Diego Fernandes Nehab

A implementação da linguagem de programação Sloth

Dissertação de Mestrado

Departamento de Informática
Programa de Pós-Graduação em Informática

Rio de Janeiro
Julho de 2002

Pontifícia Universidade Católica
do Rio de Janeiro

Diego Fernandes Nehab

A implementação da linguagem de programação Sloth

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Departamento de Informática da PUC-Rio

Orientador: Roberto Ierusalimschy

Rio de Janeiro
Julho de 2002

Diego Fernandes Nehab

Graduou-se em Engenharia de Computação pela PUC-Rio em 1999, increvendo-se logo em seguida no Programa de Pós-Graduação em Informática da mesma universidade. Trabalha junto ao TecGraf/PUC-Rio desde 1995, quando ingressou no Programa de Iniciação Científica, até os dias de hoje, realizando projetos na área de Software Básico. Em 2001, foi agraciado com a Bolsa Nota Dez da FAPERJ. Em 2002, foi aceito pelo programa de Ph.D. em Ciência da Computação de Princeton, onde pretende dar continuidade à sua carreira acadêmica.

Ficha Catalográfica

Nehab, Diego Fernandes

A implementação da linguagem de programação Sloth / Diego Fernandes Nehab; orientador: Roberto Ierusalimschy. Rio de Janeiro: PUC, Departamento de Informática, 2002.

v., xi+58 f.: il.; 29,7 cm

1. Dissertação (mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas

1. Informática — Teses. 2. Linguagens Funcionais. 3. Redução de Grafos. 4. Combinadores. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. III. Título.

Folha de aprovação

Dissertação de Mestrado apresentada por **Diego Fernandes Nehab**, em 12 de julho de 2002, ao Departamenteo de Informática da PUC-Rio e aprovada pela Comissão Julgadora formada pelos seguintes professores:

Prof. **Roberto Ierusalimschy** (Orientador) — PUC-Rio

Prof. **Edward Hermann Haeusler** — PUC-Rio

Prof[a]. **Noemi de La Rocque Rodriquez** — PUC-Rio

"Dê uma tafera difícil a um preguiçoso e ele encontrará a forma mais simples de realizá-la."

– Lei de Hlade

Agradecimentos

— a Marcelo Gattass, pelos valiosos conselhos e por toda a atenção dispensada;
— a Roberto Ierusalimschy, orientador da dissertação, pela confiança;
— ao Tecgraf, pelo melhor ambiente de trabalho concebível;
— à CAPES e FAPERJ, pela ajuda financeira ao longo do curso;
— à PUC-Rio, por todas as oportunidades que me ofereceu ao longo de sete anos;
— e à minha família, claro, por tudo isso, pela paciência e pelo carinho.

Resumo

Este trabalho descreve uma nova linguagem de programação funcional e sua implementação. Sloth é uma linguagem funcional pura interpretada, não estrita, que suporta funções de ordem elevada como valores de primeira classe, tipos de dados estruturados e definições múltiplas de funções selecionadas por casamento de padrões.

Sloth foi criada para o uso no meio acadêmico, em atividades relacionadas ao uso e à implementação de linguagens funcionais. Sendo assim, durante o desenvolvimento de Sloth, uma grande preocupação com a simplicidade direcionou tanto a especificação da linguagem quanto sua implementação. Como resultado, a sintaxe da linguagem é modesta, mas expressiva o bastante para permitir seu uso em aplicações práticas. Da mesma forma, a grande simplicidade e portabilidade de sua implementação não prejudicam severamente sua eficiência.

As principais contribuições do trabalho são o projeto e desenvolvimento de Sloth, uma otimização à redução de grafos combinatórios como alternativa ao uso dos combinadores de Turner, uma arquitetura inovadora para a implementação de linguagens funcionais interpretadas, baseada no uso cooperativo de duas linguagens de programação, e uma análise comparativa entre a eficiência de Sloth e outras linguagens semelhantes.

Palavras-chave: Linguagens Funcionais, Redução de Grafos, Combinadores

Abstract

This work describes a new functional programming language and its implementation. Sloth is a non-strict purely functional interpreted programming language, which supports high order functions as first class values, structured data types, and pattern-matching based function definitions.

Sloth was created for academical use in activities related to the use and to the implementation of functional programming languages. Therefore, there has been a great concern with the simplicity of both the language specification and its implementation. As a result, Sloth's syntax is modest, but expressive enough for practical applications. Furthermore, the great simplicity and portability of its implementation has no serious impact on its performance.

The main contributions of this work are the design and implementation of Sloth, an optimization to combinator graph reduction as an alternative to the use of Turner combinators, a new architecture for the implementation of interpreted functional programming languages, based on the cooperation between two programming languages, and the performance comparison between Sloth and similar languages.

Keywords: Functional Languages, Graph Reduction, Combinators

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Linguagens de programação funcionais podem ser classificadas por diversos critérios. Linguagens *estritas*, como Scheme e ML, determinam os valores de cada argumento de uma função antes de invocá-la. Em contraste, linguagens *procrastinantes*, como Haskell e Miranda, atrasam a avaliação de cada argumento até o momento em que seu valor se torna indispensável para a determinação do valor final da função. Em linguagens procrastinantes, torna-se importante evitar-se a avaliação repetida de partes do programa. Esta última característica é conhecida como *avaliação preguiçosa*.[1]

Outro critério relevante discrimina linguagens funcionais quanto a sua *transparência referencial*. Em uma linguagem funcional *pura*, como Haskell e Miranda, as computações realizadas se restringem à avaliação de funções, de modo que expressões de mesmo valor podem ser substituídas livremente umas pelas outras e a ordem em que a avaliação de subexpressões é realizada não interfere no resultado final, caso a avaliação termine. Em linguagens *impuras*, como Scheme e ML, o cálculo do valor de uma expressão pode deixar rastros, denominados *efeitos colaterais*, que permanecem acessíveis durante a avaliação de expressões subseqüentes.

Este trabalho diz respeito a linguagens de programação funcionais puras procrastinantes. Para tornar o texto mais leve, os qualificadores "puro" e "procrastinante" são considerados implícitos em expressões como "linguagens funcionais" ou "programas funcionais".

Um programa funcional geralmente consiste em uma expressão a ser avaliada. A avaliação desta expressão, normalmente representada por um grafo, é realizada por uma seqüência de *reduções*. Cada redução substitui uma subexpressão redutível do grafo por sua forma reduzida. O processo de avaliação termina quando não há mais subexpressões redutíveis, situação na qual a expressão se encontra em sua *forma normal*. Como a cada momento pode haver mais de uma subexpressão redutível no grafo, o processo de redução pode seguir diversas ordens. A ordem de avaliação seguida pela maioria das implementações é a *ordem normal de redução*, que seleciona sempre a subexpressão redutível mais externa no grafo. Expressões avaliadas nesta ordem alcançam a forma normal sempre que for possível alcançá-la por qualquer outra ordem.

Naturalmente, o processo de avaliação de funções se encontra no coração de qualquer implementação de uma linguagem funcional. O trabalho a ser realizado é sempre o mesmo: a avaliação de uma função na presença do número suficiente de argumentos corresponde à avaliação do corpo da função, durante a qual a avaliação de um parâmetro formal resulta no valor do argumento correspondente.

[1] Neste texto, usamos criteriosamente o termo *avaliação preguiçosa* para caracterizar a não repetição de computações e o termo *procrastinante* para caracterizar o atraso na avaliação de argumentos. Em inglês, o termo *lazy* é usado para definir ambas as características.

Apesar de representarem expressões de forma semelhante e de seguirem a mesma ordem de avaliação, as diversas técnicas para a implementação de linguagens funcionais diferem bastante pela forma como avaliam funções. As variações encontram-se, principalmente, no método pelo qual os valores dos argumentos de uma função são acessados durante o processo de avaliação.

Em 1979, David Turner publicou um artigo [50] no qual descrevia um método inovador para a implementação da avaliação de funções em linguagens aplicativas. Em relação às técnicas anteriores, a proposta de Turner se destacava tanto pela simplicidade e eficiência quanto pela elegância com que resolvia o problema do acesso aos argumentos de funções durante sua redução. Turner dividiu sua implementação em duas etapas. Na etapa inicial, denominada *compilação*, todas as ocorrências de variáveis ligadas são eliminadas (ou *abstraídas*) das expressões, tirando partido de resultados da Lógica Combinatória [13]. Na etapa final, a avaliação do programa é realizada por uma máquina capaz de compreender a representação que resulta da compilação, em um processo denominado *redução de grafos combinatórios*.

Desde então, enquanto as deficiências do método eram analisadas e algumas variações propostas na tentativa de eliminá-las [33, 44, 35], novas técnicas surgiram [21, 24, 11, 15] mostrando-se mais eficientes. Os ganhos em eficiência, entretanto, vieram ao custo de um considerável aumento na complexidade das implementações. Por essas razões, a proposta de Turner passou a ser mais respeitada por sua elegância que por sua eficiência.

Uma das linhas desenvolvidas por este trabalho propõe uma nova técnica para a implementação de linguagens funcionais baseada no método de Turner, os *combinadores microprogramados*. A nova técnica tenta aumentar a eficiência da redução de grafos combinatórios atacando algumas de suas deficiências, porém sem comprometer sua simplicidade.

Para tornar claras as motivações por trás da criação dos combinadores microprogramados, o capítulo 3 traz uma introdução sobre a implementação de linguagens funcionais. São apresentadas uma introdução ao λ-cálculo, um exemplo do processo de transformação de um programa funcional ao λ-cálculo e a análise de diversas técnicas de implementação, dando maior ênfase à técnica de Turner. Uma vez tendo sido estabelecidos os conceitos necessários, o capítulo 4 descreve detalhadamente os combinadores microprogramados.

A segunda linha desenvolvida neste trabalho é o projeto de uma nova linguagem de programação, denominada Sloth[2], descrita no capítulo 2. Sloth foi criada para ser usada no meio acadêmico, especialmente em atividades didáticas que se beneficiem de uma linguagem simples e pequena. A decisão pela criação de uma nova linguagem, em contraste à adoção de um subconjunto de alguma linguagem já consolidada, como Hope, Miranda ou Haskell, vem da insatisfação com a sintaxe das linguagens analisadas.

Naturalmente, a implementação da linguagem Sloth é baseada nos combinadores microprogramados, o que torna possível a análise concreta da eficiência da nova técnica. No capítulo 6 são apresentadas comparações entre as eficiências dos combinadores microprogrados e dos combinadores de Turner, assim como comparações entre a eficiência de Sloth e a eficiência de outras linguagens de programação com características semelhantes (Gofer, Hugs e Hope).

A terceira e última linha de pesquisa apresentada neste trabalho consiste em uma arquitetura inovadora para a implementação de linguagens funcionais, que tira partido de duas linguagens de programação totalmente integradas. Durante o desenvolvimento de Sloth, várias decisões foram tomadas de modo a manter a implementação o mais simples possível, para permitir que a implementação possa ser compreendida, até mesmo por alunos de graduação, em um intervalo relativamente curto de tempo. Em parte, a nova arquitetura é resultado desta preocupação.

[2] O nome Sloth, que significa *bicho-preguiça* em inglês, é um trocadilho com o fato da linguagem avaliar expressões de forma preguiçosa.

Pela arquitetura proposta, na construção das etapas de compilação, que submetem o programa a transformações que praticamente exigem algum tipo de gerência automática de memória, é usada a linguagem Lua [22]. Já para a implementação do redutor de grafos, na qual a eficiência é a preocupação crucial, é usada a linguagem de programação C [37]. A associação entre as vantagens das duas linguagens permitiu a criação de um sistema conciso, autocontido, portátil e eficiente. Os detalhes da implementação de Sloth são descritos no capítulo 5.

Finalmente, o capítulo 7 conclui o trabalho com a análise dos resultados obtidos e a proposta de possíveis trabalhos futuros nas mesmas linhas de pesquisa.

Capítulo 2

A linguagem de programação Sloth

A primeira versão de Sloth foi criada em 1999, como Trabalho Final de Curso em Engenharia de Computação [42], e teve como motivação principal o estudo de algumas das idéias que deram origem aos combinadores microprogramados descritos no capítulo 4. Os resultados preliminares pareceram promissores, de modo que uma nova versão de Sloth foi criada com mais recursos, para torná-la mais atraente ao meio acadêmico. Os principais recursos adicionados foram o suporte a tipos de dados estruturados e a definições múltiplas de funções selecionadas por casamento de padrões. A lista abaixo resume as principais características da versão atual de Sloth, buscando situá-la frente às demais linguagens de programação funcionais:

- Sloth é uma linguagem aplicativa, funcional pura;
- Sloth é procrastinante e preguiçosa;
- Sloth suporta tipos de dados estruturados;
- A tipagem de Sloth é dinâmica;
- Funções de qualquer ordem são valores de primeira classe;
- Funções podem ter definições múltiplas selecionadas por casamento de padrões;
- O sistema de escopo de Sloth é estático;
- Sloth é interpretada.

A sintaxe de Sloth, descrita a seguir, é vagamente inspirada nas sintaxes de Haskell e ML. Entretanto, de modo a simplificar a especificação de Sloth, vários recursos dessas linguagens foram deliberadamente omitidos, como abrangências de listas (*list comprehensions*) e o suporte à definição de novos operadores infixados.

2.1 Convenções léxicas

Identificadores em Sloth são quaisquer seqüências de letras, dígitos ou dos caracteres `_` e `'`, não iniciados por um dígito ou pelo caractere `'`. São reservadas as seguintes palavras, que não podem ser usadas como identificadores:

```
case    else  if    in    let
letrec  of    then  type
```

Sloth é sensível a caixas alta e baixa, de modo que as cadeias de caracteres `Let`, `LET` e `let` são diferentes e apenas a última é reservada. Adicionalmente, as seguintes seqüências de caracteres têm significados especiais para a linguagem, descritos na seção 2.2:

```
;   |   &   =   [   ]   (
)   "   ,   :   !   <   >
.   +   -   *   /   %   ==
->  ++  !=  <=  >=  ||  &&
```

Ao contrário de Haskell, que atribui valor sintático à indentação, Sloth é uma linguagem de formato livre. Em Sloth, quebras-de-linha, espaços brancos e tabulações servem apenas para separar os elementos léxicos e são de outra forma ignorados pela linguagem. Comentários são iniciados pela seqüência `--` e continuam até o fim da linha corrente.

Numerais são definidos com parte fracionária e expoente decimal opcionais. Alguns exemplos de constantes numéricas são: `42`, `3.1415926585` e `6.02e23`. Caracteres são especificados da mesma forma que em C, inclusive os caracteres especiais. Assim, `'a'`, `'0'`, `'*'` e `'\n'`, por exemplo, são aceitos como caracteres pela linguagem. Seqüências de caracteres podem ser definidas entre aspas duplas, como em `"foo\nbar"`.

2.2 Sintaxe

Um programa Sloth consiste em uma série de declarações globais. Declarações globais podem definir novos tipos de dados ou associar nomes a expressões. A execução de um programa Sloth consiste no cálculo e exibição do valor da expressão associada ao nome `main`. Para abreviar e simplificar a exposição da sintaxe da linguagem, cada estrutura sintática será apresentada por meio de exemplos nas próximas seções.

2.2.1 Tipos de dados

Os únicos tipos de dados primitivos em Sloth são os tipos `Number` e `Char`, representando números em ponto-flutuante e caracteres, respectivamente. Os demais tipos de dados são construídos a partir dos tipos primitivos, seguindo-se a sintaxe usual para declaração de tipos na forma de *soma de produtos*, como nos exemplos a seguir:

```
type Bool
    = true
    | false;

type List a
    = cons a (List a)
    | nil;
```

Os tipos `Bool` e `List` são pré-declarados em Sloth. `Bool` é um tipo enumerado (no qual nenhum construtor recebe argumentos) composto por dois construtores, `true` e `false`. A declaração de `List` faz uso da variável esquemática `a` para permitir a criação de listas com elementos de qualquer tipo. As listas `(cons 1 (cons 2 nil))` e `(cons 'a' (cons 'b' nil))` são dos tipos `(List Number)` e `(List Char)`, respectivamente. Usando a mesma sintaxe, o programador pode definir seus próprios tipos de dados estruturados, como os exemplos `Tree` e `Week` a seguir:

```
type Tree a
    = branch (Tree a) (Tree a)
    | leaf a;

type Week
    = monday | tuesday
    | wednesday | thursday
    | friday | saturday
    | sunday;
```

2.2.2 Expressões infixadas

Por ser uma linguagem aplicativa, Sloth foi projetada para trabalhar com expressões prefixadas. Entretanto, Sloth permite o uso de operadores infixados como uma alternativa mais confortável para a construção de expressões complexas. A Tabela 2.1 mostra os operadores suportados, em ordem decrescente de precedência, listando também o tipo de associatividade de cada operador.

operação	associatividade	observações
a b	esquerda	aplicação
`!`	nenhuma	negação booleana
`-`	nenhuma	menos unário
`*`	esquerda	
`/ %`	nenhuma	
`+`	esquerda	
`-`	esquerda	
`.`	direita	composição de funções
`== != > >= < <=`	nenhuma	
`&&`	esquerda	
`||`	esquerda	
`:`	direita	construção de listas
`++`	esquerda	concatenação de listas

Tabela 2.1: Os operadores infixados, suas precedências e associatividades.

Expressões infixadas são imediatamente convertidas para expressões prefixadas equivalentes durante a análise sintática. Ao longo do processo de conversão, os operadores são substituídos pelos nomes das funções que representam, conforme mostra a Tabela 2.2.

operador	função	operador	função	operador	função
`-` (unário)	`neg`	`-` (binário)	`sub`	`+`	`add`
`*`	`mul`	`/`	`div`	`%`	`mod`
`==`	`eq`	`!=`	`neq`	`>`	`gt`
`>=`	`ge`	`<`	`lt`	`<=`	`le`
`.`	`bola`	`:`	`cons`	`!`	`not`
`&&`	`and`	`||`	`or`	`++`	`concat`

Tabela 2.2: As funções prefixadas representando cada operador infixado.

A seguir, vemos alguns exemplos de expressões infixadas e suas equivalentes prefixadas, convertidas segundo as regras expostas nas Tabelas 2.1 e 2.2:

```
a + sin b * c  →  add a (mul (sin b) c)
     1:2:nil   →  cons 1 (cons 2 nil)
```

Seguindo a mesma filosofia, a construção *if-then-else* presente na linguagem Sloth é apenas um açúcar sintático, e é também substituída por sua equivalente prefixada durante a análise sintática:

`if` $expr_1$ `then` $expr_2$ `else` $expr_3$ $\rightarrow$ `cond` $expr_1$ $expr_2$ $expr_3$

2.2.3 Listas e tuplas

Conforme a definição apresentada na seção 2.2.1, listas são seqüências possivelmente infinitas de elementos de um mesmo tipo. Por serem tão comuns em programas funcionais, listas podem ser definidas de forma abreviada. As expressões `(cons 1 (cons 2 nil))`, `1:2:[]` e `[1,2]` são equivalentes. Seqüências de caracteres, definidas entre aspas duplas, são transformadas imediatamente para listas de caracteres durante a análise sintática, o que torna equivalentes as expressões `"foo"` e `['f','o','o']`.

Tuplas, também conhecidas como conjuntos ordenados, agrupam uma série de elementos de tipos possivelmente distintos. A definição de tuplas de qualquer dimensão é feita conforme a notação matemática mais comum: com os elementos entre parênteses, separados por vírgulas. São exemplos de tuplas as expressões `(1,2)` e `("foo",2,'a')`.

2.2.4 Declarações de funções

A sintaxe para as declarações de funções é muito flexível e é introduzida por meio de vários exemplos. A função `suc`, um dos exemplos mais simples de função, retorna o sucessor de seu argumento:

```
suc x = x + 1;
```

Naturalmente, funções podem ter definições recursivas, como ilustrado pela declaração (ingenuamente ineficiente) da função `fib`, que calcula o enésimo número de Fibonacci:

```
fib x = if x < 2 then 1 else fib (x-1) + fib (x-2);
```

Funções podem ter definições múltiplas selecionadas por casamento de padrões. Neste caso, a definição apropriada é escolhida em tempo de execução pela análise dos construtores dos parâmetros recebidos, pela análise de seus valores, ou pela avaliação de condições genéricas especificadas pelo programador, as *condições guarda*.

Um exemplo interessante é a função `cond`, predefinida em Sloth. Caso a avaliação de seu primeiro argumento retorne `true`, a primeira opção é selecionada. Caso contrário, a segunda opção é selecionada:

```
cond true  e1 e2 = e1
|    false e1 e2 = e2;
```

Já na definição de `fat`, que faz uso de expressões guarda, o compilador avalia a guarda `x < 2` e, caso obtenha o valor `true`, seleciona a primeira opção. Caso contrário, a segunda opção é escolhida (já que a guarda `otherwise` é sempre verdadeira):

```
fat x = 1,             x < 2
|   x = x * fat (x-1), otherwise;
```

O casamento de padrões se estende a tipos de dados estruturados, e pode ser usado em conjunto com expressões guarda, como demonstrado pela função `isort` e suas auxiliares, que implementam o método de ordenação de listas por inserção:

```
foldr f z []     = z
|     f z (x:xs) = f x (foldr f z xs);

insert x []      = [x]
|      x (y:ys)  = x:y:ys,         x <= y
|      x (y:ys)  = y: insert x ys, otherwise;

isort            = foldr insert [];
```

2.2.5 Definições locais

Expressões `let` e `letrec` permitem a definição de valores locais a uma expressão. A diferença entre as duas construções está no fato de `letrec`, ao contrário de `let`, permitir definições mutuamente recursivas. Definições locais são úteis tanto para abreviar expressões quanto para evitar o cálculo repetido de subexpressões. Ambas as vantagens são exploradas pelo exemplo a seguir, que retorna um par ordenado com as raízes da equação do segundo grau $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c} = 0$:

```
roots (a, b, c) = let
        delta = sqrt (b*b - 4*a*c) &
        da    = 2*a
    in ((-b + delta)/da, (-b - delta)/da);
```

Funções também podem ser definidas localmente. Na função `gcd` a seguir, o cálculo do maior divisor comum entre dois números é feito em uma função local, que é aplicada sobre argumentos tratados de modo a tornar a definição mais robusta:

```
gcd x y = letrec
        gcd' x 0 = x
        |    x y = gcd' y (x % y)
    in gcd' (floor (abs x)) (floor (abs y));
```

2.2.6 Seleção por construtor

As declaração de tipos por soma de produtos permite a especificação de tipos com mais de um construtor, os *tipos soma*. Geralmente, a distinção entre os possíveis construtores de um tipo soma é feita indiretamente, por casamento de padrões. Expressões *case* permitem a avaliação de uma entre várias expressões, selecionada pelo construtor de um valor. Além disso, a construção permite o acesso aos campos do valor estruturado, como nos exemplos `cond` e `length` a seguir:

```
cond b v f = case b of
    true -> v |
    false -> f;

length l = case l of
    nil -> 0 |
    cons x xs -> 1 + length xs;
```

2.2.7 Funções anônimas

Assim como a maioria das linguagens de programação nas quais funções são valores de primeira classe, Sloth permite a definição de funções anônimas, também conhecidas como *abstrações lambda*. Funções anônimas podem ser usadas nas mesmas situações nas quais é válido o uso de funções nomeadas, mas são especialmente úteis em conjunto com funções de ordem elevada. No exemplo a seguir, a função anônima `(\x -> 2*x)` é usada em conjunto com a função `map` na

construção da função `double`. Quando aplicada a uma lista, `double` retorna outra lista, na qual os elementos valem o dobro dos da lista original:

```
map f []     = []
|   f (x:xs) = f x: map f xs;

double       = map (\x -> 2*x);
```

Funções anônimas com mais de uma variável podem ser definidas de forma simplificada. A declaração `(\x y -> x+y)` é equivalente à declaração `(\x -> \y -> x+y)`.

2.3 Exemplos completos

O apêndice B apresenta vários exemplos de programas completos em Sloth, usados durante os testes realizados. Os programas implementam a solução do problema das Torres de Hanoi, o cálculo dos números primos pelo Crivo de Eratóstenes e a ordenação de listas numéricas pelos métodos *Quick Sort*, *Insertion Sort* e *Merge Sort*.

Capítulo 3

A implementação de linguagens funcionais

Na década de 1930, Alonzo Church e Alan Turing formalizaram a noção de função computável seguindo caminhos distintos. O trabalho de Turing deu origem a uma classe de máquinas abstratas, as *máquinas de Turing* [48]. A teoria de Church baseou-se na formulação de um sistema formal, o λ-*cálculo* [9]. Posteriormente, Turing provou que os dois formalismos definem a mesma classe de funções [47].

Os computadores modernos são baseados nas *máquinas de von Neumann*, que são semelhantes a máquinas de Turing, estendidas para permitir acesso aleatório à memória. Linguagens de programação imperativas, como C e Pascal, funcionam na mesma linha, especificando um conjunto de instruções executadas em seqüência, que manipulam a memória disponível. Por outro lado, linguagens de programação funcionais puras, como Sloth, são diretamente relacionadas ao λ-cálculo por lidarem exclusivamente com a avaliação de expressões.

A primeira etapa na implementação de uma linguagem de programação funcional consiste na eliminação das construções de alto nível presentes na linguagem, levando o programa a uma forma intermediária, geralmente muito semelhante ao λ-cálculo. A segunda etapa transforma o programa resultante para uma forma concreta que permite uma avaliação mais eficiente em máquinas de von Neumann. Juntas, as duas etapas constituem o que chamamos de *compilação*. A Figura 3.1 ilustra o processo.

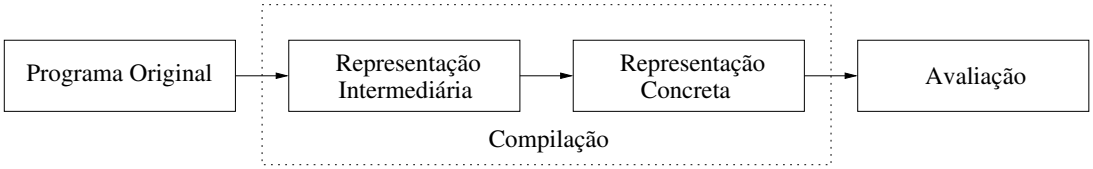


Figura 3.1: Etapas da implementação de linguagens funcionais.

O conjunto de transformações que leva um programa à sua representação no λ-cálculo segue técnicas amplamente publicadas [29], e é de pouca relevância para este trabalho. Seguindo uma breve introdução ao λ-cálculo, o processo de tradução de Sloth a sua representação intermediária é ilustrado pela transformação da função `insert`, na seção 3.2. Em seguida, a seção 3.3 detalha o processo de avaliação de expressões, apresentando a descrição de diversas técnicas usadas para a implementação de linguagens funcionais.

3.1 Introdução ao lambda cálculo

O λ-cálculo é construído a partir de um conjunto infinito de variáveis pelo uso de *aplicações* e *abstrações* (de funções). A formulação Λ dada a seguir é uma pequena extensão ao λ-cálculo, freqüentemente encontrada na literatura, que permite o uso de constantes em expressões. A referência padrão sobre o assunto é o trabalho de Barendregt [4], mas a discussão abaixo se baseia em outro trabalho, mais acessível, do mesmo autor [5].

$$\langle \mathit{variável} \rangle \in \Lambda$$
$$\langle \mathit{constante} \rangle \in \Lambda$$
$$M \in \Lambda \wedge N \in \Lambda \Rightarrow (M\ N) \in \Lambda$$
$$M \in \Lambda \Rightarrow (\lambda \langle \mathit{variável} \rangle . M) \in \Lambda$$

É comum o uso dos símbolos u, v, x, y, $z \ldots$ denotando variáveis genéricas, dos símbolos M, N, E, $F \ldots$ denotando λ-termos genéricos e do símbolo c denotando constantes. Uma considerável diminuição do número de parênteses é obtida pela omissão dos parênteses mais externos e pelo uso das regras de precedência

$$\lambda x_1 . \lambda x_2 \cdots \lambda x_n . M \equiv \lambda x_1 . (\lambda x_2 \cdots (\lambda x_n . M) \cdots)$$
$$F\ M_1\ M_2 \cdots M_n \equiv (\cdots ((F\ M_1)\ M_2) \cdots M_n)$$

Uma abstração $\lambda x.M$ é interpretada como a função que constrói o valor M a partir do valor de x. A aplicação $(M\ N)$ é interpretada como a função M aplicada sobre o argumento N. Alguns exemplos de λ-termos são dados abaixo:

$$\begin{array}{ccc} x & x\ y & x\ y\ z \\ \lambda x.x & \lambda x.y\ x & \lambda x.z\ (x\ y) \\ & \lambda x.x\ (\lambda y.y\ x)\ z & \end{array}$$

Formalmente, o conjunto de *variáveis livres* em um λ-termo M, denotado por $\mathrm{FV}(M)$, é definido por indução segundo as regras abaixo:

$$\begin{aligned} \mathrm{FV}(c) &= \emptyset \\ \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(M\ N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) - \{x\} \end{aligned}$$

Variáveis que não são livres, são ditas *ligadas*. No termo $(\lambda x.y\ x)$, por exemplo, a variável x é ligada e a variável y é livre.

O axioma do λ-cálculo de maior relevância para a implementação de linguagens funcionais (de fato, o único axioma em algumas formulações) é denominado *β-conversão* e estabelece a relação entre abstrações e aplicações: $(\lambda x.M)\ N \underset{\beta}{\leftrightarrows} M[x/N]$. Nesta fórmula, "$[x/N]$" representa a operação de substituição das ocorrências livres de x por N, que pode ser definida por indução segundo as regras abaixo:

$$\begin{aligned} c[x/N] &\equiv c \\ x[x/N] &\equiv N \\ y[x/N] &\equiv y, \quad x \not\equiv y \\ (E\ F)[x/N] &\equiv (E[x/N])\ (F[x/N]) \\ (\lambda x.E)[x/N] &\equiv \lambda x.E \\ (\lambda y.E)[x/N] &\equiv \lambda z.E[y/z][x/N], \qquad x \not\equiv y \wedge z \notin \mathrm{FV}(E\ N) \end{aligned}$$

Implementações de linguagens funcionais têm maior interesse na passagem do lado esquerdo para o lado direito da *β-conversão*, que é denominada *β-redução*. Várias das técnicas apresentadas na seção 3.3 são simplesmente diferentes métodos para a implementação eficiente da *β-redução*.

A outra forma de redução que é de interesse para a implementação de linguagens funcionais consiste na redução de funções primitivas, representadas como constantes na versão entendida do λ-cálculo. Termos encabeçados por funções primitivas são redutíveis quando aplicados a um número suficiente de argumentos. A redução do termo consome os argumentos (e a função) e resulta no valor da função quando avaliada sobre os argumentos.

Quando não há mais subtermos redutíveis em um λ-termo, dizemos que o termo está em sua *forma normal*. Os exemplos a seguir mostram o processo de reduções sucessivas, tanto β-reduções quanto reduções de funções primitivas, que levam dois termos a suas formas normais:

$$(\lambda x.\texttt{add}\ x\ \texttt{3})\ (\texttt{sub 2 1}) \xrightarrow{}_{\beta} \texttt{add}\ (\texttt{sub 2 1})\ \texttt{3} \rightarrow \texttt{add 1 3} \rightarrow \texttt{4}$$
$$(\lambda x.\lambda y.\texttt{mul}\ x\ y)\ \texttt{1 2} \xrightarrow{}_{\beta} (\lambda y.\texttt{mul 1}\ y)\ \texttt{2} \xrightarrow{}_{\beta} \texttt{mul 1 2} \rightarrow \texttt{2}$$

Freqüentemente, há mais de um subtermo redutível em um λ-termo, de modo que o processo de redução pode seguir diversas ordens. Parece possível que ordens de redução diferentes levem um λ-termo a formas normais distintas. Além disso, alguns λ-termos nunca chegam a uma forma normal[1]. Por isso, para um mesmo λ-termo, uma ordem de redução pode chegar a uma forma normal, enquanto outras podem jamais alcançá-la.

Felizmente, dois teoremas devidos a Church e Rosser resolvem ambos os problemas. O primeiro garante a igualdade de formas normais atingidas por ordens de redução diferentes. O segundo apresenta uma ordem de redução que sempre atinge uma forma normal, caso alguma exista. Essa ordem, denominada *ordem normal de redução*, escolhe para redução sempre o λ-termo redutível mais a esquerda, mais externo, e é a ordem seguida pela maioria das implementações de linguagens funcionais procrastinantes.

3.2 Tradução de Sloth ao lambda cálculo

A tradução de um programa Sloth à sua representação intermediária consiste na eliminação de todas as construções de alto nível oferecidas pela linguagem, até que o programa atinja uma representação equivalente no λ-cálculo. O processo é ilustrado pela transformação da função `insert`:

```
insert x []     = [x]
|      x (y:ys) = x: y: ys,       x <= y
|      x (y:ys) = y: insert x ys, otherwise;
```

Algumas das facilidades oferecidas por Sloth, denominadas *açúcares sintáticos*, são simplesmente formas mais confortáveis para a declaração de construções disponíveis de outras formas na linguagem. O suporte a operadores infixados e a forma compacta para a declaração de listas são exemplos deste tipo de facilidade. A eliminação dos açúcares sintáticos é feita durante a análise sintática, e resulta em um programa equivalente (muitas vezes menos legível). A versão *desaçucarada* de `insert` é dada a seguir:

```
insert x nil         = cons x nil
|      x (cons y ys) = cons x (cons y ys),   le x y
|      x (cons y ys) = cons y (insert x ys), otherwise;
```

[1] A redução do λ-termo $((\lambda x.x\ x)\ (\lambda x.x\ x))$ resulta no mesmo λ-termo, de modo que sua avaliação não termina.

A tradução de definições múltiplas de funções selecionadas por casamento de padrões, entretanto, exige transformações mais complexas, que são realizadas em uma etapa posterior à análise sintática, denominada *compilação de casamento de padrões*. O processo transforma definições múltiplas em definições simples eficientes:

```
insert = \x -> \l -> case l of
    nil -> cons x nil |
    cons y ys -> cond (le x y) (cons x (cons y ys)) (cons y (insert x ys));
```

Em uma etapa seguinte, as expressões *case* introduzidas pelo compilador de casamento de padrões (ou pelo próprio usuário) são eliminadas. A transformação faz uso de duas famílias de operações primitivas: *switch* e *select*. As operações *switch-*⟨*Type*⟩ baseia-se no construtor de seu primeiro argumento para retornar um dentre seus demais argumentos. Já o conjunto de operações *select-*⟨*field*⟩*-*⟨*constructor*⟩ extrai um dos campos do valor estruturado que recebe como argumento:

```
insert = \x -> \l-> switch-List l (cons x nil)
    (let y  = select-1-cons l &
         ys = select-2-cons l
    in cond (le x y) (cons x (cons y ys)) (cons y (insert x ys)));
```

Finalmente, as definições locais são eliminadas. A falta de recursos sintáticos no lambda cálculo é a grande responsável pela ilegibilidade da expressão resultante. Entretanto, o pequeno número de construções permitidas torna as fases subseqüentes muito mais simples e imunes a alterações na sintaxe da linguagem, de modo que a representação intermediária é extremamente vantajosa:

```
insert = \x -> \l -> switch-List l (cons x nil)
    (\y -> \ys -> cond (le x y) (cons x (cons y ys)) (cons y (insert x ys))
        (select-1-cons l) (select-2-cons l));
```

Na realidade, para alcançarmos efetivamente a representação de `insert` no λ-cálculo, seria necessária mais uma etapa, que se encarregasse de eliminar as definições recursivas. Apesar de algumas implementações realizarem essa última transformação, Sloth trata recursão diretamente, conforme mostra a seção 5.6.

3.3 Avaliação de programas

Após traduzir os programas a equivalentes no λ-cálculo, o processo de avaliação é feito por um conjunto de reduções. De modo a garantir o término do processo sempre que for possível, linguagens de programação procrastinantes costumam seguir a ordem normal de avaliação. O processo é ilustrado pela avaliação de um programa simples na linguagem Sloth:

```
simples = \g -> g ((\x -> g x 1) 2);
main = simples add 3;
```

A seqüência de reduções abaixo pode ser dividida em três grupos. As duas primeiras reduções, marcadas pelo símbolo $\Rightarrow$, costumam ser realizadas em uma etapa anterior à avaliação, por ligação (*linking*). As duas reduções seguintes, que se encontram marcadas pelo símbolo $\xrightarrow{\beta}$, correspondem à β-redução do λ-cálculo. As duas reduções finais, marcadas pelo símbolo $\rightarrow$, são reduções de funções primitivas:

```
main ⇒
simples add 3 ⇒
(\g -> g ((\x-> g x 1) 2)) add 3 →β
add ((\x -> add x 1) 2) 3 →β
add (add 2 1) 3 →
add 3 3 →
6
```

Dos três tipos de redução, a de maior interesse é a β-redução, que corresponde à avaliação de funções definidas pelo usuário. Várias técnicas foram desenvolvidas para a implementação eficiente da avaliação de funções e as próximas seções deste capítulo se devotam a este tema.

3.3.1 Técnicas anteriores à de Turner

Em uma implementação direta, o tratamento de funções é feito pela *instanciação de moldes* (*template instantiation*). Cada função é armazenada como um *molde*, na forma de uma árvore. Quando a avaliação de uma expressão demanda a avaliação de uma função, seu molde é percorrido e uma nova instância de seu corpo é criada, na qual ocorrências dos parâmetros formais foram substituídas pelos argumentos correspondentes. Essa nova instância sobrescreve a ocorrência da função e de seus argumentos na expressão original. O processo de avaliação pode então prosseguir normalmente.

A instanciação de moldes tem inúmeras desvantagens, dentre as quais uma das mais graves é a repetição desnecessária do trabalho de percorrimento do molde durante o processo de clonagem.

Uma alternativa consiste no uso de *ambientes* para associar valores a nomes de variáveis. Essa é a alternativa usada pela máquina *SECD*, de Landin [39], na qual o valor de um programa passa a ser definido por duas estruturas separadas: um ambiente e uma expressão a ser avaliada. Variáveis presentes em expressões têm seus valores determinados por consultas ao ambiente. Durante a avaliação de uma função, o ambiente é estendido para conter a associação entre os parâmetros formais da função e os argumentos correspondentes. Após a avaliação, as novas associações são removidas do ambiente. Assim, o processo de instanciação não precisa analisar variáveis.

Infelizmente, passa a ser possível que expressões assumam valores diferentes na presença de ambientes distintos, o que pode impedir que expressões sejam sobrescritas por um valor e causar a avaliação repetida de subexpressões. Além disso, o custo computacional com a manutenção de ambientes pode prejudicar a eficiência do processo.

3.3.2 Técnica de Turner, análises e variações

A Lógica Combinatória, criada por Schönfinkel [46] e melhor desenvolvida por Curry e Feys [13], abriu um novo ramo nos fundamentos da matemática, ao criar o conceito de funções genéricas, cujos argumentos podem ser outras funções (sem restrições). O trabalho de Schönfienkel é, por exemplo, o responsável pela forma simplificada com a qual linguagens de programação funcionais manipulam funções aplicadas a um subconjunto de seus argumentos, conhecida como *currying*[2].

Outra grande contribuição da Lógica Combinatória consiste em um método para a especificação de computações sem o uso de variáveis, desenvolvida originalmente para simplificar a manipulação de funções de ordem elevada: a eliminação de variáveis ligadas por abstrações lambda pela introdução de algumas funções simples, denominadas *combinadores*. Nas expressões

[2] Apesar de ter sido criada por Schönfienkel, a técnica foi melhor desenvolvida por Haskell Curry, e por isso é conhecida por seu nome.

resultantes, os combinadores determinam gradualmente os locais para os quais devem ser remetidos os argumentos, fazendo com que alcancem as posições onde se encontrariam as variáveis correspondentes. Apesar de resultar em expressões praticamente ilegíveis, o método efetivamente torna desnecessário o uso de variáveis. É notável que bastem dois combinadores, `S` e `K`, para tornar o processo viável. O combinador `I` pode ser escrito em função dos demais e serve apenas para abreviar as expressões resultantes:

```
S f g x = (f x) (g x);
K c x = c;
I x = S K K x = x;
```

O efeito de cada combinador pode ser facilmente interpretado: `K` descarta seu segundo argumento, `S` encaminha seu terceiro argumento tanto a seu primeiro quanto a seu segundo argumento e `I` representa a identidade.

Formalmente, o processo de eliminação (ou *abstração*) da variável x em uma expressão M, denotado por $\lambda^* x.M$, pode ser definido pelas regras abaixo[3]:

$$\begin{array}{rcll} \lambda^* x.x & \underset{\lambda^*}{\rightarrow} & \mathtt{I} & \\ \lambda^* x.M & \underset{\lambda^*}{\rightarrow} & \mathtt{K}\ M, & x \not\propto M \\ \lambda^* x.(U\ x) & \underset{\lambda^*}{\rightarrow} & U, & x \not\propto U \\ \lambda^* x.(M\ N) & \underset{\lambda^*}{\rightarrow} & \mathtt{S}\ (\lambda^* x.M)\ (\lambda^* x.N) & \end{array}$$

Turner notou que o tratamento de variáveis é uma das maiores deficiências das técnicas descritas na seção 3.3.1. Em seu artigo original [50], Turner propõe que, em uma etapa anterior à avaliação, semelhante a um processo de *compilação*, todas as variáveis sejam eliminadas das expressões pelo uso de técnicas baseadas na Lógica Combinatória.

De fato, seguindo as regras acima, é possível eliminar-se, uma a uma, todas as variáveis ligadas em uma expressão do λ-cálculo. O processo de compilação, que realiza essa tarefa, é definido pela transformação $\mathcal{C}$, apresentada a seguir:

$$\begin{array}{rcl} \mathcal{C}[\![x]\!] & = & x \\ \mathcal{C}[\![c]\!] & = & c \\ \mathcal{C}[\![M\ N]\!] & = & \mathcal{C}[\![M]\!]\ \mathcal{C}[\![N]\!] \\ \mathcal{C}[\![\lambda x.M]\!] & = & \lambda^* x.\mathcal{C}[\![M]\!] \end{array}$$

O aplicação da transformação $\mathcal{C}$ sobre um programa resulta em uma *árvore combinatória*, formada apenas por constantes, já que os combinadores são meras funções primitivas. O programa `simples`, apresentado no início da seção 3.3, por exemplo, é transformado segundo as etapas abaixo:

$$\begin{array}{l} \mathcal{C}[\![\lambda g.g\ ((\lambda x.g\ x\ 1)\ 2)]\!]\ = \\ \lambda^* g.g\ ((\lambda^* x.g\ x\ 1)\ 2) \underset{\lambda^*}{\rightarrow} \\ \lambda^* g.g\ (\mathtt{S}\ g\ (\mathtt{K}\ 1))\ 2) \underset{\lambda^*}{\rightarrow} \\ \mathtt{S}\ \mathtt{I}\ (\mathtt{S}\ (\mathtt{S}\ \mathtt{S}\ (\mathtt{K}\ (\mathtt{K}\ 1)))\ (\mathtt{K}\ 2)) \end{array}$$

Em seguida, uma máquina estendida com as definições de cada combinador se encarrega de avaliar a expressão compilada. Nesta máquina, a avaliação das árvores combinatórias corresponde

[3]Os predicados $x \propto M$ e $x \not\propto M$ são abreviações para os predicados $x \in \mathrm{FV}(M)$ e $x \notin \mathrm{FV}(M)$, respectivamente.

a execução automática do trabalho de instanciação de moldes. À medida que cada combinador é reduzido, a expressão resultante vai se aproximando da forma original do corpo da função, no qual cada argumento foi encaminhado para a posição onde ocorria o parâmetro correspondente. O processo de redução passa a realizar apenas reduções de funções primitivas.

Seguindo com nosso exemplo, uma vez determinada a forma compilada de `simples`, a avaliação da expressão `(simples add 3)` seguiria a seguinte seqüência de reduções:

```
main ⇒
simples add 3 ⇒
S I (S (S S (K (K 1))) (K 2)) add 3 →
I add (S (S S (K (K 1))) (K 2) add) 3 →
add (S (S S (K (K 1))) (K 2) add) 3 →
add ((S S (K (K 1)) add) (K 2 add)) 3 →
add (S add (K (K 1) add) (K 2 add)) 3 →
add (add (K 2 add) (K (K 1) add (K 2 add))) 3 →
add (add 2 (K (K 1) add (K 2 add))) 3 →
add (add 2 (K 1 (K 2 add))) 3 →
add (add 2 1) 3 →
add 3 3 →
6
```

Já em seu primeiro trabalho sobre o assunto, Turner sentiu a necessidade de tornar as árvores combinatórias o mais concisas possível. Naturalmente, quanto maior o número de combinadores, maior o consumo de memória com a armazenagem das árvores combinatórias. O problema mais grave, entretanto, reside no fato de que cada avaliação de um combinador corresponde a uma transição entre representações intermediárias, a caminho da expressão final na qual cada argumento alcançou sua posição correta. A presença de um número muito grande de combinadores resulta em um número muito grande de representações intermediárias. Além do impacto negativo sobre o tempo de execução, ocorre uma sobrecarrega no coletor de lixo, que precisa reaver o espaço ocupado pelas representações intermediárias.

Os primeiros combinadores adicionais sugeridos por Turner foram os combinadores `B` e `C`, já usados por Curry e Feys [13], que evitam encaminhar o argumento correspondente à variável abstraída a ramos nos quais certamente será descartado:

```
B f g x = f (g x);
C f g x = (f x) g;
```

Para permitir o uso destes combinadores, é necessário que o processo de abstração de variáveis seja estendido com duas novas regras (e que a máquina redutora seja estendida com os novos combinadores):

$$\lambda^* x.(M\ N) \quad \underset{\lambda^*}{\rightarrow} \quad \texttt{B}\ M\ (\lambda^* x.N), \quad x \not\circ M \wedge x \circ N$$
$$\lambda^* x.(M\ N) \quad \underset{\lambda^*}{\rightarrow} \quad \texttt{C}\ (\lambda^* x.M)\ N, \quad x \circ M \wedge x \not\circ N$$

A redução no tamanho das árvores combinatórias que resulta do uso dos combinadores `B` e `C` é observável até mesmo em programas pequenos, como é o caso de `simples`, cuja forma compilada passa de (`S I (S (S S (K (K 1))) (K 2))`) para (`S I (C (C C 1) 2)`). A nova versão compilada de `simples` permite a avaliação do programa `main` em um um número menor de reduções:

```
main ⇒
simples add 3 ⇒
```

```
S I (C (C C 1) 2) add 3 →
I add (C (C C 1) 2 add) 3 →
add (C (C C 1) 2 add) 3 →
add (C C 1 add 2) 3 →
add (C add 1 2) 3 →
add (add 2 1) 3 →
add 3 3 →
6
```

Infelizmente, programas mais complexos, ou até mesmo programas simples que exijam a eliminação de diversas variáveis, continuam sofrendo um aumento considerável no tamanho das árvores combinatórias em relação ao tamanho do programa original, quando medidos pelo número de constantes e variávies.

O programa $(\lambda a.\lambda b.\lambda c.\lambda d.d\ c\ b\ a)$, que inverte a ordem de seus argumentos, é um exemplo de programa que sofre um grande aumento de tamanho quando tem suas variáveis eliminadas. Sua tradução, mesmo com o uso dos novos combinadores, cresce desproporcionalmente, resultando na expressão (`C (B C (B (B C) (C (B C (C I))))))`).

Para amenizar o problema, em um trabalho subseqüente [49], Turner concebeu os *combinadores de longo alcance*, que lidam com um número maior de argumentos. Neste trabalho, foram propostos três novos combinadores, `S′`, `B′` e `C′`, que funcionam como os combinadores `S`, `B` e `C`, porém *pulando* sobre o primeiro argumento:

```
S′ c f g x = c (f x) (g x);
B′ c f g x = c f (g x);
C′ c f g x = c (f x) g;
```

Naturalmente, as regras para a abstração de variáveis foram estendidas novamente, para lidar com os novos combinadores. Como resultado da adição das regras abaixo, o exemplo que inverte a ordem de seus argumentos passa a ser compilado para a expressão (`C′ (C′ C) (C′ C (C I))`):

$$\begin{aligned}
\lambda^* x.(E\ M\ N) &\underset{\lambda^*}{\rightarrow} \texttt{S}'\ (\lambda^* x.M)\ (\lambda^* x.N), && x \not\circ E \wedge x \circ M \wedge x \circ N \\
\lambda^* x.(E\ M\ N) &\underset{\lambda^*}{\rightarrow} \texttt{B}'\ M\ (\lambda^* x.N), && x \not\circ E \wedge x \not\circ M \wedge x \circ N \\
\lambda^* x.(E\ M\ N) &\underset{\lambda^*}{\rightarrow} \texttt{C}'\ (\lambda^* x.M)\ N, && x \not\circ E \wedge x \circ M \wedge x \not\circ N
\end{aligned}$$

Ainda assim, Kennaway [34] mostrou que o tamanho das árvores combinatórias pode crescer de forma $\Theta(n^2)$. Por isso, ao longo dos anos seguintes, vários trabalhos propuseram formas alternativas para a codificação de árvores combinatórias, na tentativa de diminuir seus tamanhos.

O balanceamento das expressões antes que sejam submetidas à abstração de variáveis consegue garantir tamanhos $O(n \log n)$ [8]. O agrupamento de combinadores referentes a abstrações de variáveis sucessivas em seqüências diretoras (*director strings*), também alcança o limite $O(n \log n)$, desde que combinadores repetidos sejam representados de forma compacta [35]. Outra codificação que resulta em expressões de tamanho comparável é descrita no trabalho de Noshita [44]. Como, no pior caso, o tamanho de uma expressão compilada sob um conjunto *finito* de combinadores está limitado por $\Omega(n \log n)$ [33], não há mais espaço para melhorias significativas nessa linha.

Para fugir das limitações constatadas pelo trabalho de Joy et al. [33], alguns trabalhos exploram um conjunto infinito de combinadores. Antes mesmo do trabalho de Turner, Abdali [1] havia proposto uma família infinita de combinadores K_n, I_n^m e B_n^m, parametrizados por inteiros m e n, e um algoritmo que abstrai diversas variáveis simultaneamente. Ao tomar conhecimento

do trabalho, Turner julgou que os possíveis ganhos em eficiência não justificariam o aumento na complexidade do processo de abstração de variáveis e da máquina redutora de grafos associada. Talvez, pelo compromisso com a simplicidade de seu método, Turner julgasse de forma semelhante o trabalho de Hughes [21], que é uma das bases para as técnicas descritas na próxima seção.

3.3.3 Técnicas mais recentes

Quatro avanços, posteriores ao trabalho de Turner, são responsáveis por um grande aumento na eficiência das implementações mais recentes de linguagens funcionais, como o Glasgow Haskell Compiler [30]. Os ganhos apresentados são resultado de um mapeamento mais imediato entre o modelo usado na avaliação de expressões, especialmente o método para construção de funções, e a arquitetura encontrada na maioria dos computadores modernos.

Inicialmente, Hughes [21] apresentou um algoritmo que transforma qualquer programa funcional em um conjunto equivalente de definições globais, nas quais não ocorrem variáveis livres ou funções anônimas. Após o trabalho de Johnsson [25], que estendeu o processo para lidar diretamente com definições locais e com definições recursivas, a técnica ficou conhecida como *lambda lifting*.

A eliminação de variáveis livres é realizada pela decomposição de definições complexas em várias definições mais simples, que recebem e passam essas variáveis como parâmetros adicionais. Por exemplo, a função

```
simples = \g -> ((\x -> g x 1) 2);
```

que contém a função anônima local `(\x -> g x 1)`, na qual ocorre a variável livre `g`, seria decomposta no conjunto de definições

```
$x g x = g x 1;
$simples g = g ($x g 2);
simples = $simples;
```

Note que não restam variáveis livres nem abstrações lambda e que o programa resultante é equivalente ao programa original. Funções sem variáveis livres ou funções anônimas locais são denominadas *supercombinadores*, por serem semelhantes a combinadores genéricos. A vantagem da nova representação reside no fato de que supercombinadores podem ser instanciados por uma seqüência fixa de operações que não depende do contexto onde se encontram. Entretanto, como não há um conjunto finito de supercombinadores, torna-se necessária alguma forma de codificá-los.

É a isso que se propõe o trabalho de Johnsson [24], que define uma máquina abstrata denominada *G-machine* de modo a permitir a codificação de supercombinadores por um conjunto linear de instruções. A vantagem imediata deste tipo de codificação em relação à instanciação de moldes vem do fato de que torna desnecessário o translado da definição do supercombinador em tempo de execução. O trabalho de análise é feito uma única vez, durante a compilação. Outra vantagem está no fato de que os parâmetros dos supercombinadores são substituidos simultaneamente, sem desperdícios com representações parciais intermediárias. Além disso, a codificação por um conjunto linear de supercombinadores abre espaço para uma série de otimizações que não são possíveis em codificações menos sofisticadas.

A codificação de *G-machine* se baseia em uma notação posfixada para as expressões que definem os supercombinadores. Essa codificação é posteriormente interpretada por uma máquina de

pilha. O supercombinador `$simples`, por exemplo, poderia ser codificado pelo seguinte conjunto de instruções:

```
PUSH 0;          -- empilha o argumento g
PUSHGLOBAL $x;   -- empilha a definição de $x
PUSH 2;          -- empilha g novamente
MKAPP;           -- cria a aplicação ($x g)
PUSHINT 2;       -- empilha o inteiro 2
MKAPP;           -- cria a aplicação (($x g) 2)
MKAPP;           -- cria a aplicação (g (($x g) 2))
UPDATE 1;        -- atualiza com o resultado
```

A instrução "`PUSH` n" reempilha o elemento a n casas do topo da pilha. A instrução "`MKAPP`" remove os dois elementos do topo da pilha e empilha um nó com a aplicação dos dois elementos. Finalmente, a instrução "`UPDATE` n" copia o elemento no topo da pilha sobre o elemento a n casas do topo da pilha.

Antes da execução do código acima, o topo da pilha contém o argumento `g` do supercombinador `$simples`. É possível notar que, ao final do processo, uma nova instância do corpo do supercombinador terá sido construída com seus parâmetros formais substituídos corretamente pelos argumentos correspondentes.

O terceiro avanço é resultado do trabalho de Fairbairn e Wray [15], mais tarde aprimorado por Argo [3]. A *Three Instruction Machine* apresenta uma forma mais eficiente para o tratamento da pilha de argumentos, que torna seu uso muito semelhante ao feito por linguagens imperativas. *TIM* estabelece que os argumentos de uma função devem ser removidos da pilha antes da avaliação de seu corpo. A remoção empacota os argumentos em estruturas externas à pilha, normalmente tuplas de valores. Essa remoção se faz necessária para evitar o embaralhamento entre os argumentos de funções de ordem elevada, um fenômeno conhecido como *registro de ativação partido*. Além disso, o empacotamento permite que subexpressões de uma função sejam compiladas separadamente (o que permite que sejam otimizadas independentemente) compartilhando uma mesma tupla de argumentos (o que diminui o trabalho de instanciação de cada subexpressão). A pilha passa a armazenar apenas *registros de ativação*, constituídos por conjuntos de pares. Cada par, denominado *fecho*, associa o código de uma subexpressão da função à tupla com seus argumentos.

A quarta técnica que contribui para a implementação do Glasgow Haskell Compiler é uma otimização do processo de redução de grafos, merecedora de nota. Durante o percorrimento dos grafos, máquinas anteriores ao trabalho de Koopman e Lee [38] discriminavam cada um dos diversos tipos de nó (aplicações, constantes, indireções, construtores etc) pela análise de rótulos. A principal contribuição da máquina *TIGRE*, desenvolvida pelo trabalho, é a observação de que a análise de rótulos pode ser evitada pela definição, em cada nó, de ponteiros ao código a ser executado durante sua redução, coleta e impressão. Essa codificação efetivamente transforma o grafo em uma estrutura de dados executável, que pode ser vista também como um programa automodificável. A técnica, que é usada na implementação de Sloth, é apresentada em maiores detalhes na seção 5.7.

Em uma linha consideravelmente distinta, foram desenvolvidas as *Máquinas Categóricas Abstratas*, ou *CAM*, descritas no trabalho de Cousineau et al. [11]. *CAM* é baseada na notação de de Bruijn [14] para o lambda cálculo. A notação de de Bruijn substitui nomes de variáveis por numerais representando o número de abstrações lambda que envolvem cada variável. O artifício permite que a β-redução seja realizada sem preocupações com a colisão ou captura de nomes de variáveis. Assim como em *SECD*, expressões são avaliadas na presença de ambientes. Entretanto,

em *CAM*, o acesso é feito por meio de índices que correspondem aos numerais de de Bruijn associados a cada variável. A representação de ambientes é feita por meio de pares aninhados, com a profundidade de cada elemento determinando seu índice. Expressões são compiladas para uma forma semelhante à usada pela máquina de Turner, estendida com combinadores específicos que criam e desmembram automáticamente os ambientes, à medida que o programa é avaliado.

Capítulo 4

Combinadores microprogramados

Uma análise mais detalhada do funcionamento dos combinadores de Turner, tanto os da formulação original quanto os de longo alcance, leva a constatação de que há uma grande uniformidade entre eles. Essa uniformidade se torna mais evidente entre combinadores que recebem o mesmo número de argumentos. O funcionamento dos combinadores `S`, `B` e `C`, que recebem três argumentos, é ilustrado pela figura 4.1.

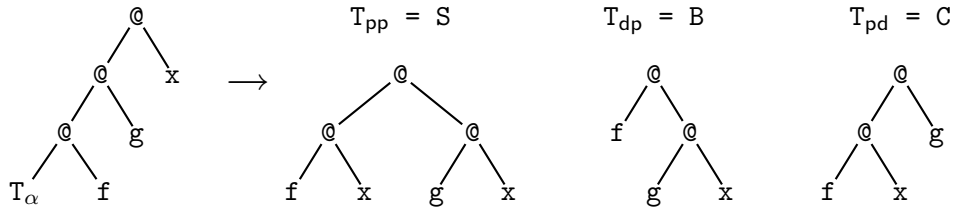


Figura 4.1: A parametrização do combinadores `S`, `B` e `C`.

Conforme mostra a figura, a tarefa realizada pelos combinadores se resume ao envio, ou não, do argumento `x` (que é o último argumento de cada combinador), aos demais argumentos (`f` e `g`, no caso). Esse comportamento pode ser modelado por um combinador genérico T_α, parametrizado por uma seqüência α com dois símbolos, que determinam se os argumentos `f` e `g` devem receber ou não o argumento `x`.

Escolhemos o símbolo `p` para denotar a passagem de `x` e o símbolo `d` para denotar seu descarte. O argumento a que se refere cada símbolo é definido pela posição do símbolo na seqüencia. O primeiro símbolo na parametrização de $B = T_{dp}$, por exemplo, se refere ao seu primeiro argumento, `f`, e determina que o argumento `x` deve ser descartado.

A mesma uniformidade é visível nas definições dos combinadores de longo alcance `S'`, `B'` e `C'`, que recebem quatro argumentos. Suas parametrizações pela família T'_α são exibidas na figura 4.2 e exigem seqüências com três símbolos.

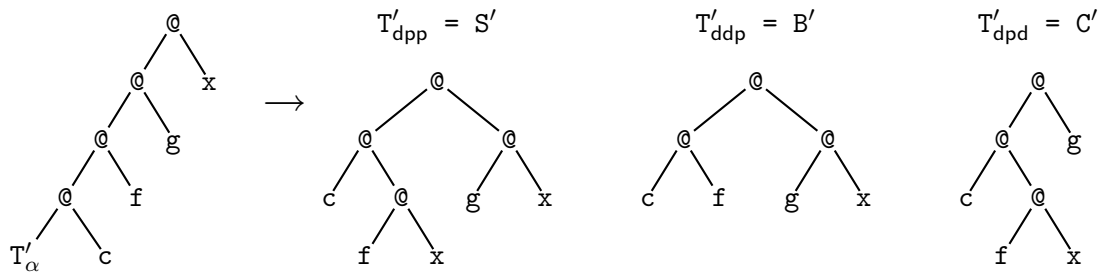


Figura 4.2: A parametrização dos combinadores `S'`, `B'` e `C'`.

A única diferença entre as famílias $\mathtt{T}_\alpha$ e $\mathtt{T}'_\alpha$ é o número de símbolos presentes no código α. Essa uniformidade leva naturalmente à generalização do tamanho das seqüências α, da qual resultam os combinadores microprogramados, representados pela família $\mathcal{L}_\alpha$ de combinadores na figura 4.3.

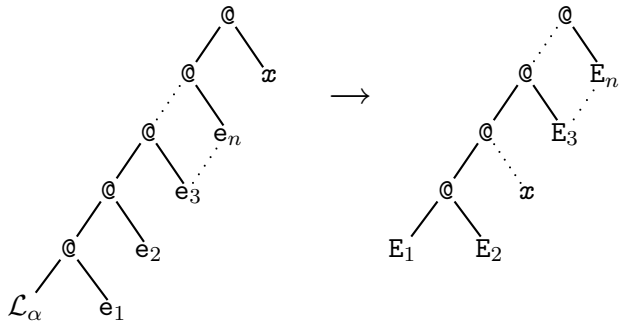


Figura 4.3: Os combinadores microprogramados.

Assim como o processo de abstração de variáveis definido na seção 3.3.2 foi estendido à medida que novos combinadores foram sendo definidos, uma nova extensão se faz necessária para permitir o uso dos combinadores microprogramados. Abaixo segue a nova regra para a abstração de variáveis em nós de aplicação. A nova regra, entretanto, alcança uma profundidade indefinida:

$$\lambda^* x.(E_1\ E_2\ E_3 \cdots E_n) \quad \underset{\lambda^*}{\Rightarrow} \quad \mathcal{L}_{(\mathsf{o}_1\mathsf{o}_2\mathsf{o}_3\cdots\mathsf{o}_n)}\ e_1\ e_2\ e_3 \cdots e_n$$

$(E_i = x) \Rightarrow$
- $e_i = \epsilon$ (omita o ramo e_i)
- $o_i = \mathsf{i}$ (identidade)

$(x \text{ oc } E_i) \Rightarrow$
- $e_i = \lambda^* x.E_i$
- $o_i = \mathsf{p}$ (passe x)

$(x \text{ o}\not{\text{c}} E_i) \Rightarrow$
- $e_i = E_i$
- $o_i = \mathsf{d}$ (descarte x)

A fórmula acima assume que a expressão E_1 não é da forma $(M\ N)$, de modo a garantir que o processo de abstração alcance a profundidade máxima possível. A cada expressão E_i corresponde uma microinstrução o_i, que depende da presença ou não da variável x na expressão E_i. Assim como na formulação original, o processo de abstração prossegue recursivamente nas expressões E_i, dando origem a expressões e_i correspondentes. Uma definição mais formal para as regras descritas acima pode ser encontrada no trabalho [43].

Uma terceira microinstrução é necessária para completar a família de combinadores microprogramados. A instrução i, de identidade, determina a construção de um ramo adicional contendo apenas a variável x, e é fundamental para a representação do combinador I. Ao contrário das microinstruções p e d, a execução da microinstrução i não diz respeito a nenhum dos argumentos e_i e simplesmente determina a posição onde deve ser inserido o ramo adicional.

Já que os combinadores de Turner podem ser interpretados como instruções compreendidas por uma máquina redutora de grafos, as operações presentes na seqüências α podem ser associadas a um nível ainda mais primitivo, correspondente ao microcódigo. Vem daí o nome "combinadores microprogramados" escolhido para a família $\mathcal{L}_\alpha$ de combinadores. Cada símbolo passa a ser identificado como uma *microinstrução*.

Usando apenas as três microinstruções, a tabela abaixo mostra como é possível a definição de todos os combinadores de Turner com o uso de combinadores microprogramados:

$$\begin{array}{ll} \mathtt{S} \equiv \mathcal{L}_{\mathsf{pp}} & \mathtt{K} \equiv \mathcal{L}_{\mathsf{d}} \\ \mathtt{B} \equiv \mathcal{L}_{\mathsf{dp}} & \mathtt{C} \equiv \mathcal{L}_{\mathsf{pd}} \\ \mathtt{I} \equiv \mathcal{L}_{\mathsf{i}} & \mathtt{S}' \equiv \mathcal{L}_{\mathsf{dpp}} \\ \mathtt{B}' \equiv \mathcal{L}_{\mathsf{ddp}} & \mathtt{C}' \equiv \mathcal{L}_{\mathsf{dpd}} \end{array}$$

Além de combinadores microprogramados equivalentes a cada combinador de Turner, o processo de abstração de variáveis descrito acima é capaz de criar novos combinadores, específicos à expressão sendo abstraída. Seguem alguns exemplos de novos combinadores criados pelo método:

```
Lpdp   f g h x     = (f x) g (h x)
Ldpdi  f g h x     = f (g x) (h x) x
Ldpip  f g h x     = f (g x) x (h x)
Ldppdd p q r s t x = p (q x) (r x) s t
```

A compilação de programas com o uso de combinadores microprogramados segue o mesmo processo descrito na seção 3.3.2, bastando apenas a adição da nova regra de abstração de variáveis. Seguindo o novo processo, o programa `simples`, usado como exemplo naquela seção, seria compilado para ($\mathcal{L}_{\mathsf{ip}}$ ($\mathcal{L}_{\mathsf{pdd}}$ $\mathcal{L}_{\mathsf{pd}}$ 1 2)). A comparação com a representação por combinadores de Turner, que resulta em (S I (C (C C 1) 2)), mostra que o número de combinadores caiu de 5 para 3. A seqüência de reduções necessárias para a avaliação do programa `main = simples add 3` com a nova versão compilada de `simples` é dada abaixo:

```
main ⇒
simples add 3 ⇒
Lip (Lpdd Lpd 1 2) add 3 →
add (Lpdd Lpd 1 2 add) 3 →
add (Lpd add 1 2) 3 →
add (add 2 1) 3 →
add 3 3 →
6
```

Os maiores benefícios da nova técnica se manifestam na compilação de expressões que exigem a eliminação de diversas variáveis. O programa ($\lambda a.\lambda b.\lambda c.\lambda d.d\ c\ b\ a$), por exemplo, que resultava na expressão (C′ (C′ C) (C′ C (C I))) caso fossem usados combinadores de Turner, é compilado para ($\mathcal{L}_{\mathsf{pd}}$ ($\mathcal{L}_{\mathsf{pdd}}$ $\mathcal{L}_{\mathsf{iddd}}$)) pela nova técnica. Onde haviam sido usados 7 combinadores, foram usados apenas 3.

Conforme notado na seção 3.3.2, expressões com menos combinadores ocupam menos espaço em memória e exigem um menor número de reduções durante sua avaliação. O menor número de reduções, por sua vez, resulta em menos representações intermediárias a serem recuperadas pelo coletor de lixo. Essas vantagens podem trazer melhorias na eficiência do processo de redução de grafos.

A análise comparativa entre as eficiências de programas compilados com combinadores de Turner e combinadores microprogramados é apresentada no capítulo 6. As implementações do processo de abstração de variáveis e da avaliação de combinadores microprogramados, usadas em Sloth, se encontram nos apêndices A.3 e A.7, respectivamente.

Capítulo 5

A implementação de Sloth

Dentre os fatores que mais influenciaram os rumos do projeto de Sloth, destacam-se as preocupações com a simplicidade, a eficiência, a portabilidade e a disponibilidade do sistema.

A preocupação com a simplicidade e com o tamanho da implementação de Sloth vem do desejo de que o sistema possa vir a ser usado em cursos sobre o desenvolvimento de linguagens de programação funcionais. Uma implementação pequena e simples, como a de Sloth, pode ser compreendida em todo seu conjunto—talvez a ponto de permitir que alunos alterem seu funcionamento—ao longo de um curso de graduação. Implementações mais complexas e sofisticadas são geralmente extensas e complicadas demais para que isso seja possível.

O cuidado com a eficiência vem do desejo de que Sloth tenha utilidade prática fora do âmbito deste trabalho, principalmente em atividades didáticas que tirem partido da simplicidade de sua especificação. A utilidade prática de Sloth seria limitada caso seu funcionamento se restringisse a uma única plataforma de desenvolvimento. Vem daí a preocupação com a portabilidade do sistema. A preocupação com a disponibilidade visa permitir que Sloth seja usada gratuitamente em aplicações acadêmicas e comerciais. Por isso, as ferramentas usadas no desenvolvimento também devem ser amplamente disponíveis.

Ocasionalmente, alguns destes fatores entram em conflito. Nesses casos, o compromisso com a simplicidade serve como árbitro na decisão pela direção a ser seguida.

5.1 Arquitetura

Durante a compilação, a natureza e quantidade das transformações estruturais pelas quais um programa é submetido, ilustradas pela seção 3.2, praticamente exigem o auxílio de algum tipo de gerência automática de memória. Evidências desse fato podem ser encontradas na implementação de Gofer [27], que usa um sistema de coleta automática de lixo atuando dentro da própria linguagem C, na qual é implementada. Apesar de engenhosa, essa solução aumenta consideravelmente a complexidade do sistema e não pode ser implementada de forma portátil, tendo sido por essas razões descartada para a construção de Sloth.

Outra alternativa, preferida pela literatura [29, 31] e adotada por um dos mais eficientes compiladores de Haskell [30], consiste no uso de uma linguagem de programação funcional para a construção de todo o sistema. De fato, essa opção é apropriada para sistemas que dispensam avaliadores, por traduzirem programas funcionais diretamente para linguagem de máquina ou para outra linguagem de programação, como é o caso da maioria dos compiladores de Haskell.

Entretanto, para tornar a implementação de Sloth mais simples e portátil, programas Sloth são avaliados por um interpretador, escrito em C. Sistemas interpretados costumam ser usados de

forma interativa pelo usuário, de modo que é desejável que não sejam visíveis os passos dados pelo sistema, desde a entrada de um programa até sua avaliação. Para isso, é necessária a integração entre o compilador e o avaliador. Infelizmente, não foi encontrada uma linguagem funcional que se integrasse bem à linguagem C sem que restringisse a portabilidade ou a disponibilidade de Sloth.

A alternativa adotada para a implementação de Sloth foi a implementação de todas as etapas anteriores a avaliação na linguagem Lua [22]. Desta forma, algumas partes de Sloth são implementadas em C, enquanto outras são escritas em Lua. Lua é uma linguagem procedural, interpretada, que oferece a flexibilidade desejada (como a coleta automática de lixo) e integra-se transparentemente à linguagem C, na forma de uma biblioteca. Além disso, Lua é uma linguagem leve, extremamente portátil e livremente disponível em código fonte.

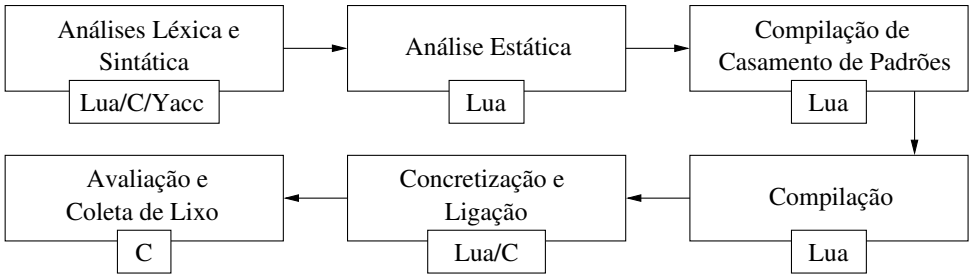


Figura 5.1: As etapas pelas quais passa um programa Sloth.

A arquitetura híbrida resultante é ilustrada pela Figura 5.1. Inicialmente, o programa Sloth é lido e sua árvore sintática é construída. Fases subseqüentes, totalmente implementadas em Lua, verificam a corretude do programa e o compilam. Em uma etapa denominada *concretização*, o programa já compilado é convertido para uma representação em C, sobre a qual o avaliador atua. Essa arquitetura permitiu a criação de um executável autocontido que associa as vantagens das linguagens de programação Lua e C. A implementação de Sloth é descrita em detalhes nas próximas seções.

5.2 Análises léxica e sintática

O analisador sintático de Sloth foi construído com o auxílio da ferramenta *Yacc* [23]. As ações semânticas associadas a cada regra constroem recursivamente a árvore sintática do programa a partir das árvores de suas partes constituintes. A cada elemento na gramática de Sloth (terminais ou não) está associada uma estrutura dinâmica com sua representação em Lua. A tabela 5.1 define a sintaxe abstrata usada na representação de programas analisados sintaticamente.

Cada elemento sintático é representado em Lua por uma tabela na qual se encontram um rótulo e campos com valores adicionais. O rótulo especifica o tipo de elemento sintático. Os demais campos definem os valores dos componentes sintáticos de cada elemento.

A representação Lua de símbolos terminais é construída pelo analisador léxico, que é escrito em C. São reconhecidas constantes numéricas, caracteres e identificadores, discriminados respectivamente pelos rótulos `number`, `char` e `ident`. Ao ler os terminais `foo` e `4.5`, por exemplo, o analisador léxico criaria as tabelas

`{tag = 'ident', value = 'foo'}` e `{tag = 'number', value = 4.5}`.

O analisador léxico coalesce, em uma única tabela, as constantes repetidas que encontra. O fato de ocorrências sucessivas de uma constante fazerem referência à uma mesma tabela Lua

Expr	::=	*Const*
	|	app (func:*Expr*) (arg:*Expr*)
	|	lambda (var:*ident*) (body:*Expr*)
	|	let (locals:[*Def*]) (body:*Expr*)
	|	letrec (locals:[*Def*]) (body:*Expr*)
	|	case (alters:[*Alt*]) (body:*Expr*)
Const	::=	ident (value:*string*)
	|	number (value:*number*)
	|	char (value:*string*)
Def	::=	def (name:*ident*) (body:[*Eqn*])
Eqn	::=	eqn (lhs:*Expr*) (rhs:*Expr*)
Alt	::=	alt (lhs:*Expr*) (rhs:*Expr*)

Tabela 5.1: Sintaxe abstrata após a análise sintática

economiza memória e simplifica o processo de comparação de constantes ao longo da implementação. Basta que as referências sejam comparadas diretamente, sem que seja necessária a comparação, campo a campo, das tabelas que representam cada constante.

O uso de tabelas Lua para a representação de todos valores semânticos associados a elementos sintáticos torna o analisador sintático mais simples e uniforme. O código C presente no analisador sintático permanece alheio aos valores internos às tabelas. A manipulação e construção de estruturas complexas é feita sempre em código Lua.

A troca de valores entre as linguagens C e Lua durante a análise sintática segue um protocolo rigoroso. Funções Lua são invocadas por meio de uma *camada de isolamento*, escrita em C. A camada de isolamento recebe as tabelas atribuídas ao lado direito da regra semântica sendo executada. Em seguida, invoca a função Lua associada à regra, passando-lhe essas tabelas. A função Lua selecionada constrói a tabela correspondente ao lado esquerdo de sua regra e a retorna à camada C.

5.2.1 Exemplo de análise sintática

O funcionamento do sistema pode ser facilmente compreendido pelo acompanhamento da expressão "`4.5+foo`", de sua leitura até a construção de sua árvore sintática. Após o reconhecimento, pelo analisador léxico, dos terminais envolvidos na expressão, será reduzida a seguinte regra sintática *Yacc*, causando a execução da ação semântica correspondente:

```
expr7 : expr7 '+' expr8
      { $$ = Syntax("binary", str2ref("add"), $1, $3, CE) }
```

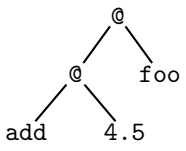
Vale lembrar que os valores semânticos associados ao lado direito da regra sintática, denotados por `$1` e `$3`, referenciam as tabelas Lua que representam os valores `4.5` e `foo`, construídas pelo analisador léxico. A função `Syntax` faz parte da camada de isolamento entre o código C do analisador sintático e o módulo Lua responsável pela criação de árvores sintáticas. A regra semântica simplesmente invoca a função Lua `Syntax.binary` e define como valor semântico da regra a tabela retornada por esta função Lua:

```
function Syntax.binary(funcname, arg1, arg2)
    return Syntax.app(Syntax.unary(funcname, arg1), arg2)
end
```

```
function Syntax.unary(funcname, arg)
    return Syntax.app(Const.ident(funcname), arg)
end

function Syntax.app(func, arg)
    return { tag = "app", func = func, arg = arg }
end
```

A função `Syntax.binary` cria uma árvore que representa a aplicação de uma função a seus dois argumentos. A função, no caso, é a função `add`, cujo identificador é passado pela auxiliar `str2ref`. O único detalhe restante consiste no uso do módulo `Const`, que é o responsável pelo coalescimento de constantes. O resultado final é mostrado na figura 5.2.




```
{ tag = 'app',
  func = { tag = 'app',
           func = { tag = 'ident', value = 'add' },
           arg = { tag = 'number', value = 4.5 } },
  arg = { tag = 'ident', value = 'foo' } }
```

Figura 5.2: Árvore sintática de `4.5+foo` e sua representação em Lua.

5.2.2 O uso de rótulos

A presença de rótulos em todos os nós permite que diversas operações sobre árvores sintáticas sejam implementadas por *tabelas de tratadores*, nas quais há uma entrada para tipo de nó. A seleção do tratador apropriado para cada nó é feita por uma função que analisa seu rótulo e invoca a entrada correspondente na tabela de tratadores. O código do invocador de tratadores é muito simples, e é reusado por diversos módulos de Sloth, que simplesmente definem tabelas de tratadores para diversas operações sobre árvores sintáticas:

```
function Driver.dispatch(driver, ...)
    local node = arg[1]
    local handler = driver[node.tag] or driver.default
    return call(handler, arg)
end
```

A operação `Const.find`, que determina se uma determinada constante ocorre em uma expressão simples poderia, por exemplo, ser implementada pelo conjunto de tratadores abaixo:

```
function Const.find(node, const)
    return Driver.dispatch(Find, node, const)
end

function Find.app(node, const)
    return Const.find(node.func, const) or Const.find(node.app, const)
end

function Find.lambda(node, const)
    return node.var == const or Const.find(node.body, const)
end

function Find.default(node, const)
    return node == const
end
```

Operações implementadas por tabelas de tratadores incluem a detecção e coleta de variáveis livres em expressões, a substituição de variáveis livres por expressões quaisquer, a abstração de variáveis, a concretização e a ligação.

5.3 Análise estática

A análise estática é responsável, dentre outras coisas, pela verificação de que a sintaxe de programas Sloth satisfaz algumas propriedades adicionais, que escapam à análise sintática. São verificadas as seguintes propriedades:

- Lados esquerdos de definições múltiplas contém o mesmo número de argumentos;
- Não ocorrem variáveis repetidas em padrões;
- Tipos são usados com o número correto de argumentos;
- Construtores nos lados esquerdos das opções em construções *case* estão saturados.

Outra tarefa realizada pela análise estática é a conversão de algumas estruturas sintáticas para novas representações, que tornam mais simples sua manipulação. Em especial, padrões são convertidos de expressões genéricas para uma representação por filas aninhadas. A nova representação torna mais simples a implementação do compilador de casamento de padrões, que realiza freqüentemente a operação de inserção e remoção de variáveis em padrões.

5.4 Compilação de casamento de padrões

A compilação do casamento de padrões, ilustrada na seção 3.2, é realizada logo após a análise estática. Apesar de constituir uma das etapas mais complexas de todo o processo de compilação, a implementação usada em Sloth segue fielmente o algoritmo descrito em Jones [29, capítulos 6 e 7] e não traz nenhuma grande inovação.

Como entrada, o compilador de casamento de padrões recebe uma definição genérica de função, possivelmente com várias alternativas:

```
Def ::= def (name:ident) (body:[Eqn])
```

O resultado da compilação é uma definição simples equivalente:

```
SimpleDef ::= simpledef (name:ident) (body:Expr)
```

O compilador de casamento de padrões também elimina expressões `case`, introduzindo as operações *switch-*⟨*Type*⟩ e *select-*⟨*field*⟩*-*⟨*constructor*⟩. A sintaxe abstrata após a compilação de casamento de padrões é exibida na tabela 5.2.

Após a compilação de casamento de padrões, o conjunto de transformações descrita na próxima seção elimina variáveis ligadas nas expressões, simplificando ainda mais a sintaxe abstrata dos programas.

5.5 Compilação para árvores combinatórias

Antes da eliminação de variáveis ligadas, os programas são submetidos a uma *análise de dependências*. A etapa se encarrega de ajustar as definições `letrec` feitas pelo usuário de modo a garantir que expressões `letrec` são usadas apenas quando necessário.

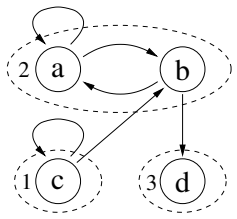
```
Expr        ::=  Const
              |  app (func:Expr) (arg:Expr)
              |  lambda (var:ident) (body:Expr)
              |  let (locals:[Def]) (body:Expr)
              |  letrec (locals:[Def]) (body:Expr)
Const       ::=  ident (value:string)
              |  number (value:number)
              |  char (value:string)
              |  switch (type:string)
              |  select (field:number) (constructor:string)
SimpleDef   ::=  simpledef (name:ident) (body:Expr])
```

Tabela 5.2: Sintaxe abstrata após a compilação de casamento de padrões

Para tanto, as definições são analisadas e os conjuntos de definições mutuamente recursivas são determinados. Esses conjuntos são isolados em uma série de expressões `let` e `letrec` aninhadas. Definições recursivas são agrupadas em expressões `letrec` e definições simples são isoladas em expressões `let`. O aninhamento final é feito respeitando-se as dependências entre os grupos.

Cada `letrec` analisado pode ser associado a um grafo G. O conjunto V de vértices do grafo corresponde ao conjunto de variáveis sendo redefinidas pelo `letrec`. O conjunto de arestas E do grafo é composto por pares (u, v_i). Para cada vértice u de V, é adicionada uma aresta partindo u para cada uma das variáveis v_i das quais depende a definição de u. O problema da análise de dependências em `letrec` corresponde aos problemas de determinação de componentes fortemente conexas e de ordenação topológica no grafo G [10, capítulo 23].

```
f = letrec
        a = a b &
        b = d a &
        c = b c &
        d = e
    in a b c d e;
```
(a)



(b)

```
f = let d = e
    in letrec
        a = a b &
        b = d a
    in letrec c = b c
    in a b c d e;
```
(c)

Figura 5.3: Análise de dependência. (a) Expressão original. (b) Grafo associado, componentes fortemente conexos e sua ordenação topológica. (c) Expressão transformada.

Definições mutuamente recursivas na expressão `letrec` pertencem ao mesmo componente fortemente conexo de G e as dependências entre os componentes é dada por sua ordenação topológica. A figura 5.3 mostra um exemplo que requer análise de dependência, o grafo associado e a expressão equivalente após a transformação.

A etapa seguinte à análise de dependências elimina todas as variáveis ligadas presentes na expressão resultante, pela introdução de combinadores. Sloth suporta mais de um método de abstração de variáveis, permitindo a opção por combinadores de Turner, por combinadores microprogramados ou por uma associação entre os dois tipos de combinador. A escolha afeta a função invocada para abstração em nós de aplicação, sem alterar o processo seguido em outros elementos sintáticos.

Em Sloth, o processo de abstração de variáveis é realizado na presença de expressões `let` e `letrec`. Expressões `let` são compiladas da mesma forma que expressões lambda, após serem transformadas pela regra

$$(\texttt{let}\ [x = M]\ \texttt{in}\ N)\ \equiv\ (\lambda x.N)\ N$$

Infelizmente, expressões `letrec` não podem ser tratadas da mesma forma, porque suas definições locais referenciam umas às outras[1]. A eliminação das ocorrências livres da variável x em uma expressão (`letrec` $[u_i = E_i]$ `in` M) deve fazer com que *cada* ocorrência de uma variável u_i redefinida pelo `letrec` receba a variável x, a não ser que *nenhuma* das definições E_i mencione x. Entretanto, se $x = u_i$, nada deve ser alterado no `letrec`, já que nenhuma ocorrência de x será livre. A solução adotada por Sloth consiste na extensão da definição λ^* do processo de abstração de variáveis para uma nova definição λ', que é consciente do conjunto de variáveis $\mathcal{F}$ que dependem da variável x sendo abstraída:

$$\lambda^* x.M \;\equiv\; \lambda'(x\,\mathcal{F}).M, \qquad \mathcal{F} = \{x\}$$

Os dois métodos funcionam de forma muito semelhante. Entretanto, sempre que $\lambda^* x.M$ toma uma decisão baseada no predicado $x \in \mathrm{FV}(M)$, a versão estendida $\lambda'(x\,\mathcal{F}).M$ se baseia no predicado $\mathcal{F} \cap \mathrm{FV}(M) \neq \emptyset$, que leva em conta o conjunto $\mathcal{F}$. Usando a versão estendida, torna-se possível a eliminação de variáveis em expressões `letrec`. O processo é definido formalmente pelas equações abaixo[2]:

$$\lambda'(x\,\mathcal{F}).(\texttt{letrec}\ [u_i = E_i]\ \texttt{in}\ M) \equiv$$
$$\begin{cases} \texttt{letrec}\ [u_i = E_i]\ \texttt{in}\ M, & \exists i(x = u_i) \\ \texttt{letrec}\ [u_i = \lambda'(x\,\mathcal{G}).E_i]\ \texttt{in}\ \lambda'(x\,\mathcal{G}).M, & \forall i(x \neq u_i) \wedge \exists i(\mathcal{F}\ \mathrm{oc}\ E_i) \wedge \mathcal{G} = \mathcal{F} \cup (\bigcup_i \{u_i\}) \\ \texttt{letrec}\ [u_i = \lambda'(x\,\mathcal{H}).E_i]\ \texttt{in}\ \lambda'(x\,\mathcal{H}).M, & \forall i(x \neq u_i \wedge \mathcal{F}\ \mathrm{o}\not{\mathrm{c}}\ E_i) \wedge \mathcal{H} = \mathcal{F} - \bigcup_i \{u_i\} \end{cases}$$

A eliminação de variáveis em expressões `letrec` segundo o processo descrito acima não resulta em expressões totalmente preguiçosas. Isto é, nas expressões resultantes, subexpressões que envolvam a variável eliminada podem vir a ser avaliadas mais de uma vez. Entretanto, definições recursivas que fazem referência a variáveis ligadas por expressões mais externas não são freqüentes. Menos freqüentes ainda são casos em que tais definições são prejudicadas seriamente pela avaliação repetida de subexpressões. Como a análise de dependências garante que `letrec` são usados somente quando indispensáveis, o problema se manifesta raramente.

```
Expr       ::=  Const
              |  app (func:Expr) (arg:Expr)
              |  letrec (locals:[SimpleDef]) (body:Expr)
Const      ::=  ident (value:string)
              |  number (value:number)
              |  char (value:string)
              |  switch (type:string)
              |  select (field:number) (constructor:string)
              |  micro (code:string)
SimpleDef  ::=  simpledef (name:ident) (body:Expr)
```

Tabela 5.3: Sintaxe abstrata após a compilação

Após a compilação, os programas já se encontram em uma forma muito simples, descrita na tabela 5.3, e estão prontos para a transformação final, que os levará a sua representação concreta.

[1] A análise de dependências garante que as definições são, de fato, mutuamente recursivas.

[2] Os predicados $\mathcal{F}$ oc M e $\mathcal{F}$ o$\not{\mathrm{c}}$ M são definidos, respectivamente, como $\mathrm{FV}(M) \cap \mathcal{F} \neq \emptyset$ e $\mathrm{FV}(M) \cap \mathcal{F} = \emptyset$.

5.6 Concretização e ligação

A *concretização* é responsável pela tradução de expressões representadas em Lua, segundo a sintaxe abstrata da tabela 5.3, para a representação esperada pelo avaliador de Sloth. Em C, as árvores combinatórias são representadas por nós do tipo `t_node`, definido no apêndice A.4. A estrutura `t_node` suporta nós de diversos tipos, incluindo nós de aplicação, constantes numéricas, nós de indireção, construtores e funções primitivas.

A *ligação* é responsável por *costurar* as expressões concretas eliminando ocorrências de variáveis livres. As variáveis encontradas são substituídas por ponteiros para as representações concretas das expressões que as definem. As definições são obtidas no escopo de alguma expressão `letrec` que seja externa à ocorrência da variável, ou no escopo global.

Na realidade, o processo de ligação é realizado ao longo do processo de concretização. Desta forma, variáveis não chegam a ser concretizadas. Se na expressão associada a uma variável y ocorre a variável x, o processo de concretização de y não prosseguirá até que obtenha a representação concreta de x. Se a representação concreta de x ainda não existir, o processo se invocará recursivamente para construir x. Na representação concreta de y, ocorrências de x são então substituídas por ponteiros para a representação concreta de x.

O tratamento de definições recursivas exige algumas precauções, entretanto. Ao iniciar a concretização de y, o processo associa a y uma representação concreta temporária. Caso a definição de y seja recursiva, a representação temporária estará disponível como se fosse a representação concreta de y. Assim, o processo não se invoca recursivamente e evita um ciclo vicioso. Ao término do processo de concretização de y, a representação temporária é sobrescrita com o resultado final. Assim, quaisquer referências que tenham selecionado a representação temporária de y ao longo de sua concretização passarão a referenciar efetivamente a representação concreta de y. Um último artifício garante que uma definição degenerada, como a função `f = f`, seja concretizada corretamente: a representação temporária é uma indireção para si mesma. Naturalmente, a avaliação dessa expressão jamais termina, o que está de acordo com a recursão imediata.

Os processos podem ser ilustrados pela concretização e ligação do programa abaixo, que define a lista com os primeiros dez números naturais:

```
naturals = 1: map (add 1) naturals;
main     = take 10 naturals;
```

As árvores abstratas para as funções `main` e `naturals` do programa acima são exibidas em cinza na figura 5.4. Na mesma figura, as representações concretas temporárias criadas durante a inicialização (auto-indireções) são exibidas em preto.

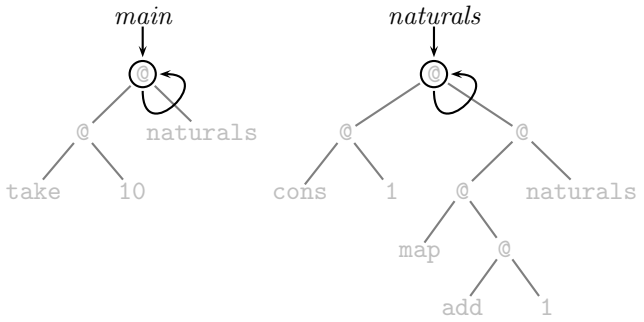


Figura 5.4: Início dos processos de concretização e ligação.

A variável `main`, que especifica a expressão a ser avaliada em um programa Sloth, é a primeira a ser selecionada para concretização. Durante o processo, são encontradas as variáveis `take` e `naturals`. Assumindo-se que a expressão concreta de `take` já tenha sido construída, o processo é interrompido apenas para a construção de `naturals`.

A concretização de `naturals`, cuja definição é recursiva, dá origem a um grafo cíclico. O resultado da concretização do programa é exibido na figura 5.5, na qual são omitidas as representações concretas das funções `take`, `map`, `cons` e `add`, que fazem parte da biblioteca padrão de Sloth.

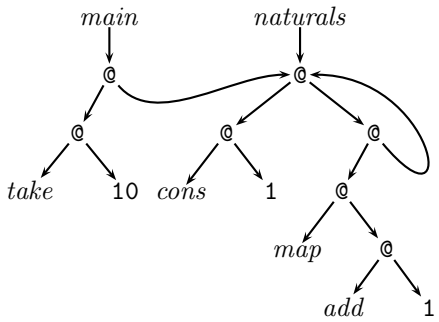


Figura 5.5: Resultado dos processos de concretização e ligação.

Após a concretização e a ligação, a expressão pode finalmente ser avaliada. O processo de avaliação seguido por Sloth é descrito na próxima seção.

5.7 Avaliação e coleta de lixo

O avaliador de Sloth recebe, como entrada, um ponteiro para a raíz do grafo combinatório concreto da expressão a ser avaliada, e imprime o resultado da avaliação no terminal. Muitas vezes, a avaliação de expressões resulta em valores estruturados. Nesses casos, o avaliador se encarrega de determinar o valor de cada um dos componentes da expressão resultante e de imprimí-los, recursivamente.

Naturalmente, o processo de avaliação de um nó depende do tipo de nó sendo avaliado. Em uma implementação tradicional, a distinção entre os possíveis tipos de nó é feita pelo uso de rótulos. O código do avaliador realiza um laço sobre um enorme comando de seleção, que analisa o rótulo dos nós encontrados e seleciona o código apropriado para a avaliação de cada nó:

```
void graph_evaluate(p_node node) {
    while (node) {
        switch (node->tag) {
            case NODE_APP:
                stack_push(spine, node->pay.app.arg);
                node = node->pay.app.func;
                break;
                ...
            case NODE_NUMBER: return;
        }
    }
}
```

Na construção de *TIGRE*, Koopman e Lee [38] observaram que o custo da análise de rótulos é responsável por uma fração considerável to tempo gasto por um redutor de grafos. Compara-

tivamente, chamadas a funções têm um custo muito mais modesto. Para tirar partido disto, na implementação de *TIGRE*, cada nó contém, no lugar de um rótulo, um ponteiro para o código responsável por sua avaliação. O grafo passa a funcionar como uma estrutura de dados executável, automodificante, e o avaliador simplesmente invoca o método do nó que deseja avaliar. A avaliação ocorre diretamente, sem nenhuma análise de rótulos.

Em Sloth, além de um ponteiro para o código redutor (`reduce`), são armazenados em cada nó ponteiros para o código que o imprime (`print`), que o coleta (`mark`) e que o compara com outros nós (`compare`). Para economizar memória, entretanto, cada nó contém na realidade um ponteiro para uma estrutura do tipo `t_driver`, que é compartilhada por todos os nós de um mesmo tipo:

```
typedef struct _t_driver {
    p_node (*reduce)  (p_node self, size_t base);
    void   (*mark)    (p_node self);
    p_node (*print)   (p_node self, int *par, int *first, int *open);
    int    (*compare) (p_node self, p_node other);
} t_driver;
```

Um último artifício, descrito no trabalho [32], garante que estruturas muito profundas não resultarão em uma cadeia de chamadas recursivas longa demais, a ponto de causar um estouro de pilha no avaliador: os métodos que implementam a redução de nós retornam um ponteiro para o próximo nó a ser avaliado ao invés de invocá-lo diretamente. O avaliador se encarrega de fazer as chamadas, mantendo o número de chamadas aninhadas ao mínimo possível. Trata-se de uma otimização explícita para chamadas recursivas finais, que dificilmente seria realizada automaticamente por um compilador para a linguagem C:

```
static p_node app_reduce(p_node self, size_t base) {
    stack_push(spine, self);
    return self->pay.app.func;
}

void graph_evaluate(p_node node) {
    size_t base = stack_gettop(spine);
    while (node)
        node = node->driver->reduce(node, base);
    mark_check(stack_getbuffer(spine), stack_gettop(spine));
    stack_settop(spine, base);
}
```

Reduções sobrescrevem as expressões encontradas no grafo combinatório com seus valores. A repetição do processo de sobrescrita resulta em uma grande quantidade de estruturas que deixam de fazer parte do grafo, e devem ser recuperadas pelo sistema de gerência de memória para uso posterior. Sloth usa o algorítmo *Mark and Sweep* [41] para a coleta de lixo. Entretanto, como não há rótulos nos nós, a marcação de nós é feita pelo método `mark` presente em cada nó.

A chamada a `mark_check`, feita ao término de cada redução, dispara o algoritmo de coleta de lixo quando determina, heuristicamente, que há muitos nós a serem recuperados. Em Sloth, o algorítmo é disparado sempre que não restam mais nós disponíveis ou quando o número de nós alocados supera o dobro do número de nós efetivamente em uso, conforme determinado durante o ciclo de coleta anterior.

Maiores informações sobre a implementação da avaliação e da coleta de lixo podem ser encontradas nos apêndices A.5 e A.6.

Capítulo 6

Testes de eficiência

Este capítulo apresenta os resultados obtidos com os testes de eficiência realizados. Os testes tiveram dois objetivos principais: a comparação da eficiência dos combinadores microprogramados com a eficiência dos combinadores de Turner e a comparação entre a eficiência da implementação de Sloth e as das implementações de outras linguagens semelhantes. Os primeiros testes tentaram medir a utilidade da nova técnica de compilação proposta no capítulo 4. Os demais testes buscaram investigar se a implementação de Sloth é eficiente o suficiente para uso didático.

6.1 Testes realizados

Os testes realizados foram baseados em medições durante a realização de cinco *tarefas* distintas. Cada tarefa foi realizada por interpretadores diferentes, com diversos parâmetros, totalizando dez *modalidades*. Para cada modalidade em cada uma das tarefas, foram realizadas dez medições de tempo de execução, das quais foram eliminados o maior e o menor tempo. Os valores registrados correspondem à média das oito medições restantes.

As cinco tarefas são as seguintes:

hanoi Cálculo da lista com os movimentos necessários para a solução do problema das Torres de Hanoi com 15 discos. A solução é um programa simples, que resulta em uma saída extensa de dados;

primes Determinação da lista com os primeiros 600 números primos pelo uso direto do Crivo de Eratóstenes. A implementação é essencialmente procrastinante e exige uma grande quantidade de operações aritméticas;

quick Ordenação de uma lista com 1500 números pelo método *quick sort*;

insert Ordenação de uma lista com 1000 números pelo método *insertion sort*;

merge Ordenação de uma lista com 1500 números pelo método *merge sort*.

A comparação entre a eficiência dos combinadores de Turner e a eficiência dos combinadores microprogramados foi realizada com o auxílio do interpretador de Sloth, que é capaz de abstrair variáveis por diversos métodos.

As modalidades criadas para este fim foram as seguintes:

sloth t-m Abstração com o uso exclusivo de combinadores de Turner. Durante a avaliação, entretanto, cada combinador de Turner é simulado por um combinador microprogramado equivalente;

sloth t Abstração com o uso exclusivo de combinadores de Turner;

sloth m Abstração exclusivamente por combinadores microprogramados;

sloth m:t Abstração por combinadores microprogramados com uso de combinadores de Turner substituindo todos os combinadores microprogramados equivalentes;

sloth m:t:nv Combinadores microprogramados junto a combinadores de Turner. O processo de avaliação assume que os programas estão corretos e não realiza nenhuma verificação de tipos.

A eficiência de Sloth também foi comparada com três implementações de interpretadores para linguagens funcionais puras e procrastinantes como Sloth: Gofer, Hugs e Hope. Gofer [26] e Hugs [28] são interpretadores para linguagens muito próximas a Haskell, ambos baseados em supercombinadores construídos por máquinas semelhantes à *G-machine*. Já o interpretador de Hope [7] é baseado na *máquina de Krivine* [12], semelhante a uma máquina categórica. As três implementações realizam a verificação estática de tipos antes da avaliação.

A grande semelhança entre as linguagens permitiu que os programas usados para os testes com Sloth fossem facilmente traduzidos para Hugs, Hope e Gofer. Todos os testes são autocontidos e não usam funções de biblioteca, para garantir que os programas testados sejam o mais semelhantes possível. Os testes em Sloth e algumas das traduções para as outras linguagens encontram-se no apêndice B.

As modalidades criadas foram as seguintes:

gofer Interpretador Gofer;

gofer nd Interpretador Gofer com programa equivalente, com a omissão dos protótipos de função;

hugs Interpretador Hugs;

hugs nd Interpretador Hugs com programa equivalente, com a omissão dos protótipos de função;

hope Interpretador Hope (protótipos obrigatórios).

As medições foram realizados em um computador com processador Pentium III 750Mhz, com 256MB de memória principal, rodando o sistema Redhat Linux 7.2, com *kernel* 2.4.7–10. Além do tempo (em segundos) gasto por cada teste, as tabelas abaixo exibem duas informações que não dependem da eficiência do sistema usado nos teses: o número de reduções realizadas e o número de ciclos de coleta de lixo executados.

hanoi			
modalidade	tempo	reduções	coletas
sloth t-m	3,63		
sloth t	3,32	6012901	22
sloth m	3,30	5128162	20
sloth m:t	3,08		
sloth m:t:nv	3,06		
gofer	1,35	507899	2
gofer nd	1,34		
hugs	3,32	2605014	21
hugs nd	3,29		
hope	3,57		8

primes			
modalidade	tempo	reduções	coletas
sloth t-m	4,32		
sloth t	3,84	6274704	23
sloth m	3,48	5321723	20
sloth m:t	3,46		
sloth m:t:nv	3,46		
gofer	1,05	1141939	1
gofer nd	1,05		
hugs	2,10	2105950	10
hugs nd	14,1		
hope	35,0		25

quick			
modalidade	tempo	reduções	coletas
sloth t-m	1,50		
sloth t	1,41	1653322	6
sloth m	1,26	1300728	5
sloth m:t	1,20		
sloth m:t:nv	1,20		
gofer	0,36	187357	0
gofer nd	0,37		
hugs	0,65	381562	2
hugs nd	0,77		
hope	1,66		4

insert			
modalidade	tempo	reduções	coletas
sloth t-m	6,46		
sloth t	6,13	6674882	28
sloth m	5,73	5017867	24
sloth m:t	5,24		
sloth m:t:nv	5,24		
gofer	0,65	421020	0
gofer nd	—		
hugs	2,26	2086964	11
hugs nd	—		
hope	6,54		15

merge			
modalidade	tempo	reduções	coletas
sloth t-m	1,68		
sloth t	1,60	1905063	7
sloth m	1,44	1602717	6
sloth m:t	1,39		
sloth m:t:nv	1,37		
gofer	0,18	76389	0
gofer nd	0,18		
hugs	0,38	215333	1
hugs nd	0,49		
hope	0,97		2

6.2 Análise dos resultados

Os valores apresentados nas tabelas acima fornecem informações sobre diversos aspectos das implementações analisadas. Os resultados são discutidos a seguir.

6.2.1 Microprogramados × Turner

Combinadores microprogramados são interpretados a partir de seu microcódigo pelo tratador `micro_reduce`, apresentado no apêndice A.7. Por outro lado, combinadores de Turner são avaliados diretamente por código compilado, como exemplificam os tratadores `S_reduce`, `K_reduce` e `I_reduce` no mesmo apêndice. A análise das modalidades sloth t-m e sloth t mostra que o impacto causado exclusivamente pela interpretação de microcódigo é inferior a 10% (e geralmente próximo aos 5%) do tempo total de execução.

Mesmo prejudicados pela interpretação, a introdução de combinadores microprogramados específicos para os programas sendo compilados mostrou-se benéfica em todas as tarefas realizadas. Excetuando-se a tarefa **hanoi**, na qual os resultados obtidos foram muito próximos, a comparação entre as modalidades sloth t e sloth m mostra melhoras em torno dos 10%. O programa **hanoi** é o mais simples de todos e conseqüentemente o menos afetado por combinadores de longo alcance.

As vantagens são, em parte, resultado do menor número de reduções realizadas pela nova técnica. Os testes realizados apresentaram uma queda de 15 a 20% no número de reduções. Esta

diminuição resulta em um menor número de estruturas intermediárias a serem coletadas, o que é confirmado pelo menor número de invocações ao coletor de lixo.

Naturalmente, combinadores microprogramados e combinadores de Turner podem ser usados em conjunto. A modalidade sloth m:t tira partido desta observação na tentativa de alcançar resultados melhores. O uso conjunto é sempre vantajoso e foi mais eficaz na tarefa **insert**, cujo tempo de execução foi reduzido em 8% em relação à modalidade sloth m.

A vantagem dos combinadores microprogramados sobre o uso exclusivo de combinadores de Turner pode ser medida pela comparação entre as modalidades sloth t e sloth m:t. Os resultados são apresentados na tabela abaixo:

	hanoi	**primes**	**quick**	**insert**	**merge**
sloth t	3,32	3,84	1,41	6,13	1,60
sloth m:t	3,08	3,46	1,20	5,24	1,39
ganho	7%	10%	15%	15%	13%

O ganho apresentado na tabela acima mostra em quantos pontos percentuais o tempo da modalidade sloth t precisa ser reduzido para alcançar o tempo da modalidade sloth m:t.

6.2.2 Sloth × Gofer, Hugs e Hope

A comparação entre a eficiência de implementações distintas para linguagens de programação funcionais distintas é uma tarefa complexa, que envolve muitos detalhes. O trabalho de Hartel e Langendoen [18] analisa algumas das dificuldades encontradas e propõe que os programas usados para testes envolvam apenas construções simples, existentes em todas as linguagens, de modo a não prejudicar qualquer uma das implementações sendo testadas. Este conselho é seguido pelos programas usados nos testes realizados neste trabalho.

Fatores que contribuem para a maior ou menor eficiência de uma implementação incluem a técnica usada para a implementação de funções, o método empregado na gerência automática de memória, o consumo de memória, a técnica empregada para a manipulação de dados estruturados, a qualidade do compilador de casamento de padrões, o uso ou não de verificação estática de tipos, a qualidade da implementação em si e até mesmo a forma como os resultados são exibidos. Em geral, não se pode afirmar qual dos fatores é o maior responsável pelos resultados obtidos sem um profundo conhecimento sobre o funcionamento interno de cada linguagem analisada. Por isso, as observações a seguir não são conclusões definitivas.

A dificuldade em chegar-se a conclusões definitivas pode ser ilustrada pela comparação entre as modalidades hope e sloth m:t. Nas tarefas **hanoi**, **quick** e **insert**, Hope mostrou-se de 15 a 28% mais lenta que Sloth. Na tarefa **primes**, Hope apresentou grandes dificuldades, chegando a ser dez vezes mais lenta que Sloth. Entretanto, na tarefa **merge**, Hope supera Sloth em 40%, o que mostra que não se pode afirmar que Sloth é *sempre* mais eficiente.

Outras tendências parecem ser mais constantes, como a superioridade de Gofer. A comparação entre as modalidades gofer e sloth m:t mostra que Gofer é consideravelmente mais eficiente. O resultado já era esperado, após a análise de um trabalho mais recente de Hartel [17]. Neste trabalho, uma comparação entre Gofer e Miranda mostra que Gofer é três vezes mais eficiente. Como Miranda [51] é uma implementação comercial, resultado da pesquisa do próprio Turner com a redução combinatória de grafos, é natural que a tendência se mantivesse em relação a Sloth. Nas tarefas de ordenação, nas quais as vantagens de Gofer se manifestaram mais, a implementação chegou a ser oito vezes mais eficiente que Sloth.

A comparação entre o número de reduções realizadas por Gofer e Sloth mostra que a redução de grafos combinatórios, mesmo com o uso dos combinadores microprogramados, ainda está muito distante de alcançar a maior granularidade apresentada pelos supercombinadores. Para cada redução executada por Gofer, Sloth parece realizar mais de dez reduções. A maior diferença foi encontrada na tarefa **merge**, para a qual Sloth precisou de vinte vezes mais reduções que Gofer.

Mesmo assim, não se pode atribuir a maior eficiência de Gofer exclusivamente às técnicas empregadas para a avaliação de funções. Basta uma simples alteração, que substitui a função `filter` pela função `filter'` na tarefa **primes**, para inverter os resultados:

```
filter _ []     = []
|    f (x:xs) = let xs' = (filter f xs) in if f x then x : xs' else xs'

filter' f l = if l == [] then []
              else let h = head l in if f h then h: (filter' f (tail l))
                   else filter' f (tail l);
```

Sloth mantém seu tempo relativamente estável, passando de 3,54s para 4,73s. Gofer, por outro lado, sofre muito com a alteração e passa de 1,35s para 6,22s. O resultado serve como uma indicação de que uma das vantagens de Gofer pode estar relacionada a uma implementação mais eficiente do casamento de padrões e impede que se afirme que Gofer seja sempre mais eficiente que Sloth.

Hugs também obteve resultados melhores que os de Sloth, em geral, apesar de menos expressivos que os de Gofer. É curioso notar a maior eficiência de Gofer em relação a Hugs, já que Hugs foi desenvolvida mais recentemente pelo mesmo autor de Gofer (Mark P. Jones). Parte da menor eficiência de Hugs pode ser atribuída a sua implementação de sobrecarga de operadores. Em alguns casos, quando a verificação estática de tipos falha em determinar unicamente uma das opções disponíveis, a escolha precisa ser feita em tempo de execução. Para demonstrar esse fato, foram criadas as tarefas gofer nd e hugs nd. Os programas usados são versões modificadas das tarefas gofer e hugs, das quais foram removidos os protótipo de função. Conseqüentemente, toda a resolução de polimorfismo precisa ser feita por inferência automática de tipos. Em alguns casos, o interpretador pode até mesmo exigir a declaração de protótipos, como ocorreu com a tarefa **insert**.

A ausência de protótipos de função não chegou a prejudicar Gofer. Entretanto, excetuando-se a simples tarefa **hanoi**, é possível perceber o impacto sofrido por Hugs. Em especial, a tarefa **primes** foi executada de forma sete vezes mais lenta na modalidade hugs nd que na modalidade hugs, sendo superada largamente por Sloth.

Como a implementação de Sloth não submete os programas a uma verificação estática de tipos, o avaliador é obrigado a realizar algumas verificações em tempo de execução. A modalidade sloth m:t:nv, que não realiza nenhuma dessas verificações, foi criada para permitir a medição do impacto das verificações dinâmicas no tempo de avaliação dos programas. A comparação com a modalidade sloth m:t, que realiza todas as verificações necessárias, mostra que o impacto é desprezível.

O próximo capítulo apresenta conclusões mais gerais sobre os resultados obtidos e conclui o trabalho com a proposta de novos rumos a serem seguidos nas mesmas linhas de pesquisa.

Capítulo 7

Conclusão

O principal objetivo deste trabalho foi a criação de uma linguagem de programação que pudesse ser disponibilizada para uso acadêmico em atividades relacionadas ao ensino do uso e da implementação de linguagens de programação funcionais.

Para isso, era fundamental que tanto a linguagem quanto sua implementação fossem o mais simples possível. A simplicidade da linguagem foi alcançada com a escolha de um subconjunto de linguagens consagradas, porém com uma sintaxe leve e de formato livre. A simplicidade da implementação foi alcançada com uso de técnicas conhecidas em uma arquitetura inovadora, que associa as vantagens de duas linguagens de programação.

Na prática, a eficiência de uma linguagem de programação é um dos fatores que determinam a sua utilidade. Na tentativa de conciliar eficiência e simplicidade, foi criada uma nova técnica para a implementação de linguagens funcionais, os combinadores microprogramados, baseados na redução de grafos combinatórios.

A análise de eficiência apresentada no capítulo 6 leva à conclusão de que, apesar dos ganhos em eficiência provenientes do uso da nova técnica de compilação terem sido modestos, o resultado final não deixa nada a desejar em relação a outras implementações de linguagens semelhantes. De fato, mesmo sem o uso dos combinadores microprogramados, Sloth se mostrou eficiente o suficiente para o uso em aplicações práticas.

Concluindo, apesar das vantagens dos combinadores microprogramados em relação aos combinadores de Turner não terem sido tão grandes quanto o desejado, todos os principais objetivos do trabalho foram alcançados.

7.1 Trabalhos futuros

A criação de Sloth abre espaço para uma série de trabalhos futuros nas linhas de pesquisa discutidas nessa dissertação. Os projetos propostos a seguir têm como objetivo a melhoria da linguagem, a simplificação de sua implementação, o aumento de sua eficiência e um aprofundamento da comparação entre diferentes técnicas para a implementação de linguagens funcionais:

- A quantidade de código C presente na implementação de Sloth poderia ser reduzida consideravelmente caso o analisador sintático fosse reescrito em Lua. As alternativas são a criação de uma ferramenta semelhante ao *Yacc*, porém que gere código Lua, ou a implementação manual do analisador;
- Como uma possível melhoria à eficiência de Sloth, poderiam ser experimentadas algumas das técnicas de coleta de lixo descritas no trabalho de Wilson [52], em especial técnicas

cujo tempo de execução seja proporcional à memória efetivamente em uso, e não ao total de memória alocada;

- Correndo o risco de prejudicar a simplicidade da implementação atual, poderia ser implementada a redução de grafos por supercombinadores, representados por instruções de *G-machine*. Além da possibilidade de aumento na eficiência da implementação, a adição tornaria possível uma comparação mais precisa entre esta técnica e a redução de grafos combinatórios;

- Para auxiliar o processo de programação em Sloth, seria interessante a adição de um verificador estático de tipos. Na mesma linha, as mensagens de erros geradas pelo compilador poderiam ser mais específicas, de modo a tornar mais simples a identificação de suas causas;

- Como melhorias à linguagem em si, poderiam ser adicionados o suporte a abrangências de listas e construções do tipo `where`, que permitam padrões genéricos nos lados esquerdos das definições;

- Finalmente, a biblioteca padrão de Sloth poderia ser estendida para tornar-se mais semelhante à de Haskell. Neste caso, a criação de um sistema de módulos seria útil facilitar a organização das bibliotecas.

Apêndice A

Trechos selecionados da implementação de Sloth

A.1 Transformações estruturais

O módulo `structure.lua` oferece uma forma confortável para a transformação de estruturas sintáticas. O usuário fornece a expressão original e dois moldes. O molde `pat` especifica a forma esperada da expressão original e subexpressões a serem capturadas. A função `Structure.transform` analisa a expressão a ser transformada, tentando casá-la com o molde `pat`. Caso obtenha sucesso, a função cria uma expressão de acordo com o molde `repl`, inserindo as subexpressões capturadas nos locais especificados por `repl`. A implementação do módulo é dada abaixo:

```
function match(expr, pat, cap)
    if type(pat) == "string" then
        if pat == "*" then tinsert(cap, expr) return cap
        elseif Const.ident(pat) == expr then return cap end
    elseif type(pat) == "table" and expr.tag == "app" then
        return match(expr.func, pat[1], cap) and match(expr.arg, pat[2], cap)
    end
end

function replace(cap, repl)
    if type(repl) == "table" then
        return Syntax.app(replace(cap, repl[1]), replace(cap, repl[2]))
    elseif type(repl) == "number" then return cap[repl]
    else return Const.ident(repl) end
end

function Structure.transform(expr, pat, repl)
    local cap = match(expr, pat, {})
    if cap then return replace(cap, repl) end
    return expr
end
```

O módulo de transformações estruturais é usado durante a compilação de casamento de padrões e durante a abstração de variáveis pelo método de Turner.

A.2 Abstração de variáveis (Turner)

A implementação da abstração de variáveis pelo método de Turner em Sloth segue uma formulação baseada em otimizações sucessivas, freqüentemente encontrada na literatura:

$$\begin{aligned}
\lambda^* x.(M\ N) &\underset{\lambda^*}{\twoheadrightarrow} \text{Opt}[\![\texttt{S}\ (\lambda^* x.M)\ (\lambda^* x.N)]\!] \\
\text{Opt}[\![\texttt{S}\ (\texttt{K}\ M)\ (\texttt{K}\ N)]\!] &= \texttt{K}\ (M\ N) \\
\text{Opt}[\![\texttt{S}\ (\texttt{K}\ M)\ \texttt{I}]\!] &= M \\
\text{Opt}[\![\texttt{S}\ (\texttt{K}\ M)\ N]\!] &= \texttt{B}\ M\ N \\
\text{Opt}[\![\texttt{S}\ (\texttt{B}\ M\ N)(\texttt{K}\ P)]\!] &= \texttt{C}'\ M\ N\ P \\
\text{Opt}[\![\texttt{S}\ M\ (\texttt{K}\ N)]\!] &= \texttt{C}\ M\ N \\
\text{Opt}[\![\texttt{S}\ (\texttt{B}\ M\ N)\ P]\!] &= \texttt{S}'\ M\ N\ P
\end{aligned}$$

O uso da linguagem Lua torna muito simples a implementação direta da formulação acima, sem receio com a coleta das representações intermediárias. Para tanto, basta que seja usado o módulo para transformações estruturais descrito no apêndice A.1:

```
function optimize(expr)
    expr = Structure.transform(expr, {{'S', {'K', '*'}}, {'K', '*'}},
        {'K', {1, 2}})
    expr = Structure.transform(expr, {{'S', {'K', '*'}}, 'I'}, 1)
    expr = Structure.transform(expr, {{'S', {'K', '*'}}, '*'}, {{'B', 1}, 2})
    expr = Structure.transform(expr, {{'S', {{'B', '*'}, '*'}}, {'K', '*'}},
        {{{"C'", 1}, 2}, 3})
    expr = Structure.transform(expr, {{'S', '*'}, {'K', '*'}}, {{'C', 1}, 2})
    expr = Structure.transform(expr, {{'S', {{'B', '*'}, '*'}}, '*'},
        {{{"S'", 1}, 2}, 3})
    return expr
end

function Turner.abstract(app, var, funcs)
    return optimize(Syntax.app(Syntax.app(Const.ident("S"),
        Bracket.abstract(app.func, var, funcs)),
        Bracket.abstract(app.arg, var, funcs)))
end
```

A.3 Abstração de variáveis (microprogramados)

Esta seção apresenta a implementação do processo de abstração de variáveis pelo uso de combinadores microprogramados, definida no capítulo 4.

A função `collect` realiza a maior parte do trabalho, coletando cada um dos componentes E_i da expressão $(E_1\ E_2 \cdots E_n)$ em `expr`, abstraindo recursivamente a variável x (`var`) em cada E_i e formando a cadeia de caracteres com o microcódigo α (`code`) do combinador microprogramado a ser usado.

Conforme descrito na seção 5.4, a abstração é realizada na presença de expressões `letrec`. Por isso, é necessário o uso da versão estendida $\lambda'(x\,\mathcal{F}).E$ do processo, que mantém um conjunto $\mathcal{F}$ (`funcs`) com as variáveis que dependem da variável `var` sendo abstraída.

```
function collect(expr, var, funcs)
    local code = ""
    local arg = {}
    while expr.tag == "app" do
        local fv_arg = Freevar.find(expr.arg, funcs)
        local fv_func = Freevar.find(expr.func, funcs)
        if fv_arg then
            if expr.arg ~= var then
                tinsert(arg, Bracket.abstract(expr.arg, var, funcs))
                code = "p" .. code
            else code = "i" .. code end
        else
            if fv_func then
                tinsert(arg, expr.arg)
                code = "d" .. code
            else break end
        end
        expr = expr.func
    end
    if expr == var then code = "i" .. code
    elseif Freevar.find(expr, funcs) then
        tinsert(arg, Bracket.abstract(expr, var, funcs))
        code = "p" .. code
    else
        tinsert(arg, expr)
        code = "d" .. code
    end
    return arg, code
end
```

A função `Microcode.abstract` cria a expressão $\mathcal{L}_\alpha\ e_1\ e_2 \cdots e_n$ a partir das expressões abstraídas e_i e do microcódigo α retornado pela função `collect`. Na versão apresentada abaixo, ocorrências de combinadores microprogramados para os quais existe um combinador de Turner correspondente são substituídas por combinadores de Turner.

```
function Microcode.abstract(expr, var, funcs)
    local arg, code = collect(expr, var, funcs)
    code = gsub(code, "^di", "p")
    local app = Syntax.micro(code)
    for i = getn(arg), 1, -1 do app = Syntax.app(app, arg[i]) end
    if code == "p" then app = app.arg
    elseif code == "i" then app = Const.ident("I")
    elseif code == "d" then app.func = Const.ident("K")
    elseif code == "pp" then app.func.func = Const.ident("S")
    elseif code == "pd" then app.func.func = Const.ident("C")
    elseif code == "dp" then app.func.func = Const.ident("B")
    elseif code == "dpp" then app.func.func.func = Const.ident("S'")
    elseif code == "dpd" then app.func.func.func = Const.ident("C'")
    end
    return app
end
```

A.4 Representação concreta

```
/* mark and sweep garbage collection control structure */
typedef struct _t_gc {
    p_node next;                /* next node in allocation chain */
    p_node flag;                /* marked flag and chain */
} t_gc;

/* numeric data type */
typedef double t_num;

/* character data type */
typedef char   t_char;

/* microcode combinator */
typedef struct _t_micro {
    cchar *code;                /* microcode to execute */
    ushort arity;               /* combinator reach */
} t_micro;
typedef t_micro *p_micro;

/* unsaturated constructor */
typedef struct _t_con {
    ushort type_id;             /* constructor type id */
    ushort con_id;              /* constructor id within its type */
    ushort arity;               /* number of arguments to saturate */
} t_con;
typedef t_con *p_con;

/* application node */
typedef struct _t_app {
    p_node func;                /* function value */
    p_node arg;                 /* argument value */
} t_app;
typedef t_app *p_app;

/* sum type switch operation */
typedef struct _t_swt {
    ushort type_id;             /* type id of sum being inspected */
    ushort arity;               /* number of options to choose from */
} t_swt;
typedef t_swt *p_swt;

/* product field selector */
typedef struct _t_sel {
    ushort type_id;             /* constructor type id */
    ushort con_id;              /* constructor id within its type */
    ushort arg;                 /* argument number to select */
} t_sel;
typedef t_sel *p_sel;
```

```
/* indirection node */
typedef struct _t_ind {
    p_node follow;              /* node to follow */
    cchar *name;                /* indirection node name (debug) */
} t_ind;
typedef t_ind *p_ind;

/* node payload defition */
typedef union {
    t_num num;                  /* numeric constant */
    t_char chr;                 /* character constant */
    t_con con;                  /* constructor */
    t_app app;                  /* application node */
    t_micro micro;              /* microcode combinator */
    t_sel sel;                  /* product field selector */
    t_swtch swt;                /* sum type switch operation */
    t_ind ind;                  /* indirection node */
} u_pay;
typedef u_pay *p_pay;

/* run-time type-checking tags */
typedef enum _e_tag {
    TAG_NUMBER,                 /* a number */
    TAG_CHAR,                   /* a character */
    TAG_CELL,                   /* a saturated cell */
    TAG_CON,                    /* a constructor */
    TAG_FAIL,                   /* a failure node */
    TAG_IND,                    /* an indirection node */
    TAG_UNTYPED                 /* not a value */
} e_tag;

/* node driver definition */
typedef struct _t_driver *p_driver;
typedef struct _t_driver {
    p_node (*reduce)  (p_node self, size_t base);
    void   (*mark)    (p_node self);
    p_node (*print)   (p_node self, int *par, int *first, int *open);
    int    (*compare) (p_node self, p_node other);
    e_tag tag;
} t_driver;

/* node definition */
typedef struct _t_node *p_node;
typedef struct _t_node {
    p_driver driver;            /* node driver */
    u_pay pay;                  /* node payload */
    t_gc gc;                    /* garbage collection control */
} t_node;
```

A.5 Redução de grafos

A função `graph_print` é a responsável pela exibição do valor de expressões. O esquema de impressão é semelhante ao sistema de avaliação, com nós retornando nós subseqüentes a serem impressos. Desta forma, campos de dados estruturados podem ser coletados antes do término da impressão do valor estruturado, o que é fundamental em estruturas recursivas potencialmente longas, como listas. Naturalmente, antes de exibir o valor do nó que recebe, `graph_print` invoca `graph_evaluate` para avaliá-lo.

```
void graph_print(p_node node, int par) {
    int open = 0, first = 1;
    graph_evaluate(node);
    while (node)
        node = node->driver->print(node, &par, &first, &open);
    while (open-- > 0) putchar(')');
}
```

Ambas as funções garantem que todos os nós ainda em uso pelo processo de avaliação estão visíveis a partir da pilha, de modo a evitar que sejam coletados injustamente. Por outro lado, as funções certificam-se também de que apenas os nós necessários permanecem na pilha, para evitar que estruturas desnecessárias permaneçam alocadas.

```
void graph_evaluate(p_node node) {
    size_t base = stack_gettop(spine);
    while (node)
        node = node->driver->reduce(node, base);
    mark_check(stack_getbuffer(spine), stack_gettop(spine));
    stack_settop(spine, base);
}
```

A.6 Tratadores selecionados

```
static p_node app_reduce(p_node self, size_t base) {
    stack_push(spine, self);
    return self->pay.app.func;
}

static p_node add_reduce(p_node self, size_t base) {
    if (stack_gettop(spine) >= base + 2) {
        p_node res, arg2, arg1;
        stack_peek(spine, 2, res);
        stack_peekarg(spine, 2, arg2);
        stack_peekarg(spine, 1, arg1);
        graph_evaluate(arg2); number_check(arg2);
        graph_evaluate(arg1); number_check(arg1);
        number_make(res, arg1->pay.num + arg2->pay.num);
        stack_drop(spine, 2);
    }
    return NULL;
}

static p_node number_print(p_node self, int *par, int *first, int *open) {
    printf("%g", self->pay.num);
    return NULL;
}

static p_node tuple_print(p_node self, int *par, int *first, int *open) {
    ushort arity = self->pay.con.arity;
    ushort i; p_node arg;
    putchar('(');
    (*open)++;
    for (i = 1; i < arity; i++) {
        stack_peek(spine, i, arg);
        graph_print(arg, 0);
        putchar(',');
    }
    stack_peek(spine, arity, arg);
    graph_evaluate(arg);
    stack_drop(spine, arity);
    *first = 1;
    *par = 0;
    return arg;
}

static void app_mark(p_node self) {
    mark_enqueue(self->pay.app.func);
    mark_enqueue(self->pay.app.arg);
}

int char_compare(p_node self, p_node other) {
    char_check(other);
    return self->pay.chr == other->pay.chr;
}
```

A.7 Avaliação de combinadores

O resultado da primeira microinstrução encontrada deve ser posicionado à esquerda de um nó de aplicação enquanto o resultado das demais deve ser posicionado á direita de um nó de aplicação. De modo a facilitar o trabalho do interpretador de microinstruções, o microcódigo é ligeiramente alterado antes de ser passado ao interpretador. A primeira instrução é sempre representada por uma letra maiúscula correspondente.

Outra otimização conveniente faz distinção entre a última microinstrução e as demais, já que apenas a última microinstrução deve sobrescrever a raíz da expressão com o resultado da avaliação do combinador. Essa distinção é feita com a alteração, no microcódigo, da última instrução por sua sucessora em ordem alfabética.

As duas otimizações transformam o microcódigo 'ppdid', por exemplo, em 'Ppdie'. Abaixo, segue a implementação completa do avaliador de combinadores microprogramados em Sloth:

```
p_node micro_reduce(p_node self, size_t base) {
    ushort arity = self->pay.micro.arity;
    if (stack_gettop(spine) >= base + arity) {
        p_node arg = NULL, res, tmp, app = NULL;
        cchar *code = self->pay.micro.code;
        stack_peek(spine, arity, res);
        stack_peekarg(spine, arity, arg);
        while (*code) {
            switch (*code) {
                case 'I': app = arg; break;
                case 'i': app = app_make(mark_new(), app, arg); break;
                case 'j': app_make(res, app, arg); break;
                case 'D': stack_poparg(spine, app); break;
                case 'd': stack_poparg(spine, tmp);
                    app = app_make(mark_new(), app, tmp);
                    break;
                case 'e': stack_poparg(spine, tmp);
                    app = app_make(res, app, tmp);
                    break;
                case 'P': stack_poparg(spine, app);
                    app = app_make(mark_new(), app, arg);
                    break;
                case 'p': stack_poparg(spine, tmp);
                    app = app_make(mark_new(), app,
                            app_make(mark_new(), tmp, arg));
                    break;
                case 'q': stack_poparg(spine, tmp);
                    app_make(res, app, app_make(mark_new(), tmp, arg));
                    break;
            }
            code++;
        }
        stack_drop(spine, 1);
        return res;
    }
    return NULL;
}
```

```
p_node S_reduce(p_node self, size_t base) {
    if (stack_gettop(spine) >= base + 3) {
        p_node res, f, g, x;
        stack_peek(spine, 3, res);
        stack_peekarg(spine, 3, x);
        stack_peekarg(spine, 2, g);
        stack_peekarg(spine, 1, f);
        app_make(res, app_make(mark_new(), f, x), app_make(mark_new(), g, x));
        stack_drop(spine, 3);
        return res; /* go on reducing from res */
    }
    return NULL;
}

p_node K_reduce(p_node self, size_t base) {
    if (stack_gettop(spine) >= base + 2) {
        p_node res, c;
        stack_peek(spine, 2, res);
        /* ignore arg at index 2 */
        stack_peekarg(spine, 1, c);
        graph_evaluate(c); /* must evaluate to keep things lazy */
        node_copy(res, c);
        stack_drop(spine, 2);
        return res; /* go on reducing from res */
    }
    return NULL;
}

p_node I_reduce(p_node self, size_t base) {
    if (stack_gettop(spine) >= base + 1) {
        p_node res, arg;
        stack_peek(spine, 1, res);
        stack_peekarg(spine, 1, arg);
        graph_evaluate(arg); /* must evaluate to keep things lazy */
        node_copy(res, arg);
        stack_drop(spine, 1);
        return res; /* go on reducing from res */
    }
    return NULL;
}
```

Apêndice B

Código fontes para os testes

B.1 Hanoi (Sloth)

```
hanoi 0 a b c   = []
|     n a b c   = hanoi (n-1) a c b ++ (a, b) : hanoi (n-1) c b a;

concat []     l = l
|      (x:xs) l = x : concat xs l;

main            = hanoi 15 'a' 'b' 'c';
```

B.2 Primes (Sloth)

```
filter _ []   = []
|    f (x:xs) = let xs' = (filter f xs) in if f x then x : xs' else xs';

take 0 _      = []
|    _ []     = []
|    n (x:xs) = x : take (n-1) xs;

sieve (p:xs)  = p : sieve (filter (\x -> (mod x p) != 0) xs);

from n        = n : from (n+1);

primes        = sieve (from 2);

main          = take 600 primes;
```

B.3 Quick Sort (Sloth)

```
qsort []     = []
|     (x:xs) = qsort (filter (gt x) xs) ++ x : qsort (filter (le x) xs);

random n     = map (mod 129970) (take n (from n));

main         = qsort (random 1500);
```

B.4 Merge Sort (Sloth)

```
evens []           = []
|     [x]          = [x]
|     (x: xs)      = x : odds xs;

odds []            = []
|    [x]           = []
|    (x: xs)       = evens xs;

merge []     ys    = ys
|     xs     []    = xs
|     (x:xs) (y:ys) = x : merge xs (y:ys),       x <= y
|     (x:xs) (y:ys) = y : merge (x:xs) ys,       otherwise;

msort []           = []
|     [x]          = [x]
|     l            = merge (msort (evens l)) (msort (odds l));

main               = msort (random 1500);
```

B.5 Insertion Sort (Sloth)

```
foldr f z []      = z
|     f z (x:xs)  = f x (foldr f z xs);

isort             = foldr insert [];

insert x []       = [x]
|      x (y:ys)   = x:y:ys,         x <= y
|      x (y:ys)   = y: insert x ys, otherwise;

main              = isort (random 1000);
```

B.6 Quick Sort (Gofer/Hugs)

```
qsort_          :: [Int] -> [Int]
qsort_ []        = []
qsort_ (x:xs)    = concat_ (qsort_ (filter_ ((>) x) xs))
                           (x: qsort_ (filter_ ((<=) x) xs))

filter_         :: (Int -> Bool) -> [Int] -> [Int]
filter_ _ []     = []
filter_ f (x:xs) = let xs' = (filter_ f xs) in if f x then x : xs' else xs'

take_           :: Int -> [Int] -> [Int]
take_ 0 _        = []
take_ _ []       = []
take_ n (x:xs)   = x : take_ (n-1) xs

from_           :: Int -> [Int]
from_ n          = n : from_ (n+1)
```

```
concat_         :: [a] -> [a] -> [a]
concat_ [] l     = l
concat_ (x:xs) l = x: concat_ xs l

map_            :: (a -> b) -> [a] -> [b]
map_ f []        = []
map_ f (x:xs)    = f x : map_ f xs

rand_           :: Int -> [Int]
rand_ n          = map_ (mod 129970) (take_ n (from_ n))

qsort_ (rand_ 1500)
```

B.7 Quick Sort (Hope)

```
dec filter_            : (num -> truval) -> list(num) -> list(num);
--- filter_ f nil     <= nil;
--- filter_ f (x::xs) <= let xsp == (filter_ f xs) in
                         if f x then (x :: xsp) else xsp;

dec take_              : num -> list(num) -> list(num);
--- take_ 0 _         <= nil;
--- take_ _ nil       <= nil;
--- take_ n (x::xs)   <= x :: take_ (n-1) xs;

dec from_              : num -> list(num);
--- from_ n           <= n :: from_ (n+1);

dec map_               : (alpha -> beta) -> list(alpha) -> list(beta);
--- map_ f nil        <=  nil;
--- map_ f (x::xs)    <= f x :: map_ f xs;

dec rand_              : num -> list(num);
--- rand_  n          <= map_ (lambda (x) => 129970 mod x) (take_ n (from_ n));

dec concat_            : list(alpha) -> list(alpha) -> list(alpha);
--- concat_ [] l      <= l;
--- concat_ (x::xs) l <= x :: concat_ xs l;

dec qsort_             : list(num) -> list(num);
--- qsort_ nil        <= nil;
--- qsort_ (x::xs)    <= concat_ (qsort_ (filter_ (< x) xs))
                              (x :: (qsort_ (filter_ (>= x) xs)));

qsort_ (rand_ 1500);
```

B.8 Primes (Gofer/Hugs)

```
filter_         :: (Int -> Bool) -> [Int] -> [Int]
filter_ _ []     = []
filter_ f (x:xs) = let xs' = (filter_ f xs)
```

```
                  in if f x then x : xs' else xs'

take_           :: Int -> [Int] -> [Int]
take_ 0 _        = []
take_ _ []       = []
take_ n (x:xs)   = x : take_ (n-1) xs

sieve_          :: [Int] -> [Int]
sieve_ (p:xs)    = p : sieve_ (filter_ (\x -> (mod x p) /= 0) xs)

from_           :: Int -> [Int]
from_ n          = n : from_ (n+1)

primes_         :: [Int]
primes_          = sieve_ (from_ 2)

:load primes.gs
take_ 600 primes_
```

B.9 Primes (Hope)

```
dec filter_            : (num -> truval) -> list(num) -> list(num);
--- filter_ f nil     <= nil;
--- filter_ f (x::xs) <= let xsp == (filter_ f xs) in
                         if f x then (x :: xsp) else xsp;

dec take_              : num -> list(num) -> list(num);
--- take_ 0 _         <= nil;
--- take_ _ nil       <= nil;
--- take_ n (x::xs)   <= x :: take_ (n-1) xs;

dec from_              : num -> list(num);
--- from_ n           <= n :: from_ (n+1);

dec sieve_             : list(num) -> list(num);
--- sieve_ (p::xs)    <= p :: sieve_ (filter_ (lambda (x) => (x mod p) /= 0) xs);

dec primes_            : list(num);
--- primes_           <= sieve_ (from_ 2);

take_ 600 primes_;
```

Bibliografia

[1] S. K. Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41(1):222–224, 1976.

[2] Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. McGraw-Hill, Novembro 1985. ISBN 0201100886.

[3] Guy Argo. Improving the Three Instruction Machine. Em *Proceedings of the Functional Programming Languages and Computer Architecture Conference*. ACM Press, Setembro 1989.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. ISBN 0-444-87508-5.

[5] H. P. Barendregt. Introduction to lambda calculus. Em *Proceedings of the 1988 Aspenæs Workshop on the Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[6] H. P. Barendregt. The impact of the lambda calculus on logic and computer science. *Bulletin of Symbolic Logic*, 3(3):181–215, 1997.

[7] R. M. Burstall, D. B MacQueen e D. T. Sanella. Hope: An experimental applicative language. Em *The 1980 LISP Conference*, páginas 136–143, Stanford, Agosto 1980. Also CSR-62-80, Dept of Computer Science, University of Edinburgh.

[8] F. W. Burton. A linear space translation of functional programs to Turner combinators. *Information Processing Letters*, 14(5):201–204, 1982.

[9] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

[10] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, Março 1990. ISBN 0070131430.

[11] G. P. Cousineau, P. L. Curien e M. Mauny. The categorical abstract machine. Em J. P. Jouannaud, editor, *Proceedings of Functional Programming Languages and Computer Architecture, Nancy, France*, páginas 50–64. Springer-Verlag, Setembro 1985. LNCS 201.

[12] P. L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, Maio 1991.

[13] H. B. Curry e R. Feys. *Combinatory Logic*, volume 1 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1958. Segunda edição 1968.

[14] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[15] Jon Fairbairn e Stuart Wray. TIM: A simple, lazy abstract machine to execute supercombinators. Em G. Kahn, editor, *Proceedings of the IFIP Functional Programming Languages and Computer Architecture Conference, Portland, Oregon, USA*, páginas 34–45. Springer-Verlag, Setembro 1987. LNCS 274.

[16] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs e Koen G. Langendoen. *Modern Compiler Design*. Worldwide Series in Computer Science. John Wiley & Sons, Agosto 2000. ISBN 0-471-97697-0.

[17] P. H. Hartel. Benchmarking implementations of functional languages with 'pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(4):587–620, 1996.

[18] P. H. Hartel e K. Langendoen. Benchmarking implementations of lazy functional languages. Em *Functional Programming Languages and Computer Architecture*, páginas 341–349, 1993.

[19] J. Roger Hindley. *An Introduction to Combinators and the Lambda-Calculus*, volume 1 de *Student Texts*. London Mathematical Society, Janeiro 1986. ISBN 0521268966.

[20] John E. Hopcroft e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Abril 1979. ISBN 020102988X.

[21] R. J. M. Hughes. Supercombinators: A new implementation method for applicative languages. Em *Proceedings of the 1982 ACM Conference on LISP and Functional Programming*, páginas 1–10, 1982.

[22] R. Ierusalimschy, L. H. de Figueiredo e W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[23] S. C. Johnson e R. Sethi. *Yacc: A parser generator*, 1990. Unix Research System Programmer's Manual, Tenth Edition, Volume 2.

[24] T. Johnsson. Efficient compilation of lazy evaluation. Em *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, páginas 58–69. ACM Press, 1984.

[25] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. Em *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, Setembro 1985.

[26] Mark P. Jones. *Introduction to Gofer 2.20*, Outubro 1991.

[27] Mark P. Jones. The implementation of the Gofer functional programming system. Research report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, Maio 1994.

[28] Mark P. Jones e A. Reid. *The Hugs 98 User Manual*, Dezembro 2001.

[29] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, Maio 1986. ISBN 0-13-453333-X.

[30] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.

[31] Simon L. Peyton Jones e David R. Lestar. *Implementing Functional Languages: A Tutorial*. Prentice-Hall, 1991.

[32] Simon L. Peyton Jones e Jon Salkild. The spineless tagless G-machine. Em *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, U. K.*, páginas 184–201. ACM Press, 1989. ISBN 0-89791-328-0.

[33] M. S. Joy, V. J. Rayward-Smith e F. W. Burton. Efficient combinator code. *Computer Languages*, 10(3/4):211–224, 1985.

[34] J. R. Kennaway. The complexity of a translation of lambda-calculus to combinators. Internal report CS/82/023/E, University of East Anglia, Norwich, U. K., 1982.

[35] J. R. Kennaway e M. R. Sleep. Variable abstraction in $O(n \log n)$ space. *Information Processing Letters*, 24(5):343–349, 1987.

[36] J. R. Kennaway e M. R. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.

[37] B. W Kernighan e D. M Ritchie. *C Programming Language (ANSI C)*. Prentice-Hall, Junho 1988. ISBN 0131103628.

[38] P. Koopman e P. Lee. A fresh look at combinator graph reduction. Em *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, páginas 21–23. ACM Press, 1989.

[39] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[40] Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, Janeiro 1990. ISBN 0201137445.

[41] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, (Part I). *Communications of the ACM*, 3(4):184–195, 1960.

[42] Diego F. Nehab. Sloth: Uma generalização dos combinadores SK. Trabalho final de curso, PUC-Rio, Departamento de Informática, Novembro 1999.

[43] Diego F. Nehab. A linguagem de programação Sloth. Em *Anais do VI Simpósio Brasileiro de Linguagens de Programação, Rio de Janeiro*, páginas 270–282. Sociedade Brasileira de Computação, Junho 2002.

[44] Kohei Noshita. Translation of Turner combinators in $O(n \log n)$ space. *Information Processing Letters*, 20(2):71–74, 1985.

[45] Mark Scheevel. NORMA: A graph reduction processor. Em *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, páginas 212–219, 1986.

[46] M. Schönfinkel. Uber die Bausteine der mathematischen Logik. *Mathematischen Annalen*, 92:305–316, 1924. Tradução para o inglês em 'From Frege to Gödel, a Source Book in Mathematical Logic'.

[47] A. M. Turing. Computability and λ-definability. *Journal of Symbolid Logic*, 2:153–163, 1936.

[48] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Em *Proceedings of the London Mathematical Society*, volume 42 de *2*, páginas 230–265, 1936/7.

[49] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2): 267–270, 1979.

[50] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[51] D. A. Turner. *Miranda system manual*. Research Software Ltd., 23 St Augustines Road, Canterbury, Kent CT1XP, England, Abril 1990.

[52] Paul R. Wilson. Uniprocessor garbage collection techniques. Em *Proceedings of the International Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag. LNCS 637.