O Projeto e Implementação da Linguagem A

Silvio Lemos Meira
Martín A. Musicante
André Santos

Universidade Federal de Pernambuco
Departamento de Informática
CP 7851, 50739 Recife - PE - Brasil

RESUMO

Este artigo descreve os principais problemas e soluções encontrados na implementação de uma linguagem funcional sofisticada. A implementação é muito eficiente, pelos padrões internacionais.

ABSTRACT

We describe the main problems of the implementation of a modern functional language and the solutions we adopted. The final result is a prototype, which can be considered as quite efficient in relation to international standards.

1 Introdução

As linguagens funcionais oferecem uma solução simples e clara a muitos dos problemas de desenvolvimento de *software*. Seu uso vai desde especificações executáveis [Tur84] e programação final de aplicações [Hug84], até código objeto na derivação de programas a partir de especificações (formais) [BM91].

Um programa funcional é formado por uma série de definições de funções (possivelmente recursivas) e por uma expressão a ser avaliada.

Entre as principais características das linguagens funcionais podemos mencionar:

- *Transparência Referencial:* Programas são expressões cujos valores dependem somente dos seus componentes, independendo do contexto da aplicação.
- *Funções de Alta Ordem:* Funções são "cidadãs de primeira classe", tratadas da mesma forma que tipos de dados como inteiros, listas, etc. Uma função não só pode ser passada como parâmetro para uma outra, como também ser o resultado da avaliação de uma expressão.
- *Propriedades Formais:* A estreita relação que existe entre as linguagens funcionais e as teorias formais serve de base para a análise da lógica de programas e suas propriedades.

2 A linguagem A

A é uma linguagem de programação puramente funcional, não estrita, polimórfica e modular, na tradição de Miranda [Tur85], Lazy ML [AJ88] e Haskell [HW+88].

Apresentamos aqui uma introdução à linguagem, que pode ser vista, exceto pela sintaxe, como uma introdução aos conceitos presentes na maioria das linguagens funcionais. A descrição é informal e feita através de exemplos escritos diretamente na sintaxe de A. Para uma descrição mais detalhada da linguagem deve-se reportar a [Mei88]. O seu sistema de tipos é descrito em detalhes em [Mou89]. A semântica denotacional do seu nível funcional é descrita em [Alb89] e sua semântica de ações em [BSM90].

2.1 Introdução

A é uma linguagem de programação puramente funcional, ou seja, sem efeitos colaterais ou carácterísticas imperativas. Um programa (ou *script*) é um conjunto de equações que definem funções e estruturas de dados.

A definição da função fatorial, por exemplo, pode ser feita por:

```
DEF fatorial;
  fatorial x = 1, x = 0;
  fatorial x = x * fatorial (x - 1), x ~= 0;
END fatorial;
```

Dentro do bloco DEF-END temos duas equações: A primeira estabelece que o fatorial de um número `x` é igual a `1`, se este número for igual a zero (`x = 0`). A segunda equação define o valor do fatorial caso o número seja diferente de zero (`x ~= 0`). O uso de uma expressão booleana após a definição estabelece o que chamamos de equações guardadas, onde a expressão é chamada de "guarda". No nosso exemplo, (`x = 0`) e (`x ~= 0`) são guardas da primeira e da segunda equação, respectivamente. A equação só será utilizada se sua guarda for verdadeira, com as equações sempre testadas de cima para baixo.

A aplicação é feita por justaposição, como em (`f x y`) que passa `x` e `y` como argumentos de `f`.

Além de equações guardadas podemos usar casamento de padrões (*pattern-matching*) na definição de funções, possibilitando definições sucintas e claras, como em

```
DEF fatorial;
  fatorial 0 = 1;
  fatorial x = x * fatorial (x - 1), x ~= 0;
END fatorial;
```

Observe que a primeira equação só será utilizada se o parâmetro passado à função for zero. Os padrões são definidos através de construtores e variáveis e podem ser tão complexos quanto se queira.

2.2 Definições Locais

A é uma linguagem estruturada, de forma que cada nome introduzido por uma definição de função tem um escopo limitado. Um exemplo disto são as definições locais:

```
DEF somaqua;
  somaqua x = quax + quax;
  DEF quax;
    'quax = x**2;
  END quax;
END somaqua;
```

em que utilizamos a definição local `quax`, que evita escrever (`x**2 + x**2`), que levaria a duas avaliações da mesma expressão. Além de definirmos constantes, como no caso acima, podemos definir também funções locais à outras funções. O uso de definições locais ocorre, em geral, por questão de eficiência e/ou modularidade.

Todas as definições em A são, a priori, recursivas, ou seja, os identificadores definidos por uma declaração podem ser utilizados dentro da própria declaração. Desta forma é deixado para o compilador a tarefa de distinguir as declarações recursivas das não recursivas.

2.3 Tipos de Dados

Os tipos básicos de A são os tipos escalares inteiro (`Int`), real (`Real`), booleano (`Bool`) e caractere (`Char`), com os operadores usuais. Além dos tipos de dados escalares as linguagens funcionais fazem uso extensivo de alguns tipos de dados estruturados, como listas e tuplas.

2.3.1 Listas

Uma lista é uma estrutura homogênea, que caracteriza uma sequência de zero ou mais elementos de um mesmo tipo. Estruturalmente o tipo lista pode ser visto como sendo a lista vazia ou tendo dois elementos: um chamado cabeça da lista, e o outro cauda, que é também uma lista. Uma lista pode ser representada de duas formas: através do uso do construtor de listas ":", forma na qual `(1:(2:(3:[])))` representa a lista dos inteiros de um a três, e `[1,2,3]`, que representa a mesma lista, sem o uso de construtores.

A primeira representação corresponde mais claramente ao processo de construção da lista. Para listas de caracteres temos outra forma de escrevê-las, entre aspas. Listas podem aparecer em padrões de qualquer dessas formas. Listas podem ser compostas por quaisquer elementos do mesmo tipo.

Os operadores pré-definidos sobre listas são: seleção da cabeça da lista, seleção da cauda da lista, concatenação de listas, comprimento de listas, diferença de listas.

Listas podem ser descritas por geradores, que são abreviações para listas de inteiros usadas frequentemente em programas funcionais. Os geradores criam listas finitas ou

infinitas de inteiros. O gerador (...) cria listas de inteiros no intervalo especificado, de forma que as duas expressões `[x..y]` e `[x,x+1,x+2,...,y]` são idênticas:

O gerador (...) denota listas infinitas. A lista de todos os inteiros, por exemplo, pode ser denotada por `[1...]`.

Compreensões de listas são uma forma ainda mais genérica de se expressar listas. Esta notação, semelhante ao conceito de compreensão de conjuntos (*set comprehension*) da teoria de conjuntos de Zermelo-Fraenkel, permite a definição de uma lista através da descrição de suas características.

Uma função que retorna a lista dos pares de uma lista através de compreensões seria

```
DEF pares;
  pares x = [n | n <- x; n % 2 = 0];
END pares;
```

que é a lista dos números (`n`) tirados da lista (`x`) tais que o resto da sua divisão por dois seja zero (`n % 2 = 0`). O resultado da avaliação da expressão (`pares [1..8]`) seria a lista `[2,4,6,8]`.

2.3.2 Tuplas

Outro tipo de dados estruturado presente em **A** são as tuplas. Os componentes das tuplas, ao contrário das listas, não possuem necessariamente o mesmo tipo. Assim uma tupla de 2 elementos (também chamada par ou 2-upla) pode possuir elementos de tipos distintos como em `(5,"casa")`, onde temos um inteiro e uma lista de caracteres.

2.3.3 Funções

O tipo função é descrito pelo tipo dos parâmetros seguido do tipo do resultado da função. Assim, temos que o tipo da função `fatorial` é `(Int -> Int)`, visto que ela recebe um elemento do tipo inteiro como parâmetro e retorna um outro como resultado.

A é uma linguagem de alta ordem, ou seja, funções são cidadãs de primeira classe, podendo ser passadas como parâmetros e retornadas como resultado de outras funções. A aplicação de funções é feita por justaposição e é associativa à esquerda.

Funções de alta ordem são usadas extensivamente em programas funcionais e permitem definições concisas e poderosas.

2.4 Avaliação Preguiçosa

Existem linguagens funcionais estritas e "preguiçosas". Em linguagens funcionais estritas como ML [GMW79] o mecanismo de avaliação de chamadas a funções é semelhante ao das linguagens imperativas: Os argumentos passados à função são avaliados e só então o código da função é executado. Em linguagens funcionais com avaliação preguiçosa os argumentos só são avaliados se e quando necessário.

O mecanismo de avaliação de funções de **A** é preguiçoso (*lazy*), isto é, nenhuma expressão é avaliada enquanto não for necessário. Este mecanismo de avaliação permite que funções indefinidas em uma linguagem estrita possam ser avaliadas em A. Outra grande vantagem do uso de avaliação preguiçosa é a possibilidade de tratar estruturas de dados infinitas.

Tanto quanto o uso de funções de alta ordem, a possibilidade de lidar com estruturas infinitas através do mecanismo de avaliação preguiçosa traz grandes mudanças no estilo de programação e na facilidade de se escrever programas concisos e muito poderosos. O uso de estruturas infinitas também permite um melhor tratamento de entrada e saída interativa, bem como da comunicação de processos sob o ponto de vista funcional.

2.5 Sistema de Tipos Polimórfico

Polimorfismo permite a definição de funções sobre famílias de tipos que tenham a mesma estrutura básica. A função, no caso, tem parâmetros de tipo que são instanciados no contexto do uso, particularizando sua aplicação a determinado domínio. Um exemplo disso é

```
DEF map : (a -> b) -> List a -> List b;
  map f [] = [];
  map f (h:t) = f a : map f t;
END map;
```

que aplica a função `f` aos elementos de uma lista qualquer de tipo (`List a`).

2.6 Tipos Algébricos

Além dos tipos pré-definidos o usuário pode definir novos tipos através do uso de definições algébricas, abstratas e sinônimas.

Os tipos de dados algébricos permitem a definição de novos tipos estruturados, além dos pré-definidos na linguagem. Os tipos algébricos são definidos pelos seus construtores e, possivelmente, por axiomas, que definem propriedades do tipo. Um exemplo de definição de tipo algébrico seria a definição de um tipo `Nat`, dos números naturais. Sua definição em **A** seria

```
ALG Nat;
  OPS Zero,Suc,Pred;
    Zero : Nat;
    Suc  : Nat -> Nat;
    Pred : Nat -> Nat;
  END Zero,Suc,Nat;
  ALL x:Nat.
    Pred (Suc  x) => x;
    Suc  (Pred x) => x;
  END;
```

```
END Nat;
```

Tipos algébricos com axiomas são construções poderosas e devem ser utilizadas com bastante cuidado, pois pode haver problemas de terminação do sistema de reescritura de termos associado aos axiomas.

2.7 Tipos Abstratos

O uso de tipos abstratos de dados se torna importante na construção de grandes sistemas, pois esconde-se a representação concreta de um determinado tipo, permitindo que se acesse apenas a sua interface, ou seja, funções que manipulam o tipo. Desta forma, mantendo-se as suas características semânticas, a representação concreta do tipo pode ser mudada para, por exemplo, fazer uso de uma estrutura de dados mais eficiente, sem que isso provoque alterações no restante do programa. Um exemplo de definição de pilhas como um tipo abstrato pode ser

```
ABS Pilha;
  OPS PV,emP,toP,desP;
    PV   : Pilha a;
    emP  : a -> Pilha a -> Pilha a;
    toP  : Pilha a -> a;
    desP : Pilha a -> Pilha a;
  END PV,emP,toP,desP;
  IMP Pilha a AS List a;
    DEF PV;   PV = [];       END PV;
    DEF emP;  emP x p = x:p; END emP;
    DEF toP;  toP p = hd p;  END toP;
    DEF desP; desP p = tl p; END desP;
  END;
END Pilha;
```

Inicialmente definimos sua interface (através do bloco `OPS-END`), ou seja, as funções através das quais se pode ter acesso ao tipo, e em seguida sua `IMP`lementação concreta, como (`AS`) listas, não acessível ao restante do programa. Em seguida definimos como as funções descritas na interface são implementadas baseadas no tipo concreto.

Um tipo abstrato de dados se distingue de um tipo algébrico por não permitir, por exemplo, o uso de casamento de padrões, visto que não há construtores e sim funções que manipulam o tipo.

2.8 Módulos

A é uma linguagem que se propõe a possibilitar a programação de grandes sistemas e para isso há um sistema de módulos que permite a compilação de programas em separado. Além de permitir uma melhor estruturação do sistema, o uso de módulos evita o esforço

computacional necessário na recompilação de um sistema, quando apenas uma pequena modificação foi efetuada.

Os módulos em **A** são definidos por um bloco `MOD-END`, onde são definidos os identificadores exportados (funções, tipos etc.) e suas respectivas declarações.

2.9 Entrada, Saída e Processos

As facilidades oferecidas pela linguagem puramente funcional para entrada e saída são bastante limitadas, mas é muito sofisticada pelo uso de processos como mecanismo de interação.

A entrada é tratada de forma preguiçosa como uma lista (possivelmente infinita) de caracteres. A saída de um programa funciona de forma idêntica, ou seja, a computação de um programa é feita sob demanda da saída, se encaixando ao conceito de processos em **A**. Um processo é visto como uma função sendo computada, e através do subconjunto de **A** para definição e manipulação de processos, pode-se conectar essas entradas e saídas (na realidade canais de entradas e canais de saída) a dispositivos ou processos, que estariam possivelmente ligados a arquivos, entrada e saída padrão, janelas etc. Para maiores detalhes sobre a sintaxe e semântica do nível de configuração e manipulação de processos de **A** (também chamado *Actions*) pode-se reportar a [Mei91, MBS91], visto que sua discussão foge ao escopo deste trabalho.

3 A Implementação da Linguagem A

Duas grandes áreas de pesquisa estão no escopo deste projeto: a definição e implementação do *front* e *back-end* de compiladores de linguagens funcionais. O primeiro trata da tradução das construções de alto nível presentes em **A** para uma versão do lambda-cálculo. O resultado deste processo de tradução é usado como representação intermediária no processo de compilação. O último proporciona um método de compilação da notação gerada na fase anterior para um código eficiente.

O *front-end* é implementado por dois sub-sistemas: um analisador sintático e um gerador de código intermediário.

O analisador sintático é escrito em C [KR78], usando ferramentas do sistema operacional UNIX para o desenvolvimento de compiladores, e traduz o código fonte para uma árvore sintática.

O gerador de código intermediário foi escrito em Lazy ML, uma linguagem funcional com características semelhantes a **A**. Ele lê a árvore sintática e efetua diversas transformações, gerando uma representação do programa original em lambda-cálculo enriquecido. O gerador de código finalmente recebe um programa nesta representação e gera o código executável.

A decisão de desenvolver o compilador em uma linguagem funcional pura tem por objetivo possibilitar que ele seja facilmente reescrito para a notação de **A**, e desta forma fazer uso da técnica de *bootstrapping*, ou seja, possibilitar que o código fonte do compi-

diretamente pelo *back-end*.

4.2 Análise de Escopo

O primeiro passo após a verificação da corretude sintática do programa através da análise sintática é a verificação da corretude do programa no que se refere aos nomes de identificadores que nele aparecem. Isto é feito através de um analisador semântico que trata condições básicas de escopo que não podem ser verificadas na análise sintática, como testar se as funções e variáveis utilizadas foram definidas e estão no escopo da expressão onde são usadas, se as equações em que se definem as funções possuem todas o mesmo número de parâmetros, etc.

4.3 Remoção de Recursão

Em **A** todas as funções globais são vistas como mutuamente recursivas, ou seja, uma definição de função (ou de tipo) pode ser usada em qualquer parte do *script*. Essas funções, no entanto, raramente são, de fato, mutuamente recursivas. Esse passo, através de análise de dependências, estabelece quais funções são mesmo mutuamente recursivas, e em seguida reordena as definições existentes (através de sort topológico) para que, do ponto de vista interno do compilador, as definições façam referências apenas a funções definidas anteriormente ou a funções que sejam mutuamente recursivas em relação à ela.

A remoção de recursão visa, no caso do compilador de **A**, simplificar o algoritmo de verificação de tipos, tornando-o mais eficiente, sendo também extremamente importante quando se processam determinadas otimizações no processo de compilação, trazendo grande impacto no desempenho do código gerado em determinadas implementações. O *back-end* do compilador de **A** não faz uso deste tipo de otimização.

4.4 Verificação de Tipos

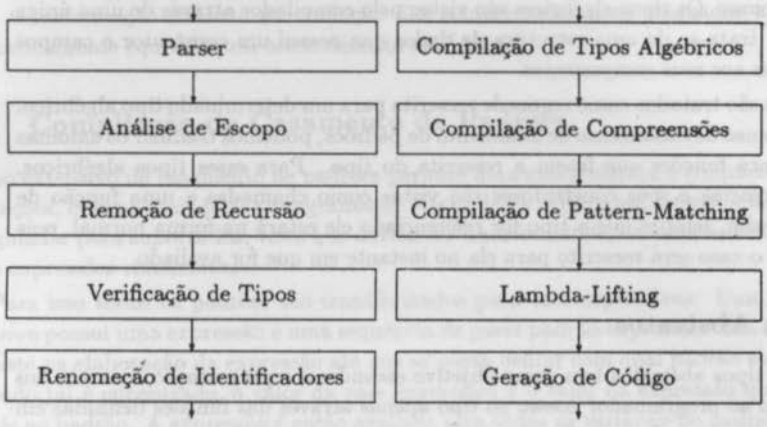
A possui um sistema de tipos fortes e polimórfico, verificado por um algoritmo de inferência de tipos . O sistema de tipos de **A** foi inicialmente descrito em [Mou89], onde também é mostrado o protótipo de um verificador de tipos para a linguagem, baseado no algoritmo de verificação de tipos de Hindley-Milner [Mil78].

Além de impedir que erros de programação passem despercebidos, a verificação de tipos permite que outros passos do compilador possam assumir que o programa está corretamente tipado. Mais ainda, esses passos podem fazer uso da informação obtida pela inferência de tipos para gerar código mais eficiente. De forma semelhante, a informação sobre os tipos permite que se utilizem representações mais eficientes para as estruturas de dados da linguagem [Ler90].

Após a passagem pelo verificador de tipos as definições de tipos algébricos e abstratos devem ser compiladas para possibilitar o uso efetivo dos tipos definidos. Esta compilação é descrita nas seções seguintes.

lador possa ser compilado pelo próprio.

A estrutura geral do compilador de **A**, com cada um de seus passos, é vista abaixo:



Alguns desses passos são feitos em conjunto, e não através de múltiplas passagens pelo código. A compilação de compreensões e de casamento de padrões, por exemplo, são efetuadas em um mesmo passo de tradução.

4 O Front-End do Compilador

A seguir descrevemos os diversos passos do compilador, destacando as transformações efetuadas no código fonte para obtermos a representação do programa no lambda-cálculo enriquecido. Em cada uma dessas transformações o programa é reescrito para uma notação mais eficiente e que usa um conjunto menor de primitivas do lambda-cálculo enriquecido.

4.1 Análise Sintática

A fase de análise sintática produz a árvore sintática abstrata para cada frase do *script* de entrada. O analisador sintático foi desenvolvido usando `lex` e `yacc`, o primeiro um gerador de analisadores léxicos e o último um gerador de analisadores sintáticos, ambos disponíveis no sistema operacional UNIX.

Ainda nesta fase são expandidas abreviações sintáticas oferecidas pela linguagem (*syntactic sugar*), tais como geradores de listas, que são transformados em chamadas a funções pré-definidas em uma biblioteca. Diversos operadores pré-definidos em **A** são tratados de forma semelhante, ou seja, não são suportados diretamente pelo *back-end*, e sim por funções comuns pré-definidas em uma biblioteca de funções. Este é o caso dos operadores `@` e `#` (concatenação e comprimento de listas) e outros. Apenas as operações básicas sobre inteiros e booleanos e as funções de conversão de tipos são suportadas

4.5 Tipos Algébricos

Na compilação de tipos algébricos devem ser considerados dois casos: tipos algébricos com e sem axiomas. Os tipos algébricos são vistos pelo compilador através de uma única representação: trata-se de uma estrutura de dados que possui um construtor e campos correspondentes aos seus componentes.

Os axiomas são tratados como regras de reescrita para um determinado tipo algébrico. Assim, fazendo uso do mecanismo de casamento de padrões, podemos traduzir os axiomas diretamente para funções que fazem a reescrita do tipo. Para esses tipos algébricos, portanto, referências a seus construtores são vistas como chamadas a uma função de reescritura. Assim, sempre que o tipo for referenciado ele estará na forma normal, pois se este não for o caso será reescrito para ela no instante em que for avaliado.

4.6 Tipos Abstratos

A definição de tipos abstratos tem como objetivo esconder a representação concreta dos tipos, deixando ao programador acesso ao tipo apenas através das funções definidas em sua interface. Assim, após a verificação de tipos simplesmente é feito um mapeamento entre as chamadas às funções do tipo abstrato para as funções definidas em sua implementação.

Passa-se para a fase de compilação apenas as informações sobre as definições de funções descritas na implementação do tipo. Assim, apesar de o usuário não ter acesso à representação do tipo, o compilador faz um mapeamento entre a sua representação abstrata e a concreta, que está presente em sua definição. O uso de tipos abstratos, portanto, não causa nenhum aumento no tempo de execução, pois a informação sobre o tipo abstrato é utilizada apenas na fase de verificação de tipos, sendo desprezada nas demais fases do compilador, onde é usada a representação concreta do tipo.

4.7 Renomeação

Muitas das transformações efetuadas no programa para permitir sua compilação para o lambda-cálculo efetuam modificações no escopo de variáveis (compilação de casamento de padrões) e definições (lambda lifting). Essas modificações podem gerar um problema semelhante ao da captura de variáveis no lambda cálculo (casamento de padrões) ou colisão de nomes (lambda lifting).

Para evitar a introdução de mecanismos de detecção e solução desses problemas, os identificadores introduzidos por definições locais e as variáveis são renomeados para nomes únicos, eliminando o problema.

Os nomes originais são substituidos e desprezados, pois os erros detectáveis pelo compilador, que gerariam mensagens de erro referenciando esses nomes, já foram detectados nas fases anteriores (análise sintática, de escopo e verificação de tipos).

A unicidade dos nomes dos identificadores é mantida pelos demais passos do compilador, pois identificadores introduzidos por transformações também preservam esta propriedade.

4.8 Compilação de Compreensões

As compreensões de listas, como as demais construções de alto nível presentes nas linguagens funcionais, devem ser transformadas em expressões do lambda-cálculo enriquecido. Essa transformação consiste na tradução das compreensões para chamadas a funções semanticamente equivalentes, como descrito em [Aug87].

4.9 Compilação de Casamento de Padrões

O uso extensivo de casamento de padrões permite uma maior clareza e simplicidade nas definições, e torna mais fácil a programação. Os padrões exigem um esforço adicional do compilador para suportá-los, visto que devem ser transformados, em uma instância final, para expressões condicionais.

Para isso todos os padrões são transformados para *case expressions*. Uma *case expression* possui uma expressão e uma sequência de pares padrão-expressão. Sua avaliação consiste na elaboração da expressão até que se possa definir com qual padrão ela "casa". Quando tal é encontrado, o valor da *case expression* é o valor da expressão que corresponde ao padrão. A expressão é então avaliada com todas as variáveis do padrão ligadas às correspondentes subpartes do padrão.

Na compilação, padrões complexos (padrões dentro de padrões) são transformados em padrões simples, que possuem construtores apenas em sua parte mais externa, e todos os seus subpadrões são variáveis. A razão para esta transformação é que o casamento de padrões simples pode ser compilado para código muito eficiente de maneira bastante simples.

A compilação eficiente de casamento de padrões foi abordada de forma semelhante, mas desenvolvida independentemente, por Augustsson [Aug85] e Wadler [PJ87]. A técnica que descrevemos é similar, porém com algumas modificações para suportar variáveis repetidas em padrões e equações guardadas.

O algoritmo que transforma padrões complexos em *case expressions* simples tem a forma geral de:

- Reordenam-se os padrões em grupos, cada grupo tendo o mesmo construtor mais externo. Esse reordenamento deve ser feito respeitando alguns critérios, pois a ordem dos padrões é importante em alguns casos. Desta forma, só os padrões que não se sobrepõem podem ser trocados sem mudar a semântica.

- Cada um desses grupos forma um único padrão e uma nova *case expression* dentro do padrão determina que padrão realmente "casa".

- Cada uma das *case expressions* introduzidas pelo passo anterior é tratada pelo algoritmo, até que tenhamos apenas padrões simples.

Um exemplo simples desta transformação é a função `last`, que retorna o último elemento de uma lista:

```
DEF last;
  last (Cons a Nil) = a;
  last (Cons a x  ) = last x;
END last;
```

que transformada em uma notação usando *case switches* leva a

```
DEF last;
  last v = CASE v OF
             Cons a Nil : a;
             Cons a x   : last x;
           END;
END last;
```

Como o construtor mais externo dos dois padrões é o mesmo, o CASE é transformado em

```
DEF last;
  last v = CASE v OF
             Cons a x : CASE x OF
                          Nil : a;
                          DEFAULT : last x;
                        END;
             DEFAULT : ERROR;
           END;
END last;
```

Para cada CASE é introduzido um caso DEFAULT que ocorre quando nenhum dos construtores é encontrado ou quando o padrão é uma variável.

A compilação de *case expressions* simples pelo *back-end* é muito semelhante à compilação de *cases* em linguagens imperativas convencionais, para a qual há métodos bastante eficientes.

4.10 Lambda Lifting

A linguagem que o *back-end* pode compilar é uma linguagem de um único nível, isto é, todas as definições de funções são globais. Isto se deve ao fato de que, além de tornar a compilação mais simples, este tipo de programa sem definições locais permite a geração de código mais eficiente.

A, como já vimos, permite definição de funções locais, que são transformadas em globais através de *lambda lifting*, uma técnica para a transformação de programas funcionais com definições de funções locais, possivelmente com variáveis livres, em um programa consistindo somente de definições de funções globais.

A dificuldade principal desta transformação é a existência de variáveis livres nas definições locais (uso de variáveis globais em definições locais). Vejamos novamente o exemplo da função que retorna a soma dos quadrados de um número:

```
DEF somaqua;
  somaqua x = quax + quax;
  DEF quax;
    quax = x**2;
  END quax;
END somaqua;
```

Se a única modificação aqui é definir quax no nível global, a variável x se tornaria indefinida. Quando as definições locais possuem variáveis livres (ou globais) estas devem ser abstraídas, alterando as definições para que recebam estas variáveis como parâmetro, tornando possível promover a definição para o nível global. A técnica adotada consiste em:

- para cada variável livre em uma definição de função local, adicionamos um argumento à função (os nomes de funções são tratados como constantes e não são abstraídos).
- aplicamos as mesmas variáveis livres a cada aplicação da função, substituindo `f` por `f` $x_1 ... x_n$; onde $x_1 ... x_n$ é o conjunto de variáveis livres na definição de `f`.

No exemplo acima teríamos qua adicionar um parâmetro a quax e passar x como argumento nas chamadas a quax em somaqua. Feito isso, poderíamos levar a definição de quax para o nível global:

```
DEF somaqua;
  somaqua x = quax x + quax x;
END somaqua;
DEF quax;
  quax x = x**2;
END quax;
```

que é equivalente à encontrada no primeiro exemplo.

A primeira impressão é que esta transformação sempre pode ser feita de maneira simples e direta. Esta técnica pode, entretanto, introduzir novas variáveis livres ao remover outras [Joh85] e o passo de abstração deve ser repetido até que não ocorram mais variáveis livres em definições. Para evitar essas transformações repetidas no programa, que são bastante ineficientes, computamos o conjunto final de variáveis que devem ser abstraídas de cada função, resolvendo um sistema de equações sobre esses conjuntos.

4.11 Módulos

A compilação de módulos em separado, do ponto de vista do *front-end*, consiste em importar informações de tipos de dados e funções definidos em outros módulos, para o processo de verificação de tipos, e exportar as respectivas informações sobre os tipos de dados e funções que o próprio módulo exporta.

Do ponto de vista do *back-end* o código objeto para cada módulo deve ser gerado mantendo as referências a funções de outros módulos, de forma que o *linker* possa resolvê-las. Tal processo é similar ao usado em linguagens imperativas.

4.12 Geração de Código Intermediário

Após as várias transformações descritas nas seções anteriores, o programa encontra-se escrito exclusivamente na notação do lambda-cálculo estendido, onde todas as construções de alto nível presentes na linguagem foram transformadas para o lambda-cálculo. Além disso as expressões lambda estão definidas em um único nível e não há variáveis livres. Ao chegar a este ponto pode-se fazer uso de uma das diversas técnicas de compilação de linguagens funcionais. Para A fez-se a opção de utilizar o *back-end* baseado em compilação para Multi-Combinadores Categóricos, desenvolvido no próprio Departamento de Informática da UFPE.

Assim, a última fase do *front-end* gera uma saída passível de tratamento por CM-C, que é um compilador de expressões do lambda-cálculo enriquecido para código de G-MC, uma máquina (absrata) de Multi-Combinadores Categóricos. Esse código é implementado em linguagem C, e então compilado para código objeto.

É importante observar que o *back-end* pode ser mudado sem que isso demande maiores modificações no *front-end*, e vice-versa. Melhoramentos e otimizações podem ser incluídas em ambos sem que um interfira (necessariamente) no outro. Esta independência entre as fases permite que o trabalho seja desenvolvido separadamente, desde que a interface entre eles seja mantida. O *front* e o *back-end* foram desenvolvidos simultaneamente, em projetos independentes.

5 O Back End do Compilador

G-MC é uma máquina de redução de grafos inspirada, no que se refere à sua dinâmica de reduções, nos Multi-Combinadores Categóricos [Lin86]. A definição de GM-C foi feita, através de um conjunto de esquemas de tradução (cuja linguagem fonte é puramente funcional), e de um sistema de transição de estados.

Para cada expressão na linguagem funcional é gerado um código seqüencial, imperativo, que será executado por GM-C.

O código intermediário passado pelo *front-end* para o tradutor de GM-C é uma representação do sistema de Multi-Combinadores Categóricos [Lin86], formalismo no qual se baseia a redução de grafos implementada por GM-C.

5.1 O Sistema de Transição de Estados

Cada estado de GM-C é uma 6-tupla,

$$\langle C, B, T, H, O, E \rangle$$

onde C é o código a ser executado, B é uma pilha de valores básicos usada na avaliação de expressões aritméticas e lógicas, T e E são pilhas usadas durante a redução de grafos, H é o *heap*, a memória onde a representação gráfica dos termos é armazenada, e O é a saída padrão.

GM-C está definida como um conjunto de regras de transição de estados, onde transições da forma

$$\langle C, B, T, H, O, E \rangle \Rightarrow \langle C', B', T', H', O', E' \rangle$$

são interpretadas como:

> "Quando a máquina está no estado dado por $\langle C, B, T, H, O, E \rangle$, ela pode passar ao estado $\langle C', B', T', H', O', E' \rangle$ em um só passo".

Por exemplo, para uma configuração arbitrária de GM-C, a seqüência de código

MKcell(1); MKcte(1)0;

que gera uma célula nova no *heap*, criando um nó contendo a constante 1, será executada passando pelos seguintes estados:

$$\begin{aligned} & \langle \text{MKcell(1).MKcte(1)0}.C, B, T, H, O, E \rangle \\ \Rightarrow\ & \langle \text{MKcte(1)0}.C, B, d.T, H[d = u_0], O, E \rangle \\ \Rightarrow\ & \langle C, B, d.T, H[d = cte(1)], O, d.E \rangle \end{aligned}$$

A instrução MKcell(1) cria uma célula de tamanho 1 na memória. O endereço da nova célula é empilhado em T (a notação "$d.T$" indica que a célula de endereço d está no topo da pilha T).

A instrução MKcte(1)0 preenche o campo 0 da célula que está no topo da pilha T com a constante 1. A notação "$H[d = cte(1)]$" indica que o *heap* contém a dita célula de constante. O conjunto completo de leis de transição é apresentado em [Mus90].

5.2 Geração de Código

A compilação dos programas na representação intermediária para o código executável por GM-C é feita usando quatro esquemas de compilação:

Esquema $\mathcal{E}$: Este esquema de compilação é o responsável pelo início da tradução, gerando o código encarregado de avaliar uma expressão. É o primeiro esquema a ser invocado e se o programa a ser traduzido é representado por e, a sua compilação gerará o código $\mathcal{E}[e]$; print

Esquema $\mathcal{T}$: Constroi uma célula para conter a representação do termo, empilhando o seu endereço na pila T.

Esquema $\mathcal{B}$: Gera os programas correspondentes a expressões aritméticas e lógicas.

Esquema $\mathcal{G}$: É chamado pelos outros esquemas para preencher as células criadas durante a avaliação.

O conjunto completo de regras de tradução pode ser encontrado em [Mus90].

5.3 Um Exemplo de Compilação

Vejamos a seguir um exemplo de tradução de uma expressão simples para o código de GM-C. Usando a função `soma` definida por:

```
DEF soma;
  soma x y = x + y;
END soma;
```

A expressão **soma 3 5** pode ser traduzida para o λ-Calculus estendido como $(\lambda xy.(x + y))$ **3 5** e para Multi-Combinadores Categóricos como $L^1(1+0)$ **3 5**

As duas expressões são semelhantes. A segunda foi obtida da primeira substituindo cada seqüência de n símbolos λ pelo combinador L^{n-1}. As variáveis ligadas são substituídas por números que representam a distância entre o sub-termo onde elas aparecem, e o elemento que as liga. As constantes são representadas em negrito. O código gerado para esta expressão é:

```
MKEcell(2);          (1)
MKEcte(3)0;          (2)
MKEcte(5)1;          (3)
get(1);              (4)
get(0);              (5)
BBADDT               (6)
```

A instrução na linha 1 cria uma célula de temanho 2 na pilha de ambientes de avaliação (de onde são consultados os valores correspondentes às variáveis ligadas). As instruções das linhas 2 e 3 preenchem os campos do ambiente correspondentes às variáveis com as constantes 3 e 5. As instruções das linhas 4 e 5 consultam os valores correspondentes às variáveis, empilhando-os no topo da pilha B de valores básicos. A instrução BBADDT soma os dois valores que aparecem no topo e sub-topo da pilha B, e deposita o resultado da soma no topo da pilha T, sendo uma das operações aritméticas e lógicas básicas de GM-C.

5.4 A Implementação da Máquina de Redução

A implementação de GM-C é feita através de um programa na linguagem C padrão. O mapeamento entre as partes componentes da definição de GM-C e o programa que a representa é dado a seguir:

C: É o programa gerado pela implementação dos esquemas mencionados na seção anterior.

B: É implementada como uma pilha de inteiros.

T: É implementada como uma pilha de apontadores a células do *heap*.

H: É uma memória dividida em duas metades usadas de forma alternada (ver abaixo).

O: É a saída padrão.

E: É a mesma pilha de apontadores que T.

O processo de avaliação de uma expressão é governado pela rotina de impressão, que chama a construção e avaliação dos grafos.

A coleta de lixo é feita mediante o uso de um *copying garbage collector* [FY69]: a memória é dividida em duas metades, que são usadas alternadamente. Quando é necessário realizar uma coleta de lixo, a parte acessível do grafo é copiada na área não corrente, compactando a representação, e modificando as referências correspondentes nas pilhas **T** e **E**. O *heap* no qual se copiou o grafo passa a ser o *heap* em uso.

Este algoritmo de coleta de lixo tem a vantagem de visitar somente as células do grafo que são acessíveis (as que não são lixo), e seu maior inconveniente está em que somente é possível acessar metade da memória prevista estaticamente. Isto não constitui uma desvantagem em computadores com memória virtual.

5.5 Otimizações

Em [Mus90] são apresentadas otimizações a GM-C. Estas otimizações trazem ganhos significativos no desempenho dos programas, tanto em tempo quanto em espaço consumidos.

Como exemplo, salientamos que a última otimização proposta para GM-C obteve um tempo medido de 28.06% do tempo da versão original na média dos casos documentados em [Mus90]. O tempo medido na última otimização para o melhor dos casos foi de apenas 8.40% do tempo medido para o mesmo programa de teste na versão original de GM-C.

6 Desempenho

A tabela a seguir compara tempos de cpu (em segundos) para uma arquitetura Sun SPARCstation 1, apresentando resultados para seis programas de teste e três implementações diferentes.

Prog	*f30*	*rev*	*crivo*	*insord*	*simlog*	*tw3lista*
GM-C	21.7	1.4	1.4	24.7	0.5	1.6
Miranda	540.9	1.4	14.0	174.9	2.6	39.3
LML	18.6	0.2	0.6	11.1	0.5	1.1

Os programas de teste usados são (cada um corresponde a uma coluna da tabela):

f30: É a função de Fibonacci, avaliada em 30, usando algoritmo exponencial.

rev3lista: É um programa totalmente dedicado à manipulação de listas. Corresponde à função

$$reverse\ reverse\ reverse\ [1 \ldots 300]$$

crivo: Corresponde à geração da lista dos números primos menores que 1000, usando o algorítmo do "Crivo de Eratóstenes".

simlog: Corresponde à geração aleatória de 100 valores booleanos.

tw3lista: Definindo as funções *twice* $f\ x = f\ (f\ x)$ e *succ* $n = v + 1$, este programa calcula "*twice twice twice succ n*" para cada um dos elementos de uma lista de 600 números inteiros.

insord: Corresponde ao tempo consumido para ordenar, por inserção, uma lista de 1000 elementos gerados aleatoriamente.

A linha *GM-C* da tabela corresponde à implementação mais eficiente de GM-C. A linha *Miranda* apresenta os dados de tempo de cpu tomados para a versão 2-014 de Miranda [Tur85], enquanto que a linha *LML* da tabela corresponde à implementação de Lazy ML versão 0.95 de Gotemburgo tal como apresentada em [Joh87]. A implementação de Miranda está baseada nos combinadores de Turner [Tur79] e a implementação de Lazy ML é baseada na máquina G, sendo uma das implementações de linguagens funcionais mais eficientes de que se dispõe.

7 Conclusões e Trabalhos Futuros

Este artigo descreveu o trabalho realizado na implementação da parte funcional e do sistema de tipos de A. Os resultados conseguidos são de qualidade comparável às melhores implementações disponíveis internacionalmente, tanto do ponto de vista do *front* como do *back-end*.

Apesar disso, há margem para muitas otimizações e melhoramentos no trabalho já realizado e não se terminou a implementação dos processos na linguagem. Este tema não foi tratado neste artigo, sendo ortogonal ao trabalho aqui descrito.

A implementação da parte seqüencial da linguagem é a tese de mestrado dos dois últimos autores e estará disponível para distribuição em 1992. A implementação de processos está em andamento e deve estar pronta em 1993.

Referências

[AJ88] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, draft edition, January 1988.

[Alb89] M. C. Albuquerque. A semântica denotacional de A. Tese de Mestrado, Departamento de Informática, Universidade Federal de Pernambuco, Agosto 1989.

[Aug85] L. Augustsson. Compiling pattern matching. Em *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 368 – 381, Nancy, September 1985. LNCS 201, Revised in [Aug87].

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Chalmers University of Technology, S–412 96 Göteborg, November 1987.

[BM91] P. H. M. Borba and S. R. L. Meira. From model based specifications to functional prototypes. Em *IEEE TENCON'91 Session on Rapid Prototyping with Functional Programming Languages*, New Delhi, India, August 1991.

[BSM90] P. H. M. Borba, A. L. M. Santos, and S. R. L. Meira. A semântica de ações de A. Relatório técnico, Departamento de Informática, Universidade Federal de Pernambuco, 1990.

[FY69] R. Fernichel and J. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *Comm. of the ACM*, 12(11):611–612, November 1969.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. Em *Springer-Verlag LNCS*, volume 78, 1979.

[Hug84] J. Hughes. Why functional programming matters. PMG Report 16, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, November 1984.

[HW+88] P. Hudak, P. Wadler, et al. *Report on the Functional Programming Language Haskell.* December 1988. Draft Proposed Standard.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. Em Jean-Pierre Jouannaud, editor, *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, Nancy, September 1985. Springer-Verlag. LNCS 201, Revised in [Joh87].

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, S–412 96 Göteborg, February 1987.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Ler90] X. Leroy. Efficient data representation in polymorphic languages. Technical Report 1264, INRIA-Rocquencourt, 1990.

[Lin86] R. D. Lins. *On the Efficiency of Categorical Combinators in Applicative Languages.* PhD thesis, Computer Laboratory, University of Kent, 1986.

[MBS91] M. A. Musicante, P. H. M. Borba, and A. L. M. Santos. An action semantics for Actions. Technical report, Departamento de Informática, Universidade Federal de Pernambuco, 1991. Em preparo.

[Mei88] S. R. L. Meira. *Introdução à Programação Funcional.* VI Escola de Computação, IMECC - UNICAMP, 1988.

[Mei91] S. R. L. Meira. Signals handling in functional languages. Relatório técnico, Universidade Federal de Pernambuco, Departamento de Informática, 1991.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mou89] H. P. Moura. Tipos de dados em linguagens funcionais. Tese de Mestrado, Departamento de Informática, Universidade Federal de Pernambuco, Setembro 1989.

[Mus90] M. A. Musicante. Redução de grafos para multi-combinadores categóricos usando G-MC. Tese de Mestrado, Departamento de Informática, Universidade Federal de Pernambuco, Setembro 1990.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Tur79] D. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

[Tur84] D. Turner. Functional programs as executable specifications. Em *Phil. Trans. Royal Soc. of London (312), 1522*, pages 363–388. ACM, 1984.

[Tur85] D. Turner. Miranda: A non strict functional language with polymorphic types. *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architechture*, pages 1–16, 1985. LNCS 201.