A linguagem de programação funcional Sloth

Diego Fernandes Nehab **Roberto Ierusalimschy**

Departamento de Informática, PUC-Rio
R. Marquês de São Vicente 225, Gávea,
Rio de Janeiro, 22453-900
{diego,roberto}@inf.puc-rio.br

Abstract. *Sloth is a simple, easy to learn, non-strict, purely functional programming language, designed to be a test bed for research on the compilation of functional programming languages. Its hybrid architecture innovates on the different programming languages chosen for its construction and resulted in the creation of a concise, self contained and portable implementation. This article introduces Sloth as a programming language, describes part of the research work currently taking advantage of the system and details its implementation, about to be completed.*

Resumo. *Sloth é uma linguagem de programação simples, de fácil aprendizagem, funcional pura e não estrita, projetada para servir como ambiente de testes para pesquisas sobre a compilacão de linguagens de programação funcionais. Sua arquitetura híbrida inova com relação às diferentes linguagens de programação escolhidas para sua construção e resultou na criação de uma implementação concisa, autocontida e portátil. Este artigo apresenta a linguagem de programação Sloth, descreve o trabalho de pesquisa que atualmente se beneficia do sistema e detalha sua implementação, prestes a ser concluída.*

1 Introdução

Sloth é uma linguagem de programação concisa e de fácil aprendizagem que surgiu da necessidade de uma análise concreta da eficiência de novas técnicas a serem usadas na compilação de programas funcionais. A decisão pela criação de uma nova linguagem, em oposição a adaptação de uma linguagem existente, teve como fatores determinantes a grande complexidade das implementações disponíveis, bem como a maior liberdade obtida com o projeto de todos os aspectos da linguagem. Por isso, mesmo oferecendo muitos dos recursos básicos existentes em linguagens de programação semelhantes, Sloth, uma linguagem funcional pura, não estrita e com tipagem parametricamente polimórfica, destaca-se por sua simplicidade.

As técnicas de compilação investigadas baseiam-se no uso de árvores combinatórias como código-objeto para programas funcionais [Turner, 1979b]. Os Combinadores Microprogramados, desenvolvidos pela pesquisa em andamento, capturam e estendem a essência dos combinadores de longo alcance descritos por [Turner, 1979a]. Sloth foi criada para auxiliar na análise da eficiência da codificação de programas funcionais com o uso dos novos combinadores e para simplificar o estudo de seu efeito sobre o tamanho das árvores combinatórias resultantes.

A experiência adquirida com a implementação de uma primeira versão de Sloth e o estudo de implementações de outras linguagens orientou a adoção de uma arquitetura híbrida pouco convencional para a construção da nova versão do sistema. Para a implementação de todas as fases intermediárias, nas quais a expressividade da linguagem de programação usada tem grande impacto, foi escolhida uma linguagem interpretada, com tipagem fraca e gerência automática de memória (Lua). Já para a construção das etapas executadas durante a avaliação do programa, nas quais se faz necessária uma maior preocupação com eficiência, foi escolhida uma linguagem de programação tradicional, compilada (ANSI C). A associação entre as vantagens das duas linguagens vem apresentando bons resultados à medida que a construção do sistema se aproxima de sua conclusão.

Dentre as preocupações que mais influenciaram os rumos do projeto de Sloth, destacam-se a simplicidade, eficiência, a portabilidade e a disponibilidade. A simplicidade permeia tanto sua especificação quanto sua implementação. Sloth não visa o estabelecimento de um novo padrão para programação funcional e tampouco ruma à criação de uma nova linguagem de uso genérico. A preocupação com a eficiência vem do desejo de que Sloth seja útil na prática, principalmente em atividades didáticas que tirem partido da simplicidade de sua especificação. A portabilidade permite que Sloth opere em uma grande variedade de sistemas operacionais e processadores. Finalmente, a licença de Sloth não restringe seu uso gratuito em aplicações acadêmicas e comerciais, e as ferramentas usadas em seu desenvolvimento são amplamente disponíveis.

Ocasionalmente, alguns desses quesitos entram em conflito. Nesses casos, o compromisso com a simplicidade acaba determinando a direção seguida.

1.1 Panorama do artigo

Este artigo prossegue com a apresentação de Sloth como uma linguagem de programação, na Seção 2. A Seção 3 descreve brevemente as novas técnicas de compilação usadas em Sloth. Em seguida, a Seção 4 detalha a arquitetura escolhida para a implementação do sistema. Por fim, a Seção 5 discute os rumos futuros da pesquisa e conclui o artigo.

2 A linguagem de programação

A sintaxe de Sloth é inspirada na sintaxe de Haskell, na qual alguns recursos foram omitidos de modo a simplificar sua especificação. A lista abaixo resume as principais características da linguagem de programação Sloth, buscando situá-la frente às demais linguagens de programação funcionais.

- Sloth é uma linguagem aplicativa, funcional pura;
- Sloth é não estrita e preguiçosa;
- Sloth suporta tipos de dados estruturados;
- A tipagem de Sloth é dinâmica e parametricamente polimórfica;
- Funções de qualquer ordem são valores de primeira classe;
- Funções podem ter definições múltiplas baseadas em casamento de padrões;
- O sistema de escopo de Sloth é estático.

Lexicamente, Sloth é uma linguagem sensível a caixas alta e baixa, de com definições convencionais para identificadores, constantes numéricas, caracteres e seqüências

de caracteres. Comentários são iniciados pela seqüência '`--`' e continuam até o fim da linha. Uma das principais diferenças entre a sintaxe de Sloth e a de Haskell reside no fato de Sloth ser uma linguagem de formato livre. Assim sendo, a indentação do código fica a critério do programador, e não tem influência no resultado final. A seguir, uma breve exposição da sintaxe da linguagem será apresentada com o auxílio de exemplos.

Os únicos tipos de dados primitivos em Sloth são os tipos `Number` e `Char`. Os tipos `Bool` e `List` são predefinidos. Os demais tipos de dados são declarados a partir dos tipos primitivos, seguindo-se a sintaxe usual para declarações na forma de soma de produtos.

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

Sloth suporta funções com definições múltiplas baseadas no casamento de padrões sobre seus argumentos. O casamento de padrões se dá pela análise dos construtores ou valores dos argumentos e se estende a tipos estruturados definidos pelo usuário.

```
fat 0 = 1
|   x = x * fat (x-1);

sign x = -1,     x < 0
|    x = 0,      x == 0
|    x = 1,      otherwise;

cond true  e2 e3 = e2
|    false e2 e3 = e3;

reflect (leaf x)     = leaf x
|       (branch a b) = branch (reflect b) (reflect a);
```

A linguagem suporta ainda operadores infixados como uma alternativa mais confortável para a construção de expressões complexas, seguindo as precedências usuais. Também são aceitas expressões na forma prefixada, usada internamente pela linguagem, com literais representando cada operação.

Listas podem ser definidas das formas tradicionalmente encontradas em outras linguagens funcionais, sendo que as listas de caracteres, por serem tão comuns, podem ser definidas entre aspas duplas.

Sloth também suporta funções anônimas, definições locais recursivas ou não e expressões *case*, conforme ilustram as declarações abaixo:

```
length = \list -> (case list of
    nil -> 0 |
    cons x xs -> 1 + length xs);

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

Por fim, apresentamos um exemplo completo com o intuito de demonstrar que a simplicidade da especificação da linguagem não prejudica sua expressividade na realização de tarefas relevantes.

Números primos

Define a lista de todos os números primos pelo Crivo de Eratóstenes.

```
sequence n = n : sequence (n + 1);

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

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

primes = sieve (sequence 2);
```

3 Combinadores Microprogramados

Apesar de ser usada por teóricos desde a década de 1920, a Lógica Combinatória [Curry et al., 1972] ganhou maior divulgação a partir do final da década de 1970, quando passou a ser usada na prática para a implementação de linguagens funcionais [Turner, 1979b]. Além de ser de fácil implementação, a técnica permite, com grande naturalidade, funções não estritas e realiza avaliação preguiçosa. Na época, implementações com essas propriedades eram muito mais complexas e muito menos eficientes.

Infelizmente, em sua formulação original, o processo de abstração de variáveis sucessivas pela introdução de combinadores pode causar uma explosão no tamanho da expressão resultante. Por conseqüência, o processo de redução torna-se muito custoso, tanto em tempo quanto em espaço. Para tornar a técnica viável, Turner adicionou os combinadores de longo alcance `S′`, `B′` e `C′` [Turner, 1979a] aos tradicionais `S`, `K`, `I`, `B` e `C` (veja a Tabela 1). Ainda assim, o tamanho de uma expressão, medido pelo número de constantes e variáveis, pode crescer de forma $\Theta(n^2)$ [Kennaway, 1982]. Além disso, a ação de cada combinador é muito restrita e o processo de redução tende a ser composto por uma grande quantidade de pequenos passos, o que pode prejudicar o uso do *cache* e sobrecarregar o coletor de lixo.

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

Tabela 1: Combinadores de Turner.

Ao longo dos anos seguintes, tentativas de solucionar esses problemas deram origem a varias alternativas. O balanceamento das expressões antes que sejam submetidas à abstração de variáveis consegue garantir tamanhos $O(n \log n)$ [Burton, 1982]. O agrupamento de combinadores referentes a abstrações de variáveis sucessivas em seqüências diretoras, quando representadas de forma compactada (*counting director strings*) [Kennaway and Sleep, 1987], alcança resultado semelhante. Como, no pior caso, o tamanho de uma expressão compilada sob um conjunto finito de combinadores está limitado por $\Omega(n \log n)$ [Joy et al., 1985], não há muito espaço para melhorias nessa linha.

Tirando partido de um conjunto infinito de combinadores, os supercombinadores [Johnsson, 1985, Hughes, 1982], é possível obter resultados que variam, em média, linearmente com o tamanho da expressão. A representação de supercombinadores por um conjunto linear de instruções (*G-Machine*) [Johnsson, 1984] aumenta a granularidade dos passos e diminui o tamanho final da representação, levando a uma das mais eficientes técnicas para a implementação de linguagens funcionais [Jones, 1992]. O sucesso desta técnica foi provavelmente um dos responsáveis pela recente perda de interesse nos combinadores de Turner, que passaram a ser mais respeitados por sua simplicidade que por sua eficiência.

Outros métodos, mais fieis às idéias de Turner, são também baseados em conjuntos infinitos de combinadores, porém mais restritivos. As restrições muitas vezes facilitam a representação dos combinadores possíveis. Representações parametrizadas por números podem ser encontradas em [Noshita, 1985] e [Abdali, 1976].

Os Combinadores Microprogramados, propostos neste artigo, adotam uma estratégia semelhante, buscando uma solução mais eficiente que os combinadores de Turner porém sem a complexidade dos supercombinadores. A parametrização, entretanto, é feita por seqüências de operações primitivas, que resultam de uma decomposição das alterações realizadas pela redução dos combinadores de Turner em operações mais básicas.

A nova codificação, além de representar uniformemente todos os combinadores da Tabela 1, dá origem a uma família infinita de combinadores com alcance ilimitado. Dado que os combinadores de Turner podem ser vistos como instruções a serem executadas por uma máquina redutora de grafos, a nova codificação pode ser associada a um nível de abstração mais baixo, correspondente ao do microcódigo. Daí o nome Combinadores Microprogramados.

3.1 A família de combinadores $\mathcal{L}_\alpha$

A avaliação de qualquer um dos combinadores da Tabela 1 afeta apenas uma pequena vizinhança da espinha da expressão na qual se encontram. Em cada nó da espinha, as alterações realizadas podem ser descritas por apenas três operações primitivas. A família de combinadores $\mathcal{L}_\alpha$ representa os combinadores que podem ser parametrizados por seqüências α dessas operações.

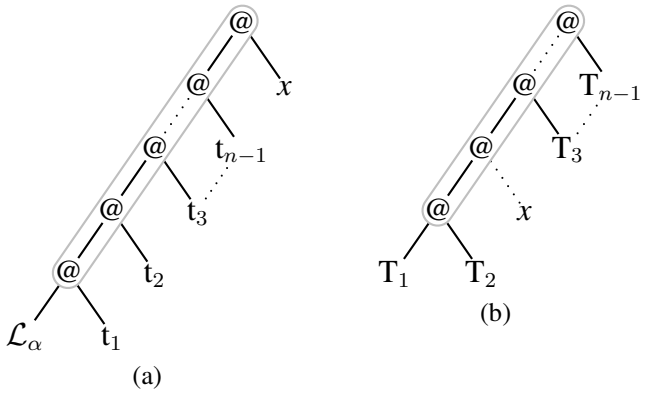

Figura 1: $\mathcal{L}_\alpha$. (b) mostra o resultado da avaliação de $\mathcal{L}_\alpha$ em (a).

Na Figura 1, o parâmetro α de $\mathcal{L}_\alpha$ controla a forma de cada T_i, determinando se assumirá o valor t_i ou $(\mathrm{t}_i\ x)$. $\mathcal{L}_\alpha$ é capaz ainda de introduzir o parâmetro x nas posições determinadas por α, entre os T_i. Essas três operações primitivas são representadas pelos seguintes símbolos:

$$\mathsf{p} \rightarrow \mathrm{T}_i = (\mathrm{t}_i\ x) \qquad \mathsf{d} \rightarrow \mathrm{T}_i = \mathrm{t}_i \qquad \mathsf{i} \rightarrow \text{inserção de } x$$

A distinção entre o tratamento dado à primeira subárvore, que é posicionada à esquerda de um nó de aplicação, e às demais, que são posicionadas à direita, é feita pelo uso de símbolos maiúsculos e minúsculos, respectivamente. A Figura 2 mostra o resultado da avaliação de um Combinador Microprogramado, sem correspondente no conjunto usado por Turner, que aparece frequentemente no código objeto gerado por Sloth.

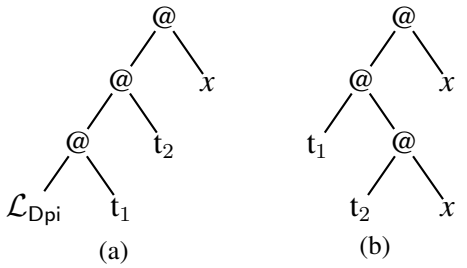

Figura 2: $\mathcal{L}_{\mathsf{Dpi}}$. (b) mostra o resultado da avaliação de $\mathcal{L}_{\mathsf{Dpi}}$ em (a).

A avaliação do Combinador Microprogramado é feita por uma máquina muito simples que, após certificar-se que o número necessário de argumentos está presente (3 no caso da Figura 2), executa as microinstruções presentes no parâmetro α do combinador e substitui a expressão original pela a expressão avaliada.

A definição recursiva para o processo de abstração de variáveis com o uso de Combinadores Microprogramados é dada abaixo sem grandes preocupações com formalismos[1] Apesar do compilador de Sloth utilizar um método iterativo mais direto para a abstração de variáveis, a representação recursiva costuma ser útil no estabelecimento de resultados teóricos e é de definição mais simples.

$$\begin{array}{ll}
\mathrm{Abst}(x, x) = \mathcal{L}_{\mathsf{I}} & \alpha(x, x) = \mathsf{I} \\
\mathrm{Abst}(x, E) = \mathcal{L}_{\mathsf{D}}\ E,\ x\ \mathrm{o}\not\in E & \alpha(x, E) = \mathsf{D},\ x\ \mathrm{o}\not\in E \\
\mathrm{Abst}(x, E\ x) = E,\ x\ \mathrm{o}\not\in E & \alpha(x, E\ x) = \mathsf{P},\ x\ \mathrm{o}\not\in E \\
\mathrm{Abst}(x, E) = \mathrm{Trim}(\mathrm{Left}(x, E, E)) & \alpha(x, F\ x) = \alpha(x, F) \cdot \mathsf{i},\ x\ \mathrm{oc}\ F \\
\mathrm{Left}(x, x, G) = \mathcal{L}_{\alpha(x,\ G)} & \alpha(x, F\ E) = \alpha(x, F) \cdot \mathsf{d},\ x\ \mathrm{o}\not\in E \\
\mathrm{Left}(x, E, G) = \mathcal{L}_{\alpha(x,\ G)}\ E,\ x\ \mathrm{o}\not\in E & \alpha(x, F\ E) = \alpha(x, F) \cdot \mathsf{p},\ x\ \mathrm{oc}\ E \\
\mathrm{Left}(x, E\ x, G) = \mathcal{L}_{\alpha(x,\ G)}\ E,\ x\ \mathrm{o}\not\in E & \mathrm{Trim}(E\ \mathcal{L}_{\mathsf{I}}) = \mathrm{Trim}(E) \\
\mathrm{Left}(x, E\ F, G) = \mathrm{Left}(x, E, G)\ \mathrm{Abst}(x, F),\ x\ \mathrm{oc}\ F & \mathrm{Trim}(E\ F) = \mathrm{Trim}(E)\ F \\
\mathrm{Left}(x, E\ F, G) = \mathrm{Left}(x, E, G)\ F,\ x\ \mathrm{o}\not\in F & \mathrm{Trim}(E) = E
\end{array}$$

A transformação $\mathrm{Abst}(x, E)$ abstrai a variável x da expressão E, fazendo uso das auxiliares Left e Trim. A transformação $\mathrm{Left}(x, E, E)$ cria um combinador apropriado para a espinha da expressão E, empregando a auxiliar α, e abstrai a variável x ao longo da espinha de argumentos de E, usando Abst. A função $\alpha(x, E)$ determina o microcódigo

[1]Nas equações, 'x oc E' deve ser lido como 'x ocorre livre em E' sendo que 'x o$\not\in E$' é a negação deste predicado.

para o combinador a ser usado na espinha de E (o operador '$\cdot$' representa a concatenação de seqüências de microcódigo). Por fim, a transformação $\mathrm{Trim}(E)$ elimina os combinadores $\mathcal{L}_{\mathsf{I}}$ (identidade) ao longo da espinha de argumentos de E, já que os mesmos estão codificados de forma mais eficiente dentro do próprio microcódigo.

3.2 Exemplos

Combinadores Microprogramados com alcance maior que o dos combinadores de Turner surgem naturalmente na abstração de variáveis em programas usuais. A tabela abaixo mostra os combinadores que surgem durante a compilação das funções da biblioteca padrão de Sloth, seguidos pela freqüência de suas ocorrências. Note que os Combinadores Microprogramados $\mathcal{L}_{\mathsf{Pp}}$, $\mathcal{L}_{\mathsf{Pd}}$, $\mathcal{L}_{\mathsf{Dp}}$, $\mathcal{L}_{\mathsf{Dpp}}$ e $\mathcal{L}_{\mathsf{Dpd}}$ são respectivamente equivalentes aos combinadores de Turner `S`, `C`, `B`, `S'` e `C'`.

$\mathcal{L}_{\mathsf{Dp}}$	222	$\mathcal{L}_{\mathsf{Dpp}}$	95	$\mathcal{L}_{\mathsf{Pd}}$	75	$\mathcal{L}_{\mathsf{Dpd}}$	75	$\mathcal{L}_{\mathsf{Dpdd}}$	48
$\mathcal{L}_{\mathsf{Pp}}$	37	$\mathcal{L}_{\mathsf{Pdp}}$	31	$\mathcal{L}_{\mathsf{Pdd}}$	14	$\mathcal{L}_{\mathsf{Dppd}}$	11	$\mathcal{L}_{\mathsf{Dpi}}$	11
$\mathcal{L}_{\mathsf{Dppdd}}$	7	$\mathcal{L}_{\mathsf{Dpdp}}$	7	$\mathcal{L}_{\mathsf{Dppp}}$	5	$\mathcal{L}_{\mathsf{Dpddd}}$	4	$\mathcal{L}_{\mathsf{Pi}}$	3
$\mathcal{L}_{\mathsf{Dppi}}$	3	$\mathcal{L}_{\mathsf{Dpdi}}$	3	$\mathcal{L}_{\mathsf{Ppp}}$	2	$\mathcal{L}_{\mathsf{Dpip}}$	2	$\mathcal{L}_{\mathsf{Dpppdd}}$	1
$\mathcal{L}_{\mathsf{Dpppd}}$	1	$\mathcal{L}_{\mathsf{Dpdpd}}$	1	$\mathcal{L}_{\mathsf{Dpddp}}$	1				

As vantagens da representação por Combinadores Microprogramados sobre a representação por combinadores de Turner ficam especialmente evidentes quando a expressão sendo transformada possui uma longa espinha, e podem ser ainda maiores na abstração de variáveis sucessivas. Os exemplos abaixo, nos quais são exibidos os resultados da abstração de variáveis por ambos os métodos, demonstram o maior poder de síntese dos Combinadores Microprogramados:

```
f x = a x (b x) c x   | g a b c d e = e d c b a
f = S (C (S a b) c) I | g = C' (C' (C' C)) (C' (C' C) (C' C (C I)))
f = Lpdi a b c        | g = Lpd (Lpdd (Lpddd Ldddd))
```

A implementação do cálculo da raiz quadrada de um número pelo método de Newton, apresentada a seguir, serve como um exemplo real. Apesar da transformação de `improve` por Combinadores Microprogramados e por combinadores de Turner resultar em estruturas idênticas, as transformações de `satis`, `until` e `sqrt` dão origem aos combinadores $\mathcal{L}_{\mathsf{Pi}}$, $\mathcal{L}_{\mathsf{Dpip}}$, $\mathcal{L}_{\mathsf{Dpi}}$ e $\mathcal{L}_{\mathsf{Dppi}}$, que tornam a representação por Combinadores Microprogramados mais concisa.

```
satis x y = y*y == x;
satis = C' eq (S mul I);
satis = Ldpd eq (Lpi mul);

improve x y = (y + x/y)/2;
improve = C' (C' div) (B (S add) div) 2;
improve = Ldpd (Ldpd div) (Ldp (Lpp add) div) 2;

until p f x = if p x then x else until p f (f x);
until = S' B (B S (C (S' cond) I)) (C' (S' B) until I);
until = Ldpp Ldp (Ldpip cond) (Ldp (Ldpi Ldp) until);

sqrt x = until (satis x) (improve x) x;
sqrt = S (S' until satis improve) I;
sqrt = Ldppi until satis improve;
```

Por usar menos combinadores, a avaliação de funções pelo método dos Combinadores Microprogramados alcança o resultado em um número menor de passos. Por

exemplo, o cálculo da expressão '`sqrt 2`' pelo método de Turner realiza 146 reduções, das quais 79 são combinadores. O cálculo da mesma expressão por Combinadores Microprogramados realiza 108 reduções, das quais apenas 48 são combinadores. Como conseqüência, no caso específico da expressão '`sqrt 2`', o tempo de avaliação é reduzido em 17%.

Além de realizar a instanciação preguiçosa de grafos, os Combinadores Microprogramados mantêm a avaliação totalmente preguiçosa de expressões, assim como fazem os combinadores de Turner. A implementação da abstração de variáveis e do interpretador para as seqüências de microinstruções pode ser feita em poucas linhas de código. Além disso, os combinadores de maior alcance representam um aumento na granularidade das etapas realizadas pelo redutor de grafos. Uma análise detalhada sobre os efeitos dessas vantagens na eficiência da avaliação de programas funcionais se encontra em andamento.

4 A arquitetura

A Figura 3 mostra as etapas pelas quais passa um programa Sloth desde sua leitura até sua avaliação. Estas etapas podem ser encontradas, com pequenas variações, na maioria das implementações de linguagens funcionais.

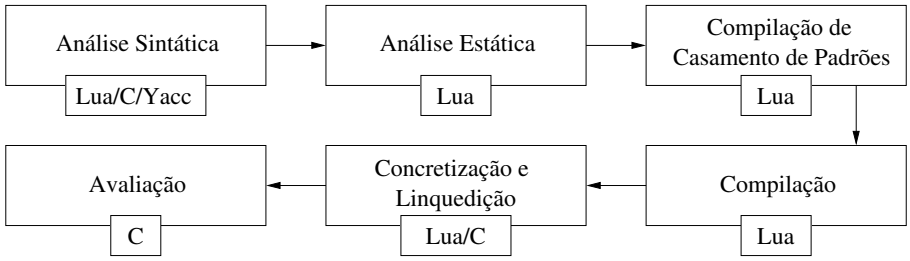

Figura 3: As principais etapas pelas quais passa um programa Sloth e as ferramentas usadas na construção de cada uma delas.

No diagrama acima, a análise sintática engloba a leitura do programa e a construção da árvore sintática correspondente. A análise estática submete a árvore sintática do programa a verificações adicionais que escapam à análise sintática. A compilação de casamento de padrões substitui declarações múltiplas de funções por declarações simples equivalentes.

A compilação transforma funções em uma representação que torna mais eficiente sua avaliação com diferentes argumentos. A fase de concretização converte as expressões compiladas do formato interno usado pelo compilador para o formato final utilizado durante sua avaliação. A linquedição substitui ocorrências de variáveis livres pelas expressões compiladas correspondentes, segundo as regras de escopo vigentes. Por fim, a etapa de avaliação realiza as computações necessárias para a exibição dos resultados.

No caso específico de Sloth, a pesquisa que está sendo realizada exige que as expressões sejam compiladas para árvores combinatórias, linqueditadas para a forma de grafos e avaliadas por um redutor de grafos. Na tentativa de conciliar portabilidade, disponibilidade e eficiência, foi escolhida a linguagem ANSI C para a implementação do avaliador. Essa decisão serviu como ponto de partida para a definição da arquitetura do restante do sistema.

Em geral, durante as fases intermediárias, a natureza e quantidade das transformações pelas quais a estrutura do programa é submetida acabam por exigir do programador tamanha disciplina com a gerência das diversas estruturas dinâmicas alocadas que o uso da linguagem C torna-se inconveniente. Evidências desse fato podem ser encontradas em outras implementações, notavelmente Hugs e Gofer [Jones, 1994], que decidiram pela criação de um sistema de coleta automática de lixo atuando dentro da própria linguagem C. 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.

Foi considerada também a alternativa adotada pela principal referência na implementação de linguagens funcionais [Jones, 1987] e por uma implementação consagrada da linguagem Haskell [Jones, 1992] que consiste na escolha de uma linguagem de programação funcional para sua construção. Essa solução se encaixa perfeitamente em sistemas que dispensam avaliadores, por traduzirem programas funcionais diretamente para linguagem de máquina ou outra linguagem de programação, como é o caso de alguns compiladores de Haskell. No caso de Sloth, entretanto, programas são avaliados por um interpretador. Para permitir que, do ponto de vista do usuário, o processo seja executado como uma única operação, é necessária uma grande integração entre o compilador e o avaliador. Infelizmente, não foi possível encontrar-se uma linguagem funcional que se integrasse bem à linguagem C, na qual é escrito o avaliador, sem que restringisse a portabilidade e a disponibilidade de Sloth.

A escolha da linguagem de programação Lua [Ierusalimschy et al., 1996] para a construção de Sloth representa uma solução intermediária entre as duas alternativas analisadas. Lua é uma linguagem procedural que, além de oferecer os recursos desejados (como a coleta automática de lixo), integra-se transparentemente à linguagem C na forma de uma biblioteca. Lua é uma linguagem leve, implementada em menos de dez mil linhas de código, extremamente portátil por ser totalmente escrita em ANSI C e gratuitamente disponível em código fonte.

4.1 Entrando em detalhes

Conforme descrito na Figura 3, o analisador sintático de Sloth é escrito em C, com o auxílio da ferramenta *Yacc*. As ações semânticas associadas a cada regra, entretanto, limitam-se à criação da árvore sintática do programa sendo analisado. A árvore sintática é diretamente criada como uma estrutura Lua, com o uso da API C oferecida pela linguagem. Desta forma, após a análise sintática, o programa '`succ x = x + 1`', por exemplo, é representado em Lua pela hierarquia de vetores associativos abaixo:

```
{tag = 'varid', value = 'succ'} = {
    tag = 'lambda',
    var = {tag = 'varid', value = 'x'},
    body = {
        tag = 'app',
        func = {
            tag = 'app',
            arg = {tag = 'varid', value = 'x'},
            func = {tag = 'varid', value = 'add'},
        },
        arg = {tag = 'number', value = 1},
    },
}
```

As fases seguintes, até a compilação, são totalmente escritas em Lua, de modo que podem tirar partido da maior expressividade e da gerência automática de memória presentes na linguagem. Em várias ocasiões, essas vantagens permitem soluções que seriam, talvez, ousadas demais caso fossem implementadas na linguagem C. Por exemplo, o processo de abstração de variáveis com o uso dos combinadores de Turner é muitas vezes descrito por um conjunto de transformações, como as abaixo:

$$\begin{aligned}
&\mathrm{Abst}(x, f_1\ f_2) = \mathrm{Opt}(\mathtt{S}\ \mathrm{Abst}(x, f_1)\ \mathrm{Abst}(x, f_2)) \\
&\mathrm{Abst}(x, x) = \mathtt{I} \\
&\mathrm{Abst}(x, c) = \mathtt{K}\ c
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Opt}(\mathtt{S}\ (\mathtt{K}\ p)\ (\mathtt{K}\ q)) = \mathtt{K}\ p\ q \\
&\mathrm{Opt}(\mathtt{S}\ (\mathtt{K}\ p)\ \mathtt{I})) = p \\
&\mathrm{Opt}(\mathtt{S}\ (\mathtt{K}\ p)\ q) = \mathtt{B}\ p\ q \\
&\mathrm{Opt}(\mathtt{S}\ (\mathtt{B}\ p\ q)\ (\mathtt{K}\ r)) = \mathtt{C}'\ p\ q\ r \\
&\mathrm{Opt}(\mathtt{S}\ p\ (\mathtt{K}\ q)) = \mathtt{C}\ p\ q \\
&\mathrm{Opt}(\mathtt{S}\ (\mathtt{B}\ p\ q)\ r) = \mathtt{S}'\ p\ q\ r
\end{aligned}$$

Por motivos de eficiência, mas principalmente por dificuldades relacionadas com a alocação dinâmica das representações temporárias, uma implementação em C evitaria uma realização explícita das transformações descritas pela função Opt acima. O mesmo não ocorre em uma linguagem com coleta automática de lixo, como Lua. O código abaixo, que pode ser usado para transformações mais genéricas em árvores sintáticas, traduz essas transformações diretamente:

```
function abstract(var, expr)
    if isExprApp(expr) then
        return optimize(app(
            app(S, abstract(var, expr.func)),
            abstract(var, expr.arg)))
    elseif expr == var then return I
    else return app(K, expr) end
end

function optimize(expr)
    expr = transform(expr,
        {{S, {K, "*"}}, {K, "*"}},
        {K, {1, 2}})
    expr = transform(expr, {{S, {K, "*"}}, I}, 1)
    expr = transform(expr, {{S, {K, "*"}}, "*"}, {{B, 1}, 2})
    expr = transform(expr,
        {{S, {{B, "*"}, "*"}}, {K, "*"}},
        {{{Cprime, 1}, 2}, 3})
    expr = transform(expr, {{S, "*"}, {K, "*"}}, {{C, 1}, 2})
    return transform(expr,
        {{S, {{B, "*"}, "*"}}, "*"},
        {{{Sprime, 1}, 2}, 3})
end

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

```
function match(expr, pat, cap)
    if pat == "*" then
        append(cap, expr)
        return cap
    elseif expr == pat then return cap
    elseif isPatApp(pat) and isExprApp(expr) then
        return match(expr.func, pat[1], cap) and
            match(expr.arg, pat[2], cap)
    else return nil end
end

function replace(cap, repl)
    if isPatApp(repl) then
        return app(replace(cap, repl[1]), replace(cap, repl[2]))
    elseif isPatNumber(repl) then
        return cap[repl]
    else return repl end
end
```

A funções `abstract` e `optimize` realizam literalmente as funções Abst e Opt, respectivamente. A função `transform` tenta casar uma expressão com o padrão definido pelo argumento `pat`, com o uso da auxiliar `match`. Durante o processo de casamento, a localização de um nó `"*"` no padrão resulta na captura da expressão correspondente. Após o casamento, a função `replace` constrói uma nova expressão, tendo como base o esqueleto definido por `repl`, na qual os nós numéricos são substituídos pelas expressões capturadas correspondentes.

Após a compilação, durante a fase de concretização, o programa é transformado de sua representação em Lua para uma série de árvores equivalentes alocadas dinamicamente em C. A fase de linquedição percorre as árvores resultantes para eliminar as variáveis livres restantes, substituindo-as por referências às expressões correspondentes, também concretizadas. Já em sua forma final, representado por um grafo, o programa é então interpretado pelo avaliador, totalmente escrito em C.

Tendo conseguido associar as vantagens das duas linguagens de programação usadas em sua implementação, Sloth está prestes a atingir todos os objetivos apresentados na Seção 1.

4.2 Alcançando os objetivos

Nossa primeira implementação de Sloth [Nehab, 1999] suportava uma versão simplificada da linguagem. Mais especificamente, não havia suporte a tipos de dados estruturados e a casamento de padrões. Implementada em 5000 linhas de código C, foi criada com o intuito de validar as técnicas de compilação descritas na Seção 3 e acabou motivando uma nova implementação, mais completa. A nova implementação, com todos os recursos descritos na Seção 2 e baseada na arquitetura descrita acima, está prestes a ser concluída, em menos de 6000 linhas de código. Estão incompletas as fases de Concretização e Linquedição.

Considerando-se o modesto tamanho da implementação e a pequena especificação da linguagem, pode-se afirmar que Sloth manteve-se fiel ao seu compromisso com a simplicidade. Sua portabilidade está assegurada pelo uso exclusivo das linguagem de

programação ANSI C e Lua (que é por sua vez implementada em ANSI C). Sua disponibilidade está garantida, já que Lua é distribuída livremente e nenhuma outra ferramenta se faz necessária (além de um compilador de C, naturalmente). Por fim, a experiência adquirida com a primeira versão da linguagem descarta a possibilidade de fracasso no que diz respeito à eficiência. Como o código gerado pelo novo compilador e as técnicas usadas pelo novo avaliador (baseado no trabalho [Koopman and Lee, 1989]) são ainda mais eficientes, os resultados só podem melhorar.

5 Conclusões

Este artigo apresentou a linguagem de programação Sloth. Sua especificação modesta facilita sua aprendizagem sem privá-la dos recursos mais comuns às linguagens de programação funcionais. A arquitetura escolhida para sua construção dá origem a uma implementação concisa, que distribui entre linguagens de programação distintas as etapas às quais melhor se adaptam. Finalmente, os Combinadores Microprogramados, nos quais se baseiam seu compilador e seu avaliador, podem trazer um aumento de eficiência em relação aos combinadores de Turner sem comprometer suas vantagens e sua simplicidade.

Dando prosseguimento ao trabalho já realizado, serão implementadas as etapas que restam para a conclusão de Sloth. Com o sistema completo, a eficiência dos Combinadores Microprogramados será comparada, na prática, com os combinadores de Turner. Logo em seguida, a eficiência de Sloth será comparada com a de outras linguagens, como Gofer e Hugs. Por fim, uma análise teórica tentará obter estimativas mais precisas quanto ao tamanho das árvores combinatórias geradas pela abstração de variáveis com o uso de Combinadores Microprogramados, já que os estudos feitos com base nos combinadores de Turner não se aplicam diretamente aos novos combinadores.

Referências

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

Burton, F. W. (1982). A linear space translation of functional programs to Turner combinators. *Information processing letters*, 14(5):201–204.

Curry, H. B.; Hindley, J. R.; Seldin, J. P. (1972). *Combinatory Logic, Volume I*. North-Holland, Amsterdam.

Hughes, R. J. M. (1982). Supercombinators: A new implementation method for applicative languages. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pp. 1–10.

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

Johnsson, T. (1984). Efficient compilation of lazy evaluation. In *Proc. of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pp. 58–69.

Johnsson, T. (1985). Lambda lifting: transforming programs to recursive equations. In *Proc. of the Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France.

Jones, M. P. (1994). The implementation of the Gofer functional programming system. Technical report, Yale University, Department of Computer Science.

Jones, S. L. P. (1987). *The implementation of functional programming languages*. Prentice Hall International series in computer science.

Jones, S. L. P. (1992). Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming 2(2)*.

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

Kennaway, J. R. (1982). The complexity of a translation of lambda-calculus to combinators. Internal report CS/82/023/E, University of East Anglia, Norwich.

Kennaway, J. R.; Sleep, M. R. (1987). Variable abstraction in $O(n \log n)$ space. *Information processing letters*, 24(5):343–349.

Koopman, P.; Lee, P. (1989). A fresh look at combinator graph reduction. In *Proc. of the SIGPLAN'89 Conf. on Programming Language Design and Implementation*, pp. 21–23.

Nehab, D. F. (1999). Sloth: Uma generalização dos combinadores `SK`. Trabalho final de curso, PUC-Rio, Departamento de Informática.

Noshita, K. (1985). Translation of turner combinators in $O(n \log n)$ space. *Information processing letters*, 20(2):71–74.

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

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