Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

Implementação de Semântica Denotacional Modular Baseada em Aspectos

Proposta de Dissertação de Mestrado

Proponente: Tays Cristina do Amaral Pales Soares

Orientador: Roberto da Silva Bigonha

Belo Horizonte, 13 de janeiro de 2006

Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

Implementação de Semântica Denotacional Modular Baseada em Aspectos

Proposta de Dissertação de Mestrado

Proponente: Tays Cristina do Amaral Pales Soares

Orientador: Roberto da Silva Bigonha

Belo Horizonte, 13 de janeiro de 2006

Sumário

1 Definição do Problema

Apesar da existência de uma diversidade de abordagens para descrições formais da semântica das linguagens e do esforço em pesquisa já realizados para o seu desenvolvimento, o seu uso real em linguagens de programação de grande porte é ainda pequeno.

Mosses, em [Mos01], enumera os usos potenciais e reais de uma descrição semântica. A semântica formal pode ser usada para registrar decisões de projeto realizadas durante a criação de linguagens ou como forma de documentação de referência. Conceitos usados nas linguagens podem ser compreendidos no estudo de especificações formais, assim como se podem ter novos *insights* sobre esses conceitos.

Compiladores, interpretadores e protótipos podem ser gerados a partir de descrições formais, permitindo a avaliação e o raciocínio sobre propriedades de programas escritos na linguagem especificada. Entretanto, os compiladores e interpretadores gerados não são eficientes o suficiente para serem usados na prática. Na verdade, o uso mais comum deste modelo é a geração de protótipos para verificação da correção da própria descrição. Em contraste ao uso restrito de descrições semânticas, as descrições formais da sintaxe de linguagens de programação são amplamentes utilizadas. Mosses atribui esse fato à sua facilidade de escrita e compreensão. Os principais conceitos como alternativas, seqüenciamento e recursão podem ser representados de maneira uniforme e existem ferramentas que suportam prototipação de gramáticas e geração de analisadores sintáticos úteis, como por exemplo o Yacc [BLM92].

Segundo Mosses [Mos01], o uso ainda pequeno de descrições formais da semântica de linguagens decorre principalmente devido à dificuldade para se escrever e ler definições usando as propostas existentes, e ao reduzido número de ferramentas e ambientes com o amparo necessário para a geração de protótipos, interpretadores e compiladores que auxiliem na validação de descrições.

A dificuldade em se especificar a semântica de linguagens de programação de grande porte é relacionada à sua complexidade. Para que seja possível controlar a complexidade é desejável que uma descrição formal possa ser decomposta em pequenos módulos independentes, e que possa ser feita de forma incremental a partir da composição desses módulos.

O objetivo deste trabalho é o desenvolvimento de um ambiente para descrições denotacionais de linguagens de programação de forma modular, buscando facilitar a elaboração de descrições modulares de linguagens de programação de grande porte. As descrições léxica, sintática e semântica de linguagens de programação serão realizadas no ambiente proposto usando uma versão estendida da linguagem *Script* [Big99]. Uma nova versão de

Script está sendo especificada como parte da tese de doutorado de Fabio Tirelo [Tir05] do DCC/UFMG. A ferramenta será capaz de gerar protótipos na forma de interpretadores para a linguagem especificada.

Não é objetivo essencial para esse trabalho que os interpretadores gerados sejam eficientes, no entanto, esforços serão feitos para que o ambiente proposto seja capaz de gerar interpretadores que apresentem tempos de execução aceitáveis e com possibilitades de uso prático.

A modularidade das descrições realizadas no ambiente proposto será alcançada pelo uso de mônadas e aspectos e será guiada pela sintaxe abstrata da linguagem. O uso de mônadas permitirá a escrita incremental de uma descrição eliminando a possível interferência decorrente da definição de um novo módulo em partes já escritas e, por sua vez, aspectos auxiliarão na escrita modular de características das linguagens de natureza transversal. No ambiente proposto os módulos que compõem a descrição formal da linguagem serão transformados em analisadores léxico, sintático e semântico para a linguagem especificada, que agrupados formarão um interpretador para a mesma.

A ferramenta gerada por esse trabalho será usada para validar por meio de aplicação a metodologia proposta em [Tir05], denominada *Semântica Multidimensional* de linguagens de programação. A *Semântica Multidimensional* é uma nova técnica para especificação da semântica denotacional de linguagens de programação com alto grau de modularidade e extensibilidade. A linguagem *Script* possuirá recursos lingüísticos para definição de módulos, controle de visibilidade, mônadas e aspectos.

2 Semântica Denotacional Modular

2.1 Introdução

A semântica e a sintaxe de linguagens de programação podem ser descritas de forma precisa por meio de definições formais. A sintaxe refere-se à forma das expressões que são permitidas pela linguagem, já as definições semânticas podem descrever os efeitos da execução de uma expressão sintaticamente correta ou um programa, ou ainda podem descrever como são executadas. A descrição da sintaxe livre de contexto das linguagens é comumente feita com gramáticas livre de contexto por meio da *Forma de Backus-Naur*, que fornece meios adequados para especificar e raciocinar sobre a gramática de uma linguagem. Em contraste, para as descrições formais da semântica das linguagens existe uma diversidade de abordagens. Dentre elas destacam-se a Semântica Axiomática [Hoa69], a Semântica Operacional [Plo81, Gur95] e a

Semântica Denotacional [Sto77]. Esta diversidade se deve principalmente ao fato de que o comportamento de um programa é naturalmente mais complexo que sua estrutura [ZX04].

Técnicas formais usadas para descrição da semântica de linguagens de programação fornecem conceitos independentes de máquina, técnicas de especificação sem ambigüidades e base teórica rigorosa que suportem raciocínio confiável para provas e deduções [Gor79]. A semântica formal é também aplicada como guia de validação de implementação e base para geração automática de compiladores [ZX04].

As seções seguintes abordam questões relacionadas à modularidade em descrições formais: a Seção 2.2 apresenta propriedades importantes inerentes a uma especificação formal; a seção 2.3 apresenta a Semântica Denotacinal tradicional; e a Seção 2.4 apresenta a Semântica de Mônadas. É importante destacar que as diferentes abordagens existentes para descrições formais da semântica de linguagens não são concorrentes entre si, mas se complementam, cada uma com suas vantagens e desvantagens.

2.2 Propiedades Desejáveis em Uma Especificação Formal

Segundo [dM96, ZX04], um método de especificação formal de linguagens de programação deve apresentar legibilidade, modularidade, capacidade de abstração, comparação, raciocínio, aplicabilidade e ferramentas de suporte.

Legibilidade é importante para facilitar a percepção de propriedades inerentes à linguagem de programação especificada por meio de uma simples análise de sua descrição, tornando a descrição acessível a todas as pessoas interessadas na linguagem.

A modularidade está relacionada a reusabilidade e modificabilidade de descrições formais, provendo meios para que módulos usados em uma especificação sejam reaproveitados em descrições de outras linguagens, e que descrições possam ser modificadas alterando-se apenas partes pontuais. Descrições de linguagens de programação reais são geralmente extensas, e a modularidade possibilita a decomposição da especificação em pequenos componentes manipuláveis independentemente, promovendo controle sobre sua complexidade.

A abstração presente no formalismo da descrição deve ser suficiente para que os projetistas da linguagem sejam capazes de se concentrarem na especificação, sem ter em que se ater aos detalhes de implementação.

A descrição formal de linguagens de programação deve permitir comparações entre linguagens de programação. Para que isso seja alcançado, é

preciso que o método de descrição apresente um certo padrão a ser seguido pelas descrições.

O objetivo do formalismo é facilitar a compreensão de programas escritos na linguagem especificada, apresentando-se como uma forma não ambígua e portanto clara sobre os aspectos específicos da linguagem de programação especificada. A descrição deve permitir raciocínio sobre programas escritos na linguagem.

O método formal deve ser capaz de descrever conceitos presentes nas linguagens de programação como estado, entrada e saída (I/O), exceções e não-determinismo, sendo assim aplicável da forma mais direta possível às linguagens de programação reais.

Ferramentas de suporte são importantes para auxiliar na escrita, verificação e leitura das descrições semânticas. O uso prático e real das descrições semânticas depende da disponibilidade de ferramentas capazes de gerar interpretadores e compiladores a partir das descrições formais.

Todas as propriedades descritas são importantes, no entanto a ausência de ferramentas de suporte a uma especificação pode impedir que linguagens de programação sejam descritas mesmo se o arcabouço utilizado na especificação possua as demais caracteríticas descritas nesta seção, pois a validação da descrição e seu uso prático estariam comprometidos.

2.3 Semântica Denotacional

Em Semântica Denotacional, a semântica de uma linguagem de programação é descrita em termos de objetos matemáticos. O termo denotacional refere-se ao fato que os construtos da linguagem são descritos por denotações, entidades matemáticas abstratas que modelam o significado de cada elemento sintático da linguagem [Gor79]. A Semântica Denotacional é composicional, ou seja, a semântica de cada construto é uma função da semântica de seus constituintes.

Uma definição tradicional em Semântica Denotacional é constituída pelas seções: domínio sintático, sintaxe abstrata, domínio semântico, funções semânticas e equações semânticas. No domínio sintático são listadas as categorias sintáticas da linguagem especificada. A sintaxe abstrata especifica os construtos pertinentes a cada categoria sintática. Domínios semânticos determinam os objetos matemáticos que compõem a semântica da linguagem. Funções semânticas mapeiam objetos do mundo sintático em objetos do mundo semântico. A seção de funções semânticas exibe as assinaturas das funções, e a seção de equações semânticas especifica a denotação semântica de cada construto sintático da linguagem. A Figura 1, extraída de [Gor79, Capítulo 2], exemplifica o uso da semântica denotacional na especificação

formal de uma pequena linguagem.

Domínio Sintático:
$Ide \quad = \quad \{I | I \textit{ é um identificador}\}$
$Exp \quad = \quad \{E | E \textit{ é uma expressão}\}$
$Com \quad = \quad \{C | C \textit{ é um comando}\}$

Sintaxe Abstrata:
$C \quad ::= \quad I := E$
$\quad\quad | \quad\quad \textbf{while } E \textbf{ do } C$
$\quad\quad | \quad\quad C_1; C_2$
$E \quad ::= \quad 0 \mid I \mid \textbf{suc } E$

Domínios Semânticos:
$\textbf{Num} = \{0, 1, 2, \cdots\}$
$\textbf{State} = [\textbf{Ide} \mapsto \textbf{Num}]$
$n \in \textbf{Num}, s \in \textbf{State}$

Funções Semânticas:
$\mathcal{E} : \textbf{Exp} \mapsto \textbf{State} \mapsto \textbf{Num}$
$\mathcal{C} : \textbf{Com} \mapsto \textbf{State} \mapsto \textbf{State}$

Equações Semânticas:
$\mathcal{E}[\![0]\!]s = 0$
$\mathcal{E}[\![I]\!]s = s\ I$
$\mathcal{E}[\![\textbf{suc } E]\!]s = \mathcal{E}[\![E]\!]s + 1$
$\mathcal{C}[\![I \ := \ E]\!]s = s[\mathcal{E}[\![E]\!]s/I]$
$\mathcal{C}[\![C_1; C_2]\!]s = \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!]s)$
$\mathcal{C}[\![\textbf{while } E \textbf{ do } C]\!]s = \mathcal{E}[\![E]\!] = 0 \rightarrow s, \mathcal{C}[\![\textbf{while } E \textbf{ do } C]\!](\mathcal{C}[\![C]\!]s)$

Figura 1: Exemplo de especificação em Semântica Denotacional

A deficiência da semântica denotacional em relação a modularidade pode ser percebida pelos exemplos da Figura 2 extraídos de [Lia98]. Na Figura 2(a) é mostrada a semântica denotacional de uma linguagem aritmética simples. A função que representa a denotação de expressões dessa linguagem mapeia termos (*term*) em valores (*value*). A modularidade de uma descrição pode ser mensurada pelas alterações necessárias quando se inclui uma nova característica na linguagem. Na Figura 2(b), é inserido um *environment* que mapeia nomes de variáveis a valores, de forma que a linguagem suporte variáveis. Nota-se que a denotação para números passa a lidar com um *environment*, apesar de não depender dele. A operação aritmética '+' também não depende diretamente do *environment*, no entanto sua denotação precisa

ser alterada para se adequar à nova funcionalidade.

Ao se incluirem sequenciadores na linguagem, pode ser necessário migrar para semântica de continuação, cujo impacto nas definições é mostrado na Figura 2(c). A inclusão dessa nova funcionalidade também implica em mudanças nas descrições já existentes. De forma geral, este exemplo ilustra a necessidade de mudanças globais nas descrições em semântica denotacional tradicional ao se acrescentarem novas funcionalidades nas linguagens de programação descritas. Essa "fragilidade" da semântica denotacional motiva o estudo de novas abordagens com propostas que visam alcançar a modularidade necessária para a descrição de linguagens de programação reais [Lia98, Mos88]. Algumas dessas propostas são descritas nas seções a seguir.

$$\begin{aligned}
&\mathcal{E} : Term \to Value \\
&\mathcal{E}[\![n]\!] = n \\
&\mathcal{E}[\![e_1 \ + \ e_2]\!] = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!]
\end{aligned}$$

(a) Definição em Semântica Denotacional de uma linguagem aritmética simples

$$\begin{aligned}
&\mathcal{E} : Term \to Env \to Value \\
&\mathcal{E}[\![n]\!] = \lambda r.n \\
&\mathcal{E}[\![e_1 \ + \ e_2]\!] = \lambda r.\mathcal{E}[\![e_1]\!]r + \mathcal{E}[\![e_2]\!]r \\
&\mathcal{E}[\![v]\!] = \lambda r.r \ v
\end{aligned}$$

(b) Inclusão variáveis à definição em semântica denotacional

$$\begin{aligned}
&\mathcal{E} : Term \to Env \to Cont \to Ans \\
&\mathcal{E}[\![n]\!] = \lambda r \ k.kn \\
&\mathcal{E}[\![e_1 \ + \ e_2]\!] = \lambda r \ k.\mathcal{E}[\![e_1]\!]r(\lambda i.\mathcal{E}[\![e_2]\!]r(\lambda j.k(i+j))) \\
&\mathcal{E}[\![e_1; e_2]\!] = \lambda r \ k.\mathcal{E}[\![e_1]\!]r(\lambda x.\mathcal{E}[\![e_2]\!]rk)
\end{aligned}$$

(c) Inclusão de continuação para suportar sequenciadores

Figura 2: A falta de modularidade em semântica denotacional

Para seus autores, Semântica de Ações se apresenta como um método bastante atraente para especificações formais de linguagens de programação. Segundo [MW86], AS tenta encontrar um caminho intermediário entre o rigor do formalismo e as armadilhas das descrições informais, buscando o reconciliação entre a teoria e a prática. Em [Mos88], Peter Mosses afirma que AS apresenta boa modularidade devido à sua notação padrão denominada *action notation*, mas observa que sua notação verbal não está relacionada a modularidade, mas à facilidade de leitura e compreensão.

Os criadores de AS afirmam que não existem garantias que os combinadores de ações existentes possam expressar todas as maneiras possíveis de combinar ações e que talvez seja necessário incluir novos combinadores [MW86]. Esse fato torna-se algo preocupante ao ser defrontado com o que se encontra em [Mos96b], onde Mosses afirma que mudanças na semântica operacional da notação de ação ou a criação de novos combinadores de ações implicariam em uma reformulação de todas as leis da notação de ação, e que a notação de ação existente é difícil de ser alterada.

2.4 Semântica de Mônadas Modular

A Semântica de Mônadas Modular (MMS)[1] [LH96, LHJ95] é uma abordagem monádica para modularizar descrições em semântica denotacional. MMS permite que linguagens de programação de grande porte sejam definidas com pequenos blocos modulares reusáveis que podem ser combinados de acordo com o conjunto de características de cada linguagem. Para seus autores, as principais contribuições dessa abordagem para a semântica denotacional convencional são as facilidades de especificação, raciocínio e implementação das linguagens de programação.

Na próximas seções são definidos os principais conceitos inerentes à Semântica de Mônadas: mônadas, unidades de construção (*building blocks*), e transformadores de mônadas.

2.4.1 Mônadas

Na Semântica de Mônadas, mônadas são usadas para capturar individualmente a essência das mais variadas características de linguagens de programação complexas, abstraindo detalhes de baixo nível como *environments* e *stores*. Simon Thompson, em [Tho99], define a classe mônada presente na linguagem *Haskell*. Uma mônada é uma família de tipos $m\ a$ com um construtor de tipo polimórfico m mostrada na Listagem 1.

Listagem 1: A classe mônada

```
class Monad m where
  (>>=)  :: m a -> (a-> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
```

[1] Do inglês *Modular Monadic Semantics*

A definição de mônadas possui declarações *default* para as funções (>>) e *fail*, por isso para se declarar uma mônada em *Haskell* é obrigatório implementar apenas as funções (>>=) e *return*. Devido a esse fato, é comum definir uma mônada como um construtor de tipos M e duas funções polimórficas *return* e *bind*, sendo a última corresponde ao '(>>=)' definido na Figura 1 [Wad92, LHJ95].

A Listagem 2 define uma mônada trivial exemplificando a instanciação de uma mônada:

Listagem 2: A mônada Id

```
data Id a = Id a

instance Monad Id where
  return x       = Id x
  (>>=) (Id x) f = f x
```

Informalmente pode-se dizer que a função *return* permite "entrar" na mônada e o operador (>>=) permite operar dentro da mônada seqüenciando funções, sem que isso implique em "sair" da mônada.

Confrontando as definições em semântica denotacional convencional (Figura 2(a)) e a semântica de mônadas (Figura 3) para a mesma linguagem nota-se que a função dessa última, que descreve a denotação dos construtos da linguagem, mapeia termos (*term*) em computações (*M Value*), onde *M* é uma mônada e *Value* denota o resultado da computação. Os detalhes intrínsecos à computação tais como *environment* e *store* são abstraídos na mônada M por meio de suas funções *return* e *bind*.

$$
\begin{aligned}
&\mathcal{E} : Term \rightarrow M\ Value \\
&\mathcal{E}[\![n]\!] = return\ n \\
&\mathcal{E}[\![e_1\ +\ e_2]\!] = (\mathcal{E}[\![e_1]\!])\ >>= \\
&\qquad\qquad (\lambda v_1.\ \ (\mathcal{E}[\![e_2]\!])\ >>= \\
&\qquad\qquad\qquad (\lambda v_2 . return(v_1\ +\ v_2)))
\end{aligned}
$$

Figura 3: Definição em Semântica de Mônadas de uma linguagem aritmética simples

No exemplo da Figura 3 a semântica do numeral n é uma computação que apenas retorna n como resultado. A semântica de $\mathcal{E}[\![e_1\ +\ e_2]\!]$ é uma computação que calcula o valor de $\mathcal{E}[\![e_1]\!]$ e o associa a v_1, calcula o valor de $\mathcal{E}[\![e_2]\!]$ e o associa a v_2, e por fim retorna $v_1 + v_2$.

Para que essa linguagem suporte variáveis é necessário incluir apenas a equação semântica para variáveis, como pode ser visto na Figura 4.

$$\mathcal{E}[\![v]\!] = do\ r \leftarrow rdEnv$$
$$\qquad\qquad return\ r\ v$$

Figura 4: Equação semântica para variáveis

Nota-se que não foi preciso, ao se incluir um *environment* na definição semântica da linguagem, alterar qualquer equação já definida; em contraste com o que ocorre em definições denotacionais tradicionais, como mostrado na Figura 2(b). Nesta equação, a função *rdEnv* recupera o *environment* corrente.

Considerando a equação semântica para $\mathcal{E}[\![e_1 + e_2]\!]$ da Figura 3, a Figura 5 ilustra a dependência das equações semânticas em relação à instanciação da mônada subjacente à descrição em semântica de mônadas da linguagem de programação especificada[2]. Dessa maneira, para adicionar uma nova construção à linguagem é preciso apenas alterar a mônada base da descrição semântica da linguagem de programação, e adicionar a equação semântica para a nova construção, sendo que nenhuma alteração nas equações já existentes é necessária.

$$\begin{array}{lcl} type\ M\ a & = & a \\ return\ x & = & x \\ bind\ e\ k & = & k\ e \end{array} \quad \Longrightarrow \quad \begin{array}{l} \mathcal{E} : Term \rightarrow Value \\ \mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!] \end{array}$$

(a) Mônada trivial

$$\begin{array}{lcl} type\ M\ a & = & Env \rightarrow a \\ return\ x & = & \lambda r.x \\ bind\ e\ k & = & \lambda r.k\ (e\ r)\ r \end{array} \quad \Longrightarrow \quad \begin{array}{l} \mathcal{E} : Term \rightarrow Env \rightarrow Value \\ \mathcal{E}[\![e_1 + e_2]\!] = \lambda r.\mathcal{E}[\![e_1]\!]\ r + \mathcal{E}[\![e_2]\!]\ r \end{array}$$

(b) Mônada com *environment*

Figura 5: Comportamento da denotação para $\mathcal{E}[\![e_1 + e_2]\!]$ de acordo com a mônada subjacente

É importante observar que a definição semântica para $\mathcal{E}[\![e_1 + e_2]\!]$ não se alterou diante da inclusão de variáveis, ou seja, inclusão do *environment*. A relação das equações semânticas das construções já existentes na linguagem com a nova característica acrescida se faz pela nova mônada definida, o que torna transparente as conseqüências provenientes da inclusão dessa nova funcionalidade. Na Figura 5(a) é mostrado o significado de $\mathcal{E}[\![e_1 + e_2]\!]$ substituindo as operações *return* e *bind* da equação da Figura 3 de acordo com a

[2]A Figura 5 é baseada na apresentada por Liang em [Lia98]

definição dessas funções da mônada subjacente, e o mesmo é feito em 5(b), onde a mônada base atua na presença de um *environment*.

2.4.2 Unidades de Construção da Semântica de Mônadas

As unidades de construção definem a semântica de mônadas de cada recurso da linguagem de programação descrita, tais como comando de atribuição, funções, operações aritméticas e continuações. Cada unidade se caracteriza por sua independência em relação às demais. No entanto, elas se baseiam em um conjunto comum de operações núcleo (*kernel*) da linguagem. Por exemplo, o comando de atribuição e as continuações de desvio podem usar o mesmo *store*. A unidade de construção para operações aritméticas é mostrada na Figura 3. A seguir será exemplificada uma unidade de construção para funções e uma para alocações e atribuições.

- Unidade de construção para funções:

  Uma unidade de construção para funções é mostrada na Figura 6. Para simplificar a leitura das equações semânticas, a função de seqüenciamento de mônadas *bind* será substituída por algo similar à notação "*do*" usada em *Haskell* [Tho99, Capítulo 18]. O tipo *Value* corresponde à união disjunta de valores básicos da linguagem e um tipo para funções, que mapeia computações em computações.

  $$\mathcal{E} : Term \rightarrow M\ Value$$
  $$\mathcal{E}[\![v]\!] = \{r \leftarrow rdEnv;\ r\ v\}$$
  $$\mathcal{E}[\![\lambda v.e]\!] = \{r \leftarrow rdEnv;\ return(\lambda x.inEnv\ r[x/v]\mathcal{E}[\![e]\!]\}$$
  $$\mathcal{E}[\![e_1\ e_2]\!] = \{f \leftarrow \mathcal{E}[\![e_1]\!]; v \leftarrow \mathcal{E}[\![e_2]\!]; f(return\ v)\}$$

  Figura 6: Definição em Semântica de Mônadas para funções

  A descrição em semântica denotacional de abstrações e aplicações de funções precisam acessar um *environment* que mapeia variáveis em computações, fazendo-se necessário o uso de duas operações núcleo que atuam sobre o *environment*:

```
type Env = Name → M Value
rdEnv    :: M Env
inEnv    :: Env → M Value → M Value
```

  A operação *rdEnv* recupera o *environment* atual e a operação *inEnv* executa uma computação em um *environment* dado. A denotação para

variáveis consiste em recuperar o *environment* atual e retornar a computação associada à variável nesse *environment*. A semântica para uma abstração é retornar uma função que espera um argumento e invoca a operação *inEnv* passando o *environment* atual acrescido da associação da variável de abstração ao argumento dessa função e a denotação da expressão da abstração. Dessa maneira, a denotação da expressão será executada no novo *environment*. Para a denotação de aplicação de função é feita a avaliação da primeira expressão, que produz uma função, seguida pela avaliação da segunda expressão, que produz um valor que é passado para a função.

- Unidade de construção para alocações e atribuições:

  Características comuns a linguagens imperativas tais como alocação de memória e atribuições de variáveis podem ser descritas por meio de um local de armazenamento usualmente denominado *store* que mapeia localizações (do tipo *Loc*) em computações. Operações para alocação, leitura e escrita em posições desse *store* são definidas:

```
type Store = Loc → M Value
alloc      :: M Loc
readS      :: Loc → M Value
write      :: (Loc,M Value) → M ()
```

  A Figura 7 mostra a semântica de mônadas para alocações e atribuições. A denotação para referenciar uma expressão corresponde a avaliar essa expressão, obter uma nova localização no *store*, escrever o valor obtido pela avaliação da expressão na nova localização e, por último, retornar o local onde o valor da expressão foi armazenado. Para "derreferenciar" uma expressão, deve-se avaliá-la para produzir uma localização que será usada para recuperar o seu valor associado no *store*. Para a atribuição, avalia-se a expressão do lado esquerdo da atribuição, produzindo uma localização, avalia-se a expressão do lado direito produzindo um valor, e associa-se no *store* a localização obtida à computação correspondente ao valor avaliado.

$$\mathcal{E}[\![ref\ e]\!] = \{v \leftarrow \mathcal{E}[\![e]\!]; l \leftarrow alloc; write(l, return\ v); return\ l\}$$
$$\mathcal{E}[\![deref\ e]\!] = \{l \leftarrow \mathcal{E}[\![e]\!]; read\ l\}$$
$$\mathcal{E}[\![e_1 := e_2]\!] = \{l \leftarrow \mathcal{E}[\![e_1]\!]; v \leftarrow \mathcal{E}[\![e_2]\!]; write(l, return\ v)\}$$

Figura 7: Definição em Semântica de Mônadas para alocações e atribuições

As operações *alloc*, *readS* e *write* abordam uma noção de estado e podem ser definidas em termos da função:

```
update :: (Store → Store) → M Store
```

para um dado *Store s*. Para recuperar o estado atual deve-se passar a função identidade para *update*, e para alterar o estado passa-se uma função de transformador de estado. Uma possível definição das funções *alloc*, *readS* e *write* é:

```
type Store = Loc → M Value

alloc       :: M Loc
alloc        = {s ← update (λi.i); return f s}

readS       :: Loc → M Value
readS l      = {s ← update (λi.i); s l}

write       :: (Loc,M Value) → M ()
write (l,v) = {_ ← update (λs.s[v/l]); return ()}
```

O *underscore* ('_') indica que o retorno da função *update* é ignorado. A função 'f', usada em *alloc* percorre o *store s* procurando por uma posição de memória que esteja disponível.

2.4.3 Transformadores de Mônadas

Diante dos exemplos das Figuras 3, 6 e 7 nota-se que as unidades de construção referentes a cada característica da linguagem de programação especificada necessitam, além das operações de *bind* e *return* da mônada subjacente, de um conjunto base de operações, como mostra a Tabela 1.

Característica	Função
Environment	`rdEnv :: M Env` `inEnv :: Env → M Value → M Value`

Store	`alloc :: M Loc`
	`readS :: Loc → M Value`
	`write :: (Loc,M Value) → M ()`

Tabela 1: Operações monádicas usadas nas definições semânticas

Em definições de semântica denotacional tradicional, ao se listarem as operações auxiliares necessárias às definições semânticas, o passo seguinte é enumerar os domínios e implementar essas funções. No entanto, se futuramente existir a necessidade de incluir um novo recurso à linguagem e novas operações, possivelmente será necessária a redefinição dos domínios semânticas e de equações já construídas.

Transformadores de mônadas permitem que a semântica para cada característica da linguagem seja tratada individualmente e, por meio da operação *lift*, é possível determinar a interação entre as características. Para que a modularidade seja alcançada, a definição semântica começa a partir de uma mônada simples e características vão sendo adicionadas. O papel do transformador de mônada é, dada uma mônada inicial, retornar uma nova mônada que incorpore a característica adicionada.

Formalmente, um transformador de mônada é um construtor de tipos t e uma função associada *lift*, onde t mapeia qualquer mônada $(m, return_m, bind_m)$ para uma nova mônada $(t\ m, return_t\ m, bind_t\ m)$. A Listagem 3 exemplifica a criação e o funcionamento de um transformador de mônada. Para elucidar o comportamento dos tranformadores de mônadas. A seguir é apresentada a definição dos trasnformadores de mônadas para *state* e *environment* seguida da interação entre eles.

Transformador de Mônadas para Estado . De forma semelhante à classe mônada da Listagem 1, os transformadores de mônadas, em *Haskell*, pertencem a uma classe de tipos denominada *MonadT* que possui uma função "*lift*" a ser implementada por suas instâncias, mostradas nas Linhas 1 e 2 da Listagem 3. Na Linha 4 é criado um tipo de dado que corresponde ao tipo da mônada *state* resultante da aplicação do trasnformador *StateT s* a uma mônada qualquer *m*. O código das Linhas 7 a 10 completa a declaração da mônada *StateT s m*, criando uma instância da classe mônada de forma que, se *m* é uma mônada, então *StateT s m* também é. Nas Linhas 12 a 14, o transformador de mônadas *StateT s* é declarado uma instância da classe *MonadT* com uma implementação da operação *lift*. Nota-se que a operação *lift* apenas insere a mônada *m* no novo contexto, enquanto o estado é preservado. Por último, uma mônada *StateMonad*, nas Linhas 16 e 17, é definida como uma subclasse de *Monad* com a adição da operação *update*, já que uma mônada para *State* deve suportar essa operação, e nas Linhas 19 e 20 *StateT s* transforma qualquer mônada *m* em uma mônada *state*, onde *update f* aplica *f* ao estado atual.

Listagem 3: Transformador de mônadas para Estado

```
1 class MonadT t where
```

```
2           lift :: (Monad m, Monad (t m)) ⇒ m a → t m a
3
4   data StateT s m a = StateM (s -> m (s,a))
5   unStateM (StateM x) = x
6
7   instance Monad m ⇒ Monad (StateT s m) where
8    return x = StateM (\s -> return (s,x))
9    (>>=) (StateM m) k = StateM(\s0->do (s1,a) <- m s0
10                                       unStateM (k a) s1)
11
12  instance MonadT (StateT s) where
13   lift m = StateM (\s -> do x <- m
14                             return (s,x))
15
16  class Monad m ⇒ StateMonad s m where
17   update :: (s->s) -> m s
18
19  instance Monad m ⇒ StateMonad s (StateT s m) where
20   update f  = StateM (\s -> return (f s,s))
```

Transformador de Mônadas para *Environment* . A Listagem 4 ilustra os passos para a criação do transformador de mônadas para *environment*.

Listagem 4: Transformador de mônadas para *Environment*

```
1   data EnvT r m a = EnvM (r -> m a)
2   unEnvM (EnvM x) = x
3
4   instance Monad m ⇒ Monad (EnvT r m) where
5    return x = \r -> return x
6    (>>=) (EnvM m) k = EnvM (\r -> do a <- m r
7                                     unEnvM (k a) r)
8
9   instance MonadT (EnvT r) where
10          lift m = EnvM{\r -> m}
11
12  class Monad m ⇒ EnvMonad env m where
13   inEnv :: env -> m a -> m a
14   rdEnv :: m env
15
16  instance Monad m ⇒ StateMonad s (StateT s m) where
17   inEnv r m = EnvM{\r' -> m r}
18   rdEnv     = EnvM{\r -> return r}
```

O transformador de mônadas "EnvT" transforma qualquer mônada m em uma mônada *environment*. As operações suportadas pelo transformador de *environment* são *inEnv*, que executa uma computação em um *env* dado ignorando o *env* presente na mônada, e a operação *rdEnv* que apenas insere o *env* dado na mônada transformada.

Interação entre Transformadores de Mônadas .

Os trasformadores de mônadas inserem as operações necessárias para cada característica suportada pela linguagem. A forma como essas operações se relacionam é dada pelo processo *lifting*, em que essas operações são transformadas por meio do uso da operação *lift* do transformador de mônadas ao qual estão sendo aplicadas. *Lifting* uma operação f para uma mônada m usando-se uma transformador t resulta em uma operação em que os tipos m a de sua assinatura são substituídos por t m a como mostra a Figura 8.

$$inEnv \ : \ r \rightarrow m \ a \rightarrow m \ a$$

(a) Assinatura da operação *inEnv* antes da operação de *lifting*

$$inEnv \ : \ r \rightarrow t \ m \ a \rightarrow t \ m \ a$$

(b) Assinatura da operação *inEnv* depois da operação de *lifting*

Figura 8: *Lifting* na operação *inEnv*

As operações *update* e *rdEnv* recebem um tipo não-monádico e retornam uma computação, ou seja, um tipo-monádico. O processo de *lift* para essas funções independe do transformador de mônadas a qual estão sendo aplicadas, como mostra a Figura 9

$$\begin{aligned} rdEnv_{t \ m} &= lift_t \ . \ rdEnv_m \\ update_{t \ m} &= lift_t \ . \ update_m \end{aligned}$$

Figura 9: *Lifting* sobre as operações *rdEnv* e *update*

Já a operação *inEnv* recebe um tipo monádico e por isso o processo de *lifting* dessa operação depende do trasnformador de mônadas ao qual está sendo aplicada. A Figura 10 mostra a operação inEnv sendo aplicada aos transformadores de *environment* *EnvT* e ao de estados *StateT*.

De posse dos transformadores de mônadas é possível construir a mônada M subjacente que suporte as unidades de construção para funções, alocações

$$
\begin{array}{lcl}
inEnv_{EnvT\ r'\ m} & : & r \to (r' \to m\ a) \to r' \to m\ a \\
inEnv_{EnvT\ r'\ m} r\ e & = & \lambda r'.inEnv_m r(e\ r') \\
 & & \\
inEnv_{StateT\ s\ m} & : & r \to (s \to m\ (s,a)) \to s \to m\ (s,a) \\
inEnv_{StateT\ s\ m} s\ e & = & \lambda s'.inEnv_m s(e\ s')
\end{array}
$$

Figura 10: *Lifting* sobre a operação *inEnv* usando os transformadores *EnvT* e *StateT*

e atribuições. A Figura 11 ilustra essa mônada subjacente, onde *Env* e *Store* são tipos de *environment* e *store*.

$$type\ M\ a\ =\ EnvT\ Env\ (StateT\ Store)\ a$$

Figura 11: Mônada M com: *environment* e *store*

2.4.4 Avaliação

A modularidade em semântica de mônadas manifesta-se em dois níveis: em alto nível com as unidades de construção e em baixo nível com os transformadores. Usando-se uma série de abstrações, semântica modular monádica transforma a estrutura monolítica da semântica denotacional tradicional em componentes reusáveis [Lia98]. Uma mônada que engloba componentes comuns às definições em semântica denotacional para linguagens de programação, tais como *environment*, continuações, *store* e reportagem de erros pode ser construída como mostra a Figura 12.

```
type M a = EnvT Env                 (environment)
             (ContT Answer          (continuação)
                (StateT Store       (store)
                   ErrT)) a         (reportagem de erro)
```

Figura 12: Mônada M que suporta *environment*, continuações, *store* e reportagem de erros

Um possível problema gerado pela adição de transformadores de mônadas nas definições em semântica de mônadas é que o número de *liftings* a serem executados pode crescer quadraticamente [LHJ95]. No entanto, ou autores desse arcabouço acreditam que não exista um grande número de transformadores

3 Programação Orientada por Aspectos

Modularidade em sistemas significa dividir o *software* em pequenas unidades funcionais, denominadas módulos, que sejam independentes e compiláveis separadamente. No entanto, existem requisitos de um sistema que são transversais a outros, ou seja, cujas implementações se atravessam, dificultando sua definição em apenas uma unidade. Programação Orientada por Aspectos [KLM+97] provê meios de implementar tais requisitos mantendo-se a modularidade geral de sistemas.

A especificação em semântica denotacional de linguagens de programação pode apresentar problemas de modularidade provenientes de entrelaçamento de conceitos presentes nas equações semânticas que as definem. Visando manter alto grau de modularidade mesmo diante deste fato, a linguagem *Script* possibilitará o uso de aspectos para a implementação de conceitos transversais nas especificações.

A falta de modularidade em sistemas pode ser percebida em manifestações de intrusão e espalhamento. A intrusão ocorre quando o código de mais de um requisito transversal está presente em uma única região do programa. O espalhamento ocorre quando o código para um determinado requisito transversal encontra-se disperso no sistema. Tirelo apresenta, em [Tir05], exemplos de recursos de linguagens de programação, tais como verificação de tipos dinâmicos, tratamento de erros e sequenciadores, cujas descrições denotacionais apresentam dificuldades de modularização, pois causam intrusão e espalhamento em descrições semânticas.

Como apresentado na Seção 5, o ambiente para definições formais proposto neste trabalho permitirá o uso de apectos para a implementação de requisitos transversais com o intuito de se manter a modularidade da especificação de linguagens. O ambiente deverá, então, ser capaz de costurar regras descritas por aspectos na especificação.

Um aspecto é normalmente implementado como um componente do sistema que define um requisito transversal, e é aplicado pelo mecanismo de orientação por aspecto em vários pontos da execução de um programa. A união entre o requisito transversal e o resto do sistema é dada pelo processo de costura (*weaving*). O processo de costura consiste em introduzir o código dos aspectos no sistema.

4 Semântica Multidimensional

Semântica Multidimensional, definida em [Tir05], é assim denominada por ser baseada na separação multidimensional de preocupações no desenvolvimento

de sistemas e no desenvolvimento de sistemas orientados por aspectos. Essa técnica permite a escrita de definições modulares e extensíveis, de modo que a inclusão de uma nova construção não implique na reescrita de outros módulos já definidos.

Essa metodologia objetiva aperfeiçoar as técnicas de definição de semântica denotacional de linguagens de programação, permitindo modularizar definições de recursos presentes nessas linguagens que se encontram dispersos em uma definição denotacional tradicional.

A decomposição da especificação de uma linguagem é normalmente guiada pelas regras de produção de sua sintaxe absrata. Semântica multidimensional é baseada na decomposição multidimensional de sistemas e seu principal diferencial é fornecer suporte à decomposição da especificação por meio de, além da sintaxe, outras propriedades da linguagem.

Existem construções em linguagens cuja compreensão não pode ser feita de forma isolada. Em semântica denotacional, essa característica das linguagens prejudica a modularidade da especificação, já que a descrição de uma construção apresenta elementos de outras, como mostrado na Figura 2 da Seção 2.3. Com o objetivo de alcançar a modularidade mesmo diante dessas características das linguagens, semântica multidimensional permite o uso de mônadas nas descrições semânticas de linguagens, permitindo uma escrita incremental de forma que novos módulos podem ser escritos, iterativamente, sem implicar em alterações nos já existentes.

A principal característica dessa abordagem é permitir o uso de aspectos nas definições da semântica de linguagens de forma a possibilitar a escrita modular de requisitos transversais que causam intrusão e espalhamento em uma descrição, como discutido na Seção 3.

5 A Linguagem *Script*

Nesse trabalho será desenvolvida uma ferramenta de suporte à especificação modular de linguagens de programação. A especificação será feita na linguagem *Script* estendida para possibilitar definições de linguagens de programação em semântica multidimensional. Será desenvolvido também um ambiente para a geração de interpretadores a partir da especificação de linguagens descritas em *Script*.

A linguagem *Script*, em processo de adaptação [Tir05], fornecerá suporte para a divisão dos módulos de acordo com os recursos da linguagem, especificação da constituição léxica e sintática da linguagem, especificação dos domínios sintáticos e da estrutura da árvore de sintaxe abstrata, especificação da semântica separada das construções e definição das regras de composição

das especificações.

A divisão dos módulos que compõem a descrição formal da linguagem é guiada pela sua sintaxe abstrata. Cada módulo em *Script* contém as especificações léxica, sintática e semântica da construção da gramática referente a esse módulo.

A especificação da constituição léxica de uma linguagem em *Script* se dará por meio de expressões regulares semelhantes às utilizadas pelo gerador de analisadores léxicos *Lex*[BLM92]. A constituição sintática será especificada pelo domínio sintático, utilizando união disjunta para domínios compostos, e pela descrição da gramática da linguagem especificada por um conjunto de regras semelhante à notação BNF.

Como ferramenta para a descrição formal de linguagens de programação, *Script*:

- é uma linguagem puramente funcional *lazy* e apresenta sintaxe semelhante à linguagem *Haskell*;
- permite a especificação léxica, sintática e semântica de uma linguagem de programação em módulos;
- permite a escrita de funções de ordem mais alta para a construção das equações que especificam a semântica dos constituintes da linguagem;
- *Script* permite a escrita de mônadas para possibilitar a especificação de forma incremental de uma linguagem de programação, fornecendo meios para abstração de elementos da definição, tais como *environments* e *stores*; mônadas e transformadores de mônadas comumente usados serão componentes do núcleo da linguagem *Script*;
- fornece construções baseadas na orientação por aspectos para implementação modular de regras de composição de requisistos transversais.
- implementa a costura de código de forma estática que é realizada pelo *back-end* de seu compilador durante a tradução das equações semânticas da definição para funções em *Haskell*, conforme detalhado na Seção 6.

6 Proposta de Trabalho

O ambiente proposto para especificação de semântica formal modular e geração de interpretadores para linguagens é constituído por uma ferramenta de suporte, onde a definição formal da linguagem será escrita em *Script*, e por um

compilador de dois passos, responsável pelo reconhecimento da linguagem especificada e pela geração de um interpretador para essa linguagem.

O *front-end* do compilador de *Script* lê os módulos da linguagem, faz as análises léxica, sintática e semântica da descrição em *Script*, e gera código intermediário. A Figura 13 ilustra o primeiro passo do compilador para a especificação de uma linguagem L. A especificação de L em *Script* é composta pelos módulos que descrevem sua parte léxica, sintática e semântica. Diante dessa especificação de L, o *front-end* do compilador reconhece seus constituintes e gera uma representação desses componentes, o código intermediário, manipulável pelo *back-end* do compilador.

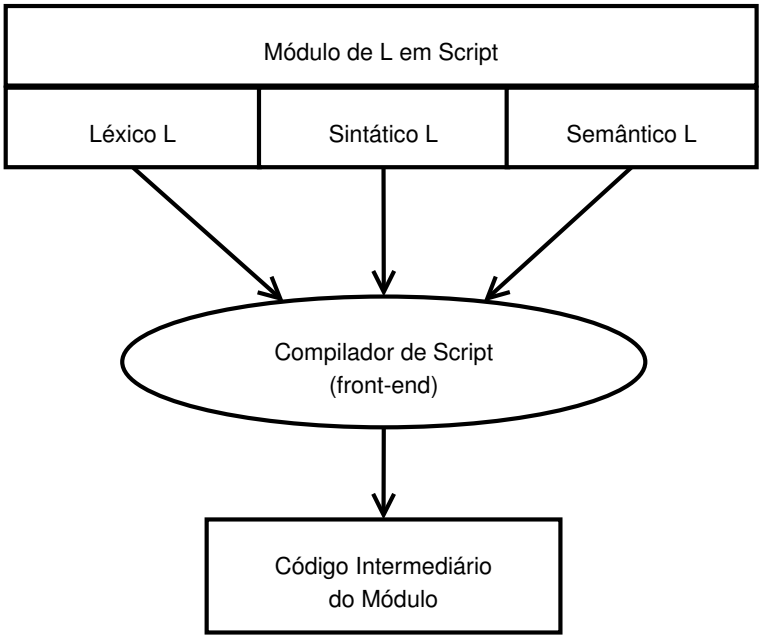

Figura 13: *Front-end* do compilador de *Script*

O *back-end* do compilador de *Script* é responsável pela leitura dos arquivos de código intermediário, e escrita de novos arquivos a serem manipulados para a geração de código em linguagem C correspondente a um interpretador da linguagem especificada. A Figura 14 ilustra o segundo passo da compilação. Cada M_i dessa figura corresponde ao código intermediário do módulo i produzido pelo *front-end* do compilador.

O *back-end* do compilador é responsável pelas gerações dos analisadores léxico e sintático da linguagem especificada e pela tradução das equações semânticas.

Inicialmente, o *back-end* do compilador identifica os elementos da definição léxica de L presentes nos módulos, e os une em arquivos no formato

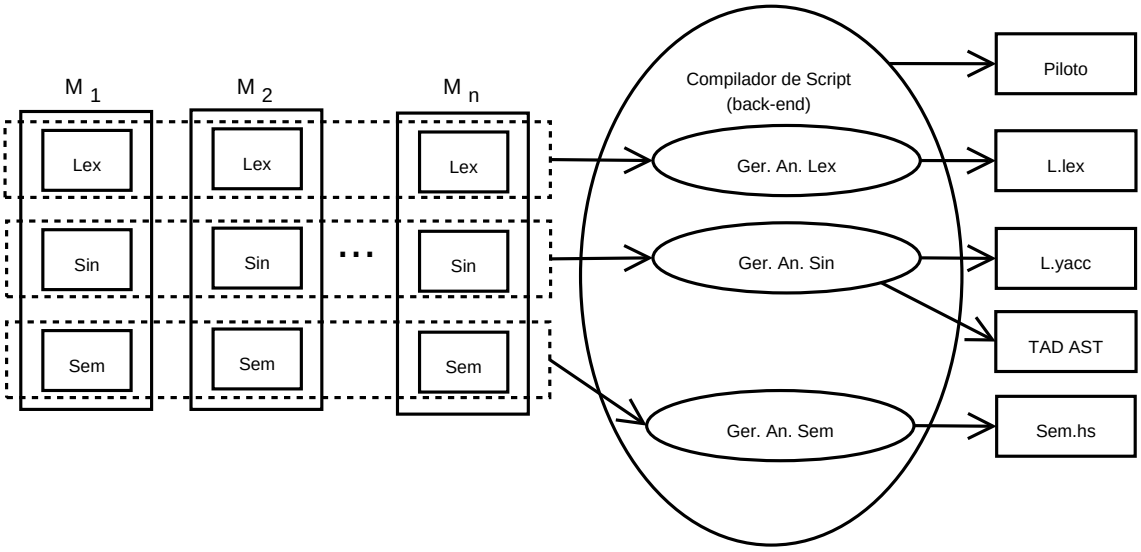

Figura 14: *Back-end* do compilador de *Script*

de entrada do gerador de analisadores léxicos *Lex* [BLM92], que então produz o analisador léxico.

De forma semelhante, o *back-end* do compilador identifica os elementos das regras de produção da gramática concreta de L presentes nos módulos, e os une em arquivos no formato de entrada do gerador de analisadores sintáticos *Yacc* [BLM92], que então produz o analisador sintático. Nesse momento também devem ser criados os tipos abstratos de dados correspondentes à árvore de sintaxe abstrata de L.

Para a construção da parte semântica da linguagem especificada, o *back-end* é responsável por gerar a partir dos domínios semânticos da especificação os *data types* em *Haskell* [Tho99] necessários para a análise semântica. As equações semânticas de cada módulo de L são traduzidas para funções em *Haskell*, e a costura de código referente aos aspectos utilizados na definição da semântica da linguagem também é realizada nesse momento pelo *back-end* do compilador.

O programa *Piloto* da Figura 14, também gerado pelo *back-end* do compilador, coordena a comunicação entre os analisadores léxico, sintático e semântico da linguagem L.

No fim, o código executável correspondente ao interpretador da linguagem especificada é gerado por meio da compilação e *link-edição* dos arquivos com programas C gerados, realizada por um compilador da linguagem C.

A fase de *link-edição*, mostrada na Figura 15, consiste na compilação dos módulos léxico, sintático e semântico para linguagem C e geração do

executável do interpretador da linguagem especificada.

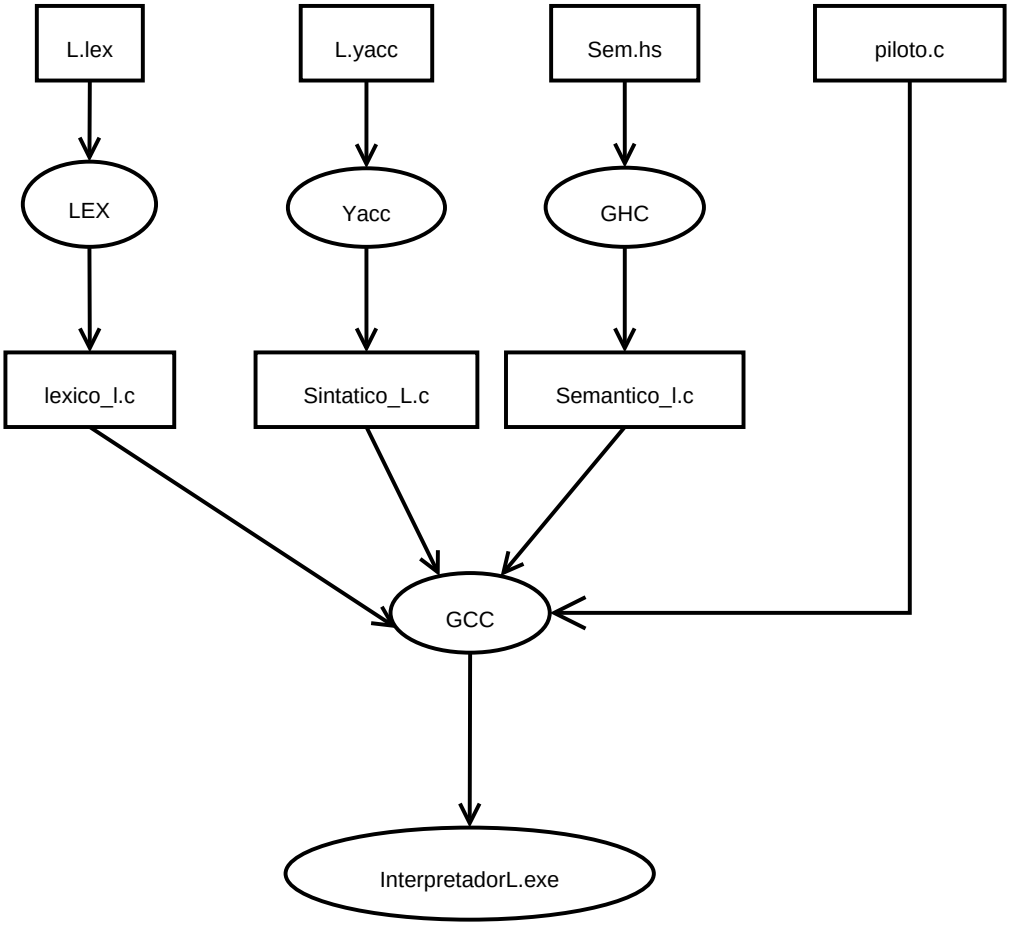

Figura 15: *Link-edição* dos analisadores léxico, sintático e semântico para L

O compilador *Glasgow Haskell Compiler*[3] (GHC) será usado para compilar o código *Haskell* referente às equações semânticas da linguagem especificada para código *C*.

A Figura 16 mostra em alto nível a geração de um interpretador a partir dos módulos da definição de L em *Script*, e o processo pelo qual um programa em L é executado a partir de sua descrição formal em *Script*; o interpretador gerado recebe um programa escrito em L e as entradas para esse programa e gera a saída.

A execução do interpretador, gerenciada pelo programa *piloto.c* (Figura 15) é ilustrada na Figura 17. Um programa escrito em L é reconhecido pelo

[3]Hospedado em http://www.haskell.org/ghc

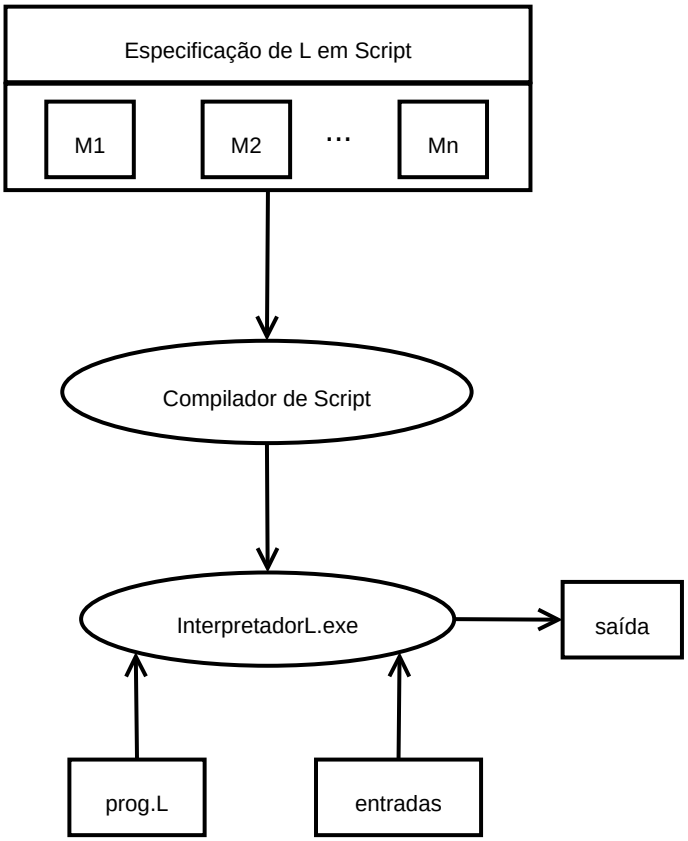

Figura 16: Geração de um interpretador para linguagem L

interpretador via análises léxica e sintática, e posteriormente sua estrutura é representada pela árvore de sintaxe abstrata que será manipulada pelo analisador semântico.

Os analisadores léxicos e sintáticos são códigos na linguagem C, diretamente gerados a partir do *Lex* e do *Yacc*. O analisador semântico é código C gerado a partir da compilação de código *Haskell* usando o GHC. Assim, a princípio, existem duas alternativas de comunicação entre o código dos analisadores sintáticos e semânticos: ou o programa em C acessa as funções em *Haskell*, ou o programa em *Haskell* acessa o programa em C. No entanto, apenas a segunda alternativa é viável, pois não existe tradução direta de *data-types* em *Haskell* para um tipo em C[Fin99].

Para que o código do analisador semântico acesse a árvore de sintaxe abstrata é preciso construir uma interface de comunicação que permita que os elementos do programa representados em C possam ser manipulados em *Haskell*. Um esquema de tradução, como mostra a Figura 18, deve ser montado para possibilitar essa comunicação entre as duas linguagens. O código em

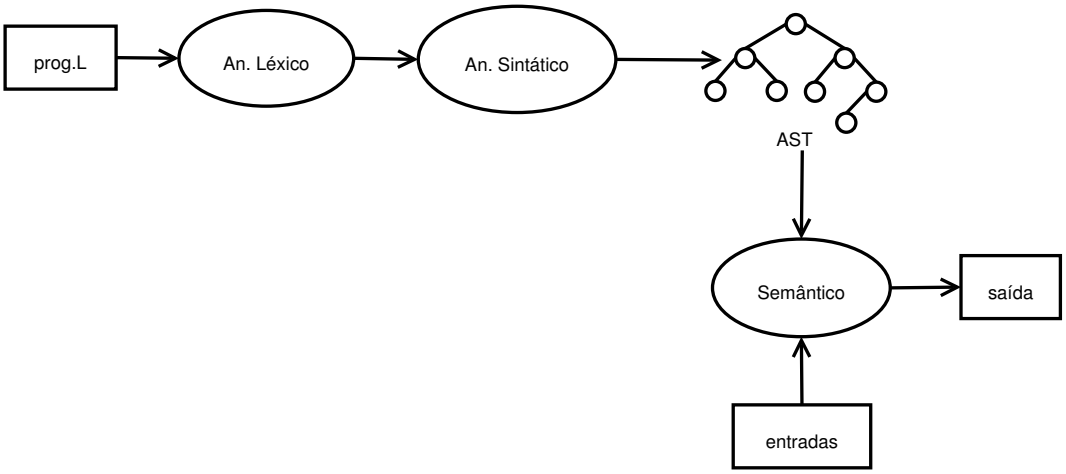

Figura 17: Execução do Interpretador

C representa a árvore de sintaxe abstrata implementada por estruturas em C, e o código em *Haskell* representa as equações semânticas da linguagem. O *tradutor 1* é responsável por determinar como as representações de dados podem ser mapeadas para *Haskell*. O *tradutor 2* obtém os recursos a serem mapeados e os transforma em representações em *Haskell*.

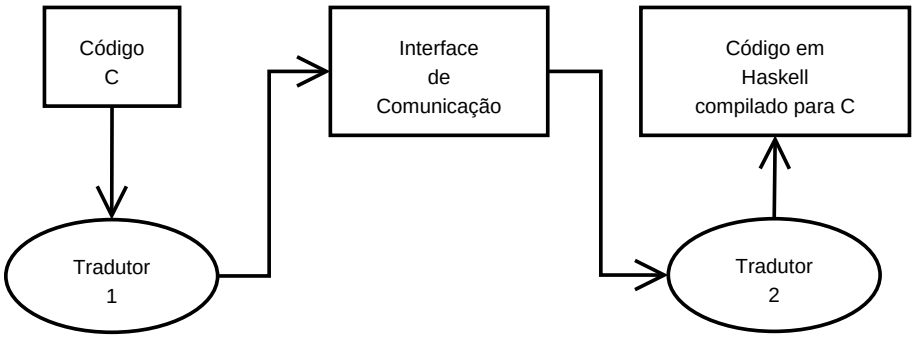

Figura 18: Esquema de tradução do código em C para código em *Haskell*

Um exemplo de uma interface de comunicação seria via arquivo, o *Tradutor 1* escreveria em um arquivo a representação da árvore de sintaxe abstrata e o *Tradutor 2* leria esse arquivo gerando *data types*. Assim a abstração de dados usada em C, por exemplo *structs*, poderia ser traduzida para *data types* em *Haskell*.

A utilização, a princípio, de *Lex* e *Yacc* para a geração dos analisadores léxicos e sintáticos de linguagens especificadas no ambiente proposto justifica-se pela estabilidade e a existência de ampla literatura e familiaridade para

esses *softwares*. No entanto, outras soluções estabelecidas na comunidade científica serão estudadas.

A possibilidade de se usar geradores de *parser* para *Haskell* pode ser interessante, pois seu uso eliminaria a necessidade da interface de comunicação entre os analisadores léxicos e sintáticos escritos em *C* e o analisador semântico escrito em *Haskell*, ilustrada na Figura 18. O geradores de analisadores sintáticos para *Haskell*, *Happy*[4] e *Frown*[5], serão analisados de forma minuciosa e, caso se mostrem suficientemente estáveis para o desenvolvimento deste trabalho, substituirão os geradores de analisadores sintáticos para linguagem *C* inicialmente escolhidos.

7 Validação

A validação deste trabalho será feita por meio de testes do compilador com definições de linguagens em *Script*. As linguagens, em ordem cronológica, a serem especificadas são:

1. Uma linguagem imperativa simples, composta por: expressões aritméticas, variáveis, atribuições, condicionais, repetição e seqüenciadores;
2. Uma linguagem funcional simples composta por: constantes, variáveis, abstração lambda e aplicação de função;
3. Uma linguagem imperativa com abstrações de comandos, expressões, tipos, declarações de variáveis;
4. Uma linguagem funcional tipada baseada no λ-cálculo tipado de [Bar92];
5. Uma linguagem orientada por objetos: Mini Java, definida em [Tir05];
6. Uma linguagem funcional de grande porte: Script, definida em [Tir05].

8 Contribuições

As principais contribuições à área de Linguagens de Programação deste trabalho são:

- desenvolvimento de um ambiente de execução para descrições formais em semântica multidimensional. Uma das propriedades desejáveis a

[4]Disponível em: http://www.haskell.org/happy/
[5]Disponível em: http://www.informatik.uni-bonn.de/~ ralf/frown/

um modelo de especificação formal é a existência de ferramentas de suporte (ver Seção 2.2). Uma nova metodologia para definições formais utilizando semântica denotacional modular está sendo desenvolvida em [Tir05], e é de suma importâncica a existência de um ambiente que auxilie na escrita e verificação das descrições semânticas, e no qual essas descrições possam ser executadas; e

- desenvolvimento de um conjunto de técnicas de compilação, a saber:
  - Costura de código de aspectos presentes nas definições formais, já que aspectos serão usados nas descrições da semântica da linguagem;
  - Definição de um método para realizar a junção dos elementos léxicos, sintáticos e semânticos divididos entre os vários módulos que formam a descrição formal da linguagem.

9 Cronograma

- Janeiro de 2006

  Estruturação da arquitetura do compilador de *Script*.

  Definição da compilação de módulos e da forma de comunicação inter-módulos.

  Escrita da Seção 3.1 da dissertação.

- Fevereiro de 2006

  Análise léxica e sintática de *Script*.

  Escrita das Seções 3.1, 3.2 e 3.3 da dissertação.

- Março de 2006

  Extração dos elementos da definição léxica da linguagem especificada e geração do código intermediário referente a esses elementos (arquivo de entrada para o lex na Figura 14 da Página 22).

  Escrita da Seção 4.2 da dissertação.

- Abril de 2006

  Extração dos elementos da definição sintática da linguagem especificada e geração do código intermediário referente a esses elementos (arquivo de entrada para o *Yacc* na Figura 14 da Página 22).

Escrita da Seção 4.3 da dissertação.

Construção das estruturas de dados para representação em C da árvore de sintaxe abstrata para a linguagem especificada.

- Maio de 2006

  Geração de *data-types* para os componentes da árvore de sintaxe abstrata em *Haskell*.

  Escrita da Seção 4.4 da dissertação.

- Junho de 2006

  Geração das funções em *Haskell* para as equações semânticas da linguagem especificada.

  Escrita da Seção 4.5 da dissertação.

- Julho de 2006

  Implementação das mônadas e transformadores de mônadas necessários para a construção do interpretador para a linguagem especificada.

  Escrita da Seção 4.6 da dissertação.

- Agosto de 2006

  Implementação dos códigos referentes aos pontos de junção e regras de junção usados na definição da linguagem.

- Setembro e Outubro de 2006

  Implementação da costura de código de aspectos utilizados na definição da linguagem.

  Escrita da Seção 4.7 da dissertação.

- Novembro de 2006

  Validação do ambiente proposto por meio da compilação de definições formais de linguagens em *Script*.

  Escrita da Seção 4.8 da dissertação.

- Dezembro de 2006

  Defesa da Dissertação.

10 Sumário da Dissertação

Nesta seção, define-se o sumário da dissertação para que seja possível identificar claramente como se dará as etapas de escrita dos capítulos da mesma e do desenvolvimento do ambiente proposto especificadas no cronograma.

1. Definição do Trabalho de Dissertação
    1.1. Introdução
    1.2. Caracterização do Problema
    1.3. Solução Proposta
    1.4. Metodologia de Desenvolvimento
    1.5. Resultados Obtidos
    1.6. Conclusão
2. Semântica Multidimensional
    2.1. Introdução
    2.2. Semântica de Linguagens de Programação
    2.3. Mônadas
    2.4. Programação Orientada por Aspectos
    2.5. Modularidade em Semântica Denotacional
    2.6. Semântica Multidimensional de Linguagens de Programação
    2.7. Conclusão
3. A Linguagem de Programação Script Estendida
    3.1. Introdução
    3.2. Especificação Léxica
    3.3. Especificação Sintática
    3.4. Especificação Semântica
    3.5. Conclusão
4. Ambiente para Especificação Modular de Linguagens de Programação
    4.1. Introdução
    4.2. Extração dos Elementos Léxicos da Linguagem Especificada
    4.3. Extração dos Elementos Sintáticos da Linguagem Especificada

4.4. Estruturas de Dados da Árvore de Sintaxe Abstrata

4.5. Tradução das Equações Semânticas da Linguagem Especificada para Funções em Haskell

4.6. Mônadas e Transformadores de Mônadas

4.7. Costura de Código

4.8. Validação

5. Conclusões

11 Conclusão

A dissertação de mestrado proposta neste texto objetiva o desenvolvimento de uma ferramenta de suporte às definições em semântica multidimensional de linguagens de programação através da implementação de um compilador para a linguagem *Script*.

O ambiente a ser desenvolvido permitirá descrições modulares, possibilitando escritas incrementais e independentes dos componentes da linguagem, fornecendo controle sobre a complexidade de descrições de linguagens de grande porte. Os módulos, uma vez definidos e testados, poderão ser compostos para formar a descrição completa de uma linguagem.

Descrições de linguagens de programação de porte e complexidades variadas serão realizadas na ferramenta proposta com o intuito de validar e verificar o seu correto funcionamento.

Referências

[Bar92] Henk Barendregt, *Lambda calculi with types*, Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon, vol. 2, 1992.

[Big99] Roberto S. Bigonha, *Script 2.0: An object oriented language for denotational semantics*, Tech. Report Technical Report LLP 013/99, Laboratório de Linguagens de Programação, UFMG, 1999.

[BLM92] Doug Brown, John Levine, and Tony Mason, *lex & yacc (A Nutshell Handbook)*, O'Reilly Media, 1992.

[dM96] Hermano Perrelli de Moura, *An Overview of Action Semantics*, I Simpósio Brasileiro de Linguagens de Programação (1996).

[Fin99] Sigbjorn Finne, *HaskellDirect User's Manual*, http://www.haskell.org/hdirect/user.html#toc2, November 1999.

[Gor79] M. J.C. Gordon, *The Denotational Description of Programming Languages - An Introduction*, Springer-Verlag, 1979.

[Gur95] Yuri Gurevich, *Evolving algebras 1993: Lipari guide*, Specification and validation methods (1995).

[Hoa69] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, 12.10 (1969).

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-oriented programming*, Proceedings of the 11th European Conference on Object-Oriented Programming. (1997).

[LH96] Sheng Liang and Paul Hudak, *Modular Denotational Semantics for Compiler Construction*, 6th European Symposium on Programming Languages and Systems (1996).

[LHJ95] Shenk Liang, Paul Hudak, and Mark Jones, *Monad Transformers and Modular Interpreters*, Conference Record of POPL'95: 22nd ACM SIGPLAN-SICACT Symposium on Principles of Programming Languages (1995).

[Lia98] Sheng Liang, *Modular Monadic Semantics and Compilation*, Ph.d. thesis, Yale University, 1998, adviser-Paul Hudak.

[Mos88] Peter D. Mosses, *The Modularity of Action Semantics*, Technical Report DAIMI IR-75 (1988).

[Mos96a] _____, *A Tutorial On Action Semantics*, Tutorial notes for FME'96: Formal Methods Europe (1996).

[Mos96b] _____, *Theory and Practice of Action Semantics*, Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science (1996).

[Mos01] _____, *What Use Is Formal Semantics?*, PSI Report, Perspective of System Informatics 2001 (2001).

[MW86] Peter D. Mosses and David A. Watt, *The Use of Action Semantics*, Technical Report DAIMI PB-217 (1986).

[Plo81] Gordon Plotkin, *A Structural Approach to Operational Semantics*, Technical Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (1981).

[Sto77] Joseph Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA (1977).

[Tho99] Simon Thompson, *Haskell: The craft of functional programming*, 2a edição ed., Addison-Wesley, 1999.

[Tir05] Fabio Tirelo, *Semântica Multidimensional de Linguagens de Programação*, Proposta de tese de doutorado, Universidade Federal de Minas Gerais, 2005, orientador: Roberto da Silva Bigonha.

[Wad92] Philip Wadler, *The essence of functional programming*, 19'th Annual Sympoium on Principles of Programming Languages (1992).

[ZX04] Yingzhou Zhang and Baowen Xu, *A survey of semantic description frameworks for programming languages*, ACM SIGPLAN Notices (2004).