Implementação de Polimorfismo de Inclusão em Linguagem Funcional Pura

Marcelo de Almeida Maia*
marcmaia@dcc.ufmg.br

Roberto da Silva Bigonha
bigonha@dcc.ufmg.br

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

* Também Universidade Federal de Ouro Preto

Resumo

As linguagens de programação evoluiram de um mundo, ou sem tipos, ou então monomórfico para alcançar a flexibilidade e confiabilidade dos sistemas de tipos polimórficos. As linguagens de programação funcional pura (ou não) popularizaram o polimorfismo paramétrico como forma de incorporar os benefícios dessa filosofia. Nesse trabalho é mostrado o polimorfismo de inclusão, bem conhecido nas linguagens imperativas, como uma alternativa para linguagens funcionais puras. Nosso objetivo é mostrar a relevância dessa idéia, assim como descrever o projeto de polimorfismo de inclusão numa linguagem funcional pura e sua respectiva implementação.

Abstract

Programming languages evolved from either an untyped or monomorphic world to achieve the flexibility and reliability of polymorphic type systems. The pure (or not) functional programming languages popularised the parametric polymorphism as a mean to incorporate the benefits of this filosophy.

In this work is showed the inclusion polymorphism, well-know on imperative languages, as an alternative to pure functional languages. Our goal is to show the significance of this idea, as well describe the inclusion polymorphism design in a pure functional language and its respective implementation.

1 Introdução

As linguagens de programação, em geral, evoluiram dos universos não tipados para sistemas de tipos monomórficos e posteriormente sistemas de tipos polimórficos. A teoria de tipos, concomitantemente, evoluiu no sentido de modelar matematicamente tais sistemas Cardelli e Wegner (1985) Danforth e Tomlinson (1988).

Em Cardelli e Wegner (1985) é feita uma classificação dos tipos de polimorfismos existentes. Cardelli dividiu o polimorfismo em 4 grupos: paramétrico, inclusão, sobrecarga e coerção.

As linguagens imperativas começaram a evoluir para um mundo polimórfico com a introdução do conceito de classes em Simula67 Dahl e et al. (1968). O grande salto em direção aos sistemas polimórficos ocorreu com a popularização da linguagem Smalltalk Goldberg e Robson (1983) e da programação orientada por objetos POO. A partir daí, as linguagens monomórficas existentes começaram a se estender no sentido de suportar o paradigma então emergente.

As linguagens funcionais (puras ou não) evoluiram a partir de LISP, que é uma linguagem que trabalha num universo não tipado, e desenvolveram seus sistemas de tipos baseadas principalmente no polimorfismo paramétrico. Importantes linguagens funcionais como ML Gordon e et al. (1978), Miranda[1] Turner (1986), Haskell Hudak e et al. (1992) têm seus sistemas de tipos baseados no estilo introduzido por Milner (1978), onde a inferência de tipos é decidível.

No sistema de tipos de ML, o polimorfismo paramétrico é conseguido com o uso de variáveis de tipo. Assim, é possível definir funções com comportamento uniforme que operam sobre um conjunto de tipos diferentes.

O sistema de tipos de Haskell foi estendido a partir do sistema de ML com o conceito de classes de tipos para controlar a sobrecarga na linguagem. O conceito de classes de tipos Wadler e Blott (1989) consiste em definir um conjunto (classe) de tipos para os quais podem ser definidas instâncias que tem um comportamento particular. Além dos métodos sobrecarregados que podem ser definidos nas instâncias, também podem é possível a declaração de métodos *default* que são usados quando não existe sua redeclaração na instância em questão. Uma vantagem com relação à POO tradicional é que os métodos são seguros com relação ao tipo, ou seja, aplicar um método a

[1] "Miranda" é marca registrada de Research Software, Ltd.

um valor que não está na classe desejada dá origem a um erro em tempo de compilação. Haskell também suporta a noção de inclusão de classes. É possível definir uma classe (subclasse) X que herda todas operações de uma classe (superclasse) Y e que tenha operações complementares definidas dentro de si. Isso possibilita a criação de contextos menores, além da possibilidade de métodos da superclasse X serem visíveis aos métodos da subclasse Y.

Um proposta alternativa ao polimorfismo com classes de tipos é apresentada. Essa proposta baseia-se no polimorfismo de inclusão ao invés do paramétrico. É apresentada a linguagem funcional pura não-estrita Script Bigonha (1994) e suas respectivas construções para suportar o polimorfismo. É apresentado também o modelo de implementação das construções polimórficas.

2 A Linguagem Script

Script é uma linguagem para prover notação legível de definições estruturadas de semântica denotacional de linguagens de programação.

Script provê características de uma linguagem de programação funcional, como funções de ordem mais alta, avaliação tardia, casamento de padrões, além de todos valores serem de primeira classe.

Script incorpora características de linguagens orientadas a objetos, como modularidade, equivalência estrutural de tipos, controle de visibilidade, encapsulamento, herança. O sistema de tipos de tipos é forte e baseado no polimorfismo de inclusão.

O projeto de Script teve como ponto de partida as notações SSL e DSL Bigonha (1981) Mosses (1975) Mosses (1978), no sentido que várias características dessas notações foram incorporadas, tais como, notação para especificação de gramática e sintaxe abstrata, tuplas, listas, nodos de árvores de derivação, padrões, notação LET e DEF. Outras características como compreensão de listas vieram de linguagens funcionais como Miranda e Haskell.

3 Extensão de Tipos em Script

Domínios em Script são c.p.o's com elemento mínimo (*bottom*) Mosses (1989). Os domínios pré-definidos padrões de Script são: N, o domínio dos números inteiros, Q, o domínio das *strings*, T, o domínio dos valores booleanos, e ?, o domínio dos valores indefinidos.

Script permite a possibilidade de criação de domínios mais complexos com o uso de operadores de domínio. Esses operadores permitem definir a união de domínios, domínio de tuplas, domínio de listas, domínio de funções contínuas e domínio de nodos de árvores. A extensão de tipos em Script é baseada na extensão do domínio de tuplas.

3.1 O Domínio de Tuplas

A expressão de domínio ($a_1 : d_1$,..., $a_n : d_n$) representa o domínio das n-**tuplas** cujo i-ésimo componente está no domínio denotado por d_i, e pode ser selecionado pelo identificador de campo a_i, onde $1 \leq i \leq n$. Essa expressão corresponde ao produto cartesiano de domínios. A seleção de campos das tuplas é expressa através da notação de ponto $t.f$, onde t é uma expressão de tupla e f é um dos identificadores de campo.

Os domínios de tuplas são extensíveis no sentido em que um domínio de tupla pode ser definido como uma extensão de outro domínio de tupla.

A expressão d_1 EXT d_2 representa o domínio de tuplas estendido a partir do domínio d_1, ou seja, denota um domínio de tuplas cujos elementos são obtidos a partir da concatenação dos elementos da tuplas em d_1 seguidos dos elementos das tuplas em d_2. Os componentes da tupla estendida que tiveram origem a partir da tupla em d_1 têm os mesmos nomes de identificadores de campo da tupla base em d_1 e o nomes dos identificadores de campo das tuplas em d_1 e d_2 não podem ser repetidos.

Seja a declaração `A = B EXT c`, onde `B` é um nome de domínio e `c` denota domínios de tuplas. Temos que `A` é definido como uma *extensão direta* do domínio `B`, e `B` é uma *base direta* de `A`.

Assim, um domínio A é definido como uma *extensão* de um domínio B, se:

1. A e B são o mesmo identificador ou
2. A é uma extensão direta de uma extensão de B.
3. Os domínios dos campos de B são equivalentes aos domínios dos campos que estão encabeçando A, na mesma ordem. Os nomes dos campos são irrelevantes.

As regras de equivalência de domínios, baseadas em equivalência estrutural, podem ser encontradas em Bigonha (1994).

3.2 Construção de Tuplas

As tuplas podem ser construídas enumerando explicitamente seus componentes com a notação $(e_1, \ ..., \ e_n)$, onde as expressões e_i definem os valores dos componentes e $1 \leq i \leq n$.

Uma nova tupla pode ser criada através da redefinição de seus campos. Este tipo de tupla é chamada de tupla de atualização, e é descrita com a notação: $t\{v_1/f_1, ..., v_n/f_n\}$, onde t é uma expressão de tupla, v_i uma expressão qualquer, f_i são identificadores de campo, e $1 \leq i \leq n$. Essa operação cria uma nova tupla a qual contém os mesmos componentes da expressão de tupla t, exceto que os componentes identificados por f_i contém os valores v_i.

Existe o operador de extensão de tupla EXT, que quando aplicado a domínios, serve para prover uma relação de concatenação entre um par de domínios. É possível criar tuplas estendidas diretamente apenas listando todos seus componentes. O operador EXT, quando aplicado a um par de tuplas, efetua a concatenção das mesmas obtendo uma tupla estendida a partir do primeiro elemento do par.

3.3 Objetos Polimórficos

A partir do momento em que é definido um objeto pertencente ao domínio das tuplas, esse objeto é tratado automaticamente como um objeto polimórfico.

O polimorfismo ocorre pelo fato de um objeto x pertencente a um domínio de tuplas X poder assumir a forma de qualquer outro objeto y pertencente ao domínio Y desde que Y seja extensão de X.

A existência de objetos polimórficos reflete diretamente na possibilidade de criação de funções polimórficas. Qualquer função que tenha algum parâmetro formal no domínio das tuplas é polimórfica por natureza. Esse fato decorre da possibilidade de se passar como argumento da função qualquer objeto que esteja num domínio estendido a partir do domínio do parâmetro formal.

3.4 Funções Virtuais e Ligação Dinâmica

A existência de extensão de tipos requer, como contrapartida, a capacidade de a linguagem permitir a sobrecarga das funções.

A sobrecarga de funções aliada ao fato de que, onde é esperado um elemento do domínio A pode ser encontrado um elemento de qualquer domínio

que seja extensão de A, impõe a necessidade da definição de um mecanismo de ligação dinâmica de funções.

Para suportar esse modelo, Script permite a definição de funções virtuais equivalentemente às linguagens imperativas de OOP. Essas funções estão associadas com algum domínio de tuplas ou existe algum parâmetro formal que pertence ao domínio de tuplas. Usa-se a notação $D.f$ para denotar que um domínio de tuplas D está associado com uma função f. Toda aplicação dessa função f deve ser qualificada por um objeto cujo domínio é uma extensão de D.

Uma função associada a um domínio de tuplas pode ser redefinida em qualquer extensão do domínio associado. A função redefinida se torna disponível daquele ponto para baixo na hierarquia de domínios. Por definição, se uma função está associada um domínio de tuplas, então a função está ligada a todos descendentes desse domínio, exceto se for redefinida.

Dado o fato de que o qualificador (ou parâmetro) formal de uma função pertença ao domínio de tuplas, a função a ser ativada no momento da aplicação deve ser aquela associada ao objeto qualificador (ou argumento) corrente, que por sua vez pode pertencer a qualquer extensão do domínio do qualificador (parâmetro) formal, sendo que esta extensão só pode ser conhecida em tempo de execução.

Nesse sentido o modelo de implementação deve ser capaz de prover uma ligação eficiente em tempo de execução, da função chamada com seu respectivo código para minimizar o *overheading* de uma aplicação de função polimórfica.

3.5 Um Exemplo

A seguir é apresentado um exemplo que demonstra o uso de polimorfismo em conjunto com a ligação dinâmica de funções. O exemplo mostra a criação de uma pilha polimórfica e a aplicação da função de translação, sobrecarregada e virtual, aos elementos polimórficos que são retângulos ou círculos.

```
MODULE Stack
EXPORTS
    Stk, stk0, Stk-elem, stk-elem0, Stk.push, Stk.pop, Stk.top, Stk.empty

DOMAINS
    Stk = Stk-elem*
```

```
        Stk-elem = ()
    DEFINITIONS
        DEF stk0 = <>
        DEF stk-elem0 = ()
        DEF Stk.push(stk-elem) : Stk = stk-elem PRE THIS
        DEF Stk.pop : Stk = LET stk-elem PRE stk1 = THIS IN stk1
        DEF Stk.empty : T = (SIZE THIS) EQ 0
        DEF Stk.top : Stk-elem =
            THIS.empty -> ?, LET stk-elem PRE stk1 = THIS IN
stk-elem
    END Stack

    MODULE User
    IMPORTS
        Stack(Stk, stk0, Stk-elem, stk-elem0)
    DOMAINS
        Stk-elem-Rect = Stk-elem EXT (x1: N, y1: N, x2: N, y2: N)
        Stk-elem-Circ = Stk-elem EXT (x: N, y: N, rad: N)
    DEFINITIONS
        DEF Stk-elem-Rect.translate (x: N, y: N) : Stk-elem-Rect =
           (THIS.x1 PLUS x, THIS.y1 PLUS y, THIS.x2 PLUS x,
THIS.y2 PLUS y)
        DEF Stk-elem-Circ.translate (x: N, y: N) : Stk-elem-Circ =
          (THIS.x PLUS x, THIS.y PLUS y, THIS.rad)
        DEF Stk.translate (x: N, y: N) : Stk =
          THIS.empty -> stk0,
              LET thetop = THIS.top
              LET thepop = THIS.pop
              LET poptrans = thepop.translate(x,y)
              IN poptrans.push(thetop.translate(x,y))
        DEF test: Stk =
          LET stk-elem-Rect = stk-elem0 EXT (1, 2, 3, 4)
          LET stk-elem-Circ = stk-elem0 EXT (5, 6, 7)
          LET stk1 = stk0.push(stk-elem-Rect)
          LET stk2 = stk1.push(stk-elem-Circ)
          IN stk2.translate(1, 1)
    END User
```

4 O Modelo de Redução de Grafos

Script está sendo implementada com o uso de técnicas bem conhecidas para implementação de linguagens funcionais puras não estritas Maia (1994). Nessa seção vamos rever a técnica utilizada inicialmente na compilação de LML Augustsson e Johnsson (1992). A máquina-G é uma máquina virtual, específica para redução de grafos, que tem os seguintes componentes:

- S - uma pilha que controla o caminhamento no grafo G, representada como $n_0 : ... : n_k$, onde n_0 é o topo da pilha, n_i são referências para o grafo e $1 \leq i \leq n$.
- G - o grafo que corresponde à expressão corrente em avaliação, representado como $G[n =< L_1...L_n > n_1 n_2]$, onde n é um nodo qualquer do grafo, L_i, $1 \leq i \leq n$, são *labels* do nodo e n_1 e n_2 são campos de informação ou referências a outros nodos.
- C - o código que resta para ser executado, representado como $c_1 : ... : c_n$, onde c_i são instruções, $1 \leq i \leq n$. Esse código é especializado em instruções para redução do grafo G,
- D - o contexto, que é uma pilha de pares (S, C),
- O - a saída produzida pelo programa.

O código da máquina-G é especializado em atualizações da pilha e do grafo, e em controle da redução do grafo. As instruções principais são de :

- Empilhamento - PUSH, PUSHINT, PUSHCHAR, ...
- Desempilhamento - POP
- Atualização na pilha - SLIDE, UPDATE, ...
- Criação de nodos no *heap* - ALLOC, MKAP, ...
- Operação pré-definidas - AND, PLUS, ...
- Redução do grafo - EVAL, UNWIND, DISPATCH
- Impressão - PRINT

4.1 Execução na Máquina-G

O programa Script para ser executado na máquina-G precisa passar por uma série de transformações (eliminação de casamento de padrões, *lambda-lifting*) até atingir o ponto em que a linguagem subjacente seja composta por supercombinadores.

Para execução de uma expressão (programa) inicialmente é empilhado o supercombinador principal, depois é executada a instrução EVAL e por fim o resultado no topo da pilha é impresso.

Quando uma aplicação de supercombinador, $f\ n_1\ ...\ n_k$, está para ser reduzida é esperado que:

- O grafo tenha o nodo n, tal que, $G[n = AP\ (AP\ ...(AP\ f\ n_1)\ ...\ n_{k-1})\ n_k]$
- A pilha tenha a configuração $f : n_1 : ... : n_k : ....$

O corpo de um supercombinador contém código para:

- construir uma instância do corpo do supercombinador no grafo, $G[n = ...Corpo...]$
- atualizar a raiz da expressão que estava sendo reduzida com uma cópia da raiz da instância, $[r : n_1 : ... : n_k : v_k : ...] \Rightarrow ([n_1 : ... : n_k : v_k : ...]$ e $G[v_k = G\ n_0])$,
- remover os parâmetros da pilha, $[n_1 : ... : n_k : v_k : ...] \Rightarrow [v_k : ...]$
- iniciar a próxima redução.

Existem uma série de otimizações para o esquema acima Peyton Jones (1987). Tem-se que a máquina-G é um modelo flexível que será estendido na próxima seção para suportar a extensão de tipos e funções virtuais.

5 Redução de Grafos com Tipos Estendidos

O modelo anterior de redução de grafos deve ser estendido para suportar a estrutura de tuplas, bem como o operador de extensão e a ligação dinâmica de funções.

Pelo fato de todos objetos da tupla serem de primeira classe eles são representados como porções do grafo. A tupla em si tem tamanho pré-definido com relação ao número de elementos e assim o acesso aos seus elementos é feito com $O(1)$.

5.1 O Operador de Extensão

O operador de extensão de tuplas EXT é definido com um mecanismo de concatenação e cópia de referências dos elementos do par de tuplas envolvido. Formalmente tem-se a seguinte transição de estado para modelar a transformação que a operação causa no grafo e na pilha:

- Grafo: $G[n_1 = [TUPLA\ (v_1, ...v_n)\ ()], n2 = [TUPLA\ (p_1, ..., p_n)\ ()] \ \Rightarrow$

$$G[n_1,\ n_2,\ n_3 = [TUPLA\ v_1, ...v_n, ..., p_1, ..., p_n)()]$$

- Pilha: $n_1 : n_2 : resto \ \Rightarrow \ n_3 : resto$

A avaliação tardia é preservada nessa operação pois os elementos da nova tupla continuam com sua avaliação no mesmo ponto que antes, e o seus elementos estão compartilhados com as tuplas n_1 e n_2.

5.2 A Ligação Dinâmica

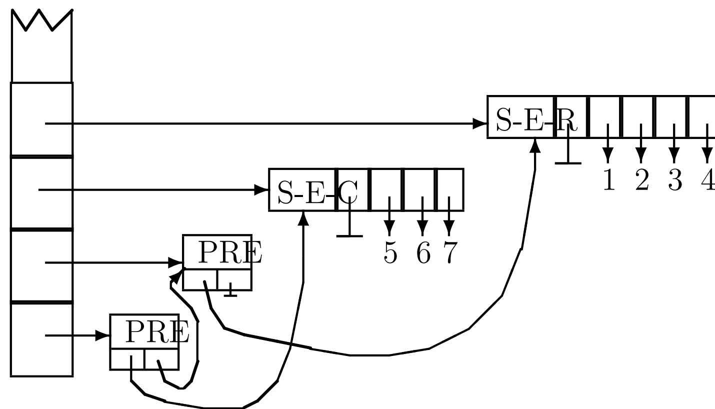
Na verdade o *label* $TUPLA$ descrito anteriormente deve ser implementado não como um valor denotando que o nodo seja uma tupla, mas sim como um valor em um intervalo de números inteiros, para ser possível a identificação do domínio de tuplas ao qual o objeto atual pertence.

Além disso, em tempo de compilação é possível criar uma tabela para cada domínio definido. A tabela é composta por pares (FS, IF), onde FS é uma referência da função sobrecarregada e IF é uma referência da instância da função para o domínio subjacente.

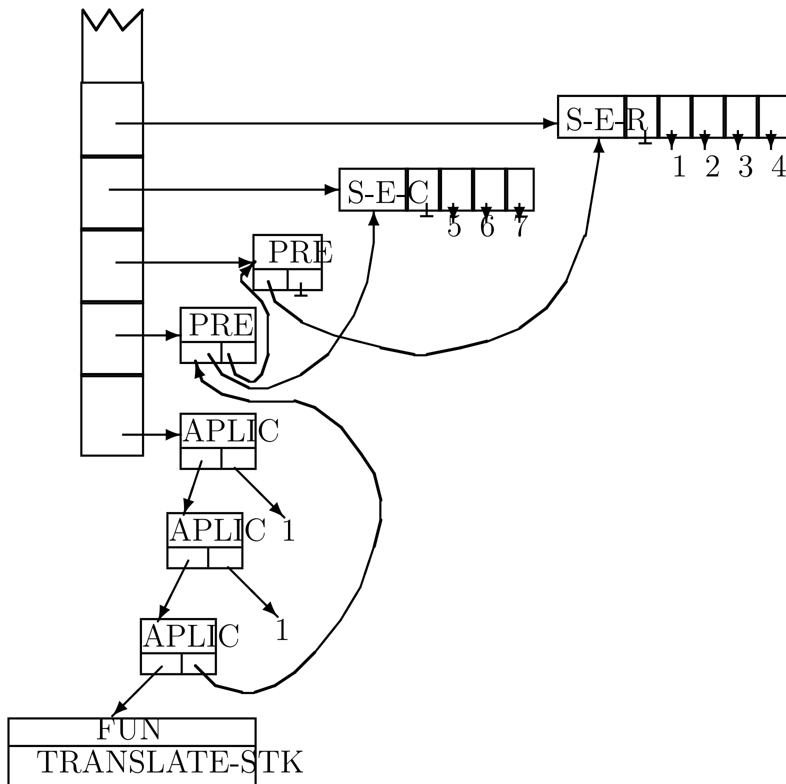
Esses ingredientes permitem resolver o problema de descobrir em tempo de execução qual instância de função deve ser utilizada numa determinada aplicação de função virtual. Quando uma aplicação necessita ser avaliada, no momento do desvio para o código da função acontece, na verdade, um desvio indireto, com acesso $O(1)$ à tabela do domínio de funções $\times$ instâncias, para o endereço da instância da função.

5.3 Um Exemplo

Executando a função *test* do módulo $User$ na Seção 3.5 tem-se após a execução das definições dos LET's a seguinte configuração:

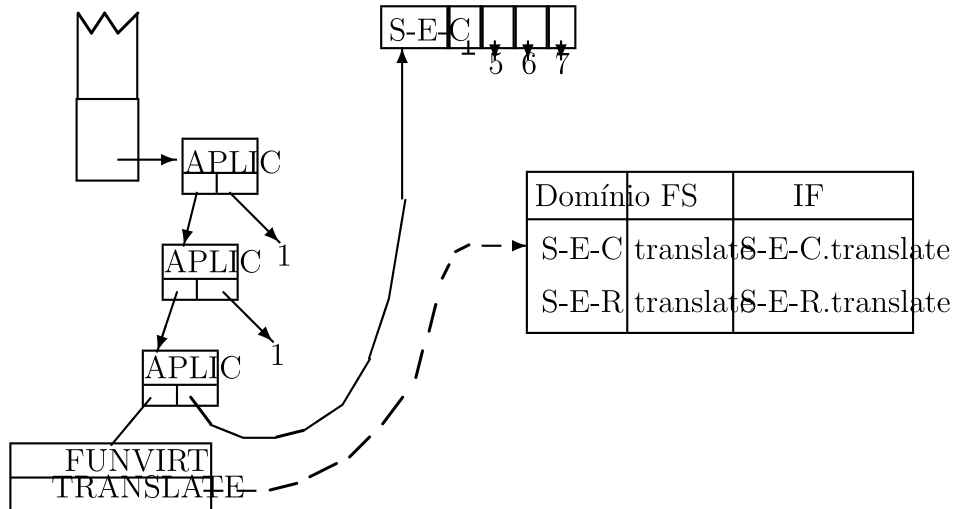


Quando a aplicaçação da função $stk.translate(1)$ está construída no grafo para ser reduzida tem-se:



É importante observar que a sobrecarga da função *Stk.translate* foi resolvida em tempo de compilação por não se tratar de uma função virtual.

No momento da translação do primeiro elemento da pilha tem-se a seguinte configuração:



Quando o desvio para a função estiver para ocorrer detecta-se que se trata de função virtual *translate*. Com base na pesquisa à tabela acima, o desvio é feito para a instância $Stk - elem - Circ.translate$.

Dessa forma, fica resolvida toda a sobrecarga na linguagem e a função retorna a mesma pilha com seu retângulo e seu círculo transladados de uma unidade.

6 Avaliação dos Resultados

Através do mecanismo de extensão de tuplas proposto, pode-se avaliar que o polimorfismo de inclusão se encaixou de maneira ortogonal no projeto de uma linguagem de programação funcional pura, mesmo com a ausência do conceito de objetos com estado atualizável.

No exemplo mostrado, houve a reutilização de código esperada, mostrando a eficácia do modelo.

Além disso, a modelo de redução de grafos não precisou sofrer nenhuma mudança mais séria para suportar o modelo de implementação subjacente, méritos devidos à flexibilidade da máquina-G.

O *overheading* adicionado pelo uso de funções virtuais sugere que o desempenho geral da máquina de redução não deva ser drasticamente afetado. O

slow-down provocado pelo uso de funções virtuais deve inclusive ser menor em linguagens funcionais do que em linguagens imperativas, dado que programas em uma linguagem funcional, em geral, tendem a executar mais lentamente que em uma linguagem imperativa e o *overheading* embutido nas funções virtuais tende a ser praticamente o mesmo em linguagens funcionais e imperativas, já as técnicas aplicadas são semelhantes.

7 Conclusões e Trabalhos Futuros

O presente trabalho mostrou a adequação da utilização do polimorfismo de inclusão em linguagens funcionais puras. Espera-se que o resultado dessa experiência possa ser utilizado para aumentar o poder de expressão das linguagens funcionais de uso geral. Nesse sentido uma linha que pode ser explorada é a unificação dos polimorfismos paramétricos e de inclusão de maneira mais incisiva em Haskell.

No estado atual da implementação de Script existe um caminho de validação a ser percorrido. É necessário a finalização da implementação de Script para uma análise qualitativa mais elaborada da eficácia do uso de polimorfismo de inclusão em problemas resolvidos no paradigma funcional puro, bem como para um possível refinamento do método.

Além disso, são necessárias medidas de performance para validar as hipóteses da seção anterior com relação ao *overhead* embutido na implementação de funções virtuais.

Outra linha que pode ser explorada é o aproveitamento da notação de funções ligadas a domínio (que sugere a relação de método com função, e de tupla com objeto) no sentido de desenvolver uma máquina de redução de grafos orientada a objetos que possibilitasse uma execução distribuída de programas funcionaisKingdon *et al.* (1991)Cavalheiro e Navaux (1993).

References

Lennart Augustsson e Thomas Johnsson. Lazy ML user's manual. Technical report, Chalmers University, Goteborg, Sweden, 1992. Department of Computer Science.

R.S. Bigonha. *A Denotational Semantics Implementation System.* Tese de

Doutorado, Computer Science Department, University of California, Los Angeles, 1981.

R.S. Bigonha. Script - An object oriented language for denotational semantics. Revised User's Manual and Reference RT 015/94, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, September 1994.

Luca Cardelli e Peter Wegner. On undertanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

G.G.H. Cavalheiro e P.O.A. Navaux. DPC++ uma linguagem para processamento distribuído. In *V SBAC-PAD*, Florianópolis, Outubro 1993.

O-J. Dahl e et al. *The SIMULA 67 Common Base Language*. Norwegian Computer Center, Oslo, 1968.

S. Danforth e C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

A. Goldberg e D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

M. Gordon e et al. Edinburgh LCF. *Lecture Notes in Computer Science, Springer-Verlag*, 78, 1978.

P. Hudak e et al. Report on the programming language Haskell. A non-strict, purely functional language version 1.2. *ACM SIGPLAN NOTICES*, 27(5), maio 1992.

H Kingdon, D. Lester, e G.L. Burn. The HDG-machine: a highly distributed graph reducer for a transputer network. *The Computer Journal*, 34(4):290–302, 1991.

M.A. Maia. Implementação eficiente de uma linguagem para definição de semântica. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Dezembro 1994.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

P. D. Mosses. *Mathematical Semantics and Compiler Generation.* Tese de Doutorado, Oxford University Computing Lab, 1975. Programming Research Group.

P. D. Mosses. SIS - A compiler-generator system using denotational semantics. Technical report, University of Aarhus, Denmark, 1978.

Peter D. Mosses. Denotational semantics. In *Lectures Notes of the State of the Art Seminar on Formal Description of Programming Concepts – IFIP TC2 WG 2.2*, Rio de Janeiro, Brazil, April 1989.

S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International Series in Computer Science, Englewood Cliffs, 1987.

David Turner. An overview of Miranda. *ACM SIGPLAN NOTICES*, 21(12):158–166, 1986.

P. Wadler e S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proceedings of 16th Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.