Compilador da Linguagem Funcional Orientada por Objetos $\mathcal{SCRIPT}$ para Haskell

Mariza A. S. Bigonha, Roberto S. Bigonha,
Wendell F. Taveira, Fabíola F. Oliveira

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte - MG - Brazil

Resumo

Este artigo mostra a compilação de programas escritos na linguagem $\mathcal{SCRIPT}$ para Haskell, especificamente, a implementação do *back-end*. $\mathcal{SCRIPT}$ é uma linguagem puramente funcional orientada por objetos, que visa prover uma notação adequada e estruturada de tal forma a permitir que as descrições de semântica denotacional possam ser efetivamente executadas e depuradas. Considerando o fato de que existem eficientes compiladores de Haskell disponíveis no mercado, o desafio de desenvolver um compilador eficiente para $\mathcal{SCRIPT}$ motivou a produção de um compilador de $\mathcal{SCRIPT}$ para Haskell. O trabalho apresentado faz parte de um projeto que é a compilação de programas em $\mathcal{SCRIPT}$ para código executável desenvolvido pela equipe de pesquisadores do Laboratório de Linguagens de Programação da UFMG.

Palavras-Chaves: Haskell, linguagem orientada por objeto, linguagem funcional, Script, Compilador.

1 Introdução

Um programa escrito no paradigma funcional é um conjunto de expressões, funções e declarações que podem chamar-se umas às outras. Como o mapeamento dos valores de entrada para os valores de saída é feito mais diretamente, por meio da construção e aplicação de funções, o paradigma funcional permite projetar programas muito simples, concisos, flexíveis e poderosos [10, 11].

É importante para qualquer linguagem de programação possuir um formalismo para definição formal de sua semântica, tanto sob o ponto de vista do projetista de processadores de linguagens, quanto sob o ponto de vista dos programadores que precisam ter um completo entendimento das construções da linguagem [6]. Sob este aspecto, linguagens funcionais se sobressaem em relação às linguagens de outros paradigmas, como por exemplo, o paradigma imperativo [9].

Considerando, portanto, que linguagens funcionais apresentam o formalismo adequado de suas construções, é importante prover aos programadores um sistema eficiente e confiável, no qual possam escrever os programas dentro da especificação da linguagem e gerar o código correspondente ao programa.

Tradicionalmente, na compilação de linguagens funcionais usa-se como código intermediário a linguagem do Cálculo-Lambda [1] com algumas construções que facilitam a especificação semântica. A partir daí, há vários caminhos possíveis para se atingir um código executável. Pode-se, por exemplo, gerar código para máquina SECD [3] ou ainda basear a compilação em combinadores [15]. Entretanto, a obtenção de compiladores eficientes para linguagens funcionais ainda é assunto de estudos e pesquisas e requer muito esforço para alcançar resultados expressivos.

$\mathcal{SCRIPT}$ é uma linguagem puramente funcional orientada por objetos, que visa prover uma notação adequada e estruturada de tal forma a permitir que as descrições de semântica denotacional possam ser efetivamente executadas e depuradas. Apresenta algumas extensões em relação aos elementos básicos de uma linguagem funcional pura, tais como: controle de visibilidade, encapsulação, herança, polimorfismo e ligação dinâmica[1], o que lhe confere característica de orientação a objetos.

Sabe-se que compilar um programa é, na maioria ds vezes, uma tarefa complicada e cara. Devido ao tamanho do trabalho envolvido e sua complexidade, um compilador quase sempre é desenvolvido por uma equipe, onde cada grupo ou pessoa da equipe é responsável por uma parte dessa tarefa. Aí surge um outro problema que está relacionado com a integração das partes. Normalmente, uma tarefa árdua se não houver uma boa coesão da equipe envolvida. O compilador de $\mathcal{SCRIPT}$ desenvolvido no Laboratório de Linguagens de Programação do DCC, devido ao seu tamanho e complexidade, foi dividido em três partes: (1) o *front-end*; (2) uma fase intermediária, o $\lambda-$*lifting* e o (3) *back-end*. O *front-end* [9] traduz a linguagem fonte, $\mathcal{SCRIPT}$, para a linguagem intermediária, $\mathcal{LAMB}$ [8, 9] e compreende as fases de análise léxica, análise sintática, criação e gerência da tabela de símbolos, verificação de tipos e geração de código intermediário. O *lifting* realiza a tradução de expressões lambda para supercombinadores [5] e constitui a etapa intermediária entre o *front-end* e o *back-end* do compilador de $\mathcal{SCRIPT}$. O *back-end* [3] consiste na compilação de supercombinadores para código em linguagem C [7].

Este artigo relata nossa experiência no desenvolvimento de uma abordagem alternativa para execução do código $\mathcal{SCRIPT}$. Em vez de se traduzir o código $\mathcal{LAMB}$ resultante para a coleção de supercombinadores e, posteriormente, para código C como mostrado acima, optou-se por uma tradução direta de $\mathcal{SCRIPT}$ para Haskell [16]. Nosso objetivo foi disponibilizar, de forma rápida e eficiente, um compilador de $\mathcal{SCRIPT}$ sem a necessidade de se implementar a parte do compilador que geraria o código executável, tarefa esta delegada ao compilador Haskell. A utilização de Haskell pode ser justificada pelo fato dessa linguagem, além de possuir compiladores eficientes, é uma linguagem de programação avançada, utilizada em aplicações do mundo real e contem várias características importantes, suportanto inclusive o estilo orientado por objetos [18].

Por questão de completeza, organizamos este texto da seguinte forma: a Seção 2 apresenta um

[1] do inglês, *dynamic binding*

resumo da linguagem $\mathcal{SCRIPT}$[2, 9], salientando seus aspectos mais relevantes. A Seção 3 discute a metodologia usada na implementação do compilador $\mathcal{SCRIPT}$, apresentando na Seção 3.2 a especificação de uma das construções da linguagem, as expressões de domínio usando esquemas de tradução para mostrar a geração do código Haskell. A Seção 4 apresenta os resultados do código compilado em Haskell.

2 A Linguagem $\mathcal{SCRIPT}$

A linguagem $\mathcal{SCRIPT}$ [2] é baseada na linguagem SDL [8]. Sua principal utilidade é prover um ambiente no qual descrições semânticas denotacionais possam ser efetivamente executadas e depuradas de maneira estruturada. $\mathcal{SCRIPT}$ define vários domínios: o domínio básico dos números inteiros representado por $\mathcal{N}$, dos *strings* representado por $\mathcal{Q}$, dos valores lógicos representado por $\mathcal{T}$, e, dos valores indefinidos representados pelo símbolo `?`.

Além desses domínios básicos, ela ainda define os domínios constantes; os o domínio de tuplas; o domínio de funções contínuas; o domínio de nodos da árvore sintática, onde os nodos de árvores sintáticas com o mesmo rótulo são representados por [$\mathtt{D}_1$ `,` ... `,` $\mathtt{D}_n$], e os domínios de listas, onde listas podem ser designadas de três formas diferentes:

1. `d`$*$: representa uma lista finita contendo qualquer número de componentes no domínio d.
2. `d`+: representa uma lista finita contendo pelo menos um componente no domínio d.
3. <$\mathtt{a}_1$ `,` $\mathtt{a}_2$ `,` ... `,` $\mathtt{a}_n$>: representa uma instância de uma lista com os componentes $\mathtt{a}_1$ `,` $\mathtt{a}_2$ `,` ... `,` $\mathtt{a}_n$, todos de um mesmo domínio d.

As expressões em $\mathcal{SCRIPT}$ podem ser expressões básicas, expressões de padrões, expressões de comparação, expressões condicionais, expressões `CASE`, expressões de abstração, expressões `LET`, aplicações de funções ou quaisquer outras combinações bem formadas de expressões mais simples com operadores. As expressões básicas são: constantes literais, variáveis, inteiros, *quotations*, expressões lógicas, expressões de listas, expressões de tuplas e expressões com nodos.

Uma expressão de padrões em $\mathcal{SCRIPT}$ pode ser uma constante literal; que casa com ela mesma; um identificador; que tratado como o valor `?` quando aplicado ao operador `IS`; o valor indefinido `?`, que casa com valores de qualquer tipo; ou então uma combinação de expressões de padrões mais simples e operadores de construção de padrão.

Para as expressões de comparação, sejam $\mathtt{e}_1$ e $\mathtt{e}_2$ expressões em $\mathcal{SCRIPT}$ e p uma expressão de padrão. Existem três formas distintas de comparar expressões em $\mathcal{SCRIPT}$:

1. $\mathtt{e}_1$ `EQ` $\mathtt{e}_2$: testa se as expressões $\mathtt{e}_1$ e $\mathtt{e}_2$ denotam o mesmo valor. É avaliada como `TT` se $\mathtt{e}_1$ e $\mathtt{e}_2$ possuírem o mesmo valor e como `FF` se possuírem valores diferentes ou se pelo menos uma delas possui valor funcional.
2. $\mathtt{e}_1$ `NE` $\mathtt{e}_2$: representa a negação da expressão $\mathtt{e}_1$ `EQ` $\mathtt{e}_2$.
3. $\mathtt{e}_1$ `IS` p: testa se a expressão $\mathtt{e}_1$ tem a forma particular, ou seja, a estrutura do padrão p.

Uma expressão condicional em $\mathcal{SCRIPT}$ é apresentada como `t` $\rightarrow$ $\mathtt{e}_1$ `,` $\mathtt{e}_2$, correspondendo ao caso onde `t` é uma expressão cuja avaliação retorna `TT`, `FF`, `?` ou $\perp$,[2] e $\mathtt{e}_1$ e $\mathtt{e}_2$ são expressões quaisquer. Caso `t` seja avaliado como `TT`, $\mathtt{e}_1$ será a expressão correspondente; se `t` for avaliado como `FF`, $\mathtt{e}_2$ será a expressão correspondente. A expressão correspondente será `?` ou $\perp$, caso `t` denote, respectivamente, `?` ou $\perp$.

Uma expressão CASE é uma construção `CASE` utilizada para pesquisar qual a estrutura ou forma de um valor. Expressões `CASE` são denotadas por uma expressão e por uma série de padrões que produzem como resultado uma expressão associada ao primeiro padrão que corresponde à estrutura do valor dado.

As expressões de abstrações de $\mathcal{SCRIPT}$podem ser abstrações `LAM` e abstrações de padrões. A operação `LAM x.e` é utilizada para representar as funções não-recursivas anônimas, sendo que a função

[2] o valor especial *bottom* ($\perp$) serve para modelar semântica de programas com execução infinita.

do operador `LAM` é associar o identificador `x` no escopo da expressão `e`. Nas abstrações de padrões, a operação `LAM p.e`, com `p` sendo um padrão e `e` uma expressão é utilizada para extrair componentes em um valor.

A forma geral das expressões `LET` é: `LET` $\mathtt{a}_1$ = $\mathtt{e}_1$ `LET` $\mathtt{a}_2$ = $\mathtt{e}_2$ ... `LET` $\mathtt{a}_i$ = $\mathtt{e}_i$ `IN e`, onde tem-se $\mathtt{a}_i$, $1 \leq i \leq n$, definida no escopo das expressões $\mathtt{e}_i$ e e. Cada $\mathtt{a}_i$ é dito estar na forma de ligação de padrões ou definição de função. Na aplicações de funções em $\mathcal{SCRIPT}$, uma expressão como `f g e` com `f` e `g` sendo expressões denotando funções e `e` uma expressão arbitrária, é construída da forma `(f(g)) (e)`. Outras formas possíveis, que eliminam o uso de parênteses, são `f; g; e` e que significa `f(g(e))`.

Em relação à passagem de parâmetros, $\mathcal{SCRIPT}$ utiliza o mecanismo de avaliação *lazy*. Outras características importantes de $\mathcal{SCRIPT}$ incluem a definição de funções associadas a domínios de tuplas e sobrecarga[3] de funções.

A estrutura completa de uma definição formal em $\mathcal{SCRIPT}$ consiste em um módulo principal, contendo a função principal definida para modelar o contexto geral da definição de semântica denotacional. Além disso, a estrutura pode conter zero ou mais módulos secundários externos que podem ser compilados junto com o módulo principal ou extraído de uma biblioteca de módulos compilados.

A função básica de um módulo é permitir que entidades relacionadas, tais como domínios e funções, sejam agrupadas para poderem ser usadas por outros módulos. Com isso, certos detalhes de definições de módulos e domínios que não precisam ser conhecidos pelos usuários de um módulo podem ficar ocultos.

Há três tipos de módulos em $\mathcal{SCRIPT}$: PROJECT, SYNTAX e MODULE. O módulo PROJECT serve para definir os parâmetros e o ambiente sobre o qual as definições devem ser avaliadas. Na seção de importação deste módulo, somente uma função pode ser importada e a mesma deve ser considerada como função principal da definição formal. O módulo SYNTAX normalmente apresenta três seções: DOMAINS, LEXIS e SYNTAX, onde: DOMAINS especifica os domínios de símbolos não-terminais, LEXIS declara as estruturas dos símbolos léxicos e SYNTAX: define as sintaxes concretas e abstratas. Um módulo MODULE é composto de quatro seções: EXPORTS, IMPORTS, DOMAINS e DEFINITIONS. A seção EXPORTS apresenta as entidades do módulo corrente que estão sendo exportadas, assim como seus níveis de encapsulamento; na seção IMPORTS são feitas as importações de símbolos, definindo seus graus de visibilidade e relacionando os módulos de onde eles serão importados; a seção DOMAINS apresenta as declarações de domínios, variáveis e funções e a seção DEFINITIONS apresenta as definições de funções e outros valores, por meio de uma lista de definições de funções. Os módulos MODULE servem para fazer encapsulações de definições de domínios e funções, e estabelecer a interface de comunicação entre os módulos.

3 Implementação do Compilador $\mathcal{SCRIPT}$

3.1 O *Front-End*

Para facilitar o entendimento de nosso projeto, mostramos primeiramente como foi implementado o *front-end* de $\mathcal{SCRIPT}$.

O projeto original do *front-end* $\mathcal{SCRIPT}$ [2, 9] foi estruturado em três passos, a saber:

1. As principais tarefas do primeiro passo foram: o processamento do texto fonte; a análise léxica; a análise sintática e a coleta e instalação de domínios e variáveis na tabela de símbolos. A tabela de símbolos de todo o programa foi implementada como uma floresta [17], uma vez que para cada módulo a ser compilado, foi montada uma árvore com as informações dos símbolos deste módulo. Os principais serviços oferecidos pela tabela de símbolos são: a gerência de escopo e visibilidade dos símbolos; a inserção, remoção e consulta de símbolos e a exportação e importação de símbolos.

[3] do inglês, *overloading*

2. As tarefas do segundo passo incluíram a fase de análise de dependência e recursividade entre funções e variáveis e consistiu na criação de listas para indicar quais identificadores aparecem livres em quais definições ou funções. A coleta deste tipo de informação termina com a incorporação das mesmas aos símbolos instalados na tabela de símbolos.

3. O terceiro passo consistiu na verificação de tipos, análise semântica e geração de código $\mathcal{LAMB}$. A verificação de tipos oferece o serviço para determinar a equivalência e compatibilidade de domínios, bem como detecção de erros relacionados ao uso de domínios. A análise semântica verifica a estrutura semântica do programa fonte, relatando erros semânticos estáticos. As informações coletadas sobre as construções foram utilizadas na fase de geração de código.

Como a geração de código ocorre somente no terceiro passo do *front-end*, em nossa parte do projeto não foi preciso implementar nenhum procedimento para instalar símbolos ou mesmo atualizar os já instalados. No entanto, se fez necessário conhecer as estruturas de dados bem como os procedimentos implementados para recuperar informações da tabela de símbolos.

As informações recuperadas da tabela de símbolos foram incorporadas aos atributos associados aos símbolos não-terminais da gramática, sendo, portanto, muitas vezes desnecessário acessar a tabela de símbolos, uma vez que tais informações podiam ser obtidas por meio destes atributos. Entretanto, devido a diferenças existentes entre a linguagem $\mathcal{LAMB}$, inicialmente usada como linguagem objeto [4, 9], e a linguagem Haskell, as informações necessárias para gerar código em cada uma das linguagens nem sempre eram as mesmas.

3.2 Especificação dos Esquemas de Tradução para a Geração de Código Haskell

Esta seção apresenta como exemplo apenas os esquemas de tradução para as expressões de domínio e módulos que correspondem ao mapeamento das construções de $\mathcal{SCRIPT}$ para Haskell. O leitor interessado nas demais expressões e padrões pode consultar [18]. A abordagem usada no processo de geração de código segue o esquema de tradução de [12].

Para a tradução deve-se considerar que em $\mathcal{SCRIPT}$: m_1,m_2,...,m_n são módulos do tipo `MODULE`; e,e_1,e_2,...,e_n são expressões; p, p_1, p_2, ..., p_n são padrões; f(f_1,f_2,...,f_n),g e h são identificadores denotando funções; F(F_1,F_2,...,F_n),G e H denotam corpos de funções; a(a_1,a_2,...,a_n),b e c são identificadores denotando variáveis quaisquer ou constantes literais, e, D denota um domínio válido. A compilação de um módulo m é m'; a compilação de uma expressão e é e$'$, e a compilação de um corpo de função F é F', seguindo assim a regra para representar as compilações dos elementos da linguagem.

Módulos	$\mathcal{SCRIPT}$	Haskell
	MODULE m_i	module m_i (p'_1,p'_2,p'_3, ..., p'_m) where
	EXPORT p_1, p_2, p_3, ..., p_m	
	DEFINITIONS DEF $f_1 = F_1$	$f'_1 = F'_1$
	DEF $f_2 = F_2$	$f'_2 = F'_2$
	DEF $f_3 = F_3$	$f'_3 = F'_3$
	$\vdots$	$\vdots$
	DEF $f_{n-1} = F_{n-1}$	$f'_{n-1} = F'_{n-1}$
	DEF $f_n = F_n$	$f'_n = F'_n$
	END m_i	

Considere o módulo m_j como sendo o módulo apresentado a seguir, onde $1 \leq x, y \leq m$, ou seja, p_x e p_y são quaisquer das funções exportadas por m_i

Módulos	$\mathcal{SCRIPT}$	Haskell
	MODULE m_j IMPORT m_i(p_1,p_3,...,p_x,...,p_y) IMPORT m_k(g) DEFINITIONS DEF h=... p_1 ... g ... p_3 ... END m_j	module m_j where import m'_i(p'_1,p'_3,...,p'_x,...,p'_y) import m'_k (g') h = ... p'_1 ... g' ... p'_3
Expressões Domínio	$\mathcal{SCRIPT}$	Haskell
a. Declaração função associada a domínio	DEF$A.f$ (a_1:D_1,a_2:D_2,...,(a_n:D_n):D_{n+1}	$f\_\_A$ *this* a_1 a_2 ... a_n
b.Seleção de campos associados a expressões	$this.a_n$	Considerando m ($m \geq n$) como número de campos do *this*, temos: let (a_1, a_2, ..., a_m) = *this* in a_n
	$e.a_n$	Considerando m ($m \geq n$) como número de campos da expressão e, temos: let (a_1, a_2, ..., a_m) = e in a_n
c.Chamada de função associada a domínio	$a.f$ (x_1) (x_2) ... (x_n)	$f\_\_A$ a (x_1) (x_2) ... (x_n) onde A é o domínio da expressão a
d.Equivalência entre parâmetros,argumento	DEF $z = p$ *(a)*	Seja m o número de campos associados ao parâmetro ao qual o argumento a está ligado, sendo n o número de campos do argumento a. Temos dois casos: 1. se $m = n$, a compilação é: $z = p$ a 2. se $n > m$, a compilação é: $z = p$ (let (a_1, a_2, ..., a_n) = a in (a_1, a_2, ..., a_m))
e.Atribuição entre tuplas	DEF d:A = c onde A é um domínio de tupla.	Seja n o número de campos de d e m o número de campos de c. Temos dois casos: 1. se $m = n$, a compilação é: d=c 2. se $n > m$, a compilação é: d=let (c_1,c_2,...,c_m)=c in (d_1,d_2,...,d_n))

3.3 Metodologia de Implementação do Código Haskell

O maior esforço na implementação do compilador foi adaptar o mecanismo de geração de código $\mathcal{LAMB}$ para Haskell. As produções estavam todas estruturadas de modo a gerar código $\mathcal{LAMB}$, dificultando a geração do código Haskell que não necessariamente possuía a mesma ordem do código $\mathcal{LAMB}$. Por isso, em muitos casos, foi preciso armazenar as informações em estruturas temporárias e só ao fim da ação semântica processar o código Haskell.

Como conseqüencia, foram implementadas ou modificadas algumas funções para a geração de código Haskell, estas funções estão basicamente relacionadas com: a geração de código para compreensão e atualização de listas; geração de código de *tokens* de uma expressão com operadores binário e unário; geração de código de *tokens* de uma lista de cláusulas em uma construção `case`; geração de

código de lista expressas por um intervalo. Além dessas funções foram implementadas funções para separar os *itens léxicos* básicos de uma estrutura mais complexa e para a gravação do código Haskell referente aos *itens léxicos* no arquivo de saída.

Em relação a verificação de tipos de Haskell não encontramos nenhuma dificuldade pois toda a verificação de tipos já havia sido feita pelo *front-end* $\mathcal{SCRIPT}$ e o código Haskell não contém informações quanto aos tipos.

Haskell é uma linguagem fortemente tipada, ela possui tipagem dinâmica, ou seja, os tipos podem ser definidos em tempo de execução, de acordo com o contexto de primeiro uso da variável no programa. Mesmo com essa característica não houve dificuldades com seu verificador de tipo. Portanto, caso a compilação ocorra com sucesso, é garantido que o código Haskell resultante não terá erros quanto ao sistema de tipos. Essa característica de tipagem dinâmica torna Haskell apropriada como linguagem intermediária, por não necessitar que as definições de tipo no código-fonte sejam compiladas para a linguagem intermediária. Este fato pode ser visualizado nos exemplos da Seção 4. Nestes exemplos todas as definições de domínios nos exemplos $\mathcal{SCRIPT}$ foram eliminadas quando compiladas para Haskell.

Apresentamos agora as soluções adotadas para dar suporte às construções mais elaboradas como orientação por objetos presentes em $\mathcal{SCRIPT}$. Apesar de $\mathcal{SCRIPT}$ ser uma linguagem orientada por objetos, as suas construções de orientação por objetos não eram totalmente compatíveis conceitualmente com as construções de Haskell. Por isso, se fez necessária a elaboração de um esquema em Haskell a partir de seus tipos básicos.

1. Classes e Objetos: em $\mathcal{SCRIPT}$, o similar a classes nas linguagens imperativas orientadas por objeto são os domínios de tuplas e o similar a objetos são as instâncias de tuplas. Este mesmo conceito de tuplas e classes foi usado na tradução, utilizando-se a estrutura de dados tuplas original em Haskell. O trecho de código $\mathcal{SCRIPT}$, ilustrado a seguir, declara um domínio de tuplas com dois componentes e define uma instância desta tupla.

   ```
   DOMAINS
   A = (a1:N, a2:Q)                                          (a)
   DEF a:A = (1, "a")
   ```

   Em Haskell, a declaração do domínio não gera código algum, apesar das informações da tupla, número de componentes, tipos, etc, serem armazenadas na tabela de símbolos para utilização em outras construções. Assim, o código Haskell resultante é:

   ```
   a = (1, "Tupla")
   ```

   Note que, em Haskell, associação dos tipos é feita em tempo de execução.

2. Herança: em $\mathcal{SCRIPT}$, o mecanismo de herança é feito por meio de extensão de tuplas. Qualquer tupla estendida é descendente de sua tupla base. Portanto, em $\mathcal{SCRIPT}$, uma hierarquia de tuplas do tipo

   ```
   DOMAINS
   A = (a1:N, a2:Q)                                          (b)
   B = A EXT (b1:N, b2:Q)
   ```

   não resulta em código direto em Haskell, mas novamente as informações de herança são armazenadas para compilação de outras construções.

3. Polimorfismo: em $\mathcal{SCRIPT}$, os dois tipos de polimorfismo permitidos são sobrecarga[4] e inclusão. Haskell suporta apenas o de inclusão, excluindo o de sobrecarga por decisão de projeto. Polimorfismo de sobrecarga ocorre quando uma ou mais função recebe o mesmo nome no mesmo escopo. Os tipos e a aridade dos argumentos devem permitir a resolução da ambigüidade do nome sobrecarregado. Devido à decisão de projeto de não gerar nenhum tipo de informação quanto a tipos no código Haskell, adicionado ao fato de que o polimorfismo de sobrecarga não pode ser simulado, uma vez que em Haskell, uma mesma função não pode ser declarada com aridades diferentes nem com tipos de parâmetros diferentes, uma função polimórfica em $\mathcal{SCRIPT}$ como a seguir:

```
DEF p(a:A):N = 1
        DEF p(b:B):N = 2                                        (c)
        DEF p(a:A, b:B):N = 3
```

gera em Haskell um conjunto de funções do tipo

```
p a = 1
p b = 2
p a b = 3
```

que não é aceito pelo processador de Haskell.

O polimorfismo de inclusão está relacionado com a capacidade de extensão de tuplas. Toda função que tem um argumento de tipo tupla também aceita argumentos de domínios que são extensões deste tipo. Isto significa que uma tupla de domínio estendido é sempre uma instância de tupla do domínio base correspondente. Por exemplo, considere o seguinte trecho de código

```
 DOMAINS
A = (a1:N, a2:Q)
B = A EXT (b1:N, b2:Q)
DEF p (a:A):N = 10
DEF a = (1, "a")                                                (d)
DEF b = (2, "b")
DEF z1 = p (a)
DEF z2 = p (b)
```

Neste exemplo, o domínio de tuplas `B` é uma extensão do domínio base `A`. A função `p` está declarada para receber como argumento uma tupla do domínio `A`. A seguir, tanto uma tupla do domínio `A` quanto do domínio `B` são aplicadas à função `p`, caracterizando o polimorfismo de inclusão.

Em Haskell, o polimorfismo de inclusão foi simulado utilizando casamento de padrões[5] a fim de selecionar os elementos de `B` comuns a `A`. O código resultante para o exemplo (d) é:

```
p a = 10
a = (1, "a")
b = (2, "b")
z1 = p a
z2 = p (let (a1, a2, a3, a4) = b) in (a1, a2))
```

[4] do inglês overloading
[5] do inglês pattern matching

Repare que as informações dos tipos das tuplas `A` e `B` e da função `p` precisam estar armazenadas na tabela de símbolos para que a expressão `let` possa ser gerada.

4. Encapsulação de Dados: tanto em $\mathcal{SCRIPT}$ como no modelo adotado em Haskell, módulo é o mecanismo para encapsular dados.

5. Funções Virtuais e Funções Associadas a Domínios: em $\mathcal{SCRIPT}$, funções podem estar associadas a domínios. Estas funções são equivalentes a funções virtuais de linguagens imperativas orientadas por objetos. Se o domínio ao qual a função está associada for uma tupla, então toda aplicação desta função deve ser qualificada por um objeto cujo domínio é o mesmo associado à função ou uma extensão deste.

```
DOMAINS
A = (a1:N, a2:Q)
DEF A.f (x:N):N = x + this.a1                                    (e)
DEF a = (1,"a")
DEF z1 = a.f(10)
```

é compilado para Haskell como:

```
f__A this x = x + let(A1, a2) = this in a1
a = (1,"a")
z1 = f__A a 10
```

As funções associadas a domínios são polimórficas, mas seus parâmetros são sempre convertidos para o tipo declarado, i.e., a passagem de parâmetro polimórfico é por valor.

O modelo completo das construções de orientação por objetos suportadas pelo compilador implementado pode ser visto em [18].

Escopo é tratado pelo *front-end* de $\mathcal{SCRIPT}$, se o escopo está errado em $\mathcal{SCRIPT}$ não há código Haskell. Em Haskell as regras de escopo verificadas obedeceram as mesmas convenções estabelecidas para $\mathcal{SCRIPT}$, por exemplo, variáveis declaradas como parâmetros só são aceitas dentro de função.

Questões relacionadas ao escopo não foram abordadas diretamente, mas um dos serviços que o módulo da tabela de símbolos presta é a gerência do escopo dos símbolos.

4 Resultados da Execução do Compilador $\mathcal{SCRIPT}$

Esta seção apresenta os resultados obtidos com a execução do compilador $\mathcal{SCRIPT}$ para alguns exemplos de programas escritos em $\mathcal{SCRIPT}$. Estes exemplos não constituem programas completos em $\mathcal{SCRIPT}$, mas procuram cobrir algumas das estruturas e construções importantes da linguagem. A ordem de apresentação dos mesmos é a seguinte: a esquerda da tabela lista-se o código do programa escrito em $\mathcal{SCRIPT}$ que serviu como entrada para o compilador. Do lado direito é listado o código Haskell obtido como saída da execução do compilador. Nos exemplos 3 e 4, além dos programas fonte e objeto, é mostrada a execução destes programas no interpretador `Hugs`.

1. Programa com construção de criação de listas usando compreensão de listas

SCRIPT	Haskell
MODULE TesteScript	module TesteScript where
DOMAINS Ndom = N ;	import Prel_Tav
Tdom = T ;	
LdomN = N*	
DEFINITIONS	
DEF ndom1= 6	ndom1 = 6
DEF ndom2= 10	ndom2 = 10
DEF t1 = "TT"	t1 = True
DEF ldomN'= <1 .. 5> ldomN'	= [1 .. 5]
DEF ldomN"= <ndom1 .. ndom2>	ldomN" = [ndom1 .. ndom2]
DEF ldomN1= < n' PLUS n" | n' ← ldomN' :: t1 |	ldomN1 = [n' + n" | n' ← ldomN' , t1 ,
n" ← ldomN" >	n" ← ldomN"]
END TesteScript	

2. Programa com tuplas e funções associadas a domínios

SCRIPT	Haskell
MODULE TesteScript	module TesteScript where
DOMAINS Adom = (m:N, T, y:N, a:T) ;	import Prel_Tav
Bdom = (m:N, T, y:N);	
Ndom = N; Tdom = T	
DEFINITIONS DEF adom1 = (11,t1,ndom1,t1)	adom1 = (11, t1, ndom1, t1)
DEF bdom1 = (11, t1, ndom1)	bdom1 = (11, t1, ndom1)
DEF ndom1 = 11	ndom1 = 11
DEF ndom2 = 4	ndom2 = 4
DEF ndom3 = ndom1 PLUS ndom2	ndom3 = ndom1 + ndom2
DEF t1 = "FF"	t1 = False
DEF Adom.f1 (x: Ndom) : Ndom =	f1__Adom this__1 x=(let(a__1,a__2,a__3,a__4)=
THIS.y PLUS x	this__1 in a__3)+x
DEF g1 (b:Bdom) (a:Bdom) (x:Ndom) :	g1 b a x =(let(a__1,a__2,a__3)=b in a__1)+
Ndom = b.m PLUS b.y	(let(a__1,a__2,a__3)=b in a__3)
DEF ndom4 = g1 adom1 bdom1 ndom1	ndom4= g1(let(param__1,param__2,param__3,param__4)
REM g1 adom1 bdom1 ndom1	(param__1,param__2,param__3)) bdom1 ndom1
END TesteScript	= adom1 in
	'mod' g1 (let (param__1,param__2,param__3,
	param__4)=adom1 in (param__1,param__2,
	param__3)) bdom1 ndom1

3. Programa com tuplas e funções associadas a domínios

SCRIPT	Haskell
MODULE TesteScript	module TesteScript where
DOMAINS Adom = (m:N, T, y:N, n:T) ;	import Prel_Tav
Ndom = N	
DEFINITIONS DEF ndom1 = 10	ndom1 = 10
DEF adom1 = (11, "TT", 11, "TT")	adom1 = (11, True, 11, True)
DEF Adom.f (x:Ndom) : Ndom = x	f__Adom this__1 x = x
DEF n1 = adom1.f (10) PLUS adom1.f (10)	n1 = f__Adom adom1 10 + f__Adom adom1 10
END TesteScript	

4. Programa com atribuição de tuplas de tamanhos diferentes

$\mathcal{SCRIPT}$	Haskell
MODULE TesteScript	module TesteScript where
DOMAINS Adom = (m:N, T, y:N) ; import	Prel_Tav
Bdom = Adom EXT (a:N);	
Ndom = N	
DEFINITIONS	
DEF bdom1 = (11, "TT, 11, 11)	bdom1 = (11, True, 11, 11)
DEF adom1 = bdom1	adom1 = let (a_1 , a_2 , a_3 , a_4) = bdom1 in
	(a_1 , a_2 , a_3)
END TesteScript	

Execução no interpretador Haskell para 3:

```
Hugs session for:
/pkg/cursos/haskell/hugs/hugs/lib/Prelude.hs
Prel_Tav.hs
ex3.hs
Type :? for help
TesteScript> ndom1
10
TesteScript> adom1
(11, True, 11, True)
TesteScript> f__Adom adom1 ndom1
10
TesteScript> n1
20
TesteScript> :q
[Leaving Hugs]
```

Execução no interpretador Haskell para 4:

```
Hugs session for:
/pkg/cursos/haskell/hugs/hugs/lib/Prelude.hs
Prel_Tav.hs
ex4.hs
Type :? for help
TesteScript> bdom1
(11, True, 11, 11)
TesteScript> adom1
(11, True, 11)
TesteScript> :q
[Leaving Hugs]
```

5 Conclusão

O compilador $\mathcal{SCRIPT}$ descrito está inserido em um ambiente cujo objetivo é oferecer aos projetistas de linguagens de programação uma ferramenta para composição de descrições de semântica denotacional de maneira estruturada. A utilização de tal compilador visa o aumento da qualidade dessas descrições, ou seja, definições mais confiáveis e fáceis de ler e compreender.

O trabalho apresentado neste artigo compreendeu a implementação da geração de código $\mathcal{SCRIPT}$ para Haskell em substituição ao gerador de código $\mathcal{LAMB}$, originalmente implementado [9]. Nosso objetivo foi disponibilizar, de forma rápida e eficiente, um compilador de $\mathcal{SCRIPT}$ sem a necessidade de se implementar a parte do compilador que geraria o código executável, tarefa esta delegada ao compilador Haskell. A utilização de Haskell pode ser justificada pelo fato dessa linguagem, além de possuir compiladores eficientes, é uma linguagem de programação avançada, utilizada em aplicações do mundo real e contem várias características importantes, suportanto inclusive o estilo orientado por objetos [18].

O compilador de $\mathcal{SCRIPT}$ para Haskell agora faz parte do *back-end* do compilador $\mathcal{SCRIPT}$. Salvo as modificações relatadas, as etapas do *front-end* do compilador como análises léxica e sintática, verificação de tipos e tabela de símbolos foram as mesmas implementadas em [9], o que diminuiu o esforço final para a produção deste compilador.

O compilador de $\mathcal{SCRIPT}$ para Haskell implementado é capaz de efetuar a leitura e impressão de expressões $\mathcal{SCRIPT}$. Para a leitura, foi utilizado um analisador LALR(1) [12]. Além disso, o compilador realizou o *pattern-matching,* seguido das redução das expressões para um padrão estrutural. O compilador foi implementado na linguagem C e gera código Haskell na versão 1.3.

References

[1] Church, A. *The Calculi of Lambda-Conversion.* Ann. of Math. Studies 6, Princeton University.

[2] Bigonha, Roberto da Silva. *The Revised Report on the $\mathcal{SCRIPT}$ Language for Denotational Semantics.* Relatório Técnico 016/97. DCC-UFMG, 07/1997.

[3] Maia, Marcelo de Almeida. *Implementação Eficiente de uma Linguagem para Definição de Semântica Denotacional.* Tese de Mestrado, DCC/UFMG, 1994.

[4] Silva, Luiz Ricardo de Faria Silva. *$\mathcal{LAMB}$ - Um Interpretador para o Cálculo-Lambda Estendido.* Anais do X Seminário Integrado de *Software* e *Hardware*, III Congresso da SBC, Campinas, 23-29/1983, páginas: 487-499, 1983.

[5] Jones, Simon L. Peyton. *The Implementation of Functional Programming Languages.* C.A.R Hoare Series Editor, 1989.

[6] Slenneger, K. and Kurts, Barry. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, 1995.

[7] Kerningham, B. and Ritchie, D. *The C Programming Language.* Prentice Hall, 1988.

[8] Mosses, P. D. *SIS - A Compiler-Generator System Using Denotational Semantics.* Technical Report, University of Aarhus, Denmark, 1978.

[9] Oliveira, Fabíola Fonseca. *Compilação de uma Linguagem Funcional, Orientada por Objetos, para Definição de Semântica Denotacional.* Tese de Mestrado, DCC/UFMG, 1998.

[10] Wadler, Philip and Bird, Richard. *Introduction to Functional Programming.* C.A.R. Hoare Series Editor, 1988.

[11] Watt, David A. *Programming Language Concepts and Paradigms.* C.A.R. Hoare Series Editor, 1989.

[12] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[13] Turner, D.A. *A New Implementation Technique for Aplicative Languages. Software - Practice and Experience.* 9:31-49, 1979.

[14] Thompson, Simon *HASKELL The Craft of Functional Programming*, Addison-Wesley, 1998.

[15] Inc. Free software Foundation. Gnu projext e compiler. Relatório Técnico, 1988.

[16] Meyer, Bertrand *Objected-Oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 2nd Edition, 1997.

[17] Bigonha, Roberto da Silva, Bigonha, Mariza A. S., Silva, Yedda A. D. *Tabela de Símbolos:Implementação e Avaliação.* Relatório Técnico LLP004/99 do Laboratório de Linguagens de Programação, DCC- UFMG, 03/1999.

[18] Taveira, Wendell F, Bigonha, Mariza A.S., Bigonha, Roberto S., *Compilador da Linguagem Funcional Orientada por Objeto $\mathcal{SCRIPT}$ para Haskell* Relatório Técnico LLP009/99 do Laboratório de Linguagens de Programação, DCC-UFMG, 09/1999.