Implementando uma Linguagem Funcional Pura com uma Máquina-G Estendida

Marcelo de Almeida Maia [1]
marcmaia@dcc.ufmg.br

Roberto da Silva Bigonha
bigonha@dcc.ufmg.br

Universidade Federal de Ouro Preto
Depto. de Computação

Universidade Federal de Minas Gerais
Depto. de Ciência da Computação

Palavras Chaves: linguagem funcional pura, máquina-G, funções especiais funções virtuais, polimorfismo de inclusão

Resumo

Esse trabalho enfoca a implementação de algumas extensões à máquina-G original de forma a tornar possível linguagens funcionais puras incorporarem polimorfismo de inclusão e funções especiais, que podem ser compiladas como funções na linguagem `C`.

Também é mostrado como uma linguagem funcional pura pode incorporar o conceito de polimorfismo de inclusão, como nas linguagens tradicionais orientadas por objetos.

[1] Doutorando em Ciência da Computação pela Universidade Federal de Minas Gerais

Abstract

This work focuses on the implementation of some extensions to the original G-machine, in order to make possible pure functional languages to incorporate inclusion polymorphism and special functions, which can be compiled into `C` language functions.

It is also shown how a pure functional language can incorporate the inclusion polymorphism concept, as in traditional object-oriented languages.

1 Introdução

A implementação eficiente de linguagens funcionais puras vem sendo um importante objeto de estudo nos últimos anos. O desenvolvimento de um compilador para LML [1] baseado na na máquina-G [6] propiciou uma proliferação da implementação de linguagens funcionais puras. Apareceram várias máquinas de redução de grafos, como Spineless G-Machine [4], Spineless Tagless G-Machine [13], dentre outras. Em [7] é proposta uma máquina para multi-combinadores categóricos cujo objetivo é transferir ao máximo o controle do fluxo execução para a linguagem `C`. Essa transferência é suportada pelo uso de funções especiais, que são funções estritas em todos seus argumentos e cujos argumentos e resultado retornam valor de tipo básico em `C`. Os resultados de performance obtidos foram bastante favoráveis ao uso de funções especiais.

Com o advento dessa tecnologia de compilação para as linguagens funcionais puras, foi percebida a viabilidade de se implementar Script [3], uma linguagem para um sistema de definição de semântica denotacional definido em [2]. Uma linguagem para esse propósito tem características muito próximas às linguagens funcionais puras de uso geral e as considerações de projeto são praticamente as mesmas, acrescentando-se algumas estruturas adicionais para definição de gramáticas.

Assim, a bem estudada máquina-G, o uso de funções especiais e a necessidade de uma programação cada vez mais modular e reutilizável influenciaram a definição e o projeto de implementação da linguagem Script.

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 pura, como funções de ordem mais alta, avaliação tardia, casamento de padrões, transparência referencial, além de que todos valores são de primeira classe.

Script ainda 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 [2] [10], 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.

2.1 Funções Especiais

As funções especiais de Script são funções estritas em todos os seus argumentos, além dos argumentos e o resultado da função serem de tipos básicos de `C`. Em Script essas funções devem ser especificadas com a sua assinatura, bem como uma anotação indicando que se tratam de funções especiais.

2.2 Extensão de Domínios

Domínios em Script são ordens parciais completas (c.p.o.[1]) com elemento mínimo (*bottom*) [11]. 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 domínios em Script é baseada na extensão do domínio de tuplas.

2.2.1 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 $1 \leq i \leq n$.

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

[1] do inglês, *complete partial orders*

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.

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 [3].

2.2.2 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 $1 \leq i \leq n$.

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.

2.2.3 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.

2.2.4 Funções Virtuais e Ligação Dinâmica

A existência de extensão de domínios 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 POO. 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. 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 A Máquina-G Original

Script está sendo implementada com o uso de técnicas bem conhecidas para implementação de linguagens funcionais puras não estritas [8][12]. Essa seção revê a técnica utilizada inicialmente na compilação de LML. 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 para atualização na pilha, criação de nodos no *heap*, operação pré-definidas, redução do grafo, impressão.

3.1 Execução na Máquina-G

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, atualizar a raiz da expressão que estava sendo reduzida com uma cópia da raiz da instância, remover os parâmetros da pilha, e iniciar a próxima redução.

Existe uma série de otimizações para o esquema acima [12].

4 Adaptações na Máquina-G

A máquina-G original como descrita na seção anterior não suporta o uso de funções especiais e funções virtuais. Apresenta-se a seguir a introdução de mecanismos na máquina de modo a suportar as características desejadas.

4.1 Esquemas de Compilação para Funções Especiais

A execução de um programa Script (supercombinadores e funções especiais) ocorre em duas máquinas hipotéticas distintas. Uma é definida pela máquina-G que executa as definições de supercombinadores. A outra é implementada pelo compilador C e executa definições de funções especiais.

A interação entre essas máquinas é feita da seguinte forma:

- A máquina principal é a máquina-G.
- Quando ocorre alguma chamada de função, é verificado se a função chamada é especial. Se não for especial continua-se executando na máquina-G. Se a função for especial executa-se uma chamada de função em linguagem C e empilha-se um apontador para o grafo contendo o resultado, volta-se para a máquina-G executando a instrução $UNWIND$, que é responsável por iniciar a próxima redução.
- Quando a execução está ocorrendo em linguagem C, e é feita alguma chamada de um supercombinador, deve ser construído o grafo para a chamada, com o ambiente da função C que estava sendo executada. O controle retorna novamente para a máquina-G para ela processar a redução local do corpo daquele supercombinador. Finalmente, o controle retorna para a função C, com o resultado acessado através de uma referência à variável do tipo grafo que contém o resultado da chamada do supercombinador. O checador de tipos deve bloquear uma chamada

de supercombinador dentro de uma função especial, caso esse não retorne um valor de tipo básico de `C`.

Para efetuar a compilação da definição de um supercombinador ou função especial, defini-se uma função **F** que recebe a definição como argumento e retorna uma definição de função `C`, contendo, ou: a definição do supercombinador compilada em código-G, ou a definição da função especial traduzida para a sintaxe de `C`.

Nesse trabalho serão mostrados apenas os esquema que compilam as funções especiais.

4.1.1 Notação

Seja a seguinte notação para as especificações:

- $[\![\ ]\!]$representam parâmetros do tipo *construções sintáticas*.
- `" "` representam valores do tipo cadeia de caracteres.
- `++` representa o operador de concatenação de cadeia de caracteres.
- $\rightarrow$ representa o operador condicional `if-then-else`. $(e \rightarrow et, ef)$
- ... representa iteração, que fica subentendida no contexto.

4.1.2 O Esquema F de Compilação

O esquema **F** é responsável pela compilação de cada definição. Ele recebe como parâmetro uma definição e tem a seguinte forma:

F $[\![t_f\ f\ t_{x1}\ x_1\ ...\ t_{xn}\ x_n\ =\ E\ ]\!]=$
$t_f\ f$ `++ "(" ++` $t_{x1}\ x_1$ `++ "," ++ ... ++ "," ++` $t_{xn}\ x_n$ `++ ")" ++`
`"{" ++` **S** $[\![$E $]\!] n\ t_f$ `++ "}"`

onde f é uma função especial,
t_f é o tipo que a função retorna e
t_{xi} $(1 \leq$ i $\leq$ n) são os tipos dos parâmetros e
x_i $(1 \leq$ i $\leq$ n) são os parâmetros.

4.1.3 O Esquema S de Compilação

O esquema **S** compila o corpo de funções especiais. Em princípio esse esquema representa apenas uma transformação sintática de uma definição es-

crita na linguagem de supercombinadores para a sintaxe da linguagem C, exceto no caso da aplicação de função como será mostrado. Os parâmetros para essa função são:

- a expressão que está sendo compilada;
- d, que é o número de parâmetros da função;
- ft, que é o tipo de valor retornado pela função.

Tem-se a seguinte definição:

- Inteiro:
  S ⟦n ⟧d ft = `"return n;"`
- Identificador:
  S ⟦id ⟧d ft = `"return id;"`
- Aplicação de função:
  S ⟦f x1 ... xn ⟧d ft =
  is_special(f) →
  `"return f(x1, ..., xn);"` ,
  CS ⟦f x1 ... xn ⟧d ft 0 `++`
  ft == NUM →
  `"return s.content[s.top--].ptr->descend.int_value;"`
  ,
  ft == STR →
  `"return s.content[s.top--].ptr->descend.str_value;"`

  Esse caso merece um comentário especial porque ele permite um chaveamento de máquinas. Isto é, caso a chamada seja para uma função especial, o esquema faz simplesmente uma tradução sintática, porém, caso a chamada seja para um supercombinador a execução se processará na máquina-G. Nesse último caso é chamado o esquema **CS** que constrói o grafo referente à aplicação, reduz essa aplicação e depois retorna um apontador para esse grafo como resultado da função.
- Expressão condicional
  S ⟦if e1 e2 e3 ⟧d ft =
  `"return"` `++` **S2** ⟦e1 ⟧d ft `++` `"?"` `++`
  S2 ⟦e2 ⟧d ft `++` `":"` `++`
  S2 ⟦e3 ⟧d ft `++` `";"`

- Operador unário *not*
  S ⟦not e ⟧=`"return (!" ++` **S2** ⟦e ⟧d ft `");"`
- Operadores binários
  S ⟦e1 op e2 ⟧=
  `"return" ++` **S2** ⟦e1 ⟧d ft `++` opbin2c op `++` **S2** ⟦e2 ⟧d ft `++ ");"`

4.1.4 O Esquema S2 de Compilação

O esquema **S2** serve para compilar as partes internas de um corpo de função. Esse esquema é idêntico ao esquema **S** exceto pelo fato de que o corpo de programa produzido não é uma chamada de retorno para o valor produzido e sim o valor propriamente dito.

Os caso possíveis para o esquema são:

- Inteiro:
  S2 ⟦n ⟧d ft = `"n"`
- Identificador:
  S2 ⟦id ⟧d ft = `"id"`
- Aplicação de função:
  S2 ⟦f x1 ... xn ⟧d ft =
  is_special(f) →
  `"f(x1, ..., xn);"` ,
  CS ⟦f x1 ... xn ⟧d ft 0 `++`
  ft == NUM →
  `"s.content[s.top--].ptr->descend.int_value;"`
  ,
  ft == STR →
  `"s.content[s.top--].ptr->descend.str_value;"`
- Expressão condicional:
  S2 ⟦if e1 e2 e3 ⟧d ft =
  S2 ⟦e1 ⟧d ft `++ "?" ++`
  S2 ⟦e2 ⟧d ft `++ ":" ++`
  S2 ⟦e3 ⟧d ft
- Operador unário *not* `++`
  S2 ⟦not e ⟧= `"!" ++` **S2** ⟦e ⟧d ft
- Operadores binários
  S2 ⟦e1 op e2 ⟧=**S2** ⟦e1 ⟧d ft `++` opbin2c op `++` **S2** ⟦e2 ⟧d ft

4.1.5 O Esquema CS de Compilação

O esquema **CS** tem o papel de construir o grafo de aplicações de supercombinadores quando essas são chamadas de dentro de uma função especial. Não é possível usar o esquema **C** convencional, pois o ambiente o qual deve ser usado na compilação da aplicação é o ambiente da função especial. Como as instruções da máquina-G que executarão a avaliação da aplicação do supercombinador em questão, estarão no escopo da função especial, essas instruções podem fazer referência aos identificadores da função especial (que deveriam estar no ambiente p caso estivessem sendo compiladas com o esquema **C** tradicional). Isto é, o ambiente para a chamada desse supercombinador é definido pela declaração de parâmetros no cabeçalho da função especial que envolve a aplicação. O parâmetro *np* é utilizado para contar o número de parâmetros de uma aplicação para criação do grafo correspondente.

Os casos possíveis são:

- Inteiro.

  CS ⟦n ⟧d ft np = `"pushint(" ++` n `++ ");"`
- Identificador. Esse o principal ponto do esquema **CS**. Os seguintes casos podem ocorrer:
  - O identificador é um supercombinador. Nesse caso é necessário compilar a aplicação tal como o esquema **R**, exceto que não é necessário atualizar a raiz e nem desempilhar os parâmetros, pois não existe contexto na pilha.
  - O identificador é uma variável do tipo inteiro ou *string*. Nesse caso é empilhado um apontador para um nodo com o conteúdo da variável.

```
CS ⟦id ⟧d ft np =
      infuntabg(id) →
            "pushglobal(id, " ++ np ");" ++
            "mkapn(" ++ np ++ ");" ++
            "unwind();"
      ,
            amb_type(id) == NUM →
                  "pushint(id);" ,
```

"pushchar(id);"

- Aplicação.
  CS ⟦e1 e2 ⟧d ft np = **CS** ⟦e2 ⟧d ft 0 **++ CS** ⟦e1 ⟧d+1 ft np+1
- Expressão condicional.
  CS ⟦if e1 e2 e3 ⟧d ft np =**S2** ⟦if e1 e2 e3 ⟧d ft
- Operadores unários.
  CS ⟦opun e ⟧d ft np = **S2** ⟦opun e ⟧d ft
- Operadores binários.
  CS ⟦opbin e1 e2 ⟧d ft np = **S2** ⟦opbin e1 e2 ⟧d ft

4.2 Redução de Grafos com Domínios Estendidos

O modelo original 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 $O(1)$.

4.2.1 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.

4.2.2 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 para a tabela do domínio de funções $\times$ instâncias, para o endereço da instância da função.

5 Conclusões

Como contribuição desse trabalho pode-se ressaltar a a definição de extensões à máquina-G para suportar funções especiais e funções virtuais, e a definição de um mecanismo de polimorfismo de inclusão em uma linguagem funcional pura.

Pode ser observado nesse projeto a flexibilidade que a máquina-G proporciona, quando é necessário incorporar novas características a ela. Poucas alterações foram necessárias nas instruções do código-G, bem como nos esquemas de compilação.

Deve-se ressaltar que o uso de funções especiais por enquanto requer a participação do programador no sentido de identificação, e se possível na intensificação, do uso de funções especiais.

Uma abordagem que pode ser seguida a partir das idéias mostradas nessa artigo é fazer com que programas escritos em C possam incorporar bibliotecas escritas em linguagens funcionais.

Com relação à implementação das funções virtuais, o modelo pode absorver técnicas já utilizadas em linguagens imperativas, e por isso a sobrecarga do uso de funções virtuais não acarretará grandes problemas na

eficiência do sistema.

O uso de polimorfismo de inclusão em linguagem funcional pura é um recurso que aumenta a reutilização de código[9] e por isso deve ser considerado no projeto dessas linguagens. Cabe ressaltar que linguagens funcionais puras consagraram o polimorfismo paramétrico, baseado no sistema de tipos de Hindley-Milner, sistema esse que apresenta dificuldades em manter a decidibilidade da inferência de tipos quando estendido para suportar polimorfismo de inclusão [5].

Além disso, outro ponto a ressaltar é que a propriedade de transparência referencial sofre danos com o uso de polimorfismo de inclusão. Nesse sentido, torna-se necessário definir uma maneira mais flexível para provar propriedades de funções definidas com polimorfismo de inclusão.

As implementações foram efetuadas em caráter experimental, apenas para validar os esquemas de compilação.

References

[1] L. Augustsson. A compiler for Lazy ML. In *ACM Symposium on Lisp an Functional Programming*, pages 218–227, Austin, August 1984.

[2] R.S. Bigonha. *A Denotational Semantics Implementation System*. PhD thesis, Computer Science Department, University of California, Los Angeles, 1981.

[3] 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.

[4] G.L. Burns, Peyton Jones, S.L., and S.L. Robson. The spineless g-machine. In *ACM Conference on Lisp an Functional Programming*, pages 244–258, Snowbird, USA, 1988.

[5] Thomas Hallgren. Subtypes in polymorphic functional languages. 1993. Licentiate Thesis.

[6] T. Johnsson. Efficient compilation of lazy evaluation. In *ACM Conference on Compiler Construction*, pages 58–69, Montreal, June 1984.

[7] R.D. Lins and B.O Lira. ΓCMC: A novel way of compiling functional languages. In *XII Congresso da Sociedade Brasileira de Computação*, pages 282–295, Setembro 1992.

[8] M.A. Maia. Implementação eficiente de uma linguagem para definição de semântica. Master's thesis, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Dezembro 1994.

[9] M.A. Maia and R.S. Bigonha. Implementação de Polimorfismo de Inclusão em Linguagens Funcionais Puras. In *XV Congresso da Sociedade Brasileira de Computação*, pages 863–876, Agosto 1995.

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

[11] 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.

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

[13] S.L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *Conference on Functional Programming and Computer Architecture*, pages 184–201, Snowbird, USA, September 1989.