs NIL) base(p (q (HEAD xs)) (TAIL xs))
```

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NIL CONS n (count(n + 1) m))
      fold = λop.λbase. $R1(fold op) (op base) base
in sumInts 100
```

...Um Exemplo ...

• O programa M2

```
$R1 p q base xs = IF(= xs NIL) base(p (q (HEAD xs)) (TAIL xs))
```

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NIL (CONS n (count(n + 1) m))
      fold = λop.λbase. $R1(fold op) (op base) base
in sumInts 100
```

• Transforma-se em M3:

```
$R1 p q base xs = IF(= xs NIL) base(p (q (HEAD xs)) (TAIL xs))
$R2 r op base = r (op base) base
```

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NIL (CONS n (count(n + 1) m))
      fold = λop. $R2($R1(fold op)) op
in sumInts 100
```

...Um Exemplo ...

• O programa M3

```
$R1 p q base xs = IF(= xs NIL) base(p (q (HEAD xs)) (TAIL xs))
$R2 r op base = r (op base) base
```

```
letrec sumInts = λn. fold (+) 0 (count 1 n)
      count = λn.λm. IF(>n m) NIL (CONS n (count(+ n 1) m))
      fold = λop. $R2($R1(fold op)) op
in sumInts 100
```

...Um Exemplo ...

• Transforma-se finalmente em M4:

```
$sumInts n = $foldPlus0 ($count1 n)
$foldPlus0 = $fold + 0
$count1 = $count 1
$count n m = IF(>n m) NIL (CONS n ($count (+ n 1) m))
$fold op = $R2 ($R1 ($fold op)) op
$prog = $sumInts 100
$R1 p q base xs = IF(= xs NIL) base (p(q(HEAD xs))(TAIL xs))
$R2 r op base = r (op base) base
```

```
$prog
```

...Um Exemplo ...

• Assim, o programa Miranda M

```

sumInts          = fold (+) 0 (count 1 n)
count n m        = [ ], n > m
count n m        = n : count (n + 1) m
fold op base []  = base
fold op base (x : xs) = fold op (op base x) xs

```

```
sumInts 100
```

...Um Exemplo

• Foi transformado em M4:

```

$sumInts n = $foldPlus0 ($count1 n)
$foldPlus0 = $fold + 0
$count1     = $count 1
$count n m = IF(> n m) NIL(CONS n ($count (+ n 1) m))
$fold op   = $R2 ($R1 ($fold op)) op
$prog      = $sumInts 100
$R1 p q base xs = IF(= xs NIL) base (p(q(HEAD xs))(TAIL xs))
$R2 r op base   = r (op base) base

```

```
$prog
```

Implementação do *lambda-lifting* Totalmente lazy...

- Para se identificar as ELM's, usa-se o conceito do nível léxico introduzido anteriormente.
- O nível léxico de uma expressão deve ser o maior nível das variáveis livres dentro da expressão.
- Quando se fizer o *lambda-lifting* de uma abstração lambda no nível n , deve-se abstrair como parâmetro extra qualquer subexpressão do corpo, cujo nível seja menor que n .

...Implementação do *lambda-lifting* Totalmente lazy...

- Seja a abstração lambda para G no nosso exemplo anterior:

$$\lambda x. \lambda y. + y (\text{sqrt } x)$$

- A abstração λx tem nível 1, λy tem nível 2
- A subexpressão $+$ tem nível 0, $(+ y)$ tem nível 2, sqrt tem nível 0, $(\text{sqrt } x)$ tem nível 1 e $(+ y (\text{sqrt } x))$ tem nível 2
- Dada uma expressão E , sua abstração lambda nativa é a abstração lambda envolvente cujo nível léxico é o mesmo de E .
- Ou ainda, é a primeira abstração lambda (de dentro para fora) que liga qualquer variável de E .

...Implementando o *lambda-lifting* Totalmente

- Toda informação sobre as ELM's e o *lambda-lifting* pode ser obtida numa única pesquisa (caminhamento) em árvore sobre a expressão:
 1. caminhar para baixo, guardando o nível de cada abstração
 2. caminhando para cima o nível de cada expressão é calculado, usando o ambiente e os níveis das suas subexpressões
 3. Se aplicada a outra expressão de mesmo nível, então as duas são unidas. Caso contrário, são dadas a elas nomes distintos.
 4. A união é o mecanismo onde expressões livres são combinadas para formar expressões livres máximas.
 5. Lambda encontrado no caminho para cima é transformado num supercombinador, e a abstração lambda é trocada pelo supercombinador aplicado às ELM's.
 6. As ELM's são aquelas subexpressões com nível menor que o da abstração lambda, depois que a união se suceder.

Lifting CAFs...

- As expressões constantes máximas (nível 0) precisam de um tratamento especial.
- Seria correto abstraí-las como parâmetro extra, mas existe uma maneira mais fácil.
- Define-se um novo supercombinador sem argumentos para substituir a expressão constante.
- Contudo, nenhum benefício será conseguido se a expressão consistir de uma única constante.

...Lifting CAFs

- Por exemplo, na expressão,

$$\lambda x. + 1 x$$

onde $(+ 1)$ é uma expressão livre máxima no nível 0, e pode ser transformada num supercombinador $\$Inc$, tal que

$$\$Inc = + 1$$

- A expressão original fica: $\lambda x. \$Inc x$
- Nesse caso, o que se conseguiu foi o compartilhamento do grafo $(+ 1)$ para cada aplicação da abstração lambda.

Ordenando os Parâmetros...

- A Inclusão de parâmetros de um supercombinador em ordem crescente de seus níveis léxicos cria oportunidades de η -redução
- A mesma ordenação é útil para as ELM's por dois motivos:
 1. a possibilidade de η -redução
 2. outro seria maximizar o tamanho de uma ELM

...Ordenando os Parâmetros

- Suponha a seguinte abstração:
 $\lambda x. (...G \dots F \dots E \dots)$
 onde E , F e G são ELM's de λx e
 o nível de $F < \text{nível de } G < \text{nível de } E$
- Seria melhor definir o supercombinador:
 $\$S f g e x = (...g \dots f \dots e \dots)$
- E substituir a abstração por
 $\$S F G E$
 porque aí $\$F G$ terá um nível menor que E e assim será tirado da abstração lambda nativa de E , como uma ELM única

Lançando para Fora os *lets* e *letrecs*...

- Supõe-se que a análise de dependência já foi feita antes de se lançar para fora todas as definições de um letrec
- Pela mesma razão, suponha que os *lets* tenham uma única definição
- Pode-se descrever o algoritmo da seguinte forma:
 Trabalhando de dentro para fora, para cada $let(rec)$ faça os seguintes passos:
 1. calcule o nível léxico de cada corpo de definição.
 - Enquanto se estiver fazendo isso para um letrec, assume-se que o nível léxico das variáveis definidas no letrec seja zero
 - A razão para isso é que o nível léxico de uma definição recursiva dependa somente de suas variáveis livres

...Lançando para Fora os *lets* e *letrecs*

2. Para cada letrec, calcule o nível léxico máximo dos corpos das definições. Esse é o nível das variáveis ligadas no letrec
 3. Para os *lets*, os níveis corretos são os calculados no primeiro passo
 4. Lance para fora as definições até que a próxima abstração lambda envolvente tenha o mesmo nível que o das variáveis definidas no letrec, o qual foi calculado no segundo passo
 5. Finalmente, se o $let(rec)$ aparece numa posição de função em uma aplicação, então continue lançando-o para fora até que isso não mais aconteça
- Se um letrec liga uma variável que já está no escopo, então ele não pode ser lançado para fora sem o risco de conflito de variável. A solução é mudar sistematicamente os nomes das variáveis.

Eliminando *full laziness* Redundante

- As transformações para conseguir *full laziness* têm um preço.
- Pode-se pagá-lo das seguintes maneiras:
 - São gerados supercombinadores com vários argumentos (para todas as ELM's)
 - Isso aumenta o tamanho do *redex* e torna a execução mais lenta
 - Podem ser gerados mais supercombinadores por causa da perda de oportunidades para η -otimizações
 - O programa é quebrado em pequenos fragmentos, fragmentos dos corpos das funções sendo exportados em partes
 - Isso pode impedir uma série de otimizações no código compilado.
- Poder-se-ia melhorar a transformação fazendo *lambda-lifting* comum (ao invés de totalmente *lazy*), seletivamente, onde não se teria ganhos com métodos totalmente *lazy*.

Funções Aplicadas a Poucos Argumentos

- Seja a seguinte abstração lambda:
 $\lambda v. \lambda x. \text{IF}(= v 0) (+ x 1) (+ x 2)$
- Um *lambda-lifter* totalmente *lazy* produziria dois combinadores:
 $\$S1 \text{ IF-v-zero } x = \text{IF-v-zero } (+ x 1) (+ x 2)$
 $\$S v = \$S1 (\text{IF } (= v 0))$
- E substituiria a abstração λv com $\$S$.
- Contudo IF requer 3 argumentos para reduzir, e aí, nenhum trabalho será economizado.
- Conclui-se que nenhum trabalho é economizado abstraindo expressões que consistem em um operador *built-in* ou supercombinador aplicados a poucos argumentos.
- O exemplo mostra que os argumentos da função devem ser considerados para abstração.

Abstrações Lambda não Compartilhadas

- Suponha que a abstração lambda em questão aparecesse num contexto como o seguinte:
 $\text{let } f = \lambda v. \lambda x. \text{IF}(= v 0) (+ x 1) (+ x 2)$
 $\text{in } \dots (f 4 5) \dots$
- E suponha que $(f 4 5)$ seja o único uso de f .
- Nesse caso, a aplicação parcial $(f 4)$ não pode ser compartilhada, já que é usada imediatamente.
- Desse exemplo pode-se derivar a regra geral:
Dada uma λ -abstração $\lambda x. E$ em um contexto em que não pode ser compartilhada, não se deve abstrair as expressões livres de E , porque elas não serão compartilhadas.
- Isto é, deve-se abstrair somente as variáveis livres.

Abstrações Lambda não Compartilhadas

- Pode-se justificar essa regra observando que expressões livres abstraídas de E não podem ser compartilhadas porque:
 - elas não são compartilhadas dentro de E , já que são abstraídas de um único lugar de E ;
 - elas não são compartilhadas fora de E , pois a abstração total, $\lambda x. E$, não é compartilhada.
- Pela primeira vez que a estratégia de *lambda-lifting* se torna dependente de contexto
- Descobrir informação sobre compartilhamento é um problema potencialmente complexo e maiores informações podem ser encontradas em [Fairbain, 1985] e [Hudak, 1985]

II. COMBINADORES SK

II. GERÊNCIA DE MEMÓRIA E COLETA DE LIXO

PARTE III

REDUÇÃO DE GRAFOS AVANÇADA

III. A MÁQUINA-G

(COLABORAÇÃO DE MARCELO DE ALMEIDA MAIA)

III. Máquina-G

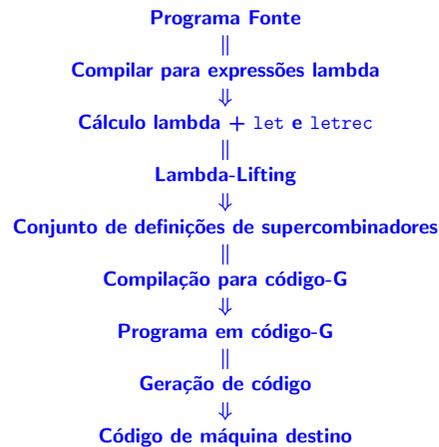
ILF

Máquina G

- Ponto central de qualquer redutor de grafos é a implementação da aplicação de função.
- Programas são transformados em um conjunto de supercombinadores, com o objetivo de compilá-los para uma sequência fixa de instruções.
- A máquina-G (Johnson e Augustsson) implementa essa compilação.
- Pode-se compilar supercombinadores para uma linguagem de máquina específica.
- É melhor compilá-los para uma linguagem intermediária, o código-G.
- As vantagens são:
 - ganho de portabilidade
 - independência de otimização para um conjunto específico de instruções.

O Compilador para Máquina-G

- A estrutura do compilador pode ser a seguinte:



Outras Implementações de Linguagens *lazy*

- Ponder [Fairbairn, 1982] propõe uma abordagem similar à máquina-G, embora desenvolvida independentemente.
- Outra abordagem é usar T [Rees e Adams, 1982] ou Scheme [Steele e Sussman, 1978], como linguagem intermediária.
 - Isso tem a vantagem de tirar proveito do imenso esforço gasto em contruir implementações eficientes de LISP.
 - Essa é a abordagem de [Hudak e Kranz, 1984].
- A implementação de Hope num VAX é baseada no código intermediário o FP/M.
 - N.B.: Hope é uma linguagem estrita.

Exemplo de Execução da Máquina-G...

- Seja o seguinte programa Miranda:

```

$from n = n : $from ($succn)
$succ n = + n 1
-----
$from ($succ 0)
  
```

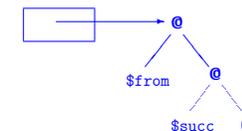
- Gerando o conjunto de supercombinadores temos:

```

$from n = CONS n ($from ($succ n))
$succ n = + n 1
$Prog   = $from ($succ 0)
-----
$Prog
  
```

...Exemplo de Execução da Máquina-G

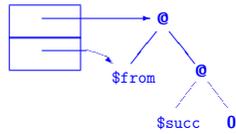
- A máquina-G usa uma pilha, e a execução começa com o topo da pilha tendo um ponteiro para o grafo inicial.



- Então é feita uma descida (*unwind*) na espinha, usando pilha, sem fazer inversão de ponteiros.

Exemplo de Execução da Máquina-G ...

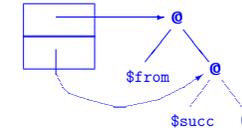
- Após a descida, temos:



- Seguindo o ponteiro do \$from a máquina-G extrai:
 1. o número de argumentos esperado,
 2. o endereço inicial do código de \$from.
- Verifica se existe na pilha argumentos suficientes para \$from.

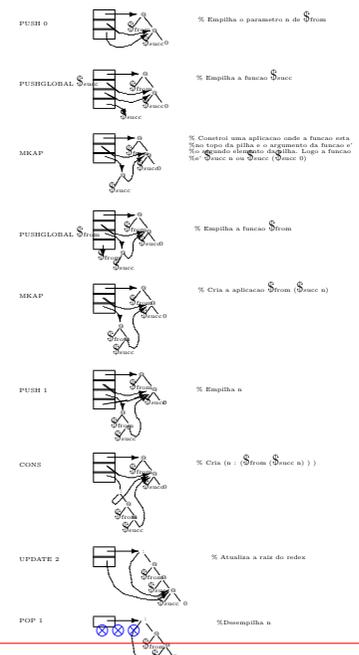
...Exemplo de Execução da Máquina-G ...

- Depois, o ponteiro para o topo da pilha é reajustado, para deixar o argumento de \$from no topo da pilha.



- Agora, faz-se um salto (*jump*) para o código de \$from.

Avaliação de \$from (\$succ 0)



Sequência de Código de \$from n...

- Na entrada:
 - parâmetro n está no topo da pilha
 - ponteiro para a raiz do *redex* está imediatamente abaixo
- Itens que não estão no topo da pilha são endereçados relativamente ao topo da pilha.
- Topo da pilha tem *offset* zero.
- Por exemplo, PUSH 1 acessa o segundo elemento da pilha (aquele logo após o topo) e empilha seu conteúdo.

...Sequência de Código de \$from n

- Algumas instruções pegam seus operandos na própria pilha e colocam os resultados de volta nela.
- É o caso de MKAP e CONS.
- UPDATE 2 atualiza a raiz do *redex* (*offset* 2 para cima) com uma cópia da raiz do resultado (topo).
- POP 1 remove um parâmetro da pilha deixando no topo um ponteiro para o grafo resultante.
- UNWIND examina o *tag* da raiz — nesse caso é uma célula CONS — indicando que a avaliação está completa.

A Linguagem Fonte para o Compilador-G...

- Compilação para o código-G começa com um programa formado por definições de supercombinadores da forma:

$\$S x_1 \dots x_n = \langle E \rangle$

onde x_i são variáveis, e $\langle E \rangle$ tem a seguinte forma:

```

<E> ::= <constante>
      | <ident>
      | <E> <E>
      | let <ident> = <E> in <E>
      | letrec {<ident> = <E>} in <E>
  
```

...A Linguagem Fonte para o Compilador-G

- Funções *built-in*:

Conjunto Tratado	Funções Similares
constantes inteiras	booleanas e caracteres
NEG(negação)	NOT
+	-, *, /, REM <, <=, =, >=, >
IF	Case-n
Fatbar	
CONS	Pack-Sum-d-r, Pack-Prod-r
HEAD	TAIL, Sel-r-i, Sel-SUM-r-i

Compilação para o Código-G

- Compilação de definições de supercombinadores para o código-G e sua execução pela máquina-G são otimizações à implementação do procedimento de instanciação.
- O programa em código-G consiste em:
 - um segmento de código de inicialização, que irá fazer qualquer inicialização em tempo de execução que se faça necessária;
 - um segmento de código-G que avalia a supercombinador especial \$Prog e imprime seu valor.
 - um segmento de código-G correspondendo a cada definição de supercombinador. Cada um desses segmentos será identificado por um rótulo;
 - segmentos de código-G, identificados por um rótulo, correspondendo a cada função *built-in*

...Compilação para o Código-G

- O segmento de código para o primeiro item é uma instrução BEGIN do código-G que rotula o início do programa, e inicializa o que for necessário.
- Para o segundo item devemos empilhar o supercombinador \$Prog, usando a instrução PUSHGLOBAL do código-G.
- Daí, avaliamos \$Prog, usando a instrução EVAL.
- Finalmente, imprimimos o resultado usando a instrução PRINT.
- A sequência que poderia ser gerada é:


```
BEGIN;           % Início do programa
PUSHGLOBAL $Prog; % Empilha $Prog
EVAL;           % Avalia $Prog
PRINT;          % Imprime o resultado
END;           % Fim do programa
```

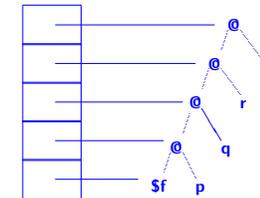
Compilação de Supercombinador

- Função F: pega a definição de um supercombinador como argumento e retorna o código-G compilado como resultado.
- F terá a seguinte forma:

$$F[\$F \ x_1 \ \dots \ x_n = E] = \dots \text{Código-G para } \$F \ \dots$$
- Função F é um esquema de compilação.
- Outros esquemas de compilação (R, C) serão definidos como funções auxiliares de F.
- Execução de \$F pressupõe certas configurações da pilha e certo contexto.

Pilhas e Contextos

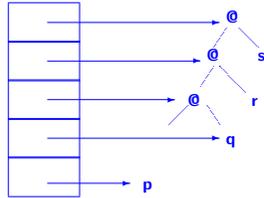
- Suponha que a máquina-G avaliando a expressão ($\$F \ p \ q \ r \ s$), onde \$F tem dois argumentos.
- Após a descida na espinha do grafo (*unwind*), a pilha estaria da seguinte forma:



- Essa configuração de pilha não é conveniente, porque para acessar os argumentos p e q é necessário acesso indireto pelas vértebras.

...Pilhas e Contextos ...

- A solução é reorganizar a pilha depois da descida e antes da execução de \$F\$, da seguinte maneira:



- O resto da espinha não foi desenhado, mas ainda continua lá.

...Pilhas e Contextos ...

- Existem três regras que devem valer:

1. Início da execução de um supercombinador: os argumentos estarão no topo da pilha e abaixo deles está um ponteiro para a raiz do *redex*.
2. Fim da execução de um supercombinador: ponteiro para o grafo reduzido permanece na pilha.
3. Grafo reduzido não precisa estar na WHNF: última instrução no supercombinador deve iniciar a próxima redução.

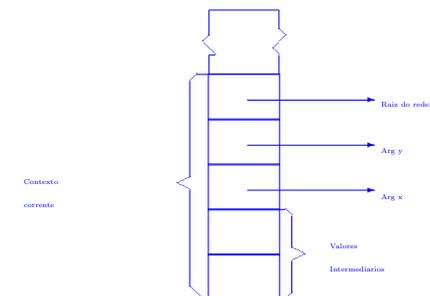
...Pilhas e Contextos ...

- O endereçamento de variáveis é relativo ao topo da pilha.
- Durante a compilação deve-se ter:
 1. ρ uma função que toma um identificador e retorna um número dando o *offset* do argumento correspondente a partir da base do contexto corrente (raiz do *redex*).
 2. raiz do *redex* tem *offset* 0
 3. último argumento tem *offset* 1.
 4. d , a profundidade do contexto corrente menos um.
- Para calcular o *offset* de uma variável a partir do topo da pilha:

$$d - \rho x,$$
 onde o topo da pilha terá *offset* 0.

...Pilhas e Contextos...

- Por exemplo, seja o seguinte contexto:



...Pilhas e Contextos

- A profundidade do contexto é 5, logo $d = 4$.
- A função ρ mapeia $x = 2$ e $y = 1$.
- Os *offsets* respectivos de x e y em relação ao topo são:

$$d - (\rho x) = 4 - 2 = 2$$

$$d - (\rho y) = 4 - 1 = 3$$

O Esquema F de Compilação

- A definição completa do esquema F de compilação é:

$$F[f \ x_1 \ \dots \ x_n = E] =$$

$$\text{GLOBSTART } f, n;$$

$$R[E] \left[\underbrace{x_1 = n, \dots, x_n = 1}_{\rho} \right] \frac{n}{d}$$

onde

- f é um supercombinador,
- GLOBSTART f, n é uma pseudo-instrução do código-G que rotula o início de uma função f que leva n argumentos.
- F chama R para compilar o corpo do supercombinador E , passando ρ e d para R.

O Esquema R de Compilação

- Código para o corpo do supercombinador deve fazer 4 coisas:
 1. construir uma instância do corpo do supercombinador usando os parâmetros da pilha,
 2. atualizar a raiz do *redex* com uma cópia do resultado,
 3. remover os parâmetros da pilha,
 4. iniciar a próxima redução.
- Isso pode traduzir diretamente num esquema de compilação para R:

$$R[E] \rho d = C[E] \rho d; \text{ UPDATE } (d + 1); \text{ POP } d; \text{ UNWIND};$$
- A nova função auxiliar C construir instância.
- Atenção: Embora R produza resultados corretos, gerará código com má performance para funções projetivas, por exemplo, $f \ x \ y \ z = y$. No capítulo sobre otimização da máquina-G veremos como melhorar esse aspecto.

O Esquema C de Compilação

- O esquema C de compilação gera código para construir uma instância de uma expressão. É uma função com o seguinte comportamento:
 - argumentos: a expressão a ser compilada, mais ρ e d , que especificam onde os argumentos do supercombinador serão encontrados na pilha;
 - resultado: uma sequência de código-G que quando executada irá construir uma instância da expressão com ponteiros para argumentos do supercombinador e deixar um ponteiro para a instância no topo da pilha.
- Para definir C que é do tipo $C[E] \rho d$ devemos escrever em termos de cada forma possível para E : constante, variável, aplicação, expressão let ou expressão letrec.

Compilação de Constante

- E é uma constante.
- Primeiro, suponha que E seja um inteiro, booleano ou outro valor *built-in* constante.
 - Deve-se empilhar um ponteiro para o inteiro i na pilha (ou o próprio inteiro numa implementação *unboxed*).
 - A regra de compilação é:

$$C[[i]]\rho d = \text{PUSHINT } i$$
- Segundo, suponha que E seja um supercombinador ou função *built-in*, chamada f .
 - Então deveremos empilhar um ponteiro para a função f .
 - A regra de compilação será:

$$C[[f]]\rho d = \text{PUSHGLOBAL } f.$$

Compilação de Variável

- E é uma variável x .
- O valor da variável x está na pilha no *offset*

$$(d - \rho x)$$
 a partir do topo.
- Podemos escrever a seguinte regra:

$$C[[x]]\rho d = \text{PUSH } (d - \rho x).$$

Compilação de Aplicações

- Seja E uma aplicação $(E_1 E_2)$ onde E_1 e E_2 são expressões arbitrárias.
- Primeiro, devemos construir uma instância de E_2 ($C[[E_2]]$) deixando um ponteiro para a instância no topo da pilha.
- Depois construímos uma célula de aplicação com os dois itens do topo da pilha, que serão desempilhados.
- Finalmente, será empilhado um ponteiro para o nó de aplicação.
- A regra ficará:

$$C[[E_1 E_2]]\rho d = C[[E_2]]\rho d; C[[E_1]]\rho (d + 1); \text{MKAP};$$

Compilação de let...

- Considere a seguinte expressão `let`:

$$C[[\text{let } x = E_x \text{ in } E_b]]\rho d.$$
- Um `let` num supercombinador é apenas um grafo.
- Podemos implementar o `let` da seguinte maneira:
 1. construímos uma instância de E_x , deixando um ponteiro para ela no topo da pilha;
 2. estendemos ρ , para que ela informe que x estará no *offset* $d + 1$ a partir da base do contexto;
 3. construímos uma instância de E_b , usando os novos valores de ρ e d , deixando um ponteiro para a instância no topo da pilha.
 4. agora temos no topo da pilha um ponteiro para a instância E_b e abaixo dele um ponteiro para a instância E_x .

...Compilação de let

- Ponteiro para E_b (topo) deve sobrepor o ponteiro para E_x , com a instrução SLIDE.

- Finalmente, teremos:

$$C[\text{let } x = E_x \text{ in } E_b] \rho d = C[E_x] \rho d;$$

$$C[E_b] \rho [x = d + 1] (d + 1);$$

SLIDE 1;

Compilação de letrec...

- Considere a seguinte expressão letrec:

$$C[\text{letrec } D \text{ in } E_b] \rho d,$$

onde D é um conjunto de definições.

- Um letrec num supercombinador é apenas a descrição de um grafo cíclico.
- Obs: Durante a instanciação, ocorrências de nomes ligados no letrec são substituídos por ponteiros para a célula vazia correspondente.

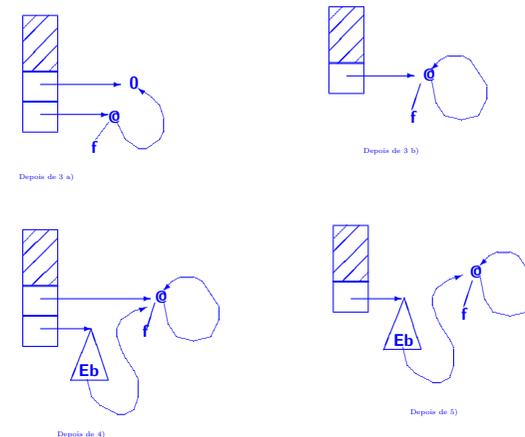
...Compilação de letrec...

- Podemos construir o grafo da seguinte maneira:

1. alocar uma célula vazia para cada definição, colocando os ponteiros para elas na pilha;
2. estender o contexto ρ e d para dizer que as variáveis ligadas pelo letrec podem ser encontradas na pilha.
3. para cada corpo da definição:
 - 3.1. construir uma instância para ele, deixando um ponteiro para a instância no topo da pilha,
 - 3.2. atualizar a célula vazia correspondente com a instância.
4. instanciar E_b , deixando um ponteiro para ele no topo,
5. eliminar os ponteiros para os corpos das definições (função SLIDE).

...Compilação de letrec...

- Seja por exemplo a execução de letrec $x = f x \text{ in } E_b$



...Compilação de letrec...

- A regra para compilação do letrec fica:

$$C[\text{letrec } D \text{ in } E_b] \rho d = \text{Clrec}[D] \rho' d'; C[E_b] \rho' d'; \\ \text{SLIDE } (d' - d)$$

onde

$$- (\rho', d') = \text{Xr}[D] \rho d$$

- O SLIDE($d' - d$) desce o topo da pilha retirando os ponteiros para E_i .

...Compilação de letrec...

$$\bullet \text{Clrec} \left\{ \begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \dots \\ x_n = E_n \end{array} \right\} \rho d = \begin{array}{l} \text{ALLOC } n; \\ C[E_1] \rho d; \text{ UPDATE } n; \\ C[E_2] \rho d; \text{ UPDATE } n - 1; \\ \dots \\ C[E_n] \rho d; \text{ UPDATE } 1; \end{array}$$

$$\bullet \text{Xr} \left\{ \begin{array}{l} x_1 = E_1 \\ x_2 = E_2 \\ \dots \\ x_n = E_n \end{array} \right\} \rho d = (\rho[x_1 = d + 1, \dots, x_n = d + n], d + n)$$

- Xr apenas calcula o ρ estendido e o novo valor de d , retornando-os como o par (ρ', d') .

...Compilação de letrec

- Existirá um problema se o corpo da definição consistir de somente um nome de variável, ligada no mesmo letrec.
- Por exemplo,


```
letrec x = y
      y = CONS 1 y
in E
```
- Isso nos dá um problema quando UPDATE tentar atualizar uma célula vazia com outra.
- Contudo, esse problema será eliminado utilizando a otimização de substituir x por y em E .

Supercombinadores sem Argumentos

- O *lambda-lifting* pode produzir supercombinadores sem argumentos, por exemplo, o \$Prog.
- Tais supercombinadores são expressões constantes (as CAFs vistas anteriormente).
- A presença de CAFs leva a duas questões:
 - compilação
 - coleta de lixo.
- Existem 2 alternativas para compilar CAFs.

Compilação de CAFs: Alternativa I

- Não compilar as *CAF*s.
- Ao invés, deixamo-las como pedaços do grafo.
- Note que, porque não funções, nunca serão copiadas, podendo facilmente ser compartilhadas.
- Entretanto, o programa destino será uma mistura de código de máquina e grafo.

Compilação de CAFs: Alternativa II

- Tratar as *CAF*s como supercombinadores de zero argumento e compilá-las para o código-G.
- Para isso, alocamos um único nodo, rotulado como uma função, que tem um ponteiro para o código compilado.
- Esse nodo é compartilhado por qualquer um que use o supercombinador.
- O esquema F compilará adequadamente o código para o corpo.
- A vantagem dessa abordagem é que o programa compilado constará quase que somente de código de máquina e apenas alguns nodos, um por supercombinador.

Coleta de Lixo das CAFs...

- Supercombinadores que têm um ou mais argumentos não precisam ser coletados pois seus tamanhos são fixos.
- CAFs podem crescer em tamanho. Por exemplo,


```
$from n = CONS n ($from (+ n 1))
$Ints   = $from 1
...

```

 onde `$Ints` é uma lista infinita de inteiros.
- A coleta associa a cada supercombinador (de zero ou mais argumentos) uma lista de CAFs às quais ele se refere, direta ou indiretamente.

...Coleta de Lixo das CAFs

- Na coleta de lixo, quando se marca um supercombinador de um ou mais argumentos, marca-se também as CAFs da lista de CAFs associadas a ele.
- Quando se marca uma CAF não reduzida, também marca-se na lista de CAFs associadas a ela.
- E por fim, uma CAF já reduzida é marcada como usual, pois ela é indistinguível de outra estrutura do *heap*.

Ajuntando os Pedacos...

- Considere a compilação do seguinte programa:

```
$F x = NEG x
$Prog = $F 3
```

```
$Prog
```

- Um possível código-G para \$Prog:

```
BEGIN;           % Começo do programa
PUSHGLOBAL $Prog % Carrega $Prog
EVAL;            % Avalia
PRINT;          % Imprime
```

....Ajuntando os Pedacos...

- Um possível código-G para $\$F x = NEG x$:

```
GLOBSTART $F, 1; % Começo de $F
PUSH 0;          % Empilha x
PUSHGLOBAL NEG; % Empilha NEG
MKAP;           % Constrói (NEG x)
UPDATE 2;       % Atualiza raiz redex
POP 1;          % Desemp. parâmetro
UNWIND;         % Continua a avaliação
```

....Ajuntando os Pedacos

- Um possível código para $\$Prog = \$F 3$:

```
GLOBSTART $Prog, 0; % Começo de $Prog
PUSHINT 3;          % Empilha 3
PUSHGLOBAL $F;     % Empilha $F
MKAP;              % Constrói ($F 3)
UPDATE 1;          % Atualiza o $Prog
UNWIND;            % Continua a avaliação
```

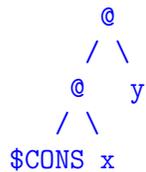
Funções Built-in...

- Funções *built-in* aparecem na implementação sob três formas.
- Por exemplo, CONS pode aparecer como:
 1. uma função *built-in* num programa de supercombinadores. Por exemplo,


```
$S x y = CONS y x;
```
 2. uma instrução do código-G, que toma os dois elementos do topo da pilha, forma uma célula CONS a partir deles e coloca um ponteiro para o resultado no topo da pilha;
 3. uma função *built-in* de tempo de execução.

...Funções *Built-in*

- Em tempo de execução a máquina pode ter que avaliar um grafo como este:



- Isso significa que deve haver uma sequência de código-G para a função \$CONS.
- Em tempo de execução, \$CONS aparece apenas como um supercombinador definido pelo usuário.

- Daqui para frente, não faremos distinção entre funções *built-in* e supercombinadores.
- *built-ins* são chamados de globais.
- Essa é a origem da instrução PUSHGLOBAL.

\$NEG, \$+ e EVAL...

- NEGate é uma função que precisa avaliar seu argumento.
- Para isto há a instrução EVAL, que avalia o item no topo da pilha, deixando o resultado no próprio topo.
- O código para \$NEG é:

```

EVAL;      % Avalia o argumento
NEG;       % Negativa-o
UPDATE 1;  % Atualiza a raiz do redex
UNWIND;    % Continua

```

...\$NEG, \$+ e EVAL

- O código para \$+ é similar:

```

PUSH 1;    % Toma o segundo argumento
EVAL;      % Avalia-o
PUSH 1     % Toma o primeiro argumento
EVAL;      % Avalia-o
ADD;       % Soma os argumentos
UPDATE 3;  % Atualiza a raiz do redex
POP 2;     % Desempilha os parâmetros
UNWIND;    % Continua

```

Instrução EVAL...

- A instrução EVAL faz o seguinte:
 1. Examina o objeto no topo da pilha.
 2. Se for uma célula CONS, um inteiro (booleano, caracter), um supercombinador ou uma função *built-in*, EVAL nada faz.
 3. Se for uma célula de aplicação, EVAL
 - cria uma nova pilha,
 - empilha o topo da pilha antiga,
 - salva o contador de programa corrente (que aponta para a instrução após EVAL),
 - executa um UNWIND.

...Instrução EVAL...

- Depois de cada redução, é executado um UNWIND.
- Se este UNWIND descobre que a expressão está na WHNF, ele recupera a pilha antiga e salta para o endereço salvo.
- A nova pilha pode ser construída em cima do topo da pilha antiga.
- Além disso, elas podem sobrepor um item, pois o topo da pilha antiga é a base da pilha nova.

...Instrução EVAL...

- Em outra pilha (dump), salvam-se duas informações:
 1. a profundidade da pilha antiga, ou o ponteiro da pilha antiga;
 2. o contador de programa antigo.
- No caso de \$NEG e \$+, o UNWIND vai sempre encontrar que a avaliação está completa, pois sabe-se de antemão que o resultado será um inteiro.
- Logo, para evitar o teste, podemos usar um RETURN ao invés de UNWIND.
- O RETURN, como o UNWIND, apenas recupera a pilha antiga e salta para o contador de programa antigo.

...Instrução EVAL

- O novo código para \$NEG seria:

```

EVAL;      % Avalia o argumento
NEG;       % Negativa-o
UPDATE 1;  % Atualiza a raiz do redex
RETURN;    % A avaliação está completa
  
```

\$CONS

- Quando o código para \$CONS é invocado, supõe-se que seus dois argumentos estão no topo da pilha, e abaixo deles um ponteiro para a raiz do *redex*.
- Assim, tem-se a seguinte sequência para \$CONS:


```

CONS;      % Forma uma célula CONS
UPDATE 1;  % Atualiza a raiz do redex
RETURN;    % Resultado está na WHNF
      
```
- O tratamento de Pack-Sum-d-r é similar, exceto que precisa-se de uma nova instrução de código-G, Pack-Sum-d-r, que construa um dado estruturado de *tagd* e *r* campos, cujos valores serão encontrados na pilha.
- CONS é equivalente a Pack-Sum-2-2
- Pack-Prod-r pode ser tratada similarmente criando-se Pack-Prod-r.

HEAD...

- HEAD é uma função que avalia seu argumento (para a WHNF).
- O argumento deve ser uma célula CONS, da qual podemos extrair a cabeça.
- O código para HEAD é:


```
EVAL;    % Avalia o argumento para WHNF
HEAD;    % Toma a cabeça
EVAL;    % Avalia a cabeça
UPDATE 1; % Atualiza a raiz do redex
UNWIND;  % Continua
```
- Note que não usamos RETURN, mesmo que a cabeça esteja na WHNF. Considere o exemplo, (\$HEAD E) 3
- A avaliação da expressão inteira não estaria completa só porque o resultado de \$HEAD E está na WHNF.

...HEADsy

- TAIL e Sel-sumr-i são análogas a HEAD. É necessário uma instrução SELSUM $r\ i$ para selecionar o i -ésimo componente do dado estruturado do tipo soma com tamanho r .
- Similarmente, Sel-r-i requer a introdução da instrução SELPRODUCT $r\ i$.

Instrução \$IF

- Para gerar código para \$IF precisa-se de duas instruções de salto (JUMP e JFALSE) e uma pseudo-instrução de rótulo LABEL:


```
PUSH 0;    % Toma o primeiro argumento
EVAL;      % Avalia-o
JFALSE L1; % Salta para L1 se falso
PUSH 1;    % Toma o segundo argumento
JUMP L2;   % Salta para L2
LABEL L1;  % Rótulo L1
PUSH 2;    % Toma o terceiro argumento
LABEL L2;  % Rótulo L2
EVAL;      % Avalia o argumento empilhado
UPDATE 4;  % Atualiza a raiz do redex
POP 3;     % Desempilha os argumentos
UNWIND;    % Continua
```

Instrução \$CASE-n

- Para implementar um \$CASE-n, precisa-se de:

CASEJUMP $L1, L2, \dots, Ln$

que examina o *tagdo* objeto no topo e salta, dependendo do seu valor, para um dos n labels.

III. O CÓDIGO-G

Código-G

- Para cada instrução do código-G, existe uma sequência de instruções de máquina.
- A saída do gerador de código é um programa em *assembly*, junto com alguma biblioteca de tempo de execução.
- Para especificar o gerador de código deve-se saber:
 1. quais são as instruções do código-G e o que cada uma faz,
 2. como os vários *bits* da máquina-G abstrata são mapeados para a máquina destino.

Máquina de Estados

- A máquina-G é uma máquina de estados finitos com os seguintes componentes:
 1. S , a pilha;
 2. G , o grafo;
 3. C , o código-G que resta para ser executado;
 4. D , o *dump*, que consiste em uma pilha de pares (S, C) , onde S é uma pilha e C é uma sequência de código.
- O estado da máquina é uma 4-tupla $\langle S, G, C, D \rangle$.
- As operações da máquina-G são descritas via transições de estado.

Notação...

- Uma pilha S cujo topo é n é escrita como, $n : S$
- Uma pilha vazia é escrita como $[]$.
- Um *dump* D cujo topo é um par (S, C) é escrito como, $(S, C) : D$
- Um *dump* vazio é escrito como $[]$.

...Notação

- Os tipos de nodos possíveis no grafo são:

INT i um inteiro
 CONS $n_1 n_2$ um nodo tipo CONS
 AP $n_1 n_2$ um nodo de aplicação
 FUN $k C$ função de k args e código C
 HOLE nodo vazio, a ser preenchido.

Serve para construir grafos cíclicos.

- A notação $G[n = AP n_1 n_2]$ representa um grafo G , no qual o nodo n é uma aplicação de n_1 a n_2 .
- A notação $G[n = G n']$ representa um grafo no qual um nodo n tem o mesmo conteúdo do nodo n' .

Transições de Estado da PUSHINT

- PUSHINT i :

$\langle S, G, \text{PUSHINT } i : C, D \rangle \Rightarrow$
 $\langle n : S, G[n = \text{INT } i], C, D \rangle$

- Isso diz que, quando PUSHINT i é a primeira instrução, a máquina-G faz uma transição (\Rightarrow) para um novo estado no qual:
 - um novo nodo n é empilhado em S ,
 - o grafo é atualizado com a informação que o nodo n é INT i ,
 - o código a ser executado é tudo após PUSHINT i ,
 - o *dump* permanece o mesmo.
- O nome n , introduzido no lado direito, é um novo e único nome de nodo.

Transições de Estado da EVAL...

$\langle n : S, G[n = AP n_1 n_2], \text{EVAL} : C, D \rangle \Rightarrow$
 $\langle n : [], G[n = AP n_1 n_2], \text{UNWIND} : [], (S, C) : D \rangle$

- Neste caso o nodo no topo da pilha é uma aplicação.
- A pilha corrente e o código são empilhados no *dump*.
- É formada uma nova pilha com o topo da pilha antiga como seu único elemento, e um UNWIND é executado.

...Transições de Estado da EVAL...

$\langle n : S, G[n = \text{FUN } 0 C'], \text{EVAL} : C, D \rangle \Rightarrow$
 $\langle n : [], G[n = \text{FUN } 0 C'], C' : [], (S, C) : D \rangle$

- Neste caso o topo da pilha é uma CAF.
- A máquina salva seu estado no *dump*,
- forma uma nova pilha tendo a CAF como seu único elemento,
- executa o código associado com a CAF (que subsequentemente irá atualizar o nodo FUN com seu valor reduzido).

...Transições de Estado da EVAL...

- $\langle n : S, G[n = \text{INT } i], \text{EVAL} : C, D \rangle \Rightarrow$
 $\langle n : S, G[n = \text{INT } i], C, D \rangle$
 - * Esta transição descreve EVAL quando o topo da pilha é um inteiro.
 - * EVAL não fará nada.
- O mesmo se aplica se o nodo no topo da pilha é um CONS ou um nodo de função não-CAF.
- Uma transição omitida indica um erro em tempo de execução, isto é, n é uma HOLE.

...Transições de Estado da EVAL

- Por questões de clareza, G é abreviado:
 $\langle n : S, G[n = \text{AP } n_1 n_2], \text{EVAL} : C, D \rangle$
 $\Rightarrow \langle n : [], G, \text{UNWIND} : [], (S, C) : D \rangle$
- A descrição completa do código é:
 - Transições de estado da máquina-G (controle)
 - Transições de estado da máquina-G (dados e pilha)

Definição de Instruções G...

EVAL:

- $\langle n : S, G[n = \text{AP } n_1 n_2], \text{EVAL} : C, D \rangle$
 $\Rightarrow \langle n : [], G, \text{UNWIND} : [], (S, C) : D \rangle$
- $\langle n : S, G[n = \text{FUN } 0 C'], \text{EVAL} : C, D \rangle$
 $\Rightarrow \langle n : [], G, C' : [], (S, C) : D \rangle$
- $\langle n : S, G[n = \text{INT } i], \text{EVAL} : C, D \rangle$
 $\Rightarrow \langle n : S, G, C, D \rangle$
- e similarmente para CONS e nodos FUN não-CAF

...Definição de Instruções G...

JUMP:

- $\langle S, G, \text{JUMP } L : \dots : \text{LABEL } L : C, D \rangle$
 $\Rightarrow \langle S, G, C, D \rangle$

JFALSE:

- $\langle n : S, G[n = \text{BOOL true}], \text{JFALSE } L : C, D \rangle$
 $\Rightarrow \langle S, G, C, D \rangle$
- $\langle n : S, G[n = \text{BOOL false}],$
 $\text{JFALSE } L : \dots : \text{LABEL } L : C, D \rangle$
 $\Rightarrow \langle S, G, C, D \rangle$

...Definição de Instruções G...

RETURN:

- $\langle v_0 : \dots : v_k : [], G, \text{RETURN} : [], (S, C) : D \rangle$
 $\Rightarrow \langle v_k : S, G, C, D \rangle$

...Definição de Instruções G...

UNWIND:

- $\langle n : [], G[n = \text{INT } i], \text{UNWIND} : [], (S, C) : D \rangle$
 $\Rightarrow \langle n : S, G, C, D \rangle$
e similarmente para nodos CONS
- $\langle v : S, G[v = \text{AP } v' n], \text{UNWIND} : [], D \rangle$
 $\Rightarrow \langle v' : v : S, G, \text{UNWIND} : [], D \rangle$
- $\langle v_0 : \dots : v_k : S,$
 $G[v_0 = \text{FUN } k C, v_i = \text{AP } v_{i-1} n_i, (1 \leq i \leq k)], \text{UNWIND} : [], D \rangle$
 $\Rightarrow \langle n_1 : \dots : n_k : v_k : S, G, C, D \rangle$
- $\langle v_0 : \dots : v_a : [], G[v_0 = \text{FUN } k C'], \text{UNWIND} : [], (S, C) : D \rangle$
 $(a < k) \Rightarrow \langle v_a : S, G, C, D \rangle$

Explicação da UNWIND...

1. O item no topo da pilha é um inteiro ou um CONS.
 - Deve haver um único elemento na pilha e já WHNF.
 - UNWIND completa avaliação, recupera pilha e código antigos do *dump* e coloca o resultado da avaliação no topo da pilha recuperada.
2. O item no topo da pilha é nodo de aplicação.
 - Nesse caso empilha-se a cabeça da aplicação na pilha e repetimos a instrução UNWIND,

...Explicação da UNWIND

3. O item no topo da pilha é uma função e existe argumentos suficientes na pilha.
 - Rearranja-se a pilha e começa-se a executar o código da função.
 - Os v_i são as vértebras na espinha, enquanto os n_i são os argumentos para a função.
4. O item no topo da pilha é uma função com argumentos insuficientes para execução ($a < k$).
 - A expressão já está na WHNF; UNWIND completa a avaliação, recupera a pilha e código antigos do *dump*, coloca o resultado da avaliação no topo da pilha recuperada.

Definição de Instruções G...

PUSH:

- $\langle n_0 : n_1 : \dots : n_k : S, G, \text{PUSH } k : C, D \rangle$
 $\Rightarrow \langle n_k : n_0 \dots n_k : S, G, C, D \rangle$

PUSHINT:

- $\langle S, G, \text{PUSHINT } i : C, D \rangle$
 $\Rightarrow \langle n : S, G[n = \text{INT } i], C, D \rangle$

PUSHGLOBAL:

- **similarmente**

POP:

- $\langle n_1 : \dots : n_k : S, G, \text{POP } k : C, D \rangle$
 $\Rightarrow \langle S, G, C, D \rangle$

...Definição de Instruções G...

SLIDE:

- $\langle n_0 : n_1 : \dots : n_k : S, G, \text{SLIDE } k : C, D \rangle$
 $\Rightarrow \langle n_0 : S, G, C, D \rangle$

UPDATE:

- $\langle n_0 : n_1 : \dots : n_k : S, G, \text{UPDATE } k : C, D \rangle$
 $\Rightarrow \langle n_1 : \dots : n_k : S, G[n_k = G n_0], C, D \rangle$

Definição de Instruções G...

ALLOC:

- $\langle S, G, \text{ALLOC } k : C, D \rangle$
 $\Rightarrow \langle n_1 : \dots : n_k : S,$
 $G[n_1 = \text{HOLE}, \dots, n_k = \text{HOLE}], C, D \rangle$

HEAD:

- $\langle n : S, G[n = \text{CONS } n_1 n_2], \text{HEAD} : C, D \rangle$
 $\Rightarrow \langle n_1 : S, G C, D \rangle$

NEG:

- $\langle n : S, G[n = \text{INT } i], \text{NEG} : C, D \rangle$
 $\Rightarrow \langle n' : S, G[n' = \text{INT } (-i)], C, D \rangle$

...Definição de Instruções G...

ADD:

- $\langle n_1 : n_2 : S, G[n_1 = \text{INT } i_1, n_2 = \text{INT } i_2],$
 $\text{ADD} : C, D \rangle$
 $\Rightarrow \langle n : S, G[n = \text{INT } (i_1 + i_2)], C, D \rangle$

MKAP:

- $\langle n_1 : n_2 : S, G, \text{MKAP} : C, D \rangle$
 $\Rightarrow \langle n : S, G[n = \text{AP } n_1 n_2], C, D \rangle$

CONS:

- **similar**

O Mecanismo de Impressão

- É necessário um mecanismo de impressão que invoca o avaliador repetidamente para reduzir as expressões para a WHNF e imprimi-las.
- A nova instrução PRINT imprime o elemento do topo da pilha.
- Para descrever essa ação, é necessário um novo componente no estado da máquina:
 O
 que é a saída produzida pela máquina.
- A saída vazia é denotada por $[]$
- $O; x$ denota a saída O seguida da saída x .

O Mecanismo de Impressão...

- A definição de PRINT:
 - $\langle O, n : S, G[n = \text{INT } i], \text{PRINT} : C, D \rangle$
 $\Rightarrow \langle O; i, S, G, C, D \rangle$
 - $\langle O, n : S, G[n = \text{CONS } n_1 n_2],$
 $\text{PRINT} : C, D \rangle$
 $\Rightarrow \langle O, n_1 : n_2 : S, G,$
 $\text{EVAL} : \text{PRINT} : \text{EVAL} : \text{PRINT} : C, D \rangle$
- Todas outras instruções deixam O sem modificações.
- Finalmente a instrução BEGIN, que inicializa a máquina:
 - $\langle O, S, G, \text{BEGIN} : C, D \rangle$
 $\Rightarrow \langle O, [], [], C, [] \rangle$.

III. IMPLEMENTAÇÃO DA MÁQUINA-G

Estratégia de Implementação da Máquina-G

- Começaremos com uma apresentação resumida da linguagem de máquina do computador VSX
- A seguir apresentamos uma discussão de como implementar a máquina abstraída pelo código-G numa máquina concreta.
- Para isso vamos prover representações concretas para cada um dos quatro componentes do estado da máquina: S, G, C, D .
- Depois será mostrado como compilar o Código-G para a linguagem de máquina concreta do VAX.

A sintaxe do assembler do VAX Unix

- **Resumo dos comandos da linguagem *assembly* do VAX:**

```
movl <conteudo-fonte>, <conteudo-destino>
                                     % move conteudo do operando
movl <endereco-fonte>, <endereco-destino>
                                     % move endereco do operando
jsb  <endereco-destino> % salta empilhando o retorno
rsb  % desempilha o retorno e volta
jmp  <endereco-codigo>  % compara operadores
jlss <endereco-codigo> % salto condicional, se for menor
cmpl <op1>, <op2>      % compara operadores
```

Endereçamento no VAX Unix

- **%reg** denota um registrador, rotulado por *reg*.
- Podem ser usados *r0* para denotar o registrador 1, *r2* para denotar o registrador 2 e assim por diante.
- Os modos de endereçamento mais comuns são:

```
n(%reg)
  endereçamento indexado = conteudo do registrador reg + n
(%reg)
  endereçamento indireto
-(%reg)
  endereçamento indireto com pré-decremento
(%reg)+
  endereçamento indireto com pós-incremento
```

Representação da Pilha

- A pilha da máquina-G é representada pela área de dados que a armazena e juntamente com um ponteiro armazenado num registrador, seja ele EP, que pode ser definido pela diretiva:


```
.set EP, 10
```
- A pilha cresce para baixo e cada elemento é uma palavra de 32-bits.
- Os elementos podem ser empilhados usando pré-decremento e desempilhados usando pós-incremento.
- Seja empilhar (e desempilhar) o conteúdo do registrador *r0*:

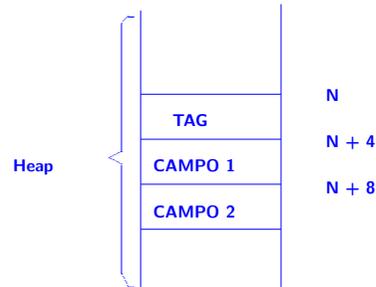

```
movl r0, -(%EP) % empilha
movl (%EP)+, r0 % desempilha
```
- É necessário a checagem de *overflow*, que precisa ser feita somente no início de uma função (o cálculo pode ser feito em tempo de compilação), exceto quando há o uso de UNWIND e EVAL.

Representação do grafo...

- O grafo é representado por uma grande área de *heap*.
- Cada nodo é representado por uma célula do *heap*.
- Cada célula consiste em um *tag* e um ou mais campos.
- Cada campo ou *tag* ocupa uma palavra VAX.

...Representação do grafo

- Por exemplo, uma célula de dois campos teria a seguinte configuração:



...Representação do grafo

- Pode parecer desperdício alocar 4 *bytes* para um *tag*, mas isso permitirá uma uniformidade de alocação de *heap* que possibilitará uma otimização simples, mas eficiente.
- Serão usadas representações *boxed* para valores básicos.
- Células serão alocadas continuamente, e assim teremos um ponteiro *HP* para a próxima célula livre.
- Essa abordagem requer um coletor de lixo compactador, sendo o coletor copiador o mais natural para ser usado.
- A checagem de *overflow* do *heap*, pode ser feita no início de um supercombinador (o espaço de *heap* pode ser calculado em tempo de compilação).

Nomenclatura

- O código é o próprio resultado da geração de código, juntamente com o contador de programa.
- *dump* é a pilha do sistema VAX juntamente com um ponteiro armazenado no registrador especial *SP*.
- Geração de Código destino via um método simples que permita demonstrar uma técnica de otimização.

Geração do Código Destino a partir do Código-G...

- Para cada instrução do código-G deve corresponder uma pequena sequência de instruções de máquina do VAX.
- Seja, por exemplo, gerar código para a instrução PUSH n:
`movl 4*n(%EP), -(%EP)`
- A origem é $4*n(EP)$, e busca a palavra que está a $4*n$ bytes do topo da pilha, o qual é apontado por EP.
- O destino é o topo da pilha e nodos pré-decrementamos o ponteiro de pilha para empilhar a nova palavra.
- Num exemplo maior vamos gerar código para a função:
`g f = NEG (f 5)`
- Primeiramente, compilaremos para o código-G, gerando:
`PUSHINT 5; PUSH 1; MKAP; PUSHGLOBAL $NEG; MKAP;
 UPDATE 2; POP 1; UNWIND;`

....Geração do Código Destino a partir do Código-G

• Uma geração de código simples geraria:

Código-G	HP	Código Vax	Comentários
PUSHINT 5	0	<i>movl I_5, -(%EP)</i>	Empilha 5
PUSH 1		<i>movl 4(%EP), -(%EP)</i>	Empilha f
MKAP	4	<i>movl APPLY, (%HP)+</i>	Tag do nodo de aplicação para o <i>heap</i>
	8	<i>movl (%EP)+, (%HP)+</i>	Função do nodo de aplicação (f)
	12	<i>movl (%EP)+, (%HP)+</i>	Argumento do nodo de aplicação (5)
		<i>movl -12(%HP), -(%EP)</i>	Resultado na pilha (f 5)
PUSHGLOBAL \$NEG		<i>movl C_NEG, -(%EP)</i>	Empilha NEG
MKAP	16	<i>movl APPLY, (%HP)+</i>	Tag do nodo de aplicação para o <i>heap</i>
	20	<i>movl (%EP)+, (%HP)+</i>	Função p/ aplicação (\$NEG)
	24	<i>movl (%EP)+, (%HP)+</i>	Argumento do nodo de aplicação (f 5)
		<i>movl -12(%HP), -(%EP)</i>	Resultado na pilha (\$NEG(f 5))
UPDATE 2		<i>movl (%EP)+, r1</i>	Resultado no registrador r1
		<i>movl 4(%EP), r2</i>	Raiz do <i>redex</i> em r2
		<i>movl (r1)+, (r2)+</i>	Cópia o tag
		<i>movl (r1)+, (r2)+</i>	Cópia o primeiro campo
		<i>movl (r1)+, (r2)+</i>	Cópia o segundo campo
POP 1		<i>movl 4(%EP), %EP</i>	Decrementa o ponteiro de pilha

- APPLY é a palavra de *tag* para um nodo de aplicação
- I_5 é o endereço de um inteiro *boxed 5*
- C_NEG é o endereço da célula com a função NEG.

Otimização Usando um Modelo de Pilha...

- A idéia dessa otimização é que durante a geração de código, deveríamos manter um modelo de como está a pilha num dado instante.
- Chamamos esse modelo de pilha simulada.
- A pilha simulada é uma pilha de tempo de compilação, que mantém os valores que estariam na pilha em tempo de execução.

...Otimização Usando um Modelo de Pilha...

- Por exemplo, poderíamos ter as seguintes entradas na pilha e seus respectivos valores:
 - 5, o valor literal 5
 - NEG, o endereço da célula com a função \$NEG
 - *heap 20*, o endereço da célula com *offset 20* a partir do ponteiro *HP* no início da execução do supercombinador
 - *stack 2*, o valor no *offset 2* a partir do ponteiro EP no início da execução do supercombinador

...Otimização Usando um Modelo de Pilha

- Abaixo ilustramos o exemplo anterior, mostrando uma redução no número de instruções VAX geradas.

Código-G	HP	Código Vax	Pilha Simulada	Comentários
	0		[]	Início
PUSHINT 5			5 : []	Empilha 5
PUSH 1			stack 0 : 5 : []	Empilha f
MKAP	4	<i>movl APPLY, (%HP)+</i>		Tag p/ <i>heap</i>
	8	<i>movl 0(%EP)+, (%HP)+</i>	5 : []	Fun p/ <i>heap</i>
	12	<i>movl I_5, (%HP)+</i>	[]	Arg p/ <i>heap</i>
			<i>heap 0</i> : []	Resultado na pilha
			NEG : <i>heap 0</i> : []	Empilha NEG
PUSHGLOBAL \$NEG				Tag p/ <i>heap</i>
MKAP	16	<i>movl APPLY, (%HP)+</i>	<i>heap 0</i> : []	Fun p/ <i>heap</i>
	20	<i>movl C_NEG, (%HP)+</i>		Arg p/ <i>heap</i>
	24	<i>movl -20(%HP), (%HP)+</i>	[]	Resultado na pilha
			<i>heap 12</i> : []	Resultado em r1
UPDATE 2		<i>movl -12(%HP), r1</i>	[]	Raiz em r2
		<i>movl 4(%EP), r2</i>		Cópia o tag
		<i>movl (r1)+, (r2)+</i>		Cópia o primeiro
		<i>movl (r1)+, (r2)+</i>		Cópia o segundo
POP 1		<i>movl 4(%EP), %EP</i>		Desempilha args

Manipulando EVALs e JUMPs...

- O EVAL gera um certo aborrecimento, pois pode exigir que muita computação ocorra.
- Isso significa que não podemos prever o quanto vamos consumir de *heap*, além do que uma coleta de lixo pode ocorrer durante a avaliação e assim atrapalhando a pilha simulada e o HP.
- Podemos lidar com essas complicações tratando separadamente os segmentos de código entre os EVALs, cada qual com seu próprio código para checar o esgotamento do *heap*.
- Todos os *offsets* da pilha e do *heap* na pilha simulada são calculados relativamente aos valores de EP e HP no início do segmento (não do supercombinador).

...Manipulando EVALs e JUMPs

- Além disso, antes do EVAL ser chamado, a pilha simulada deve ser descarregada na pilha real.
- Considerações similares se aplicam a seções de código quebradas com instruções JUMP.
- Se existem duas rotas diferentes levando a uma certa porção do código, então quantidades diferentes de *heap* devem ter sido alocadas pelas duas rotas e o conteúdo da pilha simulada também deve ser diferente.

Mais sobre a Representação do Grafo...

- Podemos pensar na máquina-G como sendo um tipo abstrato de dados, onde temos o grafo como sendo o dado em si, o mecanismo de execução como sendo as operações que atuam sobre aquele dado.
- Essa nítida separação apesar de prover uma grande flexibilidade pode causar uma certa penalidade na performance.

...Mais sobre a Representação do Grafo

- Contudo, podemos distinguir entre dois tipos de operações:
 1. Operações específicas ao nodo.
 - Por exemplo, a instrução do código-G HEAD, é executada somente quando existe um nodo CONS no topo da pilha.
 - Operações específicas ao nodo normalmente podem ser compiladas numa única instrução da máquina destino.
 2. Operações genéricas.
 - Por exemplo, quando a instrução UNWIND é executada, nada se sabe sobre o que está no topo da pilha.
 - A primeira coisa que UNWIND deve fazer é uma análise de caso no tipo do nodo, tornando essas operações normalmente mais caras.

Implementando a Análise de Caso dos *tags*

-
- Na máquina-G de Chalmers o *tag* de uma célula é uma palavra e aponta para uma pequena tabela com pontos de entrada para código, um ponto para cada operação genérica.
- Cada tipo de nodo distinto tem uma tabelas de entradas diferente, tal que a análise de casos numa célula pode ser feita com um salto para a entrada apropriada apontada a partir do *tag* da célula.

Implementando EVAL...

- Vamos dividir em duas partes:
 1. o código que é gerado para uma instrução EVALqualquer;
 2. o código de EVAL para cada *tag* da tabela de entradas.
- Primeiro vamos com o código VAX para uma instrução EVAL qualquer:


```
movl (%EP),r0    % Topo da pilha para r0
movl (r0), r1    % {\it Tag} para r1
jsb  *O_Eval(r1) % Chamada para o c\{o}digo de Eval
```
- O elemento no topo da pilha é colocado em r0 (sem desempilhar), seu *tag* é colocado em r1, e a instrução final é uma chamada indexada de subrotina, onde O_Eval é o *offset* de Eval na tabela de entradas.

...Implementando EVAL

- Agora consideraremos o código de Eval que será chamado.
- Suponha que a célula em questão é um inteiro (vamos supor uma implementação *boxed*).
- Nesse caso nenhuma avaliação será necessária e o código simplesmente retorna da subrotina:

```
INT_EVAL : rsb    Retorna de Eval
```

- O mesmo se aplica a células de função e células CONS.

...Implementando EVAL

- Para nodos de aplicação devemos empilhar a pilha corrente (*EP*) no *dump D*, e depois fazer um UNWIND na aplicação.

```
AP_EVAL :
```

```
<Teste para overflow de SP>
```

```
    movl %EP,-(SP) % Empilha EP no dump
```

```
<executa um AP_UNWIND>
```

```
    % V. proxima secao
```

Implementando UNWIND...

- Aqui está a sequência de código VAX que deve ser gerada para uma instrução de código-G UNWIND:

```
movl (%EP,r0)      % Topo da pilha em r0
movl (r0), r1      % Tag em r1
jmp *O_Unwind(r1) % Salto para Unwind
```

- O elemento no topo da pilha é colocado em r0, sem desempilhar, o tag é colocado em r1 e é feito um salto (sem retorno) indexado para o código de UNWIND.
- Quando a célula em questão for uma aplicação, deveríamos empilhar a cabeça da célula na pilha e executar um UNWIND novamente.

Implementando UNWIND...

- Lembrando que r0 aponta para a célula em questão, temos o código do Unwind para uma aplicação:

AP_UNWIND:

<Checa overflow de EP>

```
movl Head(r0),r0$ % Toma a cabe\c{c}a
movl r0,-(%EP)$ % Empilha-a
movl (r0),r1$ % P\~{o}e o tag em r1
jmp *O_Unwind(r1) % Executa o unwind.
```

- Se a célula for um inteiro, a especificação de UNWIND nos diz que a célula de inteiro deve ser a única coisa na pilha, e devemos retornar ao chamador, recuperando a pilha antiga mas colocando o elemento do topo da pilha corrente no topo da pilha a ser recuperada.

Implementando UNWIND...

- Código do Unwind para um inteiro:

INT_UNWIND:

```
movl (SP)+,%EP % Recupera o ponteiro da pilha antiga
rsb % Retorna para o chamador
```

- Se a célula for de uma função global, a especificação de UNWIND requer um teste para checar se existem argumentos suficientes na pilha para que a função possa executar.
- A máquina-G de Chalmers usa um tag separado para cada função, completado com uma tabela de entradas separada.
- Isso significa que ao invés de termos código para FUN_UNWIND, teremos um pedaço de código F_UNWIND para cada função global F (supercombinador ou função *built-in*).

Implementando UNWIND...

- Suponha que F tenha dois argumentos.
- Então tem-se o código para F_Return:

% Chega-se aqui caso tenha poucos args

F_Return:

```
movl (SP)+,%EP % Recupera o ponteiro de pilha
rsb % Retorna para o chamador
```

Implementando UNWIND...

- E o código para F_UNWIND:

```
F_Unwind: % Nota: ponteiro para FUN ainda está na pilha
    movl 8(%EP),r0 % r0 aponta para a base do contexto
    cmpl (SP), r0 % A base do contexto está abaixo da pilha
    jless F_Return % Retorna se poucos argumentos
```

```
% Agora rearruma a pilha
    movl 4(%EP),r0 % Vértebra do topo em r0
    movl Tail(r0),(%EP) % Empilha sua cauda (escreve sobre FUN)
    movl 8(%EP),r0 % Próxima vertebra em r0
    movl Tail(r0),4(%EP) % Cauda na pilha
```

```
F_EXEC % Agora vem o c\'}{o}digo para F
    . . .
```

Nodos de Indireção...

- A maior vantagem desse método de implementar operações genéricas é que novos tipos de nodos podem ser inseridos sem mudança alguma, exceto prover uma nova entrada na tabela.
- Até agora descrevemos uma implementação da máquina-G que executa a atualização ao final da redução, copiando a raiz do resultado sobre a raiz do *redex*.

...Nodos de Indireção...

- Uma outra alternativa discutida em capítulo anterior era re-escrever a raiz do redex com uma indireção para o resultado.
- Para isso teremos que fazer duas modificações:
 1. devemos introduzir um novo tipo de célula, a célula de indireção, juntamente com sua tabela de entradas. Ela terá somente um campo, que conterá o ponteiro de indireção;
 2. devemos modificar a implementação da instrução UPDATE.
- O único trabalho associado com a primeira modificação é prover seqüências de código de máquina, para cada operação genérica.

...Nodos de Indireção...

- Por exemplo, IND_UNWIND, o código Unwind para uma célula de indireção, será:

```
movl 4(r0),r0 % Toma o ponteiro de indirecao
movl r0,(%EP) % Sobrescreve o elemento do topo da pilha
movl (r0),r1 % Toma o tag
jmp *0\_Unwind(r1) % Salta para o codigo Unwind
```

...Nodos de Indireção...

- Similarmente `IND_EVAL` será:

```
movl 4(r0),r0    % Toma o ponteiro de indirecao
movl r0,(%EP)   % Sobrescreve o elemento do topo da pilha
movl (r0),r1    % Toma o tag
jmp  *0_Eval(r1) % Salta para o codigo Eval
```

- A segunda coisa que devemos fazer é alterar a implementação de `UPDATE`.

- Lembre que `UPDATE k` atualiza a raiz do *redex*, que é apontada pelo *k*-ésimo elemento da pilha, com o resultado que está no topo da pilha.

...Nodos de Indireção...

- A nova implementação de `UPDATE` deve fazer três coisas:

1. Sobrescrever a vértebra apontada do *k*-ésimo elemento da pilha com um nodo de indireção, cujo ponteiro de indireção aponta para o resultado.
2. Sobrescrever o *k*-ésimo elemento da pilha para apontar diretamente para o resultado (não para o nodo de indireção).
3. Isso é somente uma otimização, mas garante que o resultado de um `EVAL` nunca será uma célula de indireção.
4. Isso é útil quando, por exemplo, o resultado de um `EVAL` é reconhecidamente um inteiro.
5. Nesse caso é um aborrecimento ter que checar também por uma indireção.
6. Desempilhar o resultado da pilha.

...Nodos de Indireção

- Isso dá a seguinte sequência de código para a instrução `UPDATE d` do código-G:

```
movl 4*d(%EP),r2    % r2 aponta para a raiz do redex
movl IND,(r2)+      % Tag IND de indire\c{c}\~{a}o
movl (%EP),(r2)     % Põe resultado numa celula de indireção
movl (%EP)+,4*d(%EP) % Sobrescreve vértebra e empilha resultado
```

Representações do Tipo *boxed* × *unboxed*

- A máquina-G de Chalmers usa representações *boxed* para todos valores básicos.

- Existem duas razões para isso:

1. Uma representação *boxed* de um valor básico tem um *tag* no mesmo lugar como qualquer outro valor.
2. Isso possibilita que as operações básicas possam ser implementadas uniformemente.
3. Com representações *unboxed*, as operações genéricas deveriam ter que fazer um teste inicial para separar ponteiros de não-ponteiros antes de fazer uma análise de caso como antes.
4. Uma representação *unboxed* necessitaria de carregar um *bit* de ponteiro em cada campo.

Sumário

Vimos que a técnica para implementar operações genéricas usando os *tags* de célula como ponteiros para tabelas de entrada fornece duas grandes vantagens:

1. é fácil adicionar novos tipos de nodos (nodos de indireção, por exemplo);
2. é rápida, pois operações genéricas serão implementadas uniformemente usando um *jump* indexado.

Agrupando os Resultados

- Como o código que geramos fica agrupado?
- Para começar, o código-G para cada supercombinador começa com uma instrução `GLOBSTART`.

...Agrupando os Resultados...

- A instrução `GLOBSTART` deve gerar os seguintes segmentos de código destino:
 - o código de `UNWIND`, que checa o número de argumentos e rearranja a pilha;
 - o código `GC`, que dependerá do coletor de lixo;
 - a tabela de entrada para o supercombinador (as entradas de `EVAL`, `PRINT`, etc. são as mesmas para todos os supercombinadores);
 - o nodo da função propriamente dita, que pode ser alocado no início do código da função, fora do *heap* principal;
 - o código de checagem de *overflow*, que precede imediatamente o código para o corpo da função e checa o *overflow* da pilha e do *heap*.

...Agrupando os Resultados

- O código destino para cada função é precedido por alguns fragmentos de código, a tabela de entrada e o nodo da função.
- Isso completa a geração de código para cada função.
- Finalmente devemos considerar sobre o que fazem as instruções `BEGIN` e `END` do código-G.
- A instrução `BEGIN` é responsável pela inicialização de todo o sistema.
- Em particular ela deve gerar código destino para:
 - inicializar o ponteiro de pilha *EP*;
 - inicializar o *heap* (em particular *HP*).
- A instrução `END` simplesmente termina a execução do programa.

III. OTIMIZAÇÕES NA MÁQUINA-G

Estratégias de Otimização

- Daremos agora uma sequência de otimizações para os esquemas de compilação do código-G.
- No geral, elas são independentes umas das outras e qualquer combinação delas pode ser implementada.
- Todas elas são baseadas na idéia de gerar um código especial de forma a evitar a construção de grafos.

Não Construção de Grafos

- Um objetivo primário de nossas otimizações será usar a pilha ao invés do *heap*, sempre que possível.
- O *heap* provê um esquema de alocação muito geral, mas também muito caro.
- Já a pilha é um mecanismo de alocação menos flexível, mas a memória alocada é recuperada imediatamente quando ela não for mais usada, e o mecanismo de recuperação é muito barato (basta decrementar o ponteiro de pilha).
- Em particular o esquema C de compilação constrói grafos no *heap*, e muitas de nossas otimizações consistirão de substituir o esquema C por esquemas mais baratos.

Preservando *laziness*

- Como mencionamos anteriormente, quando introduzimos a primeira versão do esquema R, vimos que ele tinha baixa performance quando o corpo do supercombinador é uma única variável.
- O que devemos fazer é redefinir R de modo a ter casos separados para cada tipo de expressão, assim como fizemos para o esquema C.
- Essa otimização deve ser tida como essencial, já que sem ela *laziness* poderia ser perdida.
- No novo esquema R o código para um corpo que é apenas uma variável, carrega o valor da variável na pilha, usa EVAL para avaliá-lo e só depois atualiza a raiz do *redex* com o resultado.
- Note que o lete e letrecsão definidos elegantemente com uma chamada recursiva ao próprio R.
- O esquema R pode ser visto no fim deste capítulo.

Execução Direta de Funções *built-in*

- Essa é provavelmente a otimização mais importante e diz respeito à compilação de expressões como

$$(P \ x_1 \ x_2)$$

quando

- P é uma função *built-in*,
- todos os argumentos estão presentes.

Otimizações do Esquema R...

- Como nosso primeiro exemplo, considere compilar $(CONS \ E_1 \ E_2)$ diferentemente do que foi apresentado anteriormente, quando construímos o grafo de $(CONS \ E_1 \ E_2)$, e depois executamos um UNWIND:

$$R[[CONS \ E_1 \ E_2]]\rho \ d = C[[E_2]]\rho \ d; C[[E_1]]\rho \ (d+1); CONS; UPDATE(d+1); POP \ d; RETURN;$$

- Nessa nova versão, construímos os grafos para E_1 e E_2 , executamos a instrução CONS do código-G para formar uma célula CONS, atualizamos a raiz do *redex* e depois retornamos.
- Dessa maneira alocamos menos nodos no *heap*, usamos menos instruções do código-G e evitamos executar o código para a função \$CONS, desenvolvida no capítulo sobre a máquina-G.

...Otimizações do Esquema R...

- Para a função IF também podemos melhorar o esquema R da seguinte maneira:

$$R[[IF \ E_c \ E_t \ E_f]]\rho \ d = C[[E_c]]\rho \ d; EVAL; JFALSE \ L; R[[E_t]]\rho \ d; LABEL \ L; R[[E_f]]\rho \ d;$$

- A função *Case-n* pode ser compilada de maneira análoga, usando um salto de vários caminhos (CASEJUMP) ao invés de um salto de dois caminhos (JFALSE).

...Otimizações do Esquema R

- As expressões $(+ \ E_1 \ E_2)$ e $(HEAD \ E)$ podem ser otimizadas usando a mesma idéia.
- Ao invés de construir o grafo e imediatamente depois fazer um UNWIND nele, nodos executaremos as expressões diretamente:

$$R[[+ \ E_1 \ E_2]]\rho \ d = C[[E_2]]\rho \ d; EVAL; C[[E_1]]\rho \ (d+1); EVAL; ADD; UPDATE \ (d+1); POP \ d; RETURN$$

$$R[[HEAD \ E]]\rho \ d = C[[E]]\rho \ d; EVAL; HEAD; EVAL; UPDATE \ (d+1); POP \ d; RETURN;$$

O esquema E

- Se fizermos uma inspeção nas regras R extras, constataremos uma ocorrência frequente da sequência
 $C[[E]]\rho d; EVAL;$
- Agora suponha que E fosse da forma $(CONS E_1 E_2)$.
- Então poderíamos gerar código para construir o grafo de $(\$CONSsy E_1 E_2)$ e imediatamente avaliá-lo.
- Mas isso é exatamente o que as otimizações anteriores procuravam evitar.
- Como poderemos fazer a mesma otimização para a sequência C-EVAL?

O esquema E

- A razão pela qual a sequência C-Eval tem uma performance fraca é que o esquema C executa na ignorância do fato que o resultado será imediatamente avaliado.
- Vamos criar um novo esquema E que é uma versão do esquema C que entrega o resultado já avaliado.
- Teremos
 $E[[E]]\rho d$
 que produz código-G que avalia E para a WHNF e deixa o resultado no topo da pilha.
- Todo o esquema E pode ser visto no final deste capítulo.

Os esquemas RS e ES...

- Agora podemos usar E para substituir todos os usos da sequência C-EVAL no esquema R com uma chamada para E.
- Veja o esquema R no final do capítulo
- Considere a expressão:
 $(HEAD E_1 E_2)$
- Esperamos que E_1 avalie para uma célula CONS, cuja cabeça será uma função que será aplicada a E_2 .
- Vamos compilar essa expressão como o esquema R :
 $R[[HEAD E_1 E_2]]\rho d = C[[HEAD E_1 E_2]]\rho d; UPDATE(d+1);$
 $POP d; UNWIND;$

...Os esquemas RS e ES...

- Nós não fomos capazes de tirar vantagem da otimização de HEAD dada nas seções anteriores, por causa do segundo argumento E_2 .
- Esse problema pode ocorrer com qualquer função *built-in* que possa devolver uma função como resultado; em particular HEAD e IF juntamente com suas análogas SEL-k-i e case.
- O que gostaríamos de gerar para o exemplo acima seria:
 $R[[HEAD E_1 E_2]]\rho d = C[[E_2]]\rho d; E[[E_1]]\rho(d+1); HEAD; MKAP;$
 $UPDATE (d+1); POP d; UNWIND;$
- Usar essa otimização, requer aplicar um esquema de compilação do tipo R recursivamente em $(HEAD E_1)$, ao invés de simplesmente usar o esquema C.
- Chamaremos esse novo esquema de compilação de esquema RS.

...Os esquemas RS e ES...

- O esquema RS será parecido com
 $RS[E_1 E_2]\rho d = C[E_2]\rho d RS[E_1]\rho (d+1);$
- Daí, poderíamos substituir
 $R[E_1 E_2]\rho d$
 por
 $RS[E_1 E_2]\rho d$
 (Obs: A regra para RS ainda não está totalmente correta).
- Assim, a compilação de HEAD $E_1 E_2$ começaria assim:
 $R[HEAD E_1 E_2]\rho d$
 $= RS[HEAD E_1 E_2]\rho d$
 $= C[E_2]\rho d; RS[E_1]\rho (d+1);$
- Com a regra RS dada acima, descemos a espinha da expressão, construindo as costelas (argumentos) usando o esquema C e colocando-as na pilha.

...Os esquemas RS e ES...

- A questão que fica:
 O que o esquema RS deve fazer quando alcança o final?
- Nesse ponto, todas as costelas da expressão estão na pilha, e o que RS deveria fazer é gerar um número apropriado de MKAPs para construir a espinha da expressão, atualizar a raiz do *redex*, desempilhar os argumentos e fazer um UNWIND.
- Isso significa que RS deve conhecer quantas costelas estão na pilha, daí a necessidade de um parâmetro extra, n , para indicar esse número.
- A regra real para RS fica:
 $RS[E_1 E_2]\rho d n = C[E_2]\rho d; RS[E_1]\rho (d+1)(n+1);$

...Os esquemas RS e ES...

- Quando RS for invocado do esquema R teremos:
 $R[E_1 E_2]\rho d = RS[E_1 E_2]\rho d 0$
- Quando alcançamos o fim, RS simplesmente constrói a espinha com n MKAPs, atualiza a raiz do *redex*, desempilha os argumentos e UNWIND:
 $RS[f]\rho d n$
 $= PUSHGLOBAL f; MKAP n; UPDATE(d-n+1); POP(d-n); UNWIND;$

 $RS[x]\rho d n$
 $= PUSH(d - \rho x; MKAP n; UPDATE(d-n+1); POP(d-n); UNWIND;$
 onde MKAP n é uma versão estendida de MKAP, equivalente a n repetições de MKAP.

...Os esquemas RS e ES...

- Os *offsets* em UPDATE e POP levam em conta o fato da pilha ter ganhado um elemento como resultado do *PUSH* inicial e ter perdido n elementos como resultado de MKAP n .
- Ocorrências de um *let* ou *letrec* poderiam causar problemas para RS, já que supões-se que as n costelas ocuparão posições sucessivas na pilha.
- Felizmente podemos garantir que RS nunca encontrará um *let* ou *letrec*, transformando qualquer expressão da forma
 $(letrec <definições> in E_1) E_2$
 em
 $letrec <definições> in (E_1 E_2)$

...Os esquemas RS e ES...

- Otimizar o esquema R nos levou a desenvolver o esquema E. O esquema RS terá em contrapartida o esquema ES.
- A estrutura do esquema ES é a mesma do esquema RS.
- Eles diferem somente nos casos de $ES[f]$ e $ES[x]$.
- Os esquemas completos RS e ES podem ser encontrados no final do Capítulo 20 do Livro-Texto.

 η -Redução e *lambda-lifting*

- No capítulo sobre *lambda-lifting* mostramos como parâmetros redundantes poderiam ser eliminados por η -redução.
- Para uma implementação da máquina-G, isso é indesejável, a menos que elimine uma definição do supercombinador, o que é sempre desejável.
- Considere a definição,

$$\$F \ x \ y = IF \ E_1 \ E_2 \ y$$
 onde E_1 e E_2 não usam y .
- Podemos ter a seguinte definição equivalente:

$$\$F \ x = IF \ E_1 \ E_2$$
- Essa última definição vai gerar um código-G menos eficiente que a primeira definição.

 η -Redução e *lambda-lifting*

- A razão é que o IF não tem parâmetros suficientes, e por isso R não pode usar a eficiente sequência *testa-e-salta* que teria gerado para a definição prévia de $\$F$.
- Isso se aplica de maneira geral, e significa que a η -redução deve ser feita somente se eliminar uma definição inteira.
- As otimizações dessa seção se aplicam a expressões como $(f \ E_1 \ E_2 \ \dots)$, quando *nodos sabemos o que f é*.
- Se não soubermos o que f é, o código gerado será de pior qualidade.

 η -Redução e *lambda-lifting*

- A maneira pela qual isso tende a ocorrer é:

$$f \ x \ g \ y = g \ (+ \ x \ y)$$
- Ou seja, quando uma função é passada como um argumento e então aplicada.
- Infelizmente, *lambda-lifting* totalmente *lazy* resulta em muitas dessas expressões, e essa é a principal motivação para eliminar *full laziness* redundante.

Compilando \square e FAIL...

- Suponha que tenhamos que compilar

$$R[[\text{Fatbar } E_1 E_2]]\rho d$$

- Primeiro, lembremos as equações semânticas para Fatbar:

$$\text{Fatbar } a \quad b = a \quad \text{se } a \neq \perp \text{ e } a \neq \text{FAIL}$$

$$\text{Fatbar FAIL } b = b$$

$$\text{Fatbar } \perp \quad b = \perp$$

- Podemos compilar da seguinte forma:

$$R[[\text{Fatbar } E_1 E_2]]\rho d j s = R[[E_1]]\rho d L d;$$

$$\text{LABEL } L; R[[E_2]]\rho d j s$$

- Juntamente com a regra

$$R[[\text{FAIL}]]\rho d j s = \text{POP}(d-s); \text{JUMP } j$$

...Compilando \square e FAIL

- O efeito é o mesmo que o anterior.
- A instrução $\text{POP}(d-s)$ atualiza a pilha para o nível esperado pelo código em j , enquanto que a instrução $\text{JUMP } j$ atualiza o contador de programa.
- Juntas, essas duas instruções colocam a máquina-G no mesmo estado que ela estaria quando executando o código em j na primeira versão.
- Todos os outros casos do esquema R passam j e s inalterados.
- Otimizações similares se aplicam aos esquemas E, RS e ES.

Avaliando os Argumentos

- Suponha que um corpo de supercombinador consiste de uma expressão da forma $(f E_1 E_2)$, onde não podemos executar f diretamente como descrito na seção anterior.
- Poderíamos compilar a aplicação de uma maneira tradicional, gerando os grafos para E_1 e E_2 .
- Ao invés, suponha que f avaliaria seu primeiro argumento.
- Nesse caso seria seguro compilar E_1 com o esquema E (que a avaliaria), evitando a construção do grafo para E_1 .
- As otimizações dessa seção tentam evitar o uso do esquema C para compilar E_1 e E_2 , quando as funções são estritas.

Otimizando Aplicações Parciais

- Suponha que estejamos compilando o supercombinador:

$$f \ x = + (\text{NEG } x)$$
- Nós não podemos aplicar a otimização $R[[+ E_1 E_2]]$ da última seção, pois é dado somente um argumento para $+$.
- Contudo, a razão pela qual estamos avaliando $(f \ x)$ deve ser para aplicar a expressão para alguma coisa, e quando aplicada a essa coisa o primeiro argumento será avaliado. Assim teremos a seguinte regra:

$$R \ [[+ E]]\rho d = E[[E]]; \text{PUSHGLOBAL } + \text{MKAP};$$

$$\text{UPDATE}(d+1); \text{POP } d; \text{UNWIND};$$
- Essa otimização se aplica a todas as funções *built-in* com mais de um argumento, que avaliam o seu primeiro argumento.
- Em particular, temos $+$, IF , e seus análogos $-$, $*$, etc, e Case- .

Otimização Usando Informações globais de Funções Estritas...

- As otimizações da seção anterior se baseiam em informações especiais sobre funções *built-in*.
- Contudo, considere os seguintes supercombinadores

$$\$F \ x \ y = + \ y \ x$$

$$\$G \ x \ = \$F \ (* \ x \ x) \ (+ \ x \ x)$$
- Sem muito esforço podemos ver que $\$F$ é estrita em ambos os argumentos. Por isso, quando formos compilar $\$G$ poderíamos usar E para compilar $(* \ x \ x)$ e $(+ \ x \ x)$.
- Infelizmente essa informação não é tão óbvia para o compilador.
- Comentários similares se aplicam para expressões *let*.

...Otimização Global de Funções Estritas...

- *Letrecs* são mais problemáticos, pois existe um certo perigo que seria tentar avaliar uma célula *HOLE*.
- Por isso não tentaremos otimizar *letrecs*.
- Gostaríamos de fazer duas coisas:
 1. Primeiro, poderíamos usar informações sobre funções estritas para anotar aplicações. Por exemplo, poderíamos anotar o corpo de $\$G$ da seguinte forma:

$$\$G \ x \ = \$F \ ! \ (* \ x \ x) \ ! \ (+ \ x \ x)$$
 onde usamos "!" para indicar uma aplicação estrita.
 2. Segundo, é que precisamos modificar nossos esquemas para tirar vantagem dessa informação.

Otimização Globais de Funções Estritas...

- A última tarefa é mais fácil.
- O que precisamos fazer é adicionar uma cláusula ao esquema *ES* para dizer que

$$RS[E_1 \ ! \ E_2 \] \rho \ d \ n = E[E_2] \rho \ d; \ RS[E_1] \rho \ (d+1) \ (n+1)$$
- E fazer uma modificação similar para o esquema *ES*.
- Isso tem o efeito de uma passagem de parâmetros por valor, na qual o argumento E_2 é avaliado antes que a função E_1 seja aplicada a ele.
- Uma modificação similar se aplica ao tratamento de expressões-*let* em *R* e *E*.

Evitando Re-Avaliação no Corpo de uma Função...

- *EVAL* provavelmente é a instrução mais cara da máquina-G e otimizações que eliminam o uso de *EVAL* são muito bem vindas.
- Considere compilar uma expressão como $(+ \ x \ x)$ com o esquema *E*.
- Teremos

$$E[+ \ x \ x] \rho \ d$$

$$= \text{PUSH}(d - \rho \ x); \text{EVAL}; \text{PUSH}(d + 1 - \rho \ x); \text{EVAL}; \text{ADD};$$
- Esse código produz desperdício, pois o segundo *EVAL* não é necessário, porque x já havia sido avaliado.
- Seria preferível gerar:

$$\text{PUSH}(d - \rho \ x); \text{EVAL}; \text{PUSH}(d + 1 - \rho \ x); \text{ADD};$$

...Evitando Re-Avaliação no Corpo de uma Função

- Vemos que seria conveniente manter guardadas quais lugares da pilha estão avaliados, ao invés de quais variáveis estão avaliadas.
- Tudo que é necessário é adicionar um parâmetro extra, σ , para cada esquema de compilação.
- O novo parâmetro dá informação de contexto similar à ρ .
- σ é uma função a qual, dado um *offset* a partir da base do contexto corrente, retorna um *flag* indicando se aquela posição da pilha foi ou não avaliada.
- Além disso, cada esquema de compilação deve retornar dois pedaços de informação, o código que ele gera e um novo σ .
- O novo σ retornado pelo esquema é o mesmo que o σ passado para ele, exceto que os *flags* em alguns lugares da pilha foram atualizados para indicar que aqueles lugares foram avaliados.

Otimização de Funções Estritas...

- Considere a seguinte definição de supercombinador:

$$\$F\ x = IF\ (= x\ 0)\ 0\ (\$F\ (- x\ 1))$$
- Temos que $\$F$ é claramente estrita em x e o analisador pode descobrir isso e anotar a definição:

$$\$F\ x = IF\ !\ (= !\ x\ !\ 0)\ 0\ (\$F\ !\ (- !\ x\ !\ 1))$$
 onde $!$ é um operador binário infixado que denota aplicação estrita.
- Assim, quando $\$F$ é chamado recursivamente, já se sabe que seu argumento está avaliado.
- Contudo, $\$F$ não sabe que isso será sempre verdade, e irá em frente e chamará EVAL para o seu parâmetro durante o cálculo de $(= x\ 0)$.

...Otimização de Funções Estritas...

- O que gostaríamos é um outro supercombinador $\$F_NOEVAL$ que se comporta como $\$F$, exceto que ela assume que seu argumento está avaliado.
- Aí poderíamos usar $\$F_NOEVAL$ para a chamada recursiva e evitar a avaliação redundante de x .

...Otimização de Funções Estritas

- É requerido que:
 - o analisador de funções estritas anote as definições de supercombinador assim como os nodos de aplicação;
 - a informação que certos argumentos foram avaliados seja mantida no contexto (σ);
 - a entrada NOEVAL de uma função seja usada quando sabemos que todos seus argumentos estão avaliados.
 - A versão apropriada da função pode ser selecionada por RS ou ES (no caso geral), dependendo se souber que seus argumentos foram avaliados.
 - Na entrada para a função, os argumentos são mantidos nas posições (anônimas) do topo da pilha, sendo por isso que σ descreve quais posições foram avaliadas.

Evitando UNWINDs Repetidos...

- Algumas vezes sabemos que um nodo de aplicação não é a raiz do *redex*, mas como essa informação não está guardada, é preciso fazer um UNWIND toda vez que formos avaliá-lo, só para constatar que o nodo já está na WHNF.
- Se essa informação estivesse presente no nodo de aplicação, EVAL poderia constatar isso e retornar imediatamente.
- Essa otimização só se torna importante quando estamos fazendo análise das funções estritas, porque nesse caso funções seriam avaliadas quando passadas como argumentos estritos.
- Podemos incorporar essa informação no nodo de aplicação com um *tag* extra, AP-WHNF.

...Evitando UNWINDs Repetidos

- Se implementamos a análise de casos como no capítulo anterior, então a entrada de EVAL na tabela de entradas de AP-WHNF será idêntica à entrada para inteiros, isto é, retornará imediatamente para o chamador.

Avaliação Gulosa...

- Sob certas circunstâncias, podemos executar alguma redução mesmo que uma implementação totalmente *lazy* adiasse aquela redução.
- Considere a compilação de $(\text{CONS } E_1 E_2)$ com o esquema C:

$$C[\text{CONS } E_1 E_2] \rho \ d = C[E_2] \rho \ d; C[E_1] \rho \ (d+1);$$
 PUSHGLOBAL CONS; MKAP; MKAP
- Mas é claro que quando avaliada, a expressão $(\text{CONS } E_1 E_2)$ simplesmente retorna uma célula CONS, com $C[E_1]$ em um galho, e $C[E_2]$ no outro.
- Seria melhor construir a célula diretamente com o seguinte código:

$$C[\text{CONS } E_1 E_2] \rho \ d = C[E_2] \rho \ d; C[E_1] \rho \ (d+1); \text{CONS}$$

...Avaliação Gulosa...

- Se tivermos a informação descrita na seção anterior, dizendo quais variáveis foram avaliadas, podemos efetuar algumas outras otimizações ao esquema C.
- C é usado quando não temos certeza de que a expressão será realmente avaliada.
- Contudo, considere a compilação de $(+ x 3)$ com o esquema C num contexto onde x já foi avaliado. Nosso esquema atual produzirá

$$C[+ x 3] \rho \ d$$

$$= \text{PUSHINT } 3; \text{PUSH } (d - \rho \ x); \text{PUSHGLOBAL } +; \text{MKAP}; \text{MKAP}$$
- Mas seria consideravelmente mais barato gerar

$$C[+ \underset{\text{avaliado já}}{x} 3] \rho \ d = \text{PUSHINT } 3; \text{PUSH}(d - \rho \ x); \text{ADD}$$

...Avaliação Gulosa...

- A mesma otimização pode ser usada para qualquer outra função *built-in*. Por exemplo,

$$C[\text{HEAD } \underset{\text{avaliado já}}{y}] \rho d = \text{PUSH}(d - \rho y); \text{HEAD}$$

- Gostaríamos também de propagar essa informação para cima.

- Gostaríamos de garantir, por exemplo, que

$$C[\text{+ } (\text{+ } x \ 5) \ y] \rho d = \text{PUSH}(d - \rho y); \text{PUSHINT5}; \\ \text{PUSH}(d - \rho x); \text{ADD}; \text{ADD}$$

x e y avaliados já

- Para garantir isso, precisamos que C retorne uma informação extra que diga quando seu resultado foi avaliado.
- Mas isso já está disponível para nodos na forma de σ , que grava quais posições da pilha foram avaliadas, e daí a otimização será facilmente incorporada.

Manipulação de Valores Básicos...

- Considere a seguinte definição de função

$$f \ x \ y = + \ x \ (+ \ y \ 1)$$

- Compilará em

$$\text{PUSHINT } 1; \text{PUSH2}; \text{ADD}; \text{PUSH1}; \text{ADD}; \text{UPDATE3}; \text{POP2}; \text{RETURN}$$

- Numa implementação que usa representação *boxed* para inteiros o primeiro ADD

1. tomará dois inteiros (y e 1) de suas caixas;
2. somará esses dois inteiros;
3. alocará uma nova caixa; e
4. colocará o resultado na nova caixa;

...Manipulação de Valores Básicos...

- O segundo ADD vai tirar o resultado da caixa e somá-lo a x .
- Assim a alocação da caixa e a ação de colocar o resultado parcial dentro dela foram perdidas.
- A ineficiência acima acontece quando manipulamos valores básicos como inteiros, caracteres, booleanos e etc.
- Um valor básico sem uma caixa é chamado de "despido" e um valor básico dentro de uma caixa é chamado de "vestido".
- Por razões de eficiência gostaríamos de trabalhar com valores despidos o máximo possível.

...Manipulação de Valores Básicos...

- Vamos definir instruções para retirar valores de suas caixas (despi-los) e instruções para vestir valores despidos.
- A instrução GET retira o item do topo da pilha fora de sua caixa e o recoloca na pilha como um valor despido.
- A instrução MKINT coloca o elemento do topo da pilha numa caixa para inteiro, reescrevendo o resultado sobre o topo da pilha. Depois redefiniremos as operações que operam sobre valores básicos, para que elas operem com valores despidos ao invés de valores vestidos.
- Como podemos compilar nossos programas para usar tais instruções?
- Começaremos definindo um novo esquema de compilação B, que é igual a E, exceto que deixa o resultado na pilha como um valor despido.

...Manipulação de Valores Básicos...

- Naturalmente, ela assume que o resultado seja um valor básico.
- O esquema B completo pode ser visto no final do capítulo.
- A instrução `PUSHBASIC i` empilha um valor básico despido na pilha. Isso implica que essa instrução serve para empilhar valores básicos de qualquer tipo.
- A instrução `JFALSE` deve ser alterada de modo a esperar que seus argumentos sejam valores básicos.
- Se B não reconhecer a expressão que ele está compilando, ele a avalia com E e depois retira o valor de sua caixa, com um `GET`.
- Tudo o que resta é modificar E e R para que usem B.
- Eles usarão B em todos os contextos, onde o resultado é reconhecidamente um valor básico.

...Manipulação de Valores Básicos...

- Ambos R e E usarão B para avaliar a condição de um IF.
- E usa B para calcular os resultados de todas as operações aritméticas, seguido de um `MKINT` para vesti-los.
- Finalmente, R tem uma otimização para quando o resultado final da redução do supercombinador é um inteiro.
- Nesse caso R usa B para calcular o inteiro despido, e depois usa `UPDINT d` para atualizar a raiz do *redex* com o valor vestido.

...Manipulação de Valores Básicos...

- O único problema restante com essa otimização diz respeito à coleta de lixo.
- Quando a coleta é iniciada, o coletor de lixo tem que percorrer todo o grafo acessível, incluindo a parte acessível somente a partir da pilha.
- Isso significa que o coletor de lixo precisa saber se um item na pilha é ou não um ponteiro.
- Infelizmente, agora a pilha contém valores despídos e valores vestidos.
- O problema todo é que um valor despido e um ponteiro são indistinguíveis.

...Manipulação de Valores Básicos...

- Existem quatro????? soluções possíveis para o problema:
 1. Marcar quais são os valores despídos da pilha.
 2. Isso jogaria fora todo esforço de otimização até agora, pois essa solução seria equivalente a vestir o valor despido.
 3. Deixar o coletor de lixo tratar os valores básicos despídos, como ponteiros e tratar qualquer estrutura, que seja acessível acidentalmente a partir deles, como estando em uso.
 4. Essa solução corre o risco de não recuperar alguma memória que não esteja em uso.
 5. Usar duas pilhas ao invés de uma. Uma pilha, *V*, para valores despídos e a pilha da espinha, *S*, para valores vestidos.
 6. É fácil decidir a qual pilha uma instrução deve se referir. O problema é que teremos ainda uma outra pilha.

...Manipulação de Valores Básicos...

7. Empilhar os valores básicos no *dump*! Esse é um truque esperto, usado pela máquina-G de Chalmers. Ele é baseado em duas premissas:
- 7.1. O coletor de lixo não precisa seguir ponteiros a partir do *dump*, já que toda memória acessível pode ser marcada a partir da pilha da espinha.
 - 7.2. Nos momentos em que formos recuperar a pilha antiga e os ponteiros para código a partir do *dump*, ou referir à pilha antiga para checar se o supercombinador corrente tem argumentos suficientes, sabemos que não existe nenhum valor básico no *dump*.

...Manipulação de Valores Básicos...

8. Vimos que podemos combinar seguramente as pilhas *V* e *D*.
9. Essa nos parece a melhor opção. Se for realmente usada, então os casos para o *let* e o *letrec* do esquema B podem terminar com POP ao invés de SLIDE, porque os valores despidos não estão na pilha *S*, mas sim no *dump*.

Otimizações Locais ao Código-G

- Um otimizador local (*Peephole*) atua entre o compilador de código-G e o gerador de código.
- Ele observa uma sequência pequena de instruções do código-G e a substitui por uma sequência menor ou mais otimizada.

Combinação de SLIDEs e MKAPMúltiplos

- Esse tipo de otimização caracteriza bem o uso dos otimizadores locais.
- A otimização pode ser descrita como:

$$\text{SLIDE } k_1; \text{ SLIDE } k_2; \Rightarrow (\text{SLIDE } k_1 + k_2)$$
- As sequências de MKAP também podem ser combinadas numa instrução MKAP *n*,

$$\text{MKAP } k_1; \text{ MKAP } k_2 \Rightarrow \text{MKAP}(k_1 + k_2);$$
 onde um simples MKAP é equivalente a MKAP 1.

Evitando EVALs Redundantes

- Seja a seguinte sequência:
PUSHGLOBAL f; EVAL
- O EVAL é redundante se f for uma função *built-in*, ou um supercombinador de um ou mais argumentos.
- Ele é necessário somente se f for uma CAF.
- O otimizador local pode eliminar o EVAL se ele for redundante e se f não for uma CAF:
PUSHGLOBAL f EVAL \Rightarrow PUSHGLOBAL f

Evitando Alocar a Raiz do Resultado...

- Considere o supercombinador
 $\$F \ x \ f = f \ x$
- Com as técnicas fornecidas até agora geraríamos o seguinte código:

PUSH 0;\$	Empilha x
PUSH 2;\$	Empilha f
MKAP;	Constrói um nodo de aplicação
UPDATE 3;	Atualiza a raiz do redex
POP 2;	Desempilha os parâmetros
UNWIND;	Continua
- Numa implementação que utiliza cópia para a instrução UPDATE, esse código gera muito desperdício, pois ele aloca uma célula de aplicação com MKAP e depois a copia imediatamente sobre a raiz do *redex*, descartando a célula recém alocada.

...Evitando Alocar a Raiz do Resultado...

- Seria melhor construir o resultado diretamente sobre o topo da pilha:

PUSH 0;	Empilha x
PUSH 2;	Empilha f
UPDAP 4;	Constrói a aplicação sobre a raiz
POP 2;	Desempilha os parâmetros
UNWIND;	Continua
- A instrução UPDAP 4 pega os dois itens do topo da pilha, e usando-os, constrói um nodo de aplicação no topo da raiz do *redex*, cuja posição é a quarta a partir do topo da pilha.
- Podemos modificar o esquema RS no caso de uma função f , para incorporar essa otimização.

...Evitando Alocar a Raiz do Resultado

- A mesma otimização pode ser feita quando o resultado da função é uma célula CONS (usaremos a instrução UPDCONS).
- De uma maneira geral o que estamos fazendo é:

MKAP n; UPDATE d;	\Rightarrow MKAP(n-1); UPDAP(d+1)
CONS n; UPDATE d;	\Rightarrow CONS(n-1); UPDCONS(d+1)

Descompactando Objetos Estruturados...

- A otimização de expressões-case, resultava em ocorrências frequentes de expressões como:

```
let  $v_1 = \text{SEL-SUM-k-1 } v$ 
...
 $v_k = \text{SEL-SUM-k-k } v$ 
in  $E$ 
```

onde v e v_i são variáveis.

- Se essa expressão fosse compilada pelo esquema R num contexto onde v já está avaliado, teríamos o seguinte código-G:

```
PUSH( $d - \rho v$ ); SELSUM  $k, 1$ ;
...
PUSH( $d+k-1 - \rho v$ ); SELSUM  $k, k$ ;
R  $[[E]] \rho' (d+k)$ 
```

onde $\rho' = \rho[v_1 = d + 1, \dots, v_n = d + k]$.

...Descompactando Objetos Estruturados

- Essa sequência de instruções PUSH/SELSUM simplesmente descompacta v na pilha e pode ser otimizada para:

PUSH($d - \rho v$); Unpack-Sum- k ;

onde UNPACKSUM k é uma nova instrução do código-G, que descompacta o elemento do topo da pilha em k componentes, colocando-os no topo da pilha.

- Tudo nessa seção também se aplica a tipos produtos, substituindo Sel- k - i e SELPRODUCT k, i ao invés de SEL-SUM- k - i e SELSUM k, i .

Pattern-Matching Revisado

- A otimização local para UNPACK mostrada acima, põe um ponto final na nossa estratégia para compilar o *pattern-matching*.

- Uma função que usa o *pattern-matching* será compilada para:

1. uma sequência de código para avaliar o argumento;
2. uma salto de multi-caminhos, baseado no *tag* de estrutura do argumento;
3. uma instrução UNPACK, que descompacta os elementos da estrutura na pilha;'
4. uma sequência de código para avaliar o lado direito apropriado função, no contexto correto.

Sumário... 1

- R $[[E]] \rho d$ gera código para aplicar um supercombinador aos seus d argumentos

R $[[i]] \rho d$	= B $[[i]] \rho d$; UPDINT ($d + 1$); POP d ; RETURN;
R $[[f]] \rho d$	= E $[[f]] \rho d$; UPDATE; ($d + 1$); POP d ; RETURN;
R $[[x]] \rho d$	= E $[[x]] \rho d$; UPDATE; ($d + 1$); POP d ; RETURN;
R $[[\text{NEG } E]] \rho d$	= B $[[\text{NEG } E]] \rho d$; UPDINT ($d + 1$); POP d ; RETURN;
R $[[+ E_1 E_2]] \rho d$	= B $[[+ E_1 E_2]] \rho d$; E $[[x]] \rho d$; UPDATE; ($d + 1$); POP d ; RETURN;
R $[[\text{CONS } E_1 E_2]] \rho d$	= E $[[\text{CONS } E_1 E_2]] \rho d$; E $[[x]] \rho d$; UPDINT ($d + 1$); POP d ; RETURN;
R $[[\text{HEAD } E]] \rho d$	= E $[[\text{HEAD } E]] \rho d$; E $[[x]] \rho d$; UPDINT ($d + 1$); POP d ; RETURN;
R $[[\text{IF } E_c E_t E_f]] \rho d$	= B $[[E_c]] \rho d$; JFALSE L ; R $[[E_t]] \rho d$; LABEL L ; R $[[E_f]] \rho d$;
R $[[E_1 E_2]] \rho d$	= RS $[[E_1 E_2]] \rho d 0$;
R $[[\text{let } x = E_x \text{ in } E]] \rho d$	= C $[[E_x]] \rho d$; R $[[E]] \rho [x = d + 1] (d + 1)$
R $[[\text{let } x! = E_x \text{ in } E]] \rho d$	= E $[[E_x]] \rho d$; R $[[E]] \rho [x = d + 1] (d + 1)$
R $[[\text{letrec } D \text{ in } E]] \rho d$	= CLetrec $[[D]] \rho' d$; R $[[E]] \rho' d$ onde $(\rho', d) = \text{Xr}[[D]] \rho d$;

...Sumário...2

- $RS [E] \rho d n$ completa uma redução de supercombinador, na qual as n costelas do topo já foram colocadas na pilha.
- RS constrói instâncias das costelas de E , colocando-as na pilha e depois completa a redução da mesma forma que R .

$RS [f] \rho d n$	= PUSHGLOBAL f ; MKAP n ; UPDATE; $(d - n + 1)$; POP $(d - n)$; UNWIND;
$RS [x] \rho d n$	= PUSH $(d - \rho x)$; MKAP n ; UPDATE; $(d - n + 1)$; POP $(d - n)$; UNWIND;
$RS [HEAD E] \rho d n$	= $E[E] \rho d$; HEAD; MKAP n ; UPDATE; $(d - n + 1)$; POP $(d - n)$; UNWIND;
$RS [IF E_c E_t E_f] \rho d n$	= $B[E_c] \rho d$; JFALSE L_1 ; $RS [E_t] \rho d n$; LABEL L_1 ; $RS [E_f] \rho d n$;
$RS [IF E] \rho d n$	= $B[E] \rho d$; JFALSE L ; $RS [K-2-1] \rho d n$; LABEL L ;
	$RS [K-2-2] \rho d n$;
$RS [E_1 E_2] \rho d n$	= $C[E_2] \rho d$; $RS [E_1] \rho (d + 1) (n + 1)$;
$RS [E_1 ! E_2] \rho d n$	= $E[E_2] \rho d$; $RS [E_1] \rho (d + 1) (n + 1)$;

- **Obs:** RS não pode encontrar nem let e nem $letrec$.

...Sumário...3

- $E[E] \rho d$ avalia E , deixando o resultado no topo da pilha

$E[i] \rho d$	= PUSHINT i ;
$E[f] \rho d$	= PUSHGLOBAL f ; EVAL
$E[x] \rho d$	= PUSH $(d - \rho x)$; EVAL
$E[NEG E] \rho d$	= $B[NEG E] \rho d$; MKINT
$E[+ E_1 E_2] \rho d$	= $B[+ E_1 E_2] \rho d$; MKINT
$E[CONS E_1 E_2] \rho d$	= $C[E_2] \rho d$; $C[E_1] \rho (d + 1)$; CONS
$E[HEAD E] \rho d$	= $E[E] \rho d$; HEAD; EVAL
$E[IF E_c E_t E_f] \rho d$	= $B[E_c] \rho d$; JFALSE L_1 $E[E_t] \rho d$; JUMP L_2 ;
	LABEL L_1 ; $E[E_f] \rho d$; LABEL L_2 ;
$E[E_1 E_2] \rho d$	= ES $[E_1 E_2] \rho d 0$;
$E[let x = E_x in E] \rho d$	= $C[E_x] \rho d$; $E[E] \rho [x = d + 1] (d + 1)$; SLIDE 1;
$E[let x! = E_x in E] \rho d$	= $E[E_x] \rho d$; $E[E] \rho [x = d + 1] (d + 1)$; SLIDE 1;
$E[letrec D in E] \rho d$	= $C[letrec D] \rho d$; $E[E] \rho d$; SLIDE $(d - d)$;
	onde $(\rho', d') = Xr[D] \rho d$;

...Sumário...4

- $ES [E] \rho d n$ completa a avaliação de uma expressão, as n costelas dela já foram colocadas na pilha.
- ES constrói instâncias das costelas de E , colocando-as na pilha e completa a avaliação da mesma forma que E .

$ES [f] \rho d n$	= PUSHGLOBAL f ; MKAP n ; EVAL
$ES [x] \rho d n$	= PUSH $(d - \rho x)$; MKAP n ; EVAL
$ES [HEAD E] \rho d n$	= $E[E] \rho d$; HEAD; MKAP n ; EVAL
$ES [IF E_c E_t E_f] \rho d n$	= $B[E_c] \rho d$; JFALSE L_1 ; $ES [E_t] \rho d n$; JUMP L_2 ;
	LABEL L_1 ; $ES [E_f] \rho d n$; LABEL L_2 ;
$ES [IF E] \rho d n$	= $B[E] \rho d$; JFALSE L_1 ; $ES [K - 2 - 1] \rho d n$; JUMP L_2 ;
	LABEL L_1 ; $RS [K - 2 - 2] \rho d n$; LABEL L_2 ;
$ES [E_1 E_2] \rho d n$	= $C[E_2] \rho d$; $ES [E_1] \rho (d + 1) (n + 1)$;
$ES [E_1 ! E_2] \rho d n$	= $E[E_2] \rho d$; $ES [E_1] \rho (d + 1) (n + 1)$;

...Sumário...5

- $B[E] \rho d$ avalia E , deixando o resultado no topo da pilha como um valor básico despido.

$B[i] \rho d$	= PUSHBASIC i ;
$B[NEG E] \rho d$	= $B[E] \rho d$; NEG;
$B[+ E_1 E_2] \rho d$	= $B[E_2] \rho d$; $B[E_1] \rho (d + 1)$; ADD;
$B[IF E_c E_t E_f] \rho d$	= $B[E_c] \rho d$; JFALSE L_1 $B[E_t] \rho d$; JUMP L_2 ;
	LABEL L_1 ; $B[E_f] \rho d$; LABEL L_2 ;
$B[let x = E_x in E] \rho d$	= $C[E_x] \rho d$; $B[E] \rho [x = d + 1] (d + 1)$; SLIDE 1;
$B[letrec D in E] \rho d$	= $C[letrec D] \rho d$; $B[E] \rho d$; SLIDE $(d - d)$;
	onde $(\rho', d') = Xr[D] \rho d$;
$B[E] \rho d$	= $E[E] \rho d$; GET; (em qualquer outro caso)

III. OTIMIZAÇÃO DA RECURSIVIDADE DE CAUDA

Recursividade de Cauda

- Suponha que estejamos compilando o corpo do seguinte supercombinador

$$\$F \ x \ y = W \ E_1 \ E_2 \ E_3$$

onde W é ou um supercombinador, ou uma função *built-in*, ou uma variável (somente x ou y seriam possível nesse caso).

- Produziremos código-G para construir uma instância do corpo de $\$F$. Contudo, no final deste código está uma instrução UNWIND que vai, pela pilha, descer a espinha da instância.
- Quando fizermos a redução de W , todas as vértebras recentemente alocadas abaixo da raiz do *redex* W virarão lixo imediatamente.

... Recursividade de Cauda ...

- Esse capítulo é devotado para técnicas projetadas para evitar alocação de vértebras que tornarão lixo logo em seguida. Durante esse capítulo, vamos supor o supercombinador $\$F$ como exemplo corrente.

- Suponha que W fosse um supercombinador ou uma função *built-in*. O código para F começaria da seguinte maneira:

```
C [[E3]]ρ d;  
C [[E2]]ρ (d + 1);  
C [[E1]]ρ (d + 2);  
PUSHGLOBALW;
```

- Se W fosse uma variável, então a única diferença é que a última instrução seria um PUSH ao invés de um PUSHGLOBAL.

... Recursividade de Cauda ...

- Isso coloca todas as costelas na pilha, mas não constrói nenhuma vértebra (o que será feito a seguir com uma instrução MKAP 3).
- Depois dessa sequência ter executado, o contexto corrente se parece com a Figura 2101:

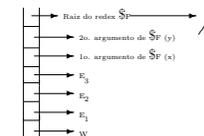


Fig 2101

- Nessa figura, todo o grafo do *redex* $\$F$ foi omitido, exceto a raiz, a qual é sempre um nodo de aplicação.

... Recursividade de Cauda

- Nesse ponto existe um número de casos a considerar, dependendo da natureza de W .
- Antes, vamos considerar um caso especial importante, a chamada de cauda. Em seguida generalizaremos esse caso especial.

Chamadas de Cauda...

- Uma chamada de cauda é o caso quando o resultado de uma função é dado por uma chamada a outra função com exatamente o mesmo número de argumentos fornecidos.
- No nosso exemplo, a chamada para W é uma chamada de cauda, se W for um supercombinador que toma exatamente três argumentos.
- Sob essas circunstâncias o corpo de $\$F (W E_1 E_2 E_3)$ é ele próprio um *redex* - ainda mais, será o próximo *redex* a ser reduzido.
- Além disso, o nodo que será atualizado pelo resultado de fazer uma redução de W é o mesmo nodo que será atualizado pelo resultado de uma redução de $\$F$.

...Chamadas de Cauda...

- Na entrada do código para o supercombinador W o corrente corrente seria como o da figura
- Uma forma de passar da figura para a figura seria completar a construção do grafo $(W E_1 E_2 E_3)$ no *heap*, atualizar a raiz do *redex* com o resultado, desempilhar os parâmetros de $\$F$ e executar um UNWIND.
- Isso desceria a espinha pela pilha, encontraria W no cóccix, rearranjaria a pilha para parecer como a figura e finalmente entrar com o código 4w4.
- Isso é exatamente o que nosso algoritmo desenvolvido anteriormente faz.

...Chamadas de Cauda...

- Porém é uma maneira muito ingênua de se proceder.

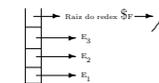


Fig 2102

- Uma maneira muito mais eficiente de passar da Figura 2101 para a Figura 2102 seria passar os quatro elementos do topo da pilha para baixo, sobrescrevendo os dois argumentos de $\$F$.
- Escrevemos a instrução
SQUEEZE 4 2
significando: mova os 4 elementos do topo da pilha, duas posições para baixo.

...Chamadas de Cauda...

- A regra para SQUEEZE é:
 $\langle n_1 : \dots : n_k : m_1 : \dots : m_d : S, G, \text{SQUEEZE } k \ d : C, D \rangle$
 $\Rightarrow \langle n_1 : \dots : n_k : S, G, C, D \rangle$
- Depois de feito isso, queremos inventar um novo código para W , então inventaremos outra nova instrução JFUN, que espera encontrar uma função no topo da pilha, desempilhá-lo e entrar com seu código.
- A definição completa de \$F seria,
 $C \llbracket E_3 \rrbracket \rho \ d; C \llbracket E_2 \rrbracket \rho \ (d + 1); C \llbracket E_1 \rrbracket \rho \ (d + 2);$
 PUSHGLOBAL W ; SQUEEZE 4 2; JFUN
- JFUN deveria entrar com o código depois da checagem de aridade e rearranjo da pilha, ou seja, deveria entrar na entrada EXEC.

...Chamadas de Cauda

- Esse código traz uma série de economias à abordagem anterior:
 - as vértebras do resultado da redução de \$F nunca serão alocadas;
 - não é necessária nenhuma atualização no final do código de \$F porque o código de W vai atualizar o mesmo nodo;
 - o SQUEEZE substitui o POP na hora de ficarmos livres dos parâmetros de \$F;
 - não é necessário nenhum UNWIND pois ele já está feito;
 - não precisamos de checar se W tem parâmetros suficientes, pois já fizemos isso em tempo de compilação.
- Porém, só obtemos esses benefícios se
 - soubermos o que W é;
 - W tomar o número exato de argumentos.

Generalizando Chamadas de Cauda...

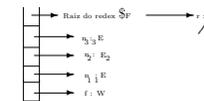
- A otimização da seção anterior se aplicava somente quando sabíamos em tempo de compilação, o que W era, e ainda se ele fosse um supercombinador com três argumentos.
- Generalizamos essa idéia para qualquer W substituindo a função JFUN no final do código de \$F por uma nova instrução
DISPATCH 3
- O argumento 3 de DISPATCH dá o número de costelas que estão na pilha.

Generalizando Chamadas de Cauda...

- O código para \$F seria

$C \llbracket E_3 \rrbracket \rho \ d; C \llbracket E_2 \rrbracket \rho \ (d+1); C \llbracket E_1 \rrbracket \rho \ (d+2);$
 PUSHGLOBAL W ; SQUEEZE 4 2; DISPATCH 3;

sem se preocupar com que seja W (exceto que o PUSHGLOBAL deveria ser um PUSH caso W fosse uma variável).



- A figura acima mostra o contexto corrente no momento em que a instrução DISPATCH é executada.

...Generalizando Chamadas de Cauda...

- Nós anotamos os nodos com um rótulo para facilitar seguir as regras de DISPATCH.
- Por exemplo, a raiz do *redex* \$F será *r* nas regras para DISPATCH.
- Quando a instrução DISPATCH 3 for executada ela terá que fazer uma análise de caso na função que está no topo da pilha.

...Generalizando Chamadas de Cauda...

- Existem várias possibilidades:
 1. W é um nodo de aplicação;
 2. W é um supercombinador com zero argumentos;
 3. W é uma função (supercombinador ou *built-in*) com exatamente três argumentos;
 4. W é uma função (supercombinador ou *built-in*) com menos que três argumentos;
 5. W é uma função (supercombinador ou *built-in*) com mais que três argumentos;
- Veremos cada caso em separado nas seções seguintes.

W é um Nodo de Aplicação

- Se W é um nodo de aplicação, então em princípio não sabemos nada sobre quantos argumentos W leva. Assim, tomaremos o seguinte caminho:
 1. construir a espinha do corpo de \$F;
 2. atualizar a raiz do *redex* \$F;
 3. UNWIND .
- Contudo, podemos fazer uma otimização.
- Ao invés de construir a espinha no *heape* então e depois descer de volta pela pilha, nodos podemos executar a primeira parte do UNWIND à medida em que construímos a espinha. Quando o DISPATCH foi feito o contexto fica da seguinte forma:



- Agora DISPATCH se comporta exatamente como UNWIND.

- Podemos formalizá-lo da seguinte maneira:

$$\langle f : n_1 : n_2 : \dots : n_k : r : S, G[f = AP \ m_1 \ m_2], DISPATCH \ k : [], D \rangle$$

$$\Rightarrow \langle f : v_1 : v_2 : \dots : v_{k-1} : r : S,$$

$$G \left[\begin{array}{l} v_1 = AP \ f \ n_1 \\ v_i = AP \ v_{i-1} \ n_i, \ (1 < i < k) \\ r = AP \ v_{k-1} \ n_k \end{array} \right], UNWIND : [], D \rangle$$

onde nodo r é a raiz do *redex* corrente nessa regra e as demais regras DISPATCH e os nodos v_i são nodos vertebrais.

W é um Supercombinador sem Argumentos

- Se W é um supercombinador com zero argumentos, não podemos melhorar o caso anterior
- Por isso DISPATCH deve atuar da mesma forma como se W fosse um nodo de aplicação.
- Se W é uma função com três argumentos então temos o caso de chamada de cauda
- E DISPATCH pode simplesmente entrar o código para W .
- Podemos expressar isso com a seguinte regra:
 $\langle f:S, G[f=FUN\ k\ C], DISPATCH\ k: [], D \rangle$
 $\Rightarrow \langle S, G, C, D \rangle$

Função W Têm Menos que 3 Argumentos...

- Se W for uma função com menos que três argumentos então uma parte do corpo de $\$F$ será o próximo *redex* a ser reduzido.
- Suponha que W tome dois argumentos.
- Então o que queremos é criar um novo contexto no qual W vai executar, com seus dois argumentos no topo da pilha e um ponteiro para a raiz do *redex* W abaixo deles.
- Podemos conseguir esse contexto, construindo somente na parte de cima da espinha do corpo de $\$F$.
- Abaixo temos a pilha logo antes de W entrar o código para W :



...Função W Têm Menos que 3 Argumentos

- O contexto para a redução de W consiste dos três elementos do topo da pilha.
- A célula HOLE deve ser alocada para receber o resultado da redução de W .
- Eis a regra formal:
 $\langle f:n_1 : n_2 : \dots : n_k:r:S, G[f=FUN\ a\ c], DISPATCH\ k: [], D \rangle$
 $(a < k) \Rightarrow \langle n_1 : \dots : n_a : v_a : \dots : v_{k-1}:r:S,$
 $G \left[\begin{array}{l} v_a = HOLE \\ v_i = AP\ v_{i-1}\ n_i, \ (a < i < k) \\ r = AP\ v_{k-1}\ n_k \end{array} \right], C, D \rangle$

Função W Tem mais que Três Argumentos...

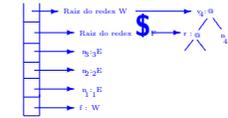
- Se W for uma função com mais que três argumentos, o corpo de $\$F$ está na WHNF, e nodos devemos atualizar a raiz do *redex* $\$F$ para refletir este fato, desde que ele possa ser compartilhado.
- Isso envolve construir a espinha no *heap* assim como fizemos para o caso de quando W era um nodo de aplicação.
- Contudo, a próxima coisa que acontecerá é tentar reduzir a aplicação de W .
- Somente se existir argumentos suficientes na pilha, a redução acontecerá.

...Função W Tem mais que Três Argumentos...

- Isso nos dá a pista do que DISPATCH deveria fazer.
- Tendo construído a espinha e atualizado a raiz do *redex* $\$F$, DISPATCH deve testar a profundidade da pilha.
- Se não houver argumentos suficientes para a redução de $\$W$, então a avaliação está completa e DISPATCH pode iniciar um *RETURN*.
- Se existem argumentos suficientes então DISPATCH pode rearranjar a pilha para W e entrar com W .

...Função W Tem mais que Três Argumentos...

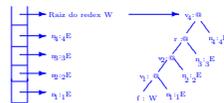
- Suponha que W tome 4 argumentos, e assim antes de DISPATCH a pilha estaria assim,



- Isso é simplesmente uma versão aumentada da figura mostrando um elemento da pilha abaixo do contexto no qual $\$F$ executa.

...Função W Tem mais que Três Argumentos...

- Nesse caso queremos que DISPATCH arrume a pilha para



- Temos duas regras para DISPATCH cobrir esse caso.

...Função W Tem mais que Três Argumentos...

- A primeira cobre o caso quando não existem argumentos suficientes para a função reduzir, então a avaliação está completa e retornamos para o chamador.

$$\langle f : n_1 : n_2 : \dots : n_k : r : v_{k+1} : \dots : v_d : [], G[f = \text{FUN } a\ c], \text{DISPATCH } k : [], (S, C) : D \rangle$$

$$(k < d < a) \Rightarrow \langle v_d : S, G \left[\begin{array}{l} v_1 = AP - \text{WHNF } f\ n_1 \\ v_i = AP - \text{WHNF } v_{i-1}\ n_i, (1 < i < k) \\ r = AP - \text{WHNF } v_{k-1}\ n_k \end{array} \right], C', D \rangle$$

- Nessa regra, k é o argumento para DISPATCH, A é a aridade da função no topo da pilha e D é o número de argumentos disponíveis.
- Veja é conhecido que as vértebras v_1, \dots, v_{k-1} estão na WHNF, e assim podemos construí-las como nodos do tipo AP-WHNF.

...Função W Tem mais que Três Argumentos...

- A segunda regra cobre o caso quando existe argumentos suficientes e e rearranjo mostrado no diagrama anterior acontece, seguido de um salto para o código da função.

$$\langle f:n_1:n_2:\dots:n_k:r:v_{k+1}:\dots:v_a:S, \\ G \left[\begin{array}{l} f = \text{FUN } a \ c \\ v_{k+1} = \text{AP } r \ n_{k+1} \\ v_i = \text{AP } v_{i-1} \ n_i, (k+1 < i \leq a) \end{array} \right], \text{ DISPATCH } k:[], D \rangle$$

$$(k < a) \Rightarrow \langle n_1:\dots:n_k:\dots:n_a:v_a:S, \\ G \left[\begin{array}{l} v_1 = \text{AP-WHNF } f \ n_1 \\ v_i = \text{AP-WHNF } v_{i-1} \ n_i, (1 < i < k) \\ r = \text{AP-WHNF } v_{k-1} \ n_k \end{array} \right], C, D \rangle$$

Esquemas de Compilação para DISPATCH

- Nessa seção discutiremos os esquemas de compilação e geração de código necessária para o uso da instrução DISPATCH.
- É simples gerar código para usar a instrução DISPATCH, através da substituição de duas regras no esquema RS abaixo.
- Essa é a razão pela qual caímos no problema de desenvolver o esquema RS:

$$\text{RS}[X]\rho \ d \ n = \text{PUSH}(d - \rho \ x); \text{SQUEEZE}(n+1)(d-n); \text{DISPATCH } n;$$

$$\text{RS}[f]\rho \ d \ n = \text{PUSHGLOBAL } f; \text{SQUEEZE}(n+1)(d-n); \text{DISPATCH } n;$$

Otimização de DISPATCH em Tempo de Compilação

- Tudo é que precisa é o gerador de código reconhecer a sequência $\text{PUSHGLOBAL } \$H; \text{SQUEEZE } p \ q; \text{DISPATCH } k;$
- Agora o gerador de código pode fazer muito da análise de caso em $\$H$ que seria feita em tempo de execução.
- Por exemplo, ele pode observar que $\$H$ toma exatamente k argumentos, em cujo caso temos uma chamada de cauda e podemos gerar código para saltar diretamente para o código de $\$H$.
- Isso nos forneceria o efeito que obtivemos nas chamadas de cauda.
- Tal salto deveria ser a entrada EXEC da função, depois do teste de aridade e da arrumação da pilha.

...Otimização de DISPATCH em Tempo de Compilação...

- Em casos particulares podemos fazer ainda melhor. Por exemplo, $\text{PUSHGLOBAL } \$CONS; \text{SQUEEZE } 3 \ q; \text{DISPATCH } 2$
- Pode ser otimizada para $\text{CONS UPDATE}(q+1); \text{POP } q; \text{RETURN};$
- Isso corresponde exatamente à otimização do $CONS$ no esquema R, mas transferido para uma otimização local no gerador de código.
- Todos os casos especiais em R podem ser movidos para o gerador de código, mas perderemos oportunidades para o uso de B.
- Na prática usaremos ambos os métodos.

...Otimização de DISPATCH em Tempo de Compilação

- O caso difícil é quando encontramos
PUSH n; SQUEEZE p q; DISPATCH k;
- ou seja, o empilhamento de uma variável.
- Nesse caso o gerador de código não pode fazer nenhuma análise de caso em tempo de compilação.
- Usando a análise de caso mostrada no capítulo de definição do código-G, nodos adicionaríamos uma entrada DISPATCH para cada tabela de entrada de *tag*.
- O código VAX para DISPATCH 4 poderia ser:

```

moval 3,r2      k é passado para o código de DISPATCH em r2
movl  (%EP)+,r0  Função POP em r0
movl  (r0),r1    Tag em r1
jmp   0_Dispatch(r1) Salto da análise de caso

```

Otimizando o Esquema E...

- As otimizações que aplicamos ao esquema RS podem ser igualmente aplicadas para o esquema ES.
- Assim como o esquema RS, o esquema ES constrói a espinha da expressão e depois faz um UNWIND nela.
- A figura acima fornece a modificação necessária.
- Primeiro ALLOCamos uma célula HOLE para conter o resultado (para o esquema RS isso já está presente na forma da raiz do *redex*).
- Depois construímos as costelas usando ES empilhando-as na pilha da espinha.

...Otimizando o Esquema E...

- Finalmente usamos uma nova instrução do código-G, CALL, para terminar o trabalho.
- A única diferença entre RS e ES é esse CALL no fim, ao invés da sequência SQUEEZE-DISPATCH.
- Novo E:

$$E[E_1 E_2] \rho d = \text{ALLOC } 1; \text{ ES}[E_1 E_2] \rho d 0;$$

- Novo ES:

$$\text{ES}[x] \rho d n = \text{PUSH } (d - \rho x); \text{ CALL } n$$

$$\text{ES}[x] \rho d n = \text{PUSHGLOBAL } f; \text{ CALL } n$$

...Otimizando o Esquema E...

- CALL é muito parecida com DISPATCH, exceto que primeiro ela salva a pilha e os ponteiros para código no *dump*
- Assim como EVAL é muito parecido com UNWIND, exceto que ela salva a pilha e os ponteiros para código primeiro.
- A regra para CALL é:

$$\langle f : n_1 : n_2 : \dots : n_k : r : S, G, \text{CALL } k : C, D \rangle$$

$$\Rightarrow \langle f : n_1 : n_2 : \dots : n_k : r : [], G, \text{DISPATCH } k : [], (S, C) : D \rangle$$

...Otimizando o Esquema E

- Os usos de CALL podem ser otimizados por um otimizador local da mesma forma que DISPATCH, exceto que ainda mais otimizações são possíveis.
- Por exemplo, a sequência

```
PUSHGLOBAL $H; CALL k
```

onde \$H toma mais que k argumentos, pode ser otimizada para

```
PUSHGLOBAL $H; MKAP k; SLIDE 1;
```

Comparações com Implementações Baseadas em Ambientes

- Nessa seção faremos uma breve comparação da nossa máquina-G final com a Máquina Funcional Abstrata de Cardelli (FAM)
- FAM é baseada em substituição *lazy*, na qual a aplicação de uma função não é feita pela construção da instância do corpo de uma função, mas sim pela avaliação do corpo da função num ambiente no qual os parâmetros formais estão ligados aos seus valores reais.
- O ambiente é um estrutura de dado que mantém os valores de todas as variáveis no escopo corrente.
- Se o resultado da avaliação da função é uma outra função, então um *closure* é retornado. O *closure* é um par consistindo de:
 - o código da função;
 - o ambiente na qual ela será executada posteriormente.

FAM e Máquina SECD...

- Essa é a abordagem da máquina SECD, e FAM pode ser considerada uma máquina SECD otimizada:
- O código da máquina SECD é geralmente implementado pela interpretação direta do código da máquina abstrata.
- FAM tem um código de máquina abstrata mais poderoso e é compilado em código de máquina (VAX).
- O ambiente da máquina SECD é geralmente implementado como uma lista encadeada e os *closures* como pares de ponteiros para o código e o ambiente.

...FAM e Máquina SECD

- FAM constrói *closures* como uma (N+1)-tupla, na qual o primeiro elemento aponta para o código da função, e os outros N elementos são os valores somente daquelas variáveis que ocorrem livres na definição da função.
- A pilha e o *dump* da máquina SECD são geralmente implementados como uma lista encadeada.
- FAM usa as pilhas da máquina destino, chamadas de AS (pilha de argumentos) e (RS) (pilha de retorno) respectivamente em [Cardelli, 1984].

FAM e Máquina-G

- Tendo dito isso, há um forte correspondência entre FAM e a máquina-G:
- O equivalente na máquina-G a um *closure* de FAM é um pedaço de grafo consistindo de um supercombinador aplicado a alguns argumentos.
- Os argumentos dão os valores das variáveis usadas no corpo do supercombinador.
- É uma consequência do algoritmo de *lambda-lifting* que todos os argumentos extra a uma função produzida por um *lambda-lifting* são usados em algum lugar do corpo do supercombinador.

FAM e Máquina-G

- Isso corresponde ao fato de que *closures* de FAM somente contém variáveis que podem ser requeridas na função.
- A execução é baseada em pilha na maior parte do tempo.
- Argumentos para a função corrente são encontrados na pilha.
- A diferença aqui é que FAM também pode acessar variáveis livres no ambiente, enquanto que supercombinadores não tem variáveis livres.
- Argumentos a serem passados para uma função são colocados na pilha antes da chamada da função.
- Esse é sempre o caso de FAM e as otimizações desse capítulo fazem com que esse geralmente seja o caso na máquina-G.

Diferenças entre FAM e a Máquina-G

1. FAM não é *lazy*.
2. É para preservar a *laziness* que máquina-G normalmente tem que escrever a espinha no *heap*, ao invés de mantê-la sempre na pilha.
3. A máquina-G é simplesmente uma implementação eficiente de redução de grafos.
4. Como veremos, a redução de grafos é um modelo natural para suportar a execução paralela.
5. Por isso, uma máquina-G paralela deverá ser mais fácil de implementar que FAM paralela.

III. ANÁLISE DE FUNÇÕES ESTRITAS

Estratégias de Avaliação

- Como vimos anteriormente, é útil que sejamos capazes de saber com antecedência se uma função necessariamente terá que avaliar seus argumentos.
- Nesse capítulo vamos discutir um método de análise em tempo de compilação, chamado análise de funções estritas que determina quais argumentos de uma função certamente deverão ser avaliados.
- Análise de funções estritas é uma das várias otimizações em tempo de compilação que pode ser feita através de uma interpretação abstrata do programa.

Um Exemplo Típico: a Regra dos Sinais...

- Interpretação abstrata é uma técnica para deduzir informação de um programa através da análise do texto, executando uma versão abstrata do programa. Escolhemos uma abstração apropriada de acordo com a informação desejada.
- Como exemplo, suponha que queiramos o sinal de:
 $34 * (-5) * (3993)$
- A maneira mais difícil de se fazer isso é executar as operações e verificar o sinal do resultado.

...Um Exemplo Típico: a Regra dos Sinais...

- Porém podemos propor um método mais simples que fazer um cálculo mais abstrato:
 $PLUS \% MINUS \% MINUS = PLUS$
- Nós substituímos cada número por uma representação abstrata (seu sinal), e substituímos a multiplicação por um operador abstrato $\%$, o qual implementa a conhecida regra dos sinais:
 $PLUS \% PLUS = PLUS$
 $PLUS \% MINUS = MINUS$
 ...
- A regra dos sinais dá um caminho direto de expressões aritméticas para o sinal de seus resultados, sem passar pela avaliação completa. É exatamente para isso que a interpretação abstrata se propõe.

...Um Exemplo Típico: a Regra dos Sinais

- Podemos especificar a seguinte função abstrata para obter o sinal de um número:
 $sgn :: Number \rightarrow Sign,$
- Por exemplo,
 $sgn\ 5 = PLUS$
- Podemos avaliar a expressão de multiplicações original usando a interpretação abstrata da regra dos sinais, e escrevemos:
 $Eval\% \llbracket 34 * (-5) * (-3993) \rrbracket = PLUS \% MINUS \% MINUS = PLUS$
- Usando a nova notação podemos expressar essa condição formalmente como:
 $sgn\ Eval\ \llbracket E \rrbracket = Eval\% \llbracket E \rrbracket$

Interpretação Abstrata na Análise de Funções Estritas

- Nessa seção desenvolveremos um domínio abstrato e mapeamento abstrato os quais são importantes na análise de funções estritas.
- Seja a seguinte questão: será que uma função sempre precisa do valor de seu argumento?
- Responder essa pergunta para todos os supercombinadores possibilita gerar melhor código para os supercombinadores, como vimos anteriormente e como veremos, vai possibilitar avaliar os argumentos de uma função em paralelo.

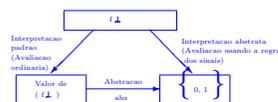
...Interpretação Abstrata na Análise de Funções Estritas...

- Como vimos a definição de uma função estrita e' : uma função é estrita se e somente se no caso da avaliação de um de seus argumentos seja infinita, então a avaliação da função também será infinita.
- A noção se estende naturalmente para funções de vários argumentos.
- Por exemplo, se g é uma função de três argumentos (x, y, z) , então dizemos que g é estrita em y se e somente se:

$$g\ x\ \perp\ z = \perp \quad \text{para qualquer } x \text{ e } z.$$

...Interpretação Abstrata na Análise de Funções Estritas...

- Podemos formalizar a questão de dada uma função f , será que $(f\ \perp) = \perp$?
- Façamos o seguinte diagrama:



- O que são o mapeamento "abstração" e o domínio abstrato?
- Queremos que a função de abstração abs distinga entre \perp e todos os outros elementos, tal que,

$$abs\ \perp = 0$$

$$abs\ x = 1 \text{ se } x \neq \perp$$

...Interpretação Abstrata na Análise de Funções Estritas

- O domínio abstrato precisa somente de dois elementos, os quais podemos arbitrariamente chamar de 0 e 1.
- Usando a notação estabelecida anteriormente, f é estrita se e somente se:

$$Evalsy[f\ \perp] = \perp$$
 o que é verdade se e somente se

$$abs\ Eval[f\ \perp] = 0$$
- Tudo o que falta é tomar uma interpretação abstrata adequada, a qual chamaremos de $Eval\#$, para distingui-la da interpretação abstrata $Eval\%$ usada para a regra do sinal.

Uma Interpretação Abstrata Adequada...

- A interpretação abstrata deve ter duas propriedades:
 1. Deve ser segura. Ela nunca deve sugerir que uma função é estrita, quando na verdade ela não é.
 2. Deve detectar o máximo possível de funções estritas.
- Como no caso da regra dos sinais, podemos dar uma expressão formal para o requisito de segurança:
 $\text{abs Eval}[E] \leq \text{Eval}\#[E]$
 para qualquer expressão E .

...Uma Interpretação Abstrata Adequada...

- A condição de segurança quer dizer que $\text{Eval}\#$ deve ter a seguinte propriedade:

$$\text{Eval}[E] = 0$$

somente se a avaliação (convencional) de E definitivamente falhar em terminar. Ou então:

$$\text{Eval}[E] = 1$$

se a avaliação convencional de E puder terminar.

...Uma Interpretação Abstrata Adequada...

- No caso de análise de funções estritas, queremos avaliar $\text{Eval}\#[f \perp]$
- Usando uma regra similar à primeira daquelas dadas para $\text{Eval}\%$, devemos proceder da seguinte maneira:
 $\text{Eval}\#[f \perp] = \text{Eval}\#[f] \text{Eval}\#[\perp]$
- Agora, certamente $\text{Eval}\#[\perp] = 0$, já que \perp certamente falha em terminar.
- Usando esse fato, juntamente com uma regra similar ao segundo $\text{Eval}\%$, temos
 $\text{Eval}\#[f \perp] = \text{Eval}\#[f] \text{Eval}\#[\perp]$
 $= f\# 0$

Desenvolvimento de $f\#$ a Partir de f

- Vamos mostrar como produzir a definição de $f\#$ a partir da definição de f com o seguinte exemplo:
 $f \text{ p } q \text{ r} = \text{IF} (= \text{p } 0) (+ \text{q } \text{r}) (+ \text{q } \text{p})$
- Tudo o que temos que fazer é tomar a interpretação abstrata do lado direito:
 $f\# \text{ p } \text{q } \text{r} = \text{Eval}\#[\text{IF} (= \text{p } 0) (+ \text{q } \text{r}) (+ \text{q } \text{p})]$
- Usando as regras da seção anterior repetidamente teremos,
 $f\# \text{ p } \text{q } \text{r} = \text{IF}\# (= \# \text{p } 0\#) (+\# \text{q } \text{r}) (+\# \text{q } \text{p})$
- Note que usamos uma regra extra:
 $\text{Eval}\#[v] = v$
 onde v é uma variável.

...Desenvolvimento de $f\#$ a Partir de f ...

- Agora, $\text{constante}\# = 1$ porque a avaliação de uma constante sempre termina
- E assim $f\# p q r = \text{IF}\# (= \# p 1) (+ \# q r) (+ \# q p)$
- Finalmente, devemos decidir o que realmente são as versões abstratas das funções *built-in*.
- Começando com a função de igualdade, sabemos que: $(= E_1 E_2)$ pode terminar se $(E_1$ puder terminar) e $(E_2$ puder terminar)
- E assim $= \# x y = \& x y$ onde definimos $\&$, como o operador booleano AND.

...Desenvolvimento de $f\#$ a Partir de f ...

- Similarmente definiremos $|$ como *OR* (no domínio abstrato).
- A definição de $+ \#$ é idêntica à de $= \#$.
- A definição de *IF* é mais interessante: $(\text{IF } E_1 E_2 E_3)$ pode terminar se $\&$ $(E_1$ puder terminar) $\&$ e $(E_2$ ou E_3 puder terminar)
- Assim, $\text{IF}\# x y z = \& x (| y z)$
- Agora podemos dar a definição final de $f\#$: $f\# p q r = \& p (\& q (| p r))$.
- Assim, para descobrir se f é estrita em r , calculamos $f\# 1 1 0 = 1$, e verificamos que f não é estrita em r .

Análise de Funções Estritas no Compilador

- Tudo que dissemos até agora assume que as funções em análise não possuem variáveis livres.
- Para contornar o problema podemos fazer a análise após o *lambda-lifting*.
- Isso faz sentido, pois são as definições de supercombinadores que queremos anotar, e não as expressões lambda originais.

Copiando com recursão

- Existe a questão de a função definida pelo usuário ser recursiva.
- Para ver que não podemos simplesmente executar a versão $\#$ da função, considere: $x y = \text{IF} (= x 0) y (f (- x 1) y)$
- A versão abstrata de f é dada por: $f\# x y = \& x (| y (f\# x y))$
- Para encontrar se f é estrita em y , avaliamos $(f\# 1 0)$, mas infelizmente essa avaliação não vai terminar.
- Isso seria um desastre já que o compilador entraria em *loop*.
- Contudo, é intuitivamente claro que f é estrita em y , e gostaríamos que o compilador deduzisse isso.

Análise Correta de uma Função Recursiva Estrita

- A única maneira correta para encontrar um ponto fixo é assegurarmos que,

$$f\#_n x y z = f\#_{n+1} x y z \quad \text{para quaisquer } x, y, z$$
- Esse teste parece ser caro, já que existe 2^3 combinações possíveis de x, y e z .
- Temos que no pior caso o custo do teste deve ser exponencial no número de argumentos, mas na prática, é necessário muito esforço para conseguir exemplos caros.
- Nos casos típicos esperamos uma rápida convergência.
- Uma abordagem promissora é desenvolver representações e heurísticas que irão trabalhar bem nos casos comuns e ainda fornecerão respostas certas (apesar que lentamente) nos casos mais difíceis.

Ordem de Análise e Recursão Mútua...

- Descrevemos como encontrar os pontos fixos de funções recursivas em si mesmo, e agora vamos estender o método para cobrir a recursão mútua.
- Considere as definições:

$$\begin{aligned} f\ x &= \dots g \dots f \dots \\ g\ y &= \dots f \dots g \dots \end{aligned}$$
- Aqui não podemos analisar uma função inteiramente antes da outra; ao invés devemos executar simultaneamente as iterações de ponto fixo, assim

$$\begin{aligned} f\#_0\ x &= 0 & g\#_0\ x &= 0 \\ f\#_1\ x &= \dots g\#_0 \dots f\#_0 \dots & g\#_1\ y &= \dots f\#_0 \dots g\#_0 \dots \\ f\#_2\ x &= \dots g\#_1 \dots f\#_1 \dots & g\#_2\ x &= \dots f\#_1 \dots g\#_1 \dots \end{aligned}$$
- É mais eficiente (e dá o mesmo resultado) usar $f\#_1$ em $g\#_1$, já que $f\#_1$ está disponível.

...Ordem de Análise e Recursão Mútua

- Suponha que a definição de uma função f envolva uma função g mas não vice-versa, assim,

$$\begin{aligned} f &= \dots g \dots f \dots \\ g &= \dots g \dots \end{aligned}$$
- Daí, seguramente podemos analisar g , encontrar o ponto fixo de $g\#$, e usar essa informação na análise subsequente de f .
- Isso se mostra muito importante quando analisamos sistemas grandes de equações
- Afinal encontrar o ponto fixo de f e g simultaneamente é muito mais caro que primeiro analisar g e depois usar essa informação para analisar f .

Extensões ao Trabalho de Mycroft e Outros Trabalhos

- O trabalho original de Mycroft estava restrito a funções de primeira ordem e domínios sem tipos de dados estruturados (Flat domains).
- Como funções de maior ordem e domínios provendo tipos de dados estruturados (que podem requerer avaliação *lazy*) são características importantes das linguagens funcionais, essas restrições eram graves e precisavam ser melhoradas.

Funções de Maior Ordem

- Brun, Hanki and Abramsky [Burn, Hankin e Abramsky, 1985] mostraram uma extensão natural das técnicas para funções de primeira ordem, que cobriam o caso para funções de maior ordem.

- Considere, por exemplo,

$$\text{hof } g \ x \ y = (g \ (\text{hof } (k \ 0) \ x \ (- \ y \ 1))) + \\ (\text{if } (= \ y \ 0) \ x \ (\text{hof } I \ 3 \ (- \ y \ 1)))$$

onde

$$K \ x \ y = x \\ I \ x = x$$

- Executando a abstração de uma maneira direta temos:

$$\text{hof}\# \ g \ x \ y = \& \ (g \ (\text{hof}\# \ (K\# \ 1) \ x \ y)) \\ (\& \ y \ (| \ x \ (\text{hof}\# \ I\# \ 1 \ y)))$$

...Funções de Maior Ordem

- Precisamos tomar algum cuidado quando olhar para um ponto fixo, para assegurar que aproximações sucessivas produzem o mesmo resultado para todos os valores de g .
- Como g é uma função, ela pode ser um reticulado de valores.
- No caso três valores:
($K \ 0$), I e ($K \ 1$)
- E isso faz com que encontrar os pontos fixos seja ainda mais caro.

Domínios com Tipos Estruturados

- Análise de domínios com tipos estruturados nos fala, por exemplo, quando que uma aplicação de CONS é estrita.
- Isso nos possibilita gerar um código melhor.
- Hughes [Hughes, 1985] e Wadler [Wadler, 1985a] oferecem extensões à análise para cobrir essa área.

Outros Trabalhos Relacionados

- Wray [Wray, 1986] descreve o algoritmo de análise de funções estritas, com uma técnica que não é baseada em interpretação abstrata.
- Outra técnica em tempo de compilação, projetada para fazer processamento de listas, de uma forma bastante eficiente, é o transformador com ausência de listas de Wadler [Wadler, 1984 e 1985b].
- Esse transformador é capaz de compilar certas funções processadoras de listas numa máquina de estados finitos, que executa sem consumir nenhum *heap*.

Anotando o Programa...

- O objetivo da análise de funções estritas é anotar o programa para beneficiar fases subsequentes da compilação.
- Até agora vimos como derivar uma versão abstrata de cada supercombinador a partir de sua definição.
- Agora veremos como usar essa informação para adicionar anotações ao programa.

...Anotando o Programa...

- Existem duas maneiras, as quais podemos usar funções abstratas para anotar o programa original de supercombinadores:
 1. Podemos anotar cada definição de supercombinador para indicar em quais argumentos ela é estrita.
 - Por exemplo, a definição

$$\$F ! x y ! z = \dots \text{corpo de } \$F \dots$$
 indica que $\$F$ é estrita em x e z , mas não em y .
 2. Podemos anotar nós individuais de aplicação, no corpo dos supercombinadores para indicar uma aplicação estrita.
 - Por exemplo, na definição

$$\$G p q = \dots (\$F ! p \text{ } \text{ } ! q) \dots$$
 a aplicação de $\$F$ a p é anotada com uma exclamação infixa para indicar uma aplicação estrita.
 - A aplicação de $(\$F p \text{ } \text{ } \text{ } q)$ ao q está similarmente anotada.

...Anotando o Programa

- Em princípio, esses dois tipos de anotações fornecem informação duplicada, e normalmente elas o fazem.
- Mas haverá certas ocasiões em que uma delas se mostrará mais apropriada.

Anotando Definições de Funções...

- Suponha que $\$F$ seja uma função que queremos anotar e que ela tenha dois argumentos.
- Para descobrir se $\$F$ é estrita no seu primeiro argumento, basta avaliar:

$$\$F \# 0 1$$
- Se a resposta for 0, $\$F$ certamente é estrita no seu primeiro argumento.
- Uma complicação é que o resultado de $\$F$ pode ser uma função.
- Nesse caso, o resultado de nossa avaliação abstrata também será uma função.

...Anotando Definições de Funções...

- E ainda, estamos interessados em saber se o resultado é o último¹ elemento do domínio da função, de tal forma que simplesmente fornecemos 1's como argumentos até que a função retorne 0 (no caso em que \$F\$ será estrita) ou 1 (no caso em que não será estrita).
- O último elemento do domínio de uma função é aquela função que retorna o último elemento do seu domínio resultante, retirando seu argumento.
- Suponha, por exemplo, que \$F\$ seja definida como:

$$F(x, y) = +(x, y)$$
Então \$F\#\$ será

$$F\#(x, y) = \&(x, y)$$
- Se avaliarmos \$F\#(0, 1)\$ teremos \$\&(0, 1)\$.

¹bottom...Anotando Definições de Funções

- Para determinar se essa função é a última, a aplicamos a 1 e teremos 0 como resultado.
- Temos então que a função era a última e que \$F\$ é estrita no seu primeiro argumento.
- Outra complicação acontece quando um argumento para \$F\$ também é uma função.
- Nesse caso, ao invés de 0 e 1, deveremos usar a primeira e a última do domínio de função apropriado.
- Isso requer saber o tipo de \$F\$, dando outro motivo para o uso de uma linguagem tipada.

Anotando Nodos de Aplicações...

- Considere a definição:

$$G(x, y) = y \ 3 \ x$$
- E suponha que no corpo de outro supercombinador ocorreu a expressão

$$\dots(G \ E \ +) \dots$$
onde \$E\$ é alguma expressão complicada.
- Claramente \$G\$ não é estrita em \$x\$, porque o argumento (função) \$y\$ pode não ser estrito no seu segundo parâmetro.

...Anotando Nodos de Aplicações...

- Contudo nessa aplicação particular de \$G\$ o segundo argumento é \$+\$, e assim \$E\$ com certeza será avaliada subsequentemente.
- Assim \$E\$ poderia ser avaliada antes da chamada de \$G\$, e poderíamos anotar a chamada assim,

$$\dots(G \ ! \ E \ ! \ +) \dots$$
- Fazer isso é muito interessante pois possibilita que as otimizações da máquina-\$G\$ sejam efetuadas, como nesse caso, avaliar \$E\$ diretamente ao invés de construir seu grafo correspondente.

...Anotando Nodos de Aplicações...

- Felizmente, é relativamente fácil deduzir essa anotação.
- Dada uma expressão $(\$G P Q)$, podemos descobrir se ela é estrita em P , avaliando $\$G\# 0 Q\#$ e em Q avaliando $\$G\# P\# 0$
- Para ver que isso é válido formalmente, considere a achar se funções $\$Dummy1$ e $\$Dummy2$ são estritas, onde $\$Dummy1 e = \$G e Q$
 $\$Dummy2 e = \$G P e$
- $\$Dummy1$ é estrita em e se e somente se a expressão $(\$G P Q)$ for estrita em P , e similarmente para $\$Dummy2$.

...Anotando Nodos de Aplicações

- Outro ponto de interesse ocorre quando analisamos uma definição como:
 $\$F x y \dots (\$G E y) \dots$
onde os parâmetros formais da definição ocorrem na subexpressão sendo analisada.
- Para verificar se a aplicação é estrita em E , avalia-se $(\$G\# 0 y\#)$
- Mas qual valor deveríamos usar para $y\#$?
- A análise que estamos fazendo deveria valer para qualquer aplicação de $\$F$, então deveríamos usar 1 para $y\#$, o que reflete nossa falta de informação sobre seu valor.
- Se o tipo do parâmetro é uma função, então substituímos as ocorrências dela com a primeira do espaço de funções abstratas.

Por Que Ambas as Anotações São Necessárias

- Agora pode parecer que a informação que a anotação dos nós de aplicação provê é sempre superior àquela provida pela anotação de definições de funções, já que a primeira tem a vantagem de obter informações do contexto.
- Mas existem duas razões pelas quais é importante anotar também a definição de função.
- Uma é que as otimizações da máquina-G requerem anotações nas definições das funções, para gerar o melhor código possível independentemente do contexto.
- Outra razão é aumentar a chance de paralelismo conservativo numa avaliação paralela como veremos adiante.

III. PRAGMATISMO EM REDUÇÃO DE GRAFOS

III. REDUÇÃO DE GRAFOS PARALELA

Introdução

- A possibilidade de execução em paralelo é normalmente colocada como uma vantagem das linguagens funcionais.
- Essa possibilidade decorre do fato das linguagens funcionais não serem inerentemente sequenciais como as linguagens imperativas.
- Podemos verificar que existem várias tarefas na redução de grafos que são passíveis de serem executadas simultaneamente.
- Vamos abordar aqui alguns tópicos que caracterizam essa área.

Programação Funcional Paralela...

- Normalmente, temos a programação paralela convencional como uma tarefa difícil, pois o programador além de se preocupar com a lógica de seu problema, deverá também ser capaz de dividi-lo em subtarefas, programar essa subtarefas de modo sequencial e organizar a comunicação entre elas.
- Além disso, esse programa paralelo será, normalmente, de difícil depuração.
- Esperamos da programação funcional paralela, a possibilidade de execução paralela de nossos programas sem termos que adicionar nenhuma construção nova à linguagem.

Programação Funcional Paralela...

- Seria de esperar que um programa funcional arbitrário executasse muito mais rápido num ambiente de execução paralelo.
- Essa é uma idéia errônea como foi mostrado em [Clack e Peyton Jones, 1985].
- Na verdade, é de responsabilidade do programador criar um algoritmo que particione a tarefa em questão em subtarefas independentes.

...Programação Funcional Paralela...

- Mas, por que então escrevermos programas paralelos numa linguagem funcional?
 1. O programador não precisa identificar todos os processos concorrentes (tarefas) de seu projeto.
 2. Em linguagens funcionais, o paralelismo pode ser dinâmico, ou seja, tarefas podem ser criadas e removidas dinamicamente.
 3. O programador não precisa se preocupar como será feita a sincronização e os protocolos de comunicação entre as tarefas.
 4. O comportamento do programa será independente do escalonamento das tarefas.

...Programação Funcional Paralela

5. Em linguagens convencionais o programador deve ter em mente todas as ordenações possíveis de tarefas no tempo, referentes as quais uma execução pode ocorrer.
 6. Para fazer um programa funcional paralelo não é requerida nenhuma construção extra na linguagem .
 7. Um programa funcional paralelo é mais fácil de ser depurado que um programa convencional paralelo.
- De fato, os benefícios apontados acima são válidos para qualquer implementação paralela de uma linguagem funcional.
 - Mas a redução de grafos é, particularmente, mais atraente.

Redução de Paralela de Grafos

- Redução de grafos é uma atividade inerentemente paralela.
- A qualquer momento o grafo pode haver um número qualquer de *redexes*, sendo natural reduzi-las simultaneamente.
- Redução de grafos é uma atividade inerentemente distribuída.
- Uma redução é uma transformação local do grafo, e nenhum gargalo compartilhado (p.ex. um ambiente) precisa ser consultado para processarmos uma redução.
- Toda comunicação é mediada através do grafo.
- Isso nos dá um modelo simples de como as atividades concorrentes cooperam, além de ser um modelo de confiança considerável (é o mesmo das nossas implementações sequenciais).
- O estado corrente da computação está bem definido a qualquer momento — é o estado corrente do grafo.

Um Modelo para Redução Paralela...

- Numa implementação sequencial a avaliação (execução) é feita chamando um avaliador, e passando para ele a raiz do grafo a ser avaliado.
- O avaliador faz uma sequência de reduções até que o grafo esteja numa forma terminal.
- Nosso modelo para redução paralela é um generalização deste.
- Nós imaginaremos um certo número de tarefas avaliadoras, trabalhando simultaneamente no grafo.
- Cada avaliador estará ocupado em reduzir seu subgrafo correspondente para a forma terminal.

...Um Modelo para Redução Paralela

- Durante a sua execução, uma tarefa pode prever que necessitará do valor de um certo subgrafo, num tempo futuro.
- Nesse caso, uma nova tarefa pode ser disparada para avaliar aquele subgrafo em paralelo, dando origem à uma hierarquia de tarefas trabalhando concorrentemente, que precisam ser sincronizadas corretamente.

Questões Lógicas sobre o Gerenciamento do Paralelismo...

- Quando os nós do grafos são ativados para criar novas tarefas?
- Essa questão decorre da possibilidade de ativar um nó com ou sem a certeza de que seu valor será usado posteriormente.
- Isso trará implicações no grau de paralelismo de uma execução.
- Veremos que a execução pode ser afetada negativamente, tanto por pouco ou muito paralelismo.

...Questões Lógicas sobre o Gerenciamento do Paralelismo

- Outro ponto é a questão do grão do paralelismo, que precisa estar no ponto certo para conseguir resultados optimais.
- O que acontece se duas tarefas começam a avaliar a mesma porção do grafo?
- Elas farão isso, provalmente, porque o mesmo nó poderia ter sido ativado duas vezes, ou porque os grafos avaliados por duas tarefas compartilham um mesmo subgrafo.
- Veremos que são necessários mecanismos para bloquear tarefas que estão avaliando uma porção do grafo que já está sendo avaliada.

Outras Questões

- Questões de Representação de Tarefas:
 - Quando uma tarefa não está sendo executada, ela deve ser armazenada de alguma forma na memória.
 - O armazenamento deve conter informações suficientes para a tarefa continuar a executar a partir do ponto onde foi suspensa.
- Questões de Localidade:
 - dizendo respeito a como empregar os recursos da máquina para executar as tarefas concorrentes, minimizando a comunicação.
- Questões de arquitetura da máquina onde está sendo feita a redução.

Máquinas Paralelas para Redução de Grafos...

- Existem vários projetos que envolvem a construção de máquinas com arquiteturas específicas para redução paralela de grafos.
- Existe um projeto da Universidade de Utah [Keller *et al.* 1979], cuja a arquitetura da máquina consiste de um conjunto de unidades de processadores/memória/chaveadores, chamados Xputers, onde a porção chaveadora dos Xputers formam, coletivamente, uma rede de comunicações em vários estágios.
- Os processadores se comunicam sobre essa rede, através de passagem de mensagens.

...Máquinas Paralelas para Redução de Grafos

- No Imperial College, London existe um projeto [Darlington e Reeve, 1981], onde a arquitetura é baseada nos Inmos Transputers.
- Outro projeto vem da Universidade de Yale, onde o modelo de redução paralela é baseada em combinadores seriais [Hudak e Goldberg, 1985b].
- O hardware básico utilizado foi um Hipercubo Intel 128-nós.
- Existe um projeto de redução paralela de grafos com supercombinadores, na Universidade de Londres [Peyton Jones *et al.*, 1985; Clack e Peyton Jones, 1986], baseado em arquitetura *bus*.

FIM DO CURSO