Arthur Branco Costa - 7278156
Daniel Paulino Alves - 7156894
Felipe Yamaguti - 7295336
Pedro Paulo Vezzá Campos - 7538743
Thiago Tatsuo Nagaoka - 7289197

SML - Programação Funcional

São Paulo, SP - Brasil

19 de Junho de 2012

Arthur Branco Costa - 7278156
Daniel Paulino Alves - 7156894
Felipe Yamaguti - 7295336
Pedro Paulo Vezzá Campos - 7538743
Thiago Tatsuo Nagaoka - 7289197

SML - Programação Funcional

Terceiro exercício-programa apresentado para avaliação na disciplina MAC0316, do curso de Bacharelado em Ciência da Computação, turma 45, da Universidade de São Paulo, ministrada pela professora Ana Cristina Vieira de Melo.

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
UNIVERSIDADE DE SÃO PAULO

São Paulo, SP - Brasil

19 de Junho de 2012

Conteúdo

Introdução

"It's going to be legen... wait for it, dary!" *–Barney Stinson*

Programação Funcional é um paradigma baseado no λ-*Calculus*, em oposição ao paradigma imperativo fundamentado na essência da máquina de Turing. Diferente desta última, caracterizada pela representação de estados e suas sucessivas transformações, a programação funcional avalia o problema segundo outra ótica, dando margem a uma interpretação diferenciada.

Ela objetiva a criação de abstrações puramente funcionais, ou seja, mapeamentos de valores de um domínio para outro, sem efeitos colaterais, haja vista que a presença de procedimentos representaria uma violação neste paradigma. Além disso, de acordo com esta concepção, uma vez que o conceito de estado não é aplicável, valores calculados, quando armazenados, o são na forma de constantes, de maneira que o uso extensivo de endereçamentos de memória e o compartilhamento de seus valores associados ao longo do programa (em suma, o emprego de variáveis) são inapropriados.

Este trabalho visa, através da aplicação dos conceitos que dão suporte ao paradigma funcional, criar um sistema de avaliação para a Linguagem Funcional Simplificada (LFSimp), originada a partir da linguagem funcional SML, de forma a manipular diferentes valores e conjuntos de dados e comparar os resultados obtidos pelas principais estratégias de redução existentes: *a priori* e sob demanda.

1 Especificação

Após considerar a linguagem funcional simplificada (LFSimp) definida no enunciado encontramos dificuldades inicialmente em seguir a sintaxe especificada devido às limitações da linguagem SML, que por ser fortemente tipificada, restringe o uso do domínio e imagem de suas funções. Além disso, por não haver uma função que devolva o tipo da varíavel recebida, a especificação fornecida nos obrigou a encontrar alguma alternativa que satisfizesse o alto grau de polimorfismo requisitado pelo programa.

Nossa abordagem ficou bastante parecida com a especificação atualizada do problema, na qual foram definidos os tipos básicos do programa e alguns protótipos.

Assim, modificamos partes da LFSimp passada, buscando alterá-la o mínimo possível. É importante frisar que nossa especificação considera que todos os dados de entrada são consistentes.

A seguir, encontram-se os tipos implementados em nosso EP, com pequenas alterações da nova especificação do enunciado, que serão destacadas adiante:

```
datatype Id = a | b | c | d | e | f | g | h | i | j | k | l | m |
  n | o | p | q | r | s | t | u | v | w | x | y | z;


datatype Expressao
= Valor of int
| Bool of bool
| id of Id
| Conta of Expressao * string * Expressao
| IfThenElse of string * Expressao * Expressao * Expressao
| Aplicacao of  DecFuncao * Expressao list

and DecFuncao = funcao of string * Id list * Expressao;
```

A primeira diferença foi que consideramos uma letra do alfabeto como sendo uma Id da linguagem e não uma *string*, como especificado anteriormente. Esse fator acarretou um leve efeito colateral no nosso programa: as variáveis não podem ter apenas uma letra em seus nomes.

Além disso, o `datatype Expressao` consiste da definição do que seria uma expressão na linguagem. Ele é composto pelos seguintes valores atômicos: Valor (um inteiro), Bool (um booleano) e Id, citado acima. É ainda composto por operações, que podem ser: aritméticas, booleanas, IfThenElse e funções.

2 *Estratégias para a sequência de reduções e avaliação dos parâmetros*

Em funções recursivas podem ser utilizadas duas estratégias para a avaliação da expressão final, *a priori* ou sob demanda, variando conforme a linguagem utilizada. Neste trabalho, forçamos a linguagem especificada a realizar ambas estratégias.

2.1 *A priori* de ou dentro para fora

Estratégia de redução mais tradicional nas linguagens de programação, na qual a função é considerada prioritária sob as outras operações, e nada mais é avaliado enquanto seu λ-termo não é totalmente reduzido. Somente após a função ser totalmente reduzida é que os termos restantes efetuam as respectivas operações.

A implementação dessa estratégia é mais simples que a da anterior, permitindo, assim, a construção de interpretadores e compiladores mais compactos.

No trabalho em questão foi implementado o seguinte algoritmo abstrato:

1. `apriori = fn : Expressao -> Expressao`
   é um alias para a função reduz.
2. `reduz = fn : Expressao -> Expressao`
   - reduz de um valor atômico devolve o próprio valor atômico;
   - reduz uma conta, invoca a função calcula;
   - reduz de um ifThenElse, trata a expressão booleana correspondente, tratando-a; e reduz o bloco correspondente à ela.
   - reduz de aplicação, implica na chamada da função aplicaFuncao

3. `calcula = fn : Expressao * string * Expressao -> Expressao`
   Sua função é reduzir ao máximo cada termo (lados esquerdo e direito) antes de realizar a operação principal; Observação: calcula não é responsável por associar Ids a seus respectivos valores, portanto, a função calcula aplicada a uma Id e uma Conta implica na redução apenas da Conta.

4. `aplicaFuncao = fn : Expressao -> Expressao`
   é responsável por desmembrar a aplicação, preparando os parâmetros para serem entregues à função evalParametros.

5. `evalParametros = fn : Id list * Expressao * Expressao list -> Expressao`
   alias para trocaTudo, que antes, minimiza todas as expressões contidas na lista de parâmetros, através da chamada da função resolveLista

6. `trocaTudo = fn : Id list * Expressao * Expressao list -> Expressao`
   é responsável por substituir todos os parâmetros da declaração da função por seus respectivos valores, que serão obtidos a partir da função resolveLista; isso ocorre, através de sucessivas chamadas da função troca;

7. `troca = fn : Id * Expressao * Expressao -> Expressao`
   responsável por substituir todas as ocorrências de um dado Id na declaração da função por seu valor reduzido; Quando o operando da função troca for uma aplicação, a função trocaLista é chamada; acarretando em uma substituição de todos os parâmetros da lista por seus respectivos valores;

8. `fn : Id * Expressao list * Expressao -> Expressao list`
   percebe-se nesta função a avaliação a priori dos parâmetros da função chamada, uma vez que todos eles são substituídos pelos seus respectivos valores, previamente analisados.

9. `resolveLista = fn : Expressao list -> Expressao list`
   chama a função reduz para cada elemento da lista de parâmetros.

Percebe-se que neste algoritmo, para a realização de qualquer operação, todos os seus operandos são reduzidos primeiramente ao máximo. Isso se aplica tanto para operações aritméticas, booleanas, IfThenElse e para aplicações de funções.

2.2 Sob Demanda ou de fora para dentro

Nestes casos, as operações são efetuadas assim que houver uma oportunidade. Assim, a resposta desejada será encontrada logo que a última ocorrência da função for reduzida.

1. `sobdemanda = fn : Expressao -> Expressao`
   é um alias para a função reduzDemanda.

2. `reduzDemanda = fn : Expressao -> Expressao`
   o comportamento desta função é o mesmo da reduz, excetuando-se o caso quando uma Aplicacao é passada, na qual é chamada a função aplicaFuncaoDemanda;

3. `calculaDemanda = fn : Expressao * string * Expressao -> Expressao`
   responsabiliza-se por reordenar a expressão passada, de maneira a permitir o cálculo dos valores assim que tornam-se disponíveis. Por exemplo, uma expressão no formato: 6 + (4 + (2 - (3 + 5))) será convertida na forma seguinte forma: (6 + 4) + (2 - (3 + 5)) e a assim, será realizado o cálculo do primeiro termo, resultando na expressão: 10 + (2 - (3 + 5)); esse processo se repete recursivamente e serão obtidas as seguintes expressões: 12 - (3 + 5) 9 - (5) 4

4. `aplicaFuncaoDemanda = fn : Expressao -> Expressao`
   é responsável por desmembrar a aplicação, preparando os parâmetros para serem entregues à função trocaTudo, observa-se que diferentemente da função aplicaFuncao, esta não invoca a resolveLista para a lista de parâmetros, como consequência a aplicação após a chamada de trocaTudo não estará minimizada, abrindo margem para a função calculaDemanda poder calcular os valores sod demanda, apenas nos momentos necessários, ou seja, quando eles serão de fato utilizados.

5. `trocaTudo = fn : Id list * Expressao * Expressao list -> Expressao`
   mesma implementação da versão a priori.

A seguinte função realiza uma redução sob demanda:

A estratégia de redução sob demanda não pode ser realizada nos operadores booleanos. Isso ocorre, pois diferente das operações binárias aritméticas, definidas na especificação, nas quais todas as operações (adição e subtração) possuem o mesmo grau de precedência, ao passo que nas operações booleanas os operadores (and e or) possuem uma precedência similar à de soma e multiplicação, por exemplo.

Dessa forma, seria necessária uma verificação de todos os valores pertencentes a expressão antes de efetivamente realizar o cálculo da expressão. Contudo, a natureza da recursão não possibilita tal procedimento.

3 Exemplos

```
val aa = Valor 7;
val bb = Valor 3;
val cc = Conta (aa, "-", bb);
val dd = Conta (cc, "-", cc);
val ee = Bool (true);
val ff = Bool (false);
val gg = Conta (ee, "And", ff);
val hh = Conta (ee, "Or", ff);
val ii = Conta (gg, "==", hh);
val pp = Conta (Valor 0, "+", Valor 4);
val mn = Conta (id x, "+", id y);
val pq = Conta (Conta (Valor 9, "-", Valor 9), "+", id x);
val kk = Conta (Valor 0, "+" , Conta (Conta (Conta (Conta (Valor 5, "+", Valor 2), "-", Valor 2), "+", Valo
val testIf = IfThenElse ("if", Conta (Valor 5, "==", Conta (Valor 2, "+", Valor 3)), Valor 4, Valor 3);
val testIfid = IfThenElse ("if", Conta (id a, "==", Conta (Valor 2, "+", id b)), Valor 4, Valor 3);
val teste2If = Conta (Valor 5, "+", IfThenElse ("if", Conta (Valor 5, "==", Valor 4), Valor 5, Valor 0));
val teste5 = Conta (Conta (Valor 7, "-", Valor 8), "-", Conta (Valor 9, "+", Valor 10));
val teste3 = Conta (Bool true, "Or", Conta (Bool true, "And", IfThenElse ("if", Conta (Valor 7, "==", Valor
val teste4 = Conta (Conta (Bool false, "Or", Bool false), "Or", Conta (Bool true, "And", Bool false));
val teste2 = Conta (Bool false, "And", Conta (Bool false, "Or", Bool true));


reduz cc;
reduzDemanda cc;
print ("\n\n");
reduz aa;
reduzDemanda aa;
print ("\n\n");
reduz dd;
reduzDemanda dd;
print ("\n\n");
reduz gg;
reduzDemanda gg;
print ("\n\n");
reduz mn;
reduzDemanda mn;
print ("\n\n");
reduz kk;
reduzDemanda kk;

print ("\n\nNovos testes\n\n");
```

```
reduz teste5;
reduzDemanda teste5;
print ("\n\n");
reduz teste3;
reduzDemanda teste3;
print ("\n\n");
reduz teste4;
reduzDemanda teste4;
print ("\n\n");
and calcula (id vala, aa, id valc) = Conta (id vala, aa, id valc)
  | calcula (id vala, aa, valc) = Conta (id vala, aa, reduz(valc))
  | calcula (vala, aa, id valc) = Conta (reduz(vala), aa, id valc)
reduz teste2;
reduzDemanda teste2;

print ("\n\nTestes mais novos!\n\n");


val func = funcao ("fun", [x, y, z], Conta(id x, "+", id y));
val aplic = Aplicacao (func, [Valor 5, Conta (Valor 3, "+", Valor 3), Conta (Valor 4, "+", Valor 4)]);

print "\n\n";
reduz  aplic;
reduzDemanda aplic;
print "\n\n";

val bla = funcao ("fun", [a, b], IfThenElse ("if", Conta (id a, "==", id b), id a, id b));
val ble = funcao ("fun", [a, c, d], Conta (Aplicacao (bla, [id a, Valor 0]), "+", id d));

reduz (Aplicacao (ble, [Bool true, Valor 0, Valor 1]));
reduzDemanda (Aplicacao (ble, [Bool true, Valor 0, Valor 1]));

print "\n\n";
```

Conclusão

O desenvolvimento deste projeto evidenciou a importância de avaliadores de linguagens de programação, seja de dentro para fora ou de fora para dentro, e como expressões idênticas, ao ser analisadas sob perspectivas distintas, podem influenciar o desempenho do programa.

Diferentes estratégias aplicáveis proporcionam diferentes sequências de reduções, sendo esta ou aquela mais apropriada para maximizar a eficiência das reduções a serem efetuadas de acordo com o caso em questão. Logo, não se pode julgar uma dada estratégia melhor do que outra para a maioria dos programas funcionais, mas sim, a que for melhor para um problema (ou uma família de problemas) em particular. Contudo, é notório como as duas opções sempre deverão encontrar resultados válidos, na medida em que satisfazem as propriedades de correção e confluência e garantem a equivalência das expressões finais, variando apenas em eficiência.

Ademais, considerando que o paradigma funcional não é tão usual quanto ao paradigma imperativo, tivemos que nos adaptar à carência de variáveis e comandos iterativos, já que, embora estas ferramentas estejam presentes na linguagem SML, restringimos o nosso arsenal de possibilidades, limitando-nos aos recursos puramente funcionais.

Por fim, é fundamental ressaltar como programar conforme o enfoque funcional constitui uma nova forma de pensar, modelar e abstrair resoluções, não se tratando, portanto, de uma discussão trivial. Assim, torna-se evidente como esta aplicação prática dos conceitos estudados ao longo do semestre foi determinante para a fixação do conhecimento aprendido, tão elaborado, e a sua inegável relevância.

Referências

[1] MELO, A. C. V. de; SILVA, F. S. C. da. *Princípios de Linguagens de Programação*. São Paulo, SP: Edgard Blucher, 2003.