Implementação da Linguagem Funcional *SCRIPT*

Fabíola F. Oliveira

Roberto S. Bigonha

Mariza A. S. Bigonha

Marco Rodrigo Costa

e–mail: {mariza, bigonha, fabiola, mrcosta} @dcc.ufmg.br

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 *SCRIPT* para C. *SCRIPT* é uma linguagem funcional, ela atende a propósitos gerais e visa prover uma notação adequada para permitir que descrições de semântica denotacional possam ser processadas. Alguns fatores determinantes para a escolha da linguagem C como linguagem objeto foram: sua vasta utilização, seu poder de expressão e o fato de poderem ser encontrados vários compiladores eficientes no mercado. O compilador foi desenvolvido pela equipe de pesquisadores do Laboratório de Linguagens de Programação da UFMG, sendo composto por três etapas: *Front-End, Lambda-Lifting* e *Back-End.* Este artigo descreve as etapas de implementação do *Front-End* e do *Lambda-Lifting.*

Palavras-Chaves: compilador, linguagem funcional, *Lambda-Lifting, SCRIPT*

1 Introdução

A importância da definição formal da semântica de uma linguagem pode ser percebida quando se desejam soluções para dúvidas em relação ao significado de suas construções. Para que seja possível verificar se um programa está correto, o programador precisa escrevê-lo dentro das regras descritas para a linguagem, que, para isso, precisam ser claras e sem ambigüidade. Torna-se necessário, portanto, construir um padrão para definição semântica de uma linguagem para fornecer aos programadores uma forma única e consistente de entender a linguagem. Ao considerar essa necessidade, verifica-se que as linguagens funcionais são capazes de expressar o formalismo desejado, provendo uma importante ajuda para as definições semânticas.

Considerando então que linguagens funcionais são um formalismo adequado para definições semânticas, mas que a implementação de um compilador para tais linguagens quase sempre resulta em implementações ineficientes, se comparadas a implementações de linguagens convencionais imperativas, procurar uma forma mais eficiente de implementar um compilador para definições semânticas de linguagens escritas em uma linguagem funcional pode ser uma tarefa bastante interessante e instigante, além de ser uma proposta com importantes aplicações nesta área de estudo.

Em geral, a implementação de compiladores para linguagens funcionais tende a ser conduzida de forma diferente da implementação dos compiladores no paradigma imperativo. No passado as implementações de linguagens funcionais baseavam-se no esquema de ambientes, e a outra tem seu ponto principal na redução de grafos. Ao considerar técnicas baseadas em ambientes, a primeira máquina abstrata desenvolvida para reduzir expressões λ-*calculus* foi a máquina SECD, descrita em [14]. As implementações mais recentes têm sido radicalmente diferentes das tecnologias convencionais para compiladores e as mudanças mais significativas têm se destacado pelo uso de novas técnicas que objetivam tornar os compiladores mais eficientes [11]. A título de exemplo, linguagens que já apresentam tais mudanças são Chalmers Lazy ML [13] e Haskell [12]. Estas abordagens são baseadas em reduções de grafos, que têm como ponto importante o trabalho de Turner [16], que descreve uma técnica para implementação dessas linguagens por meio de redução por combinadores SK. A propriedade crucial é que qualquer expressão λ pode ser transformada em uma expressão constituída somente por tais combinadores. Outras formas de combinadores surgiram com o objetivo de otimizar aqueles propostos por Turner, tais como os multicombinadores categóricos [17] e os supercombinadores [6]. Os supercombinadores são compostos por funções que podem ser transformadas em um combinador por meio da adição de parâmetros formais extras, correspondendo às variáveis livres que aparecem no corpo da função. Para isso, seus corpos não podem ser abstrações lambda, e qualquer abstração lambda no corpo de um supercombinador deve ser um supercombinador. A máquina-G também é uma técnica para implementação eficiente de linguagens funcionais *lazy* desenvolvidos por Augustsson [18] e Johnsson [13], que consiste em uma implementação rápida da redução de grafos baseada na compilação de supercombinadores. Seus trabalhos devem ser vistos como uma prova matemática formal de que a máquina-G é correta do ponto de vista das especificações de semântica denotacional de linguagens simples.

Este artigo relata nossa experiência na construção de um compilador para $\mathcal{SCRIPT}$ desenvolvido em três etapas: o *front-end*, o *lambda-lifting* e o *back-end*. Enfatizaremos os pontos mais desafiantes do compilador, especificamente no desenvolvimento do *front-end* e do *lambda-lifting*. $\mathcal{SCRIPT}$ é uma linguagem funcional orientada por objetos com recursos para definições de semântica denotacional de linguagens de programação. A semântica denotacional mapeia um programa diretamente para o seu significado. Nesse sentido ela é util para especificar a funcionalidade de novas construções de linguagem que ainda estão para ser implementadas, tornando sua utilização uma contribuição em projetos de geração automática de compiladores.

Para apresentar os maiores desafios que deparamos durante o desenvolvimento desse compilador considere a descrição de duas características importantes de $\mathcal{SCRIPT}$: (1) Um programa em $\mathcal{SCRIPT}$

é escrito em módulos que podem ser compilados separadamente, mas que troca informações entre si. Outra característica importante e que ainda diz respeito às condições de importação de exportação de $\mathcal{SCRIPT}$ é o controle de visibilidade de um símbolo que foi exportado ou importado, caracterizado como *aberto* ou *fechado* quando ocorre a importação/exportação. Essa denominação indica que um símbolo fechado só terá seu nome conhecido no módulo no qual foi importado. Já um símbolo aberto terá tanto seu nome quanto sua *assinatura de domínio* disponíveis para os módulos que o importarem. Essa característica dos módulos foi concretizada por meio do uso de técnicas de importação e exportação de símbolos e levou à necessidade de um gerenciamento adequado de escopo e visibilidade para que pudesse garantir sua perfeita realização. Portanto, este gerenciamento pode ser considerado um dos aspectos principais desenvolvidos neste trabalho.(2) A implantação da disciplina de tipos para verificar a compatibilidade e equivalência estrutural de domínios[1].

Como $\mathcal{SCRIPT}$ é uma linguagem com características de orientação por objetos, pois contempla os conceitos básicos de classe, objetos, herança de tipo, encapsulação de dados, polimorfismo, funções virtuais e ligação dinâmica, o compilador trata situações referentes a herança, encapsulação e polimorfismo de sobrecarga e inclusão. $\mathcal{SCRIPT}$ também apresenta o mecanismo de funções virtuais, que podem estar associadas ou ligadas a domínios. Levando em conta essas duas características, polimorfismo e funções virtuais, pode-se dizer que $\mathcal{SCRIPT}$ apresenta associação dinâmica de funções. Este também é um aspecto relevante da linguagem que é tratado pelo compilador, contudo, por questões de espaço a metodologia empregada pelo *front-end* para estas características não são abordadas neste artigo.

2 Compilação de Linguagens Funcionais

Em 1978 um dos primeiros sistema para geração de compiladores usando semântica denotacional, **SIS**, foi proposto por Mosses [9]. Seu gerador de código é produzido a partir de uma semântica formal da linguagem de programação, a semântica denotacional. Ele recebe uma árvore de *parser* de um programa e retorna o que pode ser entendido como uma expressão básica em notação lambda, chamada $\mathcal{LAMB}$. O algoritmo de redução implementado nesse sistema utiliza a estratégia de chamada por necessidade [2].

Em 1980, Robin Milner projetou a linguagem **ML**, inicialmente como uma metalinguagem. Seu objetivo foi usá-la em um sistema de verificação de programas. As características marcantes desta linguagem que nascia eram: linguagem de programação funcional, com declaração de tipos, fortemente tipada, com avaliação estrita, que usa inferência de tipo. Desde então **ML** ganhou muitas variantes. A técnica básica de compilação utilizada pelos desenvolvedores de compiladores **ML** foi a máquina-G proposta inicialmente em [18] e [13]. O compilador de **ML** mais utilizado é o *Standard ML of New Jersey*, desenvolvido pelos laboratórios da `AT&T` Bell e a Universidade de Princeton.

Uma outra linguagem de programação puramente funcional foi desenvolvida também na década de 1980, Miranda [5]. Além da linguagem, um ambiente de programação interativo também foi desenvolvido. Em [6] é apresentada uma técnica de compilação para linguagens funcionais, transformando um subconjunto da linguagem Miranda em uma versão enriquecida do λ-*calculus* , para depois transformar essa expressão em uma expressão ordinária.

Até esta época, não existia nenhuma linguagem de programação puramente funcional e não-estrita que pudesse ser considerada uma linguagem padrão. Um comitê com o objetivo de projetar tal linguagem foi montado, e a linguagem Haskell [12] é o resultado desta análise, com sua primeira versão datada de abril de 1990. A partir do surgimento de Haskell começaram a aparecer compiladores para

[1] Domínio em $\mathcal{SCRIPT}$ pode ser definido como uma entidade matemática usada para representar conjuntos com ordem parcial com um elemento mínimo *bottom*. Tipo e domínio têm neste texto o mesmo sentido.

[2] do inglês *call–by–need*.

esta linguagem. O compilador para Haskell da Universidade de Glasgow utiliza na sua compilação a técnica da máquina-G sem espinha e sem *tag*, com coletor de lixo. O compilador da Universidade de Chalmers para Haskell é baseado no compilador **LML** de Augustsson e Johnsson, e possui uma interface para *X windows*. O compilador para Haskell da Universidade de Yale possui um ambiente de programação interativo e flexível, com otimizações que inclui a análise de estringência.

3 A Linguagem $\mathcal{SCRIPT}$

$\mathcal{SCRIPT}$ é uma linguagem funcional que visa prover uma notação adequada para formular descrições semânticas denotacionais de linguagens de programação de uma forma estruturada. Para isto, questões como modularização de descrições, equivalência estrutural de tipos, encapsulação, herança, hierarquia de tipo e associação dinâmica foram incorporados na estrutura da linguagem.

Por razões de completeza, esta seção irá mostrar um resumo da descrição de $\mathcal{SCRIPT}$, cujos detalhes podem ser vistos em [1]. Em uma visão geral, um programa $\mathcal{SCRIPT}$ é formado por módulos de três tipos. Um destes módulos é o módulo PROJECT que define os parâmetros e o ambiente sobre o qual as definições devem ser avaliadas. Outro módulo distinto é o SYNTAX, que define a sintaxe concreta e abstrata de uma linguagem. O último tipo de módulo $\mathcal{SCRIPT}$ é o MODULE onde são encontrados domínios e definições de funções. MODULE é composto de quatro seções: EXPORTS, IMPORTS, DOMAINS e DEFINITIONS. Cada módulo $\mathcal{SCRIPT}$ pode ser compilado separadamente, sendo depois agrupados para formar uma máquina de processamento de definições denotacionais.

$\mathcal{SCRIPT}$ define vários domínios: o domínio básico dos números inteiros, dos *strings*, dos valores lógicos (TT, FF), e, dos valores indefinidos (?). Além desses domínios, ela ainda define os domínios constantes; de tuplas; de funções contínuas; de nodos da árvore sintática, e de listas, onde listas podem ser designadas de três formas diferentes: (1) `d*`: lista finita contendo qualquer número de componentes no domínio *d*. (2) `d+`: lista finita contendo pelo menos um componente no domínio *d*.
(3) <$\mathtt{a}_1$,$\mathtt{a}_2$,...,$\mathtt{a}_n$>: instância de uma lista com componentes $\mathtt{a}_1$,$\mathtt{a}_2$,...,$\mathtt{a}_n$, todos de um mesmo domínio.

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

Expressão de padrões pode ser constante literal; 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.

A seguir o símbolo `e`, possivelmente indexado, denota uma expressão $\mathcal{SCRIPT}$ qualquer e `p`, uma expressão de padrão. Há três formas para comparar expressões:

1. $\mathtt{e}_1$ `EQ` $\mathtt{e}_2$: testa se $\mathtt{e}_1$ e $\mathtt{e}_2$ denotam o mesmo valor. Avalia 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 $\mathtt{e}_1$ tem a forma particular, ou seja, a estrutura do padrão *p*.

Expressão condicional 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 $\perp$,[3] 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 $\perp$.

Expressão CASE é uma construção `CASE` usada para determinar qual a estrutura ou forma de um valor. Elas 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.

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

Expressões de abstrações podem ser abstrações `LAM` e abstrações de padrões. A operação `LAM x.e` é usada para representar as funções não-recursivas anônimas, sendo que a função do operador `LAM` é associar o identificador `x` no escopo da expressão `e`. Em abstrações de padrões, a operação `LAM p.e`, com `p` sendo um padrão e `e` uma expressão é usada para extrair componentes em um valor.

A forma geral das expressões `LET` é: `LET` a_1 = e_1 `LET` a_2 = e_2 ... `LET` a_i = e_i `IN e`, onde tem-se a_i, $1 \leq i \leq n$, definida no escopo das expressões e_i e e. Cada a_i é dito estar na forma de ligação de padrões ou definição de função. Na aplicações de funções, uma expressão como `f g e`, com `f` e `g` sendo expressões denotando funções e `e` uma expressão arbitrária, é interpretada como `(f(g)) (e)`. Outra forma possível, que elimina o uso de parênteses, é `f; g; e`, que significa `f(g(e))`.

Em relação à passagem de parâmetros, $\mathcal{SCRIPT}$ usa o mecanismo de avaliação *lazy*. Outras características incluem a definição de funções associadas a domínios de tuplas e sobrecarga[4] de funções.

3.1 O Sistema de tipos $\mathcal{SCRIPT}$

Nesta seção apresentamos as noções sobre a disciplina de tipos de $\mathcal{SCRIPT}$. O projeto de um verificador de tipo para uma linguagem está baseado nas informações sobre suas construções sintáticas, nas noções de tipos, e nas regras de atribuição de tipos para a construção da linguagem. Quando um símbolo é utilizado em uma expressão ele precisa estar dentro da hierarquia de tipos definida para essa expressão. Para verificar isto, devem-se utilizar os conceitos de compatibilidade de tipos caminhando-se na árvore da tabela de símbolos.

$\mathcal{SCRIPT}$ é uma linguagem polimórfica, fortemente tipada e usa uma equivalência estrutural para ver se dois tipos são ou não equivalentes. Nos conceitos de equivalência estrutural dois tipos são o mesmo se suas definições vêm a ser as mesmas quando expandidas [10]. Ou seja, quando todas as expressões constantes são trocadas por seus valores e todos os nomes de tipos são trocados por suas definições. No caso de tipos recursivos a expansão é o limite infinito da expansão parcial dele. A disciplina de tipos de $\mathcal{SCRIPT}$ requer que variáveis só sejam usadas em contextos onde seus domínios sejam compatíveis com os domínios esperados para tais contextos. A noção de compatibilidade de $\mathcal{SCRIPT}$ surgiu dos conceitos de equivalência estrutural e de inclusão de tipo, conceito presente em muitas linguagens imperativas, também mostrado como herança ou subtipagem [10].

A importação de uma variável pode ser feita independentemente da importação do domínio, ou de mais de um para o caso de ser uma expressão, aos quais ela está associada. Quando uma variável é importada também é importada a *assinatura de seu domínio*, sendo importante mencionar que apenas os nomes de domínios estão nessa *assinatura*, mas não suas definições. Essa *assinatura* é, então, associada à variável importada. Porém os nomes que estiverem mencionados nesta *assinatura* não serão importados, a menos que sejam explicitamente importados, o que é importante para o caso de ocorrer alguma referência a esses domínios quando do uso da variável. As principais condições para a verificação de equivalência e compatibilidade de tipos estão descritos a seguir.

3.1.1 Compatibilidade de Tipos

Um domínio D_1 é compatível com um domínio D_2, se pelo menos uma das condições abaixo é satisfeita:

- Domínios D_1 e D_2 possuem o mesmo nome de domínio.
- Nomes dos domínios D_1 e D_2 são diferentes, cada um sendo uma referência a uma definição reflexiva de domínio na qual as referências recursivas ocorrem nas mesmas posições nas suas estruturas de domínios.

[4] do inglês, *overloading*

- Domínio D_2 é uma expressão de domínio, e D_1 um nome de domínio, cuja definição visível é tal que $D_1 = d$ ou D_1=D_{11}=D_{12}=D_{13}=...=D_{1n}=d, onde cada D_{1i}, para $1 <= i <= n$, é um nome de domínio, e a expressão de domínio d é compatível com a expressão de domínio D_2.

- Domínio D_1 é uma expressão de domínio, e D_2 é nome de domínio, cuja definição visível é tal que D_2=d ou D_2=D_{21}=D_{22}=D_{23}=...=D_{2n}=d, onde cada D_{2i}, para $1<=i<=n$, é um nome de domínio, e a expressão de domínio D_1 é compatível com a expressão de domínio d.

- Domínios D_1 e D_2 são domínios de listas, D_1 é da forma d_1*, D_2 é da forma d_2*, e d_1 é compatível a d_2.

  Os domínios D_1 e D_2 são domínios de listas, D_1 é da forma d_1+, D_2 é da forma d_2+, e d_1 ser compatível com d_2.

- Domínios D_1 e D_2 são domínios de nodos, D_1 é da forma [d_{11}, d_{12}, ..., d_{1n}], D_2 ser da forma [d_{21}, d_{22}, ..., d_{2n}], e para cada i tal que $1 <= i <= n$, d_{1i} é compatível com d_{2i}.

- Domínios D_1 e D_2 são domínios de funções contínuas, D_1 é da forma d_{11} –> d_{12}, D_2 ser da forma d_{21} –> d_{22}, d_{11} é compatível com d_{21}, e d_{12} é compatível com d_{22}.

- Domínio D_1 é domínio polimórfico de lista vazia, e D_2 é o domínio de lista de qualquer tipo.

- Domínios D_1 e D_2 são domínios de tuplas e D_1 é uma extensão de D_2.

- Domínios D_1, D_2 são domínios de tuplas, D_1 é extensão de outro domínio de tuplas D_3; D_3 é da forma (d_{31},d_{32},...,d_{3n}), onde d_{3i}, para $1<=i <=n$, denota domínio dos componentes de D_3;D_2 ser da forma (d_{21},d_{22},...,d_{2n}), onde d_{2i}, para $1<=i <=n$, denota domínio dos componentes de D_2 e cada d_{3i} é compatível com seu correspondente d_{2i}, para $0<=i<=n$. Nomes de campos das tuplas não são considerados para o propósito de determinar compatibilidade de domínios.

- Domínio D_1 é domínio de tuplas da forma (D_{11}), e D_{11} é compatível com D_2.

- Domínio D_1 é uma expressão de domínio, o domínio D_2 é uma expressão de domínios da forma d_{21} | d_{22} | ... | d_{2n}, e D_1 ser compatível com algum d_{2i}, para $1 <= i <= n$.

- Domínio D_1 é um domínio de constante de uma quotation, e D_2 é um domínio built-in `Q` .

- O domínio D_1 é o domínio de valores indefinidos **"?"**.

4 A Linguagem Intermediária $\mathcal{LAMB}$

A escolha de $\mathcal{LAMB}$ como linguagem objeto da primeira etapa da compilação de $\mathcal{SCRIPT}$ baseia-se principalmente no fato de $\mathcal{LAMB}$ ser uma versão particular enriquecida de uma notação lambda. O uso do cálculo lambda em uma linguagem intermediária na compilação de linguagens funcionais de alto nível facilita a representação abstrata das funções matemáticas que são a base de tais linguagens. $\mathcal{LAMB}$ é uma linguagem simples, porém com sua semântica formal rigorosamente definida [7]. Apresenta poucas construções sintáticas e uma semântica de fácil entendimento, condições importantes para uma linguagem de implementação de cálculo lambda que visa demonstrar a correção de programas.

Contudo, apesar de seu poder de expressão e sua simplicidade semântica, $\mathcal{LAMB}$ tal como foi proposta por Mosses [7] não ajuda o trabalho dos usuários, por ser uma linguagem de difícil manipulação, portanto, uma extensão de $\mathcal{LAMB}$ foi definida, para que esta nova linguagem $\mathcal{LAMB}$ pudesse ser usada como a linguagem objeto da compilação do *front-end* com poder suficiente para compor todos

os programas funcionais. Extensões incorporadas a $\mathcal{LAMB}$ consistem em construções LET e DEF, o operador SPECIAL e operadores primitivos de Script.

Um programa em uma linguagem funcional pode ser visto como uma expressão, e executar esse programa seria avaliar esta expressão. Portanto, pode-se dizer que um programa $\mathcal{LAMB}$ é composto por uma expressão $\mathcal{LAMB}$. Essas expressões são formadas por padrões e operadores e estão associadas a domínios, sendo esses os elementos básicos condiderados na linguagem $\mathcal{LAMB}$ desta compilação.

5 O Compilador de $\mathcal{SCRIPT}$

O *Front-End* do compilador traduz $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$ [2]. O *Lambda-Lifting* traduz $\mathcal{LAMB}$ para Supercombinadores [3]. O *Back-End*, fortemente influenciado por [6], traduz Supercombinadores para código C usando uma máquina-G estendida [4]. A abordagem adotada corresponde a mesma técnica usada para implementar compiladores de linguagem funcionais como ML [18]. Também foi usada a idéia de funções especiais [19]. Estas funções são consideradas como supercombinadores especiais.

Para efetuar a compilação de um supercombinador foi definida uma função que recebe a definição de um supercombinador, já analisado léxica e sintaticamente, como argumento e retorna uma definição de função na linguagem C, contendo o código do corpo do supercombinador compilado em código-G, caso o supercombinador corresponda à uma função ordinária, ou então, o código do corpo do supercombinador traduzido para a sintaxe da linguagem C, caso o supercombinador corresponda a uma função especial. Os detalhes de implementação do *back-end* podem ser encontrados em [4, 8].

5.1 *Front-End*

O *Front-End* do compilador para $\mathcal{SCRIPT}$ está organizado em três passos, conforme ilustra a Figura 1. Duas características de $\mathcal{SCRIPT}$ que influenciaram a decisão de fazer essa etapa em três passos foram: (a) a necessidade de identificação de rotinas semânticas recursivas, que exigem inspeção no texto, e (b) a possibilidade de declarações aparecerem após o uso.

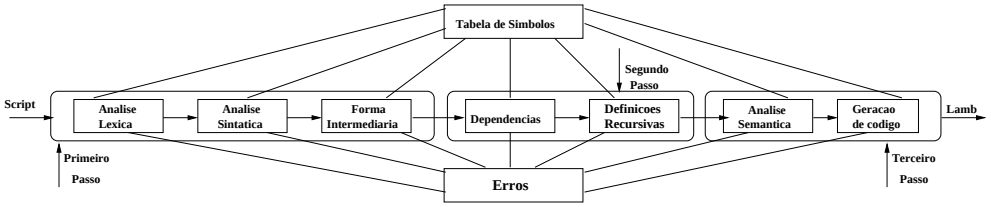

Figure 1: Divisão das Fases em Cada Passo do *Front-End*

No primeiro passo está o processamento do texto fonte, a análise léxica e sintática, a coleta dos domínios e variáveis e sua inserção na tabela de símbolos. Os dados da tabela de símbolos são armazenados em uma forma intermediária e usadas como entrada para o segundo passo, durante o qual as definições recursivas e as dependências entre as definições são coletadas, sendo novamente necessária uma forma intermediária para guardar a tabela de símbolos para o terceiro passo. Este passo realiza a análise semântica, verificação de tipos e geração do código $\mathcal{LAMB}$.

Cada um desses passos compreende a construção de uma gramática [5] com rotinas semânticas associadas, que servem de entradas para o Yacc [15]. Símbolos encontrados ao longo do código de entrada que foram inseridos na tabela e informações sobre definições recursivas existentes no código fonte, que

[5] O termo gramática nesta seção deverá ser entendido como a gramática com rotinas semânticas associadas que serve de entrada para o Yacc.

foram tratadas pela primeira e segunda gramáticas, são armazenados em uma forma intermediária para serem consultadas pelo terceiro passo. Com isso, a gramática do terceiro passo fica responsável pelas ações semânticas de verificação dos tipos, pois os domínios e variáveis já estão todos instalados e definidos na tabela. Essa terceira gramática também realiza a geração de $\mathcal{LAMB}$, contando com informações sobre definições que são recursivas, para a decisão na montagem do código final.

O segundo passo faz a análise de dependência entre as funções e variáveis declaradas em um programa $\mathcal{SCRIPT}$. Uma análise do fluxo dos dados nas definições é importante para verificar as condições de recursividade ou não dos símbolos, com o objetivo de gerar um código $\mathcal{LAMB}$ mais eficiente. Para auxiliar na verificação destas dependências, foram criadas listas para indicar que identificadores aparecem livres em quais definições ou funções. Esta informação passou a fazer parte da tabela de símbolos, ajudando na etapa de geração de código objeto. Com a coleta dessas informações, a recursividade dos símbolos é então analisada e é incorporada aos atributos referente a cada símbolo na tabela.

No terceiro passo é realizada a análise semântica, verificação de tipos e geração de código $\mathcal{LAMB}$. Para a verificação de tipos foi implementado um tipo abstrato correspondente à gerência da estrutura de domínios de $\mathcal{SCRIPT}$ que procura oferecer serviços de determinação de equivalência e compatibilidade de tipos ou domínios. A disciplina de tipos de $\mathcal{SCRIPT}$ foi definida de tal forma que possibilitasse sua verificação em tempo de compilação dado que os objetos não podem carregar as informações de seus tipos para a linguagem objeto.

A análise semântica verifica o programa fonte procurando por erros semânticos estáticos e deve acumular informações a respeito dos tipos das construções, por meio de atributos associados a cada identificador, com o objetivo de auxiliar na fase de geração de código. Ao definir o projeto da compilação, cada símbolo não-terminal da gramática deve ter ligado a ele um conjunto de atributos. E a cada produção deve-se associar um conjunto de regras semânticas, regras que irão manipular os valores dos atributos associados aos símbolos que aparecem na produção.

Associado aos passos do *front-end* temos o módulo gerenciador da tabela de símbolos responsável por quatro serviços. O primeiro deles é a própria tabela de símbolos. O segundo serviço é a gerência da tabela de símbolos dentro de cada módulo. A estrutura de dados utilizada para representar a tabela de símbolos é a árvore binária. A tabela de símbolos de todo o programa pode ser entendida como uma floresta, pois foi implementada de forma a conter todas as árvores de símbolos dos vários módulos (escopos) existentes, e seu acesso é feito por meio de um vetor de apontadores para árvores.

O terceiro serviço é a gerência de escopo dos símbolos. Quando ocorre a procura de um nome do texto fonte na tabela de símbolos, a localização da declaração apropriada do nome deve ser retornada. As regras de escopo da linguagem fonte devem ser próprias para determinar qual é a declaração apropriada. Esta gerência foi implementada por um mecanismo que funciona como uma pilha, armazenando os escopos e utilizando o método LIFO (*Last in, First Out*) para controlar a entrada e saída nos mesmos. O quarto serviço deste módulo é a gerência de importação e exportação de símbolos. Ao chegar ao fim de um módulo, o mecanismo de compilação deve reconhecer as variáveis e funções deste módulo que podem ser exportadas para outros módulos. Para facilitar esse reconhecimento podemos classificar a Tabela de Símbolos em três tipos de tabelas. O primeiro tipo, a *Tabela de Símbolos Global*, guarda informações do escopo principal do módulo. Um outro tipo, a *Tabela de Símbolos Locais*, representa a árvore dos símbolos de um escopo mais interno do módulo. Um mesmo módulo pode ter somente uma Tabela de Símbolos Global, mas pode ter zero ou mais Tabelas de Símbolos Locais. Com esta implementação, a busca por variáveis ou funções exportáveis fica mais fácil pois ela se restringe à Tabela de Símbolos Global. É neste momento que aparece o terceiro tipo de tabela de símbolos, a *Tabela de Símbolos Exportáveis*. Monta-se a árvore com os símbolos que são exportáveis neste módulo e a guarda para ser pesquisada sempre que necessário.

Ao entrar no módulo M1 o compilador deve reconhecer os módulos importados por ele. Para cada um desses módulos, o compilador deve buscar a árvore com os símbolos disponíveis para exportação

para verificar as exportações disponíveis e as importações necessárias. A árvore referente à *Tabela de Símbolos Global* do módulo M1 é inicializada com esses símbolos importados. Depois da fase de inserção dos símbolos importados, segue-se a fase de inserção de símbolos do próprio módulo, caracterizando a Tabela de Símbolos Global. Por fim, ocorre a montagem das árvores das *Tabelas de Símbolos Locais*.

A tabela de símbolos para programas $\mathcal{SCRIPT}$ é montada durante o primeiro e segundo passos. As informações nela contidas são necessárias durante a verificação de tipos para assegurar a correta equivalência e compatibilidade dos tipos em cada operação de construção de produções. Durante a geração de código essas informações são consultadas, visando a análise de escopo das variáveis utilizadas e a geração de definições, que utilizam as informações de recursividade e dependência.

5.2 *Lambda-Lifting*

Esta etapa do compilador trata da compilação de programas escritos em $\mathcal{LAMB}$ para $\mathcal{SUPER}$, um cálculo de supercombinadores [3]. Este processo é conhecido como *Lambda-Lifting* e consiste em transformar abstrações lambda que contenham variáveis livres em abstrações equivalentes que não contenham variáveis livres e por isso podem sem compiladas para uma seqüência fixa de código.

A estratégia usada pelo compilador de $\mathcal{LAMB}$ é ilustrada na Figura 2. A função $\mathcal{LL}$ ($\mathcal{L}$ ambda $\mathcal{L}$ ifting), também chamada esquema de compilação [6, 4] é definida como $\mathcal{LL} : ExpL \mapsto ExpS$, onde $ExpL$ representa uma expressão $\mathcal{LAMB}$, e $ExpS$ representa uma expressão $\mathcal{SUPER}$ de supercombinadores. O esquema $\mathcal{LL}$ utiliza dois esquemas de compilação como funções auxiliares, os esquemas representados pela função $\mathcal{P}$ e pela função $\mathcal{L}$. $\mathcal{P}$ tem a função de processar e eliminar padrões da expressão dada.

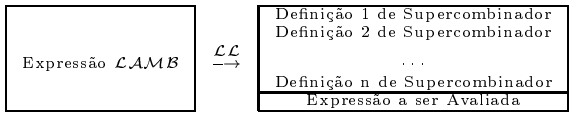

Figure 2: Esquema do Processo de Compilação de $\mathcal{LAMB}$

A função $\mathcal{P}$ tem tipo $\mathcal{P} : ExpL \mapsto ExpB$, onde $ExpL$ é o domínio das expressões $\mathcal{LAMB}$, e $ExpB$ o domínio das expressões $\mathcal{LAMBASIC}$, que representa um cálculo lambda estendido, mas sem padrões.

A função $\mathcal{L}$ faz o *lifting* da expressão lambda. A função $\mathcal{L}$ tem tipo: $\mathcal{L} : (ExpB \times B \times N) \mapsto (ExpS \times ExpS \times Free)$, onde B é o domínio das variáveis ligadas, armazenadas em uma pilha na forma (ide ∈ Ide, n ∈ N, t ∈ Type, x ∈ Ide); o 'ide' representa o identificador de tipo 't' ligado no nível léxico 'n' com nome de supercombinador 'x' associado; N representa o domínio dos inteiros, e; Type representa os tipos primitivos Q, N, T e '?'. Por exemplo, uma tupla (x,3,Q,?) indica que o identificador 'x' tem tipo 'Q', está ligado no nível léxico 3 e não tem nome de supercombinador associado.

Ainda analisando a assinatura de $\mathcal{L}$, o terceiro domínio, identificado por N, representa o nível léxico corrente, fazendo alusão ao escopo da expressão. $Free$ é um domínio de tuplas (ide ∈ Ide,n ∈ N,t ∈ Type,x ∈ Ide) representando as variáveis livres identificadas na expressão; 'x' é o nome do supercombinador associado a "ide", se houver, senão é o próprio idenficador. Por exemplo, uma tupla (y,2,y,N) indica que o identificador 'y' do tipo 'N' ocorre livre no nível léxico 2 e é representado por ele mesmo, não é um supercombinador. Já na tupla (y,0,$S1,N), o identificador 'y' ocorre livre no nível léxico 0, sendo caracterizado como um supercombinador com nome $S1.

Internamente, os programas em $\mathcal{LAMB}$ são representados por meio de três tipos de estruturas: representação de expressões, representação de padrões e representação de pares formados por lados esquerdo e direito de uma definição LET/DEF.

O processo de tradução de $\mathcal{LAMB}$ para supercombinadores é feito em dois passos, porque a presença de padrões na linguagem fonte inviabiliza a geração do código em apenas um passo. O primeiro passo desta etapa se encarrega apenas do casamento de padrões, ou seja, elimina os padrões que eventualmente possam estar presentes nas expressões. Aquelas expressões que não apresentam padrões, são apenas copiadas para que se possa realizar o *lambada-lifting* no segundo passo. O primeiro passo usa o esquema $\mathcal{P}$ definido anteriormente e representa uma função que recebe uma árvore com padrões e retorna uma outra árvore equivalente sem padrões. Os padrões podem ocorrer de três formas em expressões $\mathcal{LAMB}$: (1) como parâmetro formal de abstração lambda, (2) no lado esquerdo de uma definição LET ou DEF e (3) como parâmetro do operador IS. No segundo passo realiza-se o *Lambda-Lifting* efetivo, ou seja, transforma a expressão gerada em uma coleção de supercombinadores mais uma expressão a ser avaliada. Para este passo usa-se a função $\mathcal{L}$ definida anteriormente.

Isto posto, resumidamente, é possível identificar o processo de tradução de $\mathcal{LAMB}$ para supercombinadores como uma composição de de funções: a função $\mathcal{L}$, que realiza o *Lambda-Lifting* efetivo, aplicada ao resultado da função $\mathcal{P}$, que realiza o casamento de padrões. Seguindo esse raciocínio, todo o processo de tradução é dado por:

$\mathcal{LL}$ (ExpL) = LET s_0 = definição de supercombinadores auxiliares
LET (s_1,e_1,f_1) = $\mathcal{L}$ ($\mathcal{P}$ (ExpL)) (ϕ) (0)
IN $s_0 \| s_1 \| e_1$

onde ϕ representa, neste caso, uma pilha inicialmente vazia e $\|$ é o operador de concatenação.

6 Resultados Obtidos

Nesta seção apresentamos três exemplos com os resultados obtidos da execução do compilador de $\mathcal{SCRIPT}$. Mostrados os programas $\mathcal{SCRIPT}$, $\mathcal{LAMB}$, $\mathcal{LAMBASIC}$ e $\mathcal{SUPER}$.

No Exemplo 1 temos um programa em $\mathcal{SCRIPT}$ e o resultado de sua compilação para a linguagem C. O Exemplo 2 mostra como se dá a tradução de constantes e operadores binários. O principal ponto a ser observado é a mudança de operador na forma infixa para a forma prefixada aos seus operandos. O Exemplo 3 ilustra a compilação de abstração lambda. As traduções mais interessantes são os supercombinadores $S38 e $S39, que foram obtidos a partir das duas abstrações lambda LAM x e LAM nn24, respectivamente, do programa $\mathcal{LAMBASIC}$.

– SCRIPT
```
MODULE C
DOMAINS
Fdom = N -¿ N;
x:N;
f:Fdom

DEFINITIONS
DEF f =
 LAM x. x PLUS 7
END C
```

– LAMB
```
LET C =
LET f =
 LAM x . x PLUS 7
IN < >
```

–LAMBASIC
```
LET C =
LET f =
 ( LAM x . x PLUS 7 )
IN < >
```

–SUPER
```
$S26 x = PLUS x 7
$S25 = ( $S26 )
$MOD_LAMB = ( )
```

–Programa C
```
#include "includes.h"
#include "globvars.h"

void $s26 (void) {
pushbasic(7);
push(1);
eval();
get();
plus();
updint(2);
pop(1);
returng();
}

void $s25 (void) {
pushglobal(1, $s26);
tuple(1);
update(1);
unwind();
}

void $mod_lamb (void) {
tuple(0);
update(1);
unwind();
}
```

Exemplo 2: Operadores Binários e Constantes	Exemplo 3: Abstração Lambda e Constantes
– SCRIPT MODULE A DOMAINS Ndom = N DEFINITIONS DEF ndom1 = 11 DEF ndom2 = ndom1 PLUS 11 DEF ndom3 = ndom2 MULT 11 END A **– LAMB** LET A = LET ndom1 = 11 IN LET ndom2 = ndom1 PLUS 11 IN LET ndom3 = ndom2 MULT 11 IN < > **– LAMBASIC** LET A = LET ndom1 = 11 IN LET ndom2 = ndom1 PLUS 11 IN LET ndom3 = ndom2 MULT 11 IN < > **– SUPER** $S35 = 11 $S36 = PLUS $S35 11 $S37 = MULT $S36 11 $MOD_LAMB = ( )	**– SCRIPT** MODULE B DOMAINS F = (N) – > N; Ndom = N; x: N DEFINITIONS DEF ndom1 = 11 DEF t1 = "FF" DEF f1 = LAM (x). x EQ 0 – > 1, 2 END B **– LAMB** LET B = LET ndom1 = 11 ALSO t1 = "FF" ALSO f1 = LAM (x) . x EQ 0 – > 1 , 2 IN < > **– LAMBASIC** LET B = LET ndom1 = 11 IN LET t1 = "FF" IN LET f1 = (LAM nn24.(LAM x. x EQ 0 – > 1 , 2) (nn24 EL 1)) IN < > **– SUPER** $S35 = 11 $S36 = "FF" $S38 x = IF EQ x 0 1 2 $S39 nn24 = AP ($S38) (EL nn24 1) $S37 = ($S39) $MOD_LAMB = ( )

7 Conclusão

Em relação às linguagens expostas neste artigo, $\mathcal{SCRIPT}$ além de se enquadrar dentro do paradigma funcional, e ter características de linguagens orientadas por objetos, ela possui uma maneira intuitiva de programar, já que as instruções são especificações matemáticas tradicionais. Além da rapidez com que se pode construir programas de propósito geral, $\mathcal{SCRIPT}$ apresenta um grande diferencial, ela apresenta facilidades para especificação denotacional de linguagens de programação.

O trabalho descrito neste artigo relata nossa experiência na tradução de $\mathcal{SCRIPT}$ para código C feita em três etapas. O *Front-End*, o *Lambda-Lifting* e o *Back-End*. Especificamente, mostramos aqui como foram tratados pelo *front-end* alguns aspectos importantes da implementação do sistema de tipos de $\mathcal{SCRIPT}$e como foi feito o *lambda-lifting*. A integração entre ss duas primeiras etapas está finalizada e testada. A parte do *Back-End* também já está integrada às demais, contudo ainda mais testes serão necessários para tornar o compilador mais robusto.

References

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

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

[3] Costa, Marco Rodrigo. *Compilação de um Cálculo Lambda Estendido para Supercombinadores.* Tese de Mestrado, DCC/UFMG, 2000.

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

[5] Turner, David *Miranda - a non-strict functional language with polymorphic types.*, Conference on Functional Programming Languages and Computer Architecture, Springer Verlag, 1985.

[6] Peyton, J. *The Implementation of Functional Programming Languages.* C.A.R HSEditor, 1987.

[7] Mosses, P. D. *Mathematical Semantics and Compiler Generation*, PHD thesis, Oxford, 1975.

[8] Maia, Marcelo A., Bigonha, R. *Implementando uma Linguagem Funcional Pura com uma Máquina-G* Anais do I SBLP, pag.141-154, Belo Horizonte, 1996.

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

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

[11] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers Principles, Techniques, and Tools.* A.W. 1986.

[12] Peterson, John and Hammond, Kevin and Wadler, Philip and Jones, Simon Peyton and Augustsson, Lennart and Boutel, Brian and Burton, Warren and Fasel, Joseph and Gordon, Andrew D. *Haskell, A Non-strict, Purely Functional Language*, Haskell 1.4 Report, yale, 1997.

[13] Johnsson, T. *Efficient Compilation of Lazy Evaluation*, ACM Montreal, 07/1984.

[14] Landin, P.J. *The Mechanical Evaluation of Expressions*, journal tcj, Volume 6, 1964.

[15] Johnson, S. *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, N. J., 1978.

[16] Turner, D. A. *A New Implementation Technique for Applicative Languages*, SPE, V. 9, 1979.

[17] Lins, R.D. *Categorical Multi-Combinators*, Functional Programming and Computer Architecture, Gilles Kahn, LNCS 274, Springer Verlag, setembro de 1987.

[18] Augustsson, L. *A Compiler for Lazy ML*, ACM SLFP, Austin, 08/1984.

[19] Lins, R.D. and Lira, B.O Γ *CMC: A Novel Way of Compiling Functional Languages*, XII Congresso da Sociedade Brasileira de Computação, Setembro de 1992.