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

Fabíola F. Oliveira, Roberto S. Bigonha,
Mariza A. S. Bigonha, Marco R. Costa

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

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

Resumo

O objetivo deste artigo é apresentar um trabalho sobre compilação de programas escritos em uma linguagem funcional, $\mathcal{SCRIPT}$, para a linguagem C. $\mathcal{SCRIPT}$ apresenta características de linguagens orientadas por objetos e atende a propósitos gerais, visando prover uma notação adequada para permitir que descrições de semântica denotacional possam ser processadas. Dentre fatores que contribuíram para a escolha da linguagem C como linguagem objeto devemos destacar sua larga utilização, seu poder de expressão e o fato de poderem ser encontrados vários compiladores eficientes no mercado. É interesante mencionar que o referido trabalho faz parte de um projeto maior 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. Este artigo trará, além de uma visão geral de todo o trabalho, uma enfase na solução para que os mecanismos de polimorfismo e funções virtuais pudessem ser efetivados em $\mathcal{SCRIPT}$, mecanismos estes implementados na etapa do *Front-End* do compilador.

Palavras-Chaves: compilador, linguagem funcional, $\mathcal{SCRIPT}$, polimorfismo, funções virtuais

1 Introdução

O interesse em verificar se programas de computador possuem o comportamento que se espera que tenham existe desde o advento dos computadores. À medida em que o tamanho e a complexidade desses programas vêm crescendo, também cresce a importância de se assegurar que

os mesmos se comportem de forma confiável. Naturalmente, as atenções se voltam para o problema de especificar precisamente o que constitui um comportamento confiável e para a tarefa de desenvolver um método minucioso de verificação que garanta que um programa sempre corresponderá às especificações. Isso levou à necessidade de algum tipo de ferramenta para análise de linguagens que tivesse como base um modelo preciso e seguro. Estudos em busca de uma forma para especificação formal de linguagens evoluíram no sentido de alcançar completude, consistência, precisão, usabilidade, eliminação de ambigüidade e facilidade de entendimento.

Introduzindo-se o formalismo, cuidados especiais devem ser considerados, pois, quanto mais formal for a definição da semântica da linguagem, mais se tornará complicada a construção do compilador. Portanto o formalismo deve ser o mais claro, objetivo e legível possível, procurando tornar mais direta a construção do compilador e mais perceptível o significado das construções dessa linguagem. E, ao considerar as necessidades levantadas, poderão ser encontradas nas linguagens funcionais características relevantes para se chegar ao formalismo almejado.

Passamos, então, a dar ênfase ao estudo de semântica denotacional que tradicionalmente tem sido descrita como a teoria sobre os significados reais de programas, ou, como a teoria sobre **o que** os programas denotam [20], [21]. Em muitas situações, denotações têm sido construídas com a ajuda de funções do universo matemático. Mas ainda existe uma visão alternativa, pela qual a semântica denotacional é vista como uma ferramenta para realizar *transformações* partindo de um sistema formal para chegar a outro. Esta segunda forma de entender a semântica denotacional tem se tornado cada vez mais popular, contribuindo para o surgimento de um paradigma computacional novo. A idéia fundamental é definir tanto objetos matemáticos para cada entidade da linguagem quanto uma função que mapeie instâncias desta entidade em instâncias do objeto matemático [22] [23].

Por ser um conceito que pode ser visto como um mapeamento de um programa diretamente para o seu significado, a semântica denotacional é útil para especificar a funcionalidade de novas construções de linguagem que ainda não foram implementadas, o que a torna uma grande contribuição em projetos de geração automática de compiladores. De acordo com [24] semântica denotacional tem muito valor do ponto de vista de um projetista, assim como as definições axiomáticas servem ao programador e a semântica operacional se adequa mais aos implementadores de linguagem. E é com esse enfoque da semântica denotacional que trabalhamos objetivando um método formal de semântica de linguagens para a descrição de $\mathcal{SCRIPT}$ [1], a linguagem fonte do presente projeto. A partir da criação da linguagem, o passo seguinte para ela ser utilizada passou a ser a construção do seu compilador.

$\mathcal{SCRIPT}$ é uma linguagem funcional orientada por objetos, usada para definições de semântica denotacional de linguagens de programação. Além de possuir os elementos básicos de uma linguagem funcional pura, $\mathcal{SCRIPT}$ possui também extensões como controle de visibilidade, encapsulamento, herança, polimorfismo, ligação dinâmica (*dynamic binding*), dentre outros, que são recursos que pedem cuidados especiais durante a construção de seu compilador.

Uma característica importante nos programas em $\mathcal{SCRIPT}$ é a facilidade de escrevê-los em módulos que podem vir a ser independentes ou que podem trocar informações entre si. Essa divisão em módulos foi possibilitada com o uso de técnicas de importação e exportação de símbolos. Dentro desse enfoque também foram levados em consideração as diferentes possibilidades de visibilidade de um símbolo que passa a ser caracterizado como *aberto* ou *fechado* quando ocorre a importação ou exportação. Com essa denominação identifica-se um símbolo

fechado como sendo aquele que só terá o seu nome conhecido no módulo no qual foi importado. Ao considerar um símbolo denominado *aberto* tanto seu nome quanto sua *assinatura de domínio* estarão disponíveis para os módulos que o importarem.

A experiência relatada nesse artigo trata do estudo e da construção de um compilador eficiente para $\mathcal{SCRIPT}$. O ambiente em que esse compilador está inserido tem por objetivo diminuir o esforço do programador para composição das descrições de uma linguagem de programação. A implementação do compilador para $\mathcal{SCRIPT}$ foi dividida em três fases: a compilação para um cálculo lambda estendido, o *Front-End* [5], o processo de *Lambda-Lifting* [6] e a geração de código, o *Back-End* [7]. A Figura 1 apresenta a sua arquitetura. Enfatizaremos os pontos mais instigantes durante a fase de *Front-End*, centrando no aspecto da solução para os mecanismos de polimorfismo e funções virtuais, o que garante a $\mathcal{SCRIPT}$ a presença de associação dinâmica de funções.

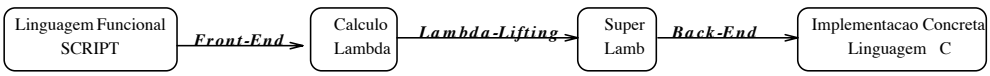

Figure 1: Compilação de $\mathcal{SCRIPT}$.

A primeira etapa, o *Front-End,* trata da compilação da linguagem fonte, $\mathcal{SCRIPT}$, para $\mathcal{LAMB}$, que é uma linguagem funcional de semântica não-estrita e que provê funções de ordem mais alta [4, 2]. Ela é utilizada para representar funções matemáticas tais como as funções de definições usadas em semântica denotacional. A segunda etapa trata do gerador de código, o *Back-End,* e consiste na compilação de Supercombinadores para código em linguagem C [8] utilizando uma máquina-G estendida [6]. A terceira etapa constitui a etapa intermediária entre o *Front-End* e o *Back-End* desse compilador. Ela consiste na realização do *Lambda-Lifting,* a tradução das expressões lambda ($\mathcal{LAMB}$) para Supercombinadores [6].

2 Técnicas de Compilação de Linguagens Funcionais

Linguagens funcionais sempre foram um assunto interessante para estudos e pesquisas. Porém, até pouco tempo atrás esses estudos e pesquisas não tinham atingido um patamar considerável com resultados favoráveis, devido, principalmente, a ineficiência dos compiladores para tais linguagens, se comparados a compiladores de linguagens tradicionais. Por essa limitação, as expectativas em relação ao futuro das linguagens funcionais eram pequenas.

O desafio de desenvolver um compilador eficiente para linguagens funcionais, especificamente para $\mathcal{SCRIPT}$, é que motivou a produção deste projeto. Em geral, a implementação de tais compiladores tende a ser conduzida de uma forma diferente da implementação dos compiladores de linguagens imperativas. As implementações mais recentes de compiladores para linguagens funcionais têm sido radicalmente diferentes das tecnologias convencionais para compiladores [11]. As mudanças mais significativas têm surgido ultimamente e se destacam pelo uso de novas técnicas que objetivam tornar os compiladores mais eficientes. A título de exemplo, linguagens que já apresentam tais mudanças são ML [25], Ponder [26], Chalmers Lazy ML [13], Haskell [12].

Utilizando estas técnicas, duas abordagens destacam-se na busca de implementações eficientes para linguagens funcionais. A primeira delas baseia-se no esquema de **ambientes**, e a outra tem seu ponto principal na redução de grafos.

Ao considerar as técnicas baseadas em **ambientes**, a primeira máquina abstrata desenvolvida para reduzir expressões λ-*calculus* foi a máquina SECD, descrita em [14]. Já a abordagem baseada nas reduções de grafos tem como ponto importante o trabalho de Turner [16], que descreve uma técnica para implementação de linguagens funcionais 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 os combinadores propostos por Turner, tais como os multicombinadores categóricos [17] e os supercombinadores. 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 transformada em um supercombinador.

A máquina-G é uma técnica para implementação eficiente de linguagens funcionais *lazy* desenvolvida 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.

Um dos primeiros sistema para geração de compiladores a utilizar a semântica denotacional foi proposto por Mosses [9], que tem como base uma semântica formal da linguagem de programação. O projeto da linguagem ML por Robin Milner [32], inicialmente uma metalinguagem, foi idealizada visando ser usada em um sistema de verificação de programas. Sua característica marcante está no fato de ser uma linguagem de programação funcional, com declaração de tipos, fortemente tipada, com avaliação estrita, e que usa inferência de tipos. Já a técnica básica de compilação utilizada pelos desenvolvedores de compiladores ML foi a máquina-G.

Projetada na mesma época da linguagem ML, Miranda [5] [1] também é uma linguagem de programação puramente funcional integrada a um ambiente de programação interativo desenvolvido especificamente para ela. Podemos encontrar em [6] uma técnica de compilação para linguagens funcionais que transforma um subconjunto da linguagem Miranda em uma versão enriquecida do λ-*calculus* e depois transforma esta expressão em uma expressão ordinária [2].

Nas décadas de 1970 e 1980, ainda não era possível encontrar uma linguagem de programação puramente funcional e não-estrita que pudesse ser considerada uma linguagem padrão. Para tanto, foi criado um comitê com o objetivo de projetar tal linguagem. Nascia assim como resultado desta análise, a linguagem Haskell. A partir do surgimento de Haskell começaram a aparecer compiladores para esta linguagem. Usando a técnica da máquina-G sem espinha e sem *tag*, e com coletor de lixo, a Universidade de Glasgow desenvolveu seu compilador para a linguagem Haskell. Já o compilador para Haskell da Universidade de Yale está inserido em um ambiente de programação interativo e flexível, que procurou incorporar otimizações que incluem a análise de estringência (*strict analysis*), e o compilador da Universidade de Chalmers

[1] Miranda é marca registrada da *Research Software Limited.*

[2] λ-*calculus*, livre de construções auxiliares.

para Haskell tem por base o compilador de LazyML de Augustsson e Johnsson.

3 A Linguagem Script

SCRIPT [27] é uma linguagem orientada por objetos cujas descrições de semântica denotacional podem ser efetivamente executadas e depuradas com o auxílio de computadores. A linguagem *SCRIPT* apresenta várias construções oriundas dos sistemas LDS [1] e SDL de Petter Mosses [9]. Ela foi acrescida com operadores e construções cujo objetivo é alcançar as questões citadas a seguir. Possui também facilidades advindas das linguagens funcionais Miranda e ML. Um programa em *SCRIPT* é formado por módulos, que podem importar e/ou exportar funções ou variáveis entre si, e apresenta uma disciplina baseada em equivalência estrutural [29] e polimorfismo de inclusão [30].

Além das características descritas, *SCRIPT* é uma linguagem funcional que possui facilidades para encapsulamento, herança e associação dinâmica, visando prover uma notação adequada para formular descrições semânticas denotacionais de linguagens de forma modular.

Nesta seção, por razões de completude, será mostrado um resumo da descrição da linguagem *SCRIPT*, cujos detalhes podem ser encontrados em [27].

Em *SCRIPT* um domínio pode ser definido como uma entidade matemática para representar conjuntos com ordem parcial, com um elemento mínimo *bottom* representado pelo símbolo $\perp$. É importante salientar que o valor especial $\perp$, que não pode ser representado diretamente em *SCRIPT*, serve para modelar a semântica de não-determinismo. Os domínios padrões em *SCRIPT* são: o domínio básico dos números inteiros, o domínio básico das cadeias de caracteres, ou *strings*, o domínio básico dos valores lógicos e o domínio básico dos valores indefinidos, que são representados pelo símbolo " ? ". Esses domínios padrões possuem, relacionados a eles, algumas operações pré-definidas. O domínio *indefinido* é utilizado para indicar o valor de qualquer expressão em *SCRIPT* que não possua um sentido semântico identificável.

SCRIPT possibilita ao usuário da linguagem construir novos tipos de domínios. Esses novos domínios podem expressar domínios ainda mais complexos com o objetivo de modelar propriedades sintáticas e semânticas de outras linguagens de programação.

Para apresentar as possíveis possibilidades de combinações de domínios, convencionamos que d, d_1, d_2, ..., d_n são expressões quaisquer de domínio, q é uma *quotation* e a_1, a_2, ..., a_n são variáveis.

- **União de domínios**: a representação de uma **união** de domínios se dá com o símbolo **|**, e pode ser exemplificada pela expressão d_1 | d_2 **|** ... **|** d_n .

- **Domínio de tuplas**: um domínio de tuplas genérico, **n-tupla**, pode ser denotado por (a_1**:**d_1**,** a_2**:**d_2**, ...,** a_n**:**d_n), e seu i-ésimo elemento tem domínio denominado por d_i e pode ser acessado com o identificador a_i, para 1 <= i <= n.

- **Domínios de listas**: o domínio das listas pode ser apresentado de três formas distintas. Uma delas é a declaração d*, que representa uma lista finita podendo ser vazia cujos elementos estão no domínio d. Outra forma é a declaração de lista finita d+, não vazia,

com elementos em d. Por último, uma instanciação de uma lista é possível através de `<` a_1, a_2, ..., a_n `>`, em que cada a_i pertence a um mesmo domínio d, posto que 1 `<=` i `<=` n.

- **Domínios de funções contínuas**: o domínio de uma função contínua é caracterizada por d_1 `->` d_2, e representa o mapeamento de um domínio d_1 em um domínio d_2.
- **Domínios de nós**: uma expressão de domínio com a forma `[` d_1 d_2 ... d_n `]` denota o domínio dos nós das árvores de sintaxe com o mesmo rótulo. O rótulo é necessário para a separação dos nós e são definidos pela *quotation* QUOTE `<` q_1, q_2, ..., q_m `>`, onde m `<=` n, e q_i é o nome do domínio d_i.

Já as expressões que podem ser encontradas em um programa $\mathcal{SCRIPT}$ são:

1. **constantes literais**, valores que pertencem aos domínios padrões **N**, **Q**, **T** e **?**;
2. **variáveis**, que denotam membros de domínios;
3. **expressões inteiras**;
4. **expressões de** *quotation*;
5. **expressões lógicas**, expressões cujos valores de retorno são `TT`, `FF`, `?` ou $\perp$;
6. **expressões de listas**, expressões construídas com `"*"`, `"+"`, `"<"` e `">"` ;
7. **expressões de tuplas**, expressões caracterizadas pelo uso de parênteses, de reconhecimento não trivial;
8. **expressões de nós**, expressões caracterizadas pelo operador `"[...]"`;
9. **expressões de comparação**, expressões apresentadas em três formas distintas. Uma forma serve para analisar se duas expressões apresentam o mesmo valor. Outra forma compara se duas expressões apresentam valores distintos. Uma última forma tem seu foco na estrutura de uma expressão e de um padrão, e não no valor encontrado após a análise das expressões, ou seja, a preocupação dessa terceira forma é com a lei de formação de uma expressão;
10. **expressões de padrões**, pode ser um identificador, uma constante literal, uma combinação de padrões mais simples com operadores construtores de padrões, ou o valor **?**;
11. **expressões condicionais**: t `->` e_1, e_2. Essa expressão corresponde à expressão e_1 caso t denote verdadeiro; corresponde à expressão e_2 caso t denote falso; corresponde a **?** se t denota **?** e corresponde a $\perp$ se t representa $\perp$;
12. **abstrações LAMBDA** são funções não-recursivas anônimas da forma `LAM` $x.e$;

13. **abstrações de padrão**, que tem o objetivo de permitir a ocorrência de padrões em expressões do tipo `LAM` $p.e$ com p sendo uma expressão de padrão, fornecendo à linguagem uma forma de extrair componentes em um valor;

14. **expressões CASE**, maneira na qual é possível pesquisar a estrutura ou forma que um valor está assumindo em determinado momento;

15. **expressões LET**, forma de dividir uma expressão em partes menores, ajudando na compreensão do significado de tal expressão;

16. **aplicações de funções**, expressões seqüenciais, que combinam expressões *SCRIPT* com operadores de seqüenciamento. Exemplo: uma expressão $f\ g\ e$, que passa a ser construída com a seguinte forma: $(f\ (\ g\ ))\ (\ e\ )$. O domínio de cada parâmetro atual precisa ser compatível com o domínio do parâmetro formal correspondente na passagem de parâmetros.

17. Qualquer combinação de expressões mais simples utilizando os devidos operadores.

A estrutura de uma definição formal em *SCRIPT* é composta por um módulo principal denominado Project, que contém a função principal, e zero ou mais módulos secundários externos. Na função principal é apresentado o contexto geral da definição de semântica denotacional. Os módulos secundários podem ser compilados juntamente com o módulo principal ou podem vir de uma biblioteca de módulos já previamente compilados.

A meta principal a ser atingida com o uso de módulos é a possibilidade de agrupar entidades correlatas, como domínios ou funções, para que elas possam ser mais facilmente usadas por outros módulos. Com isso entidades que não precisam ser conhecidas pelos usuários de um módulo poderão ficar ocultas. Somente informações relevantes para o usuário são apresentadas.

O objetivo do módulo **PROJECT**, o módulo principal, é definir os parâmetros e o ambiente sobre o qual as definições formais devem ser trabalhadas. Os módulos secundários podem ser de dois tipos: **SYNTAX** e **MODULE**. No primeiro são apresentadas três seções: **DOMAINS**, que especifica os domínios de símbolos não-terminais, **SYNTAX**, onde são definidas as sintaxes concretas e abstratas, e a seção **LEXIS**, para a declaração das estruturas dos símbolos léxicos. No último tipo de módulo, **MODULE**, podem ser encontradas quatro seções. A seção **EXPORTS** apresenta as entidades do módulo corrente que estão sendo exportadas, assim como seus níveis de encapsulamento, em **IMPORTS** são feitas as importações de símbolos definindo seus graus de visibilidade e relacionando os módulos de onde eles serão importados, já **DOMAINS** é a seção onde são encontradas as declarações de domínios, variáveis e funções, e a última seção é **DEFINITIONS**, onde são definidas funções e outros valores.

Em relação à passagem de parâmetros em *SCRIPT* uma informação importante é que os argumentos só são efetivamente avaliados quando necessários. Isso porque *SCRIPT* é uma linguagem pura, preguiçosa (lazy, não-estrita). Também é importante ressaltar que em *SCRIPT* é possível definir funções associadas a domínios de tuplas, e que sobrecarga (*overloading*) de funções é permitida, e a ambiguidade de nomes é resolvida verificando-se os domínios dos argumentos que cada uma delas recebe.

4 O Compilador de *SCRIPT*

Começando pelo *Front-End* do compilador para *SCRIPT*, podemos caracterizá-lo pelas fases de análise léxica e sintática, criação e geração da tabela de símbolos e verificação de tipos até a geração do código intermediário para a linguagem *LAMB*. Essas fases foram divididas em três diferentes passos. A razão desta organização advém do fato de programas *SCRIPT* poderem ser escritos como uma seqüência não ordenada de comandos de declaração e comandos que usam as declarações. Devido a esse não ordenamento, não é possível garantir que as informações necessárias para produzir uma instrução no código objeto final em um determinado momento da compilação estará disponível no momento desejado. As duas principais situações que levaram a isso foram a ocorrência de declarações podendo aparecer após o uso e a possibilidade de encontrar rotinas semânticas recursivas, ambas exigindo inspeção no texto para avaliá-las.

O primeiro passo processa o texto fonte realizando análise léxica e sintática, 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. Já a responsabilidade pela identificação das definições recursivas e das dependências entre as definições, está a cargo do segundo passo. No fim deste passo, novamente a tabela de símbolos é guardada na mesma forma intermediária utilizada entre o primeiro e segundo passos. Após a geração desse código intermediário, contendo o código fonte já processado léxica e sintaticamente, e os símbolos já devidamente analisados em termo de suas dependências e recursividades, é que o terceiro passo é ativado. Ele realiza a análise semântica, verificação de tipos e produz o código *LAMB*.

Associada a todos os passos do *Front-End* está a tabela de símbolos, que é responsável primordialmente por quatro serviços. Um desses serviços é a própria tabela de símbolos, ou seja, sua interface. Outro pode ser visto como uma gerência de escopo dos símbolos. Quando é necessário procurar um nome do texto fonte na tabela de símbolos, a localização da declaração apropriada do nome deve ser recuperada. As regras para verificação do escopo da linguagem fonte devem ser únicas e bem definidas para que se torne precisa qual é a declaração apropriada a cada momento. Esta gerência pode ser entendida como um mecanismo de pilha, onde são armazenandos os escopos e sua utilização caracteriza-se pelo método LIFO (*Last in, First Out*) para controlar a entrada e saída nos mesmos.

Outro serviço oferecido é a gerência da tabela de símbolos dentro de cada módulo, neste caso a estrutura de dados utilizada para representá-la é a árvore binária. Podemos entender a tabela de símbolos de todo o programa como uma **floresta** [33], já que foi implementada em uma forma que irá englobar todas as árvores de símbolos dos vários módulos (escopos) existentes. Seu acesso é feito por meio de um vetor de apontadores para árvores.

O último serviço deste módulo é a gerência de importação e exportação de símbolos. Chegando ao fim de um módulo, o mecanismo de compilação deve ser capaz de encontrar as variáveis e funções deste módulo que podem ser exportadas para outros módulos. Para facilitar essa busca, a tabela de símbolos pode ser classificada como sendo três tipos de tabelas distintas. A primeira delas, a Tabela de Símbolos Global, contém informações do escopo principal do módulo. A Tabela de Símbolos Locais representa a árvore dos símbolos de um escopo mais interno do módulo. É importante ressaltar que um mesmo módulo pode ter somente uma Tabela de Símbolos Global, porém 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. E é nesse contexto que pode ser encontrado o terceiro tipo de tabela de símbolos, a Tabela de Símbolos Exportáveis. Uma árvore com os símbolos que são exportáveis é criada e salva neste módulo de forma que ela possa ser pesquisada sempre que necessário. Ao entrar em um módulo qualquer, o compilador deverá reconhecer os módulos importados por ele, e para cada um desses módulos importados o compilador deve incorporar a árvore com os símbolos disponíveis para exportação, verificando se as importações necessárias são possíveis considerando-se as exportações disponíveis. A árvore referente a Tabela de Símbolos Global do módulo referido é inicializada com esses símbolos importados e, então, ocorre a fase de inserção de símbolos do próprio módulo, caracterizando a Tabela de Símbolos Global. Por fim, acontece a montagem das árvores das Tabelas de Símbolos Locais.

Cada um desses passos é dirigido por uma gramática com rotinas semânticas associadas, que servem de entradas para o Yacc [3] [28]. As gramáticas [4] do primeiro, segundo e terceiro passos são distintas, mas as linguagens por elas definidas são as mesmas, a menos de algumas informações especiais inseridas. Símbolos encontrados ao longo do código de entrada que foram inseridos na tabela de símbolos e informações sobre definições recursivas existentes no código fonte, tratadas pela primeira e segunda gramáticas, são armazenados em uma forma intermediária para serem consultadas pelo terceiro passo. A segunda e terceira gramáticas devem esperar como entrada um arquivo que não precisa ser analisado lexicamente. Com isso, a gramática do terceiro passo fica responsável pelas ações semânticas de verificação dos tipos, pois os domínios e variáveis já estão instalados e devidamente definidos na tabela de símbolos. Essa terceira gramática também deve realizar a geração de $\mathcal{LAMB}$, contando com as informações sobre as definições que são recursivas, para a decisão na montagem do código final.

Ao apresentar o *Back-End* é necessário ressaltar que o objetivo desta parte do trabalho foi o estudo e validação de técnicas para a geração de código eficiente para $\mathcal{SCRIPT}$. Ela foi fortemente influenciada por [6].

A técnica básica usada no *Back-End* é a mesma utilizada para implementar compiladores de linguagem funcionais como ML [18] e outras. Trata-se da máquina-G proposta inicialmente por [13] [18]. Também foi usada a idéia de funções especiais [19]. A escolha por esse método é por se tratar um método eficiente na implementação de linguagens funcionais puras [6]. O compilador tem como entrada um programa em linguagem $\mathcal{SUPER}$ já analisado léxica e sintaticamente. A saída é um programa equivalente escrito em linguagem C, veja a Figura 2.

Essa fase é responsável basicamente por duas etapa: (a) compilar para código-G, e (b) compilar de código-G para código em C. O primeiro recebe um programa que é um conjunto de definições de supercombinadores e uma expressão a ser avaliada e retorna um programa correspondente escrito em código-G. Cada instrução do código-G pode ser compilada para poucas instruções em linguagem C. A segunda tem como função transformar cada instrução do código-G gerada pela etapa anterior em instruções da linguagem C.

A terceira etapa do compilador de $\mathcal{SCRIPT}$ corresponde à parte do trabalho que trata da

[3] Um gerador LALR de reconhecedores sintáticos.

[4] O termo gramática neste capítulo deverá ser entendido como a gramática com rotinas semânticas associadas que serve de entrada para o Yacc.

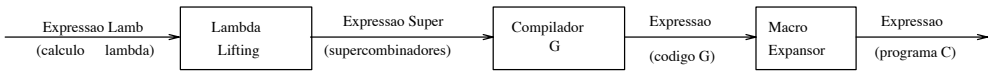

Figure 2: *Back-End*

compilação de programas escritos em $\mathcal{LAMB}$ para $\mathcal{SUPER}$, um cálculo de supercombinadores. Este processo é conhecido como *Lambda-Lifting* e consiste em transformar as abstrações lambda em abstrações equivalentes que não contenham variáveis livres.

Um supercombinador é como uma função em linguagem imperativa com vários parâmetros, que não usa variáveis globais e nem provoca efeitos colaterais. Formalmente, um supercombinador, \$S, de aridade n é uma abstração lambda da forma $LAMx_1.LAMx_2.\cdots LAMx_n.E$, onde: E não é uma abstração lambda; \$S não tem variáveis livres; qualquer abstração lambda em E é um supercombinador, e; n ≥ 0. A estratégia usada para a especificaçãao do compilador de $\mathcal{LAMB}$ consiste, a princípio, na transformação da expressão $\mathcal{LAMB}$ que se quer compilar em um conjunto de definições de supercombinadores mais uma expressão a ser avaliada.

O compilador define uma função $\mathcal{LL}$ ($\mathcal{L}$ ambda $\mathcal{L}$ ifting), que recebe como argumento uma expressão $\mathcal{LAMB}$ e produz como resultado o código de supercombinadores. Logo: $\mathcal{LL}$: $ExpL \mapsto ExpS$ onde: a função $\mathcal{LL}$ é chamada um esquema de compilação [6, 4], e utiliza outros esquemas de compilação como funções auxiliares, por exemplo os esquemas representados pela função $\mathcal{P}$ e os esquemas representados pela função $\mathcal{L}$. A notação de esquemas foi adotada por permitir expressar as técnicas de compilação de maneira compacta e elegante. $ExpL$ representa uma expressão $\mathcal{LAMB}$. $ExpS$ representa uma expressão $\mathcal{SUPER}$ de supercombinadores.

A presença de padrões na linguagem $\mathcal{LAMB}$ faz com que o processo de *Lambda-Lifting* seja feito após um passo importante de tradução: a compilação do casamento de padrões. Isto posto, é possível identificar o processo de tradução de $\mathcal{LAMB}$ para supercombinadores como uma composição de tarefas, ou melhor, uma composição de funções. É possível perceber a necessidade de se fazer, primeiramente, a compilação de casamento de padrões nas construções onde possa haver a ocorrência interna dos mesmos, podendo essa fase ser entendida como uma função que recebe uma árvore com padrões e retorna uma outra árvore equivalente sem padrões. Assim, temos a função $\mathcal{L}$, que efetua o *Lambda-Lifting* efetivo, aplicada ao resultado da função $\mathcal{P}$, que realiza o casamento de padrões, como a composição de funções descrita.

5 Classe, Herança, Polimorfismo e Funções Virtuais em $\mathcal{SCRIPT}$

Em $\mathcal{SCRIPT}$, os domínios de tuplas e as instâncias de tuplas implementam o conceito similar de classe e objeto nas linguagens imperativas orientadas por objetos. Por ser o mecanismo de implementação de classe, um objeto d de domínio de tupla D precisa guardar os valores associados a cada campo existente em D. A estrutura usada para guardar essas informações é a lista. A herança foi implementada pelo mecanismo de extensão de tuplas, onde qualquer tupla estendida é descendente de sua tupla base. A herança pode ser encontrada em $\mathcal{LAMB}$ pela consulta à lista gerada pela tupla estendida, como poderá ser visto mais a diante.

Em $\mathcal{SCRIPT}$ podem ser definidas funções associadas a domínios, sendo que esses domínios precisam ser tuplas. Uma função f associada a um domínio de tupla D só pode ser usada se estiver ligada a um objeto do domínio D ou ligada a um objeto de um domínio que é uma extensão direta ou indireta de D. Por isso foram criadas listas para indicar quais funções estão associadas a quais domínios. Essas listas se comportam como tabelas contendo os métodos virtuais de um domínio. Na tradução para $\mathcal{LAMB}$ esta lista deve ter as seguintes características:

- Primeiro item: Para o caso do domínio D ser uma extensão de outro domínio A, o primeiro item contém uma indicação para a lista dos métodos virtuais do domínio A. Se o domínio D não for uma extensão de outro domínio, o primeiro item deve ser uma indicação de que não há um domínio para o qual D seja uma extensão.

- Outros itens: Os itens seguintes nessa lista de métodos virtuais são os nomes das funções associadas a esse domínio. Caso D seja uma extensão de outro domínio A, as primeiras funções da lista são as funções do domínio A, seguidas, então, pelas novas funções atribuidas a D. Para o caso de funções atribuídas a D que sejam redefinições de funções de A, seus nomes são colocados no local em que estariam os nomes das funções originais.

Além dos valores de cada campo, o objeto d precisa fazer referência às funções que foram associadas a D. Portanto, na tradução para $\mathcal{LAMB}$ esta lista referente ao objeto d deve ter as seguintes características:
• **primeiro item:** nome da lista que contém as informações das funções associadas a D.
• **Outros itens**: valores referentes aos campos de D.

Portanto, a definição, onde D é uma extensão de A e f_i são funções associadas ao domínio A e g_i uma função associada a D,

MODULE Exemplo
DOMAINS
A = (x: X, y: Y, z: Z)
D = A EXT (w: W, k: K)
$a_1 : A$
$d_1 : D$
DEFINITIONS
DEF $A.f_1$ = ...
DEF $A.f_2$ = ...
DEF $A.f_3$ = ...
DEF $A.f_4$ = ...
DEF $D.g_1$ = ...
DEF $D.f_2$ = ...
DEF $D.g_2$ = ...
DEF $a_1 = (x_1, y_1, z_1)$
DEF $d_1 = (x_2, y_2, z_2, w_1, k_1)$
END Exemplo

deve, então, gerar definições auxiliares em $\mathcal{LAMB}$, que são:

DEF Vmt_A = < < > , f_{1A}, f_{2A}, f_{3A}, f_{4A} >
DEF Vmt_D = < Vmt_A , f_{1A}, f_{2D}, f_{3A}, f_{4A} , g_{1D}, g_{2D} >
DEF a_1 = < Vmt_A, x_1, y_1, z_1 >
DEF d_1 = < Vmt_D, x_2, y_2, z_2, w_1, k_1 >

sendo que Vmt_A e Vmt_D são geradas no início do terceiro passo do *Front-End* e os códigos de a_1 e d_1 são gerados à medida que essas variáveis forem aparecendo no programa.

Os tipos de polimorfismos permitidos em $\mathcal{SCRIPT}$ são sobrecarga e inclusão. A sobrecarga ocorre quando uma ou mais função recebe o mesmo nome no mesmo escopo, como pode ser visto no exemplo anterior, onde o domínio D, extensão do domínio A, redefiniu a função f_2. A ambiguidade desse tipo de definição é resolvida pelos tipos e a aridade dos argumentos das funções em questão. A inclusão está relacionada 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. Esse polimorfismo foi resolvido com a inclusão de um recurso a cada célula da tabela de símbolos permitindo identificar o pai de qualquer tupla, no caso de ser extensão.

6 Resultados Obtidos

Por razões de espaço apresentaremos aqui apenas três exemplos com os resultados obtidos da execução do compilador de $\mathcal{SCRIPT}$. Para os exemplos 1 e 2, ilustrados na Figura 3, são apresentados os programas $\mathcal{SCRIPT}$, $\mathcal{LAMB}$, $\mathcal{LAMBASIC}$ e $\mathcal{SUPER}$. O terceiro exemplo, Figura 4, apresenta o resultado da compilação de outro exemplo escrito na linguagem $\mathcal{SCRIPT}$ para a linguagem C.

O Exemplo 1 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 2 mostra 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 program $\mathcal{LAMBASIC}$.

7 Conclusão

Linguagens funcionais sempre propuseram, e continuam propondo, desafios especiais para projetistas de compiladores. Mesmo com os recentes progressos, compiladores para essas linguagens geram programas significativamente mais ineficientes que compiladores de linguagens convencionais. Programas em linguagens funcionais possuem duas propriedades que são particularmente problemáticas para o *back end* dos compiladores: eles fazem muitas chamadas recursivas e acessam intensivamente o *heap*. Portanto, procurar soluções para suprir tais problemas, como tentar minimizar o uso de definições recursivas, foi um dos objetos de estudo deste trabalho.

O compilador implementado para a linguagem $\mathcal{SCRIPT}$ faz parte de um ambiente cujo

Exemplo 1: operadores binários e constantes	Exemplo 2: abstração lambda e constantes
– SCRIPT MODULE TestScript DOMAINS Ndom = N DEFINITIONS DEF ndom1 = 11 DEF ndom2 = ndom1 PLUS 11 DEF ndom3 = ndom2 MULT 11 END TestScript – LAMB LET TesteScript = LET ndom1 = 11 IN LET ndom2 = ndom1 PLUS 11 IN LET ndom3 = ndom2 MULT 11 IN < > – LAMBASIC LET TesteScript = 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 TestScript 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 TestScript – LAMB LET TesteScript = LET ndom1 = 11 ALSO t1 = “FF” ALSO f1 = LAM (x) . x EQ 0 – > 1 , 2 IN < > – LAMBASIC LET TesteScript = 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 = ( )

Figure 3: Exemplos

principal objetivo é oferecer ao seu usuário ferramentas que diminuam o esforço de composição de descrições de linguagens de programação. Nele destacam-se dois pontos importantes:

- Possibilidade de testar, por meio do computador, as descrições semânticas para definição de linguagens de programação, podendo avaliar os resultados obtidos. Isto tornará as definições mais confiáveis.

- Ampliação do uso das técnicas de projeto baseadas em definições semânticas denotacionais. Isso deverá ocorrer pois o projetista de linguagem terá seu trabalho facilitado e com possibilidades de verificação e correção durante a fase de projeto.

Ao compararmos com as linguagens expostas na Seção 2.1, $\mathcal{SCRIPT}$ além de ser enquadrada dentro do paradigma funcional, apresentanto características de linguagens orientadas por objetos, que permitem alto grau de modularização. Além da rapidez com que se pode construir programas de propósito geral, $\mathcal{SCRIPT}$ apresenta um grande diferencial: apresenta facilidades para especificação denotacional de linguagens de programação.

O trabalho apresentado neste artigo relata nossa experiência na tradução de $\mathcal{SCRIPT}$ para código C. Uma visão geral das características relavantes das três etapas em que foi dividido o trabalho, *Front-End*, *Lambda-Lifting* e *Back-End*, foi um dos enfoques dados ao artigo. A integração entre as etapas de *Front-End* e *Lambda-Lifting* está finalizada e testada. A parte do *Back-End* também já está integrada às demais, contudo mais testes serão necessários para tornar o compilador mais robusto. Exemplos simples já são compilados para código C, conforme pode ser visto nas Figuras 4 e 3.

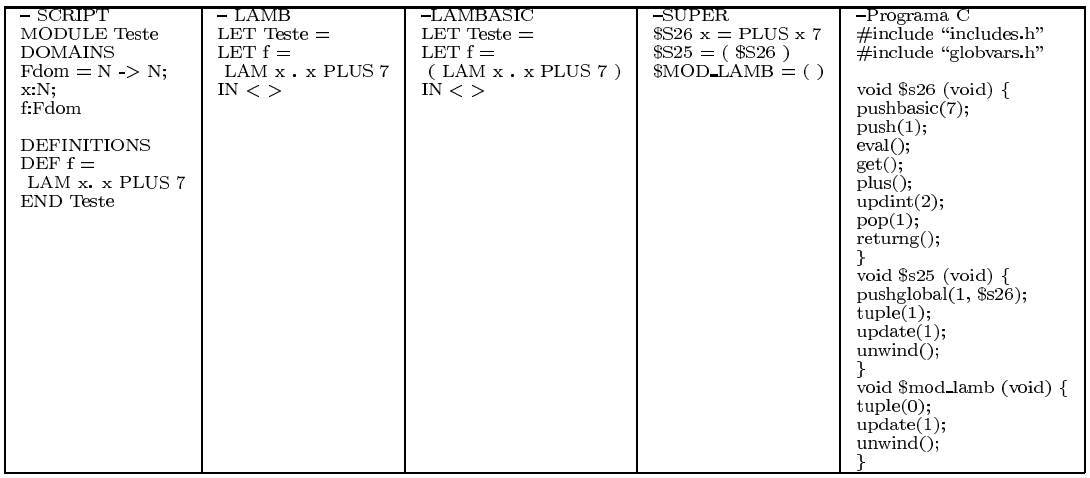

Figure 4: Geração de Código C

Problemas de $\mathcal{SCRIPT}$, tratados durante a etapa do *Front-End* e relacionados a controle de visibilidade, encapsulamento, herança, polimorfismo e ligação dinâmica tiveram especial atenção neste artigo, principalmente o tratamento de polimorfismo e funções virtuais. O controle de visibilidade foi feito implementando a tabela de símbolos como uma estrutura que representa uma floresta de árvores binárias [31] [33], o que facilitou o acesso a símbolos locais, importados e exportados. O controle da herança foi obtido principalmente por intermédio de um recurso colocado em cada célula da tabela de símbolos com o objetivo de identificar o pai daquele símbolo, caso existisse. Um mecanismo para gerência da importação e exportação de símbolos foi a solução adotada para contemplar a capacidade de encapsulamento presente na linguagem.

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

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

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

[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.* Addison Wesley, 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 Conference on Compiler Construction, Montreal, junho de 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 Symposium on Lisp an Functional Programming, Austin, agosto de 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.

[20] Jung, Achim *Domains and Denotational Semantics: History, Accomplishments and open Problems*, Lecture Notes in Computer Science, Springer-Verlag, 1996

[21] Milner, R. *Fully abstract models of typed lambda-calculi*, "Theoretical Computer Science", página 4:1-22, 1977

[22] Silva Júnior, J.L. *Linguagem de Definição e Geração de Analisadores Sintáticos em Semântica Denotacional Legível*, Dissertação de Mestrado - UFMG, Março de 1993

[23] Rodrigues, W.A. *Compilação e Otimização de uma Linguagem para Definição Denotacional de Semântica*, Dissertação de Mestrado - UFMG, Abril de 1993

[24] Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977

[25] Cardelli, L. *The Functional Abstract Machine*, journal Polymorphism, Volume 1, Number 1, 1983.

[26] Fairbairn, J. *Ponder and its type system*, Computer Lab., Cambridge, Technical Report 31, novembro de 1982.

[27] Bigonha, Roberto da Silva. *Script - An Object Oriented Language for Denotational Semantics*, Revised User's Manual and Reference, Technical Report 010/95. Department of Computer Science - UFMG, 07/1995.

[28] Johnson, Stephen. *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, 1978.

[29] Berry, D. M. and Schwartz, R. L. *Type Equivalence in Strongly Typed Languages: One More Look.*, In ACM SIGPLAN Notices, 1979, pages 158-166.

[30] Cardeli, L. and Wegner, P. *On Understanding Types, Data Abstraction and Polymorphism.*, In ACM Computing Surveys, 1985, number 17, pages 471-522.

[31] Aho, A.V., Hopcroft, J. and Ullman, J.D. *Data Structures and, Algorithms.* Addison Wesley, 1993.

[32] Milner, R. and Tofte and M. Harper, R. *The Definition of Standard ML* Cambridge, Massachusetts, 1990.

[33] Yedda A. D. Silva, Mariza A. S. Bigonha, Roberto S. Bigonha, *Tabela de Símbolos: Implementação e Avaliação*, LLP004/99, DCC-ICEx, UFMG, 1999.