O Uso das Semânticas Indutiva e Denotacional na Tradução Fonte-a-Fonte de Linguagens de Programação

Alberto Pardo
Silvio Lemos Meira

Departamento de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851, 50739 Recife - PE - Brasil

RESUMO

Consideramos o problema da tradução fonte-a-fonte entre linguagens de programação visando a geração de tradutores. Nossa abordagem é deduzir as regras de compilação a partir da semântica formal de ambas linguagens usando um meta-compilador. Neste sentido fazemos um estudo de dois formalismos para a especificação formal da semântica de linguagens na procura de um método que se adapte a nossa aplicação. Regras para a derivação da semântica de frases compostas numa linguagem são apresentadas para ambos formalismos. Mostramos como estas regras podem servir para a prova de propriedades sem uso de indução, e como isto pode assistir o processo de dedução de regras de compilação.

ABSTRACT

We study the automatic generation of source-to-source programming language transiators, using a meta-compiler to process the formal semantics of such languages. In particular, two different semantic environments are considered in order to try and establish their adequacy for the task. Rules for the semantics of compound phrases are presented and used to prove properties of languages in an induction-free style. We also show how this can help the translation process.

1 Introdução

Semântica formal de linguagens de programação fornece um marco adequado para descrever precisa e concisamente os conceitos referentes a linguagens, para raciocinar acerca de propriedades das linguagens, assim como para projeto. Diversos métodos para a definição formal da semântica de linguagens são disponíveis, cada um deles com diferentes estilos e fundamentos matemáticos.

Neste trabalho concentramos o estudo em dois estilos de definição semântica atualmente bem difundidos, que são o *denotacional* e o *operacional indutivo*. O objetivo é aprofundar o estudo destes métodos a fim de resolver o problema de tradução fonte-a-fonte, o qual podemos formular nos seguintes termos:

> dadas quaisquer duas linguagens de programação L_{origem} e $L_{destino}$, gerar um tradutor T que converta programas escritos em L_{origem} em programas, *semanticamente equivalentes*, em $L_{destino}$.

Neste sentido definimos um ambiente no qual, dadas as descrições semânticas das duas linguagens (origem e destino), possamos derivar em forma semi-automática as regras de tradução entre as linguagens. O objetivo final que buscamos é a construção de um meta-compilador que tenha como entrada as descrições semânticas e como saida o tradutor entre as linguagens, ou seja:

$\mathcal{MC}$: DescLing → DescLing → Ling → Ling

tal que

$\mathcal{T} = \mathcal{MC}\ \mathrm{DescL}_{origem}\ \mathrm{DescL}_{destino}$

2 Semântica Formal de Linguagens de Programação

Nesta seção fazemos um estudo comparativo entre Semântica Denotacional e Semântica Indutiva Estruturada. Este estudo tem como objetivo clarificar os conceitos e fundamentos teóricos envolvidos nos dois métodos, vitais para o desenvolvimento da solução do problema da tradução fonte-a-fonte.

Do ponto de vista semântico, pouco importa a particular e concreta representação das construções sintáticas da linguagem, mas sim sua estrutura abstrata. Por esta razão é que os dois métodos utilizam sintaxe abstrata como forma de definir a sintaxe das linguagens.

Primeiramente apresentaremos brevemente cada método para em seguida estudar as relações entre estes.

2.1 Semântica Indutiva Estruturada

Semântica Indutiva Estruturada (SIS) [Ast89] é uma nova versão da Semântica Operacional Estruturada introduzida e desenvolvida por Plotkin [Plo81]. A SIS de uma linguagem é uma definição indutiva, em um sistema lógico de dedução, guiada pela sintaxe abstrata da linguagem.

SIS permite definições mediante sistemas indutivos que descrevem diferentes estratégias de computação em vários níveis de detalhe. Por exemplo, é possível escrever definições semânticas com o uso de sistemas "*big-*", "*small-*" ou "*mixed-step*".

Um sistema "big-step" define a avaliação dos objetos sintáticos diretamente, sem passos intermediários. Uma definição "big-step" pode ser canonicamente convertida em uma definição denotacional e vice-versa, como será visto posteriormente. Os sistemas "small-" e "mixed-step" definem a avaliação dos objetos sintáticos passo a passo. Neste trabalho vamos assumir que as definições das linguagens são feitas usando sistemas "big-step".

Uma SIS é um conjunto de relações em um sistema indutivo. As cláusulas que definem cada relação são guiadas pela estrutura sintática dos objetos cuja semântica está sendo definida (neste sentido é que dizemos que são definições estruturais). Estas relações associam um valor em um domínio semântico a cada objeto sintático da linguagem.

Exemplo: Expressões aritméticas com variáveis (Sistema "big-step").

• Sintaxe Abstrata:

$$\mathbf{e} ::= \mathbf{e}_1 + \mathbf{e}_2 \mid \mathbf{e}_1 * \mathbf{e}_2 \mid \mathbf{n} \mid \mathbf{id}$$

onde $\mathbf{e}, \mathbf{e}_1, \mathbf{e}_2 \in \mathsf{Exp}$; $\mathbf{n} \in \mathsf{Num}$; $\mathbf{id} \in \mathsf{Id}$.

Neste exemplo, os domínios semânticos são os inteiros (Int) e a memória (Memória), que associa identificadores com valores. Formalmente Memória = $(\mathsf{Id} \rightarrow \mathsf{Int})_f$, onde Id é um conjunto de identificadores, e $(\mathsf{A} \rightarrow \mathsf{B})_f$ é o domínio dos mapeamentos finitos de A para B.

A semântica de uma expressão **e**, denotada por $\mathcal{E}[\![\mathbf{e}]\!]$, é uma função que, dada uma memória como entrada, resulta num valor inteiro.
O tipo de $\mathcal{E}$ é:

$$\mathcal{E}\text{: Exp} \rightarrow (\text{Memória} \rightarrow \text{Int})$$

Vamos escrever $\langle \mathbf{e}, m\rangle \xrightarrow{\mathcal{E}} v$ em vez de $\mathcal{E}[\![\mathbf{e}]\!]m = v$. A relação $\xrightarrow{\mathcal{E}}$ é definida indutivamente pelas cláusulas a seguir:

$$\langle \mathbf{n}, m\rangle \xrightarrow{\mathcal{E}} v \quad v = \overline{\mathbf{n}}$$

$$\langle \mathbf{id}, m\rangle \xrightarrow{\mathcal{E}} m(\mathbf{id}) \quad \mathbf{id} \in dom(m)$$

$$\frac{\langle \mathbf{e}_1, m\rangle \xrightarrow{\mathcal{E}} v_1 \quad \langle \mathbf{e}_2, m\rangle \xrightarrow{\mathcal{E}} v_2}{\langle \mathbf{e}_1 + \mathbf{e}_2, m\rangle \xrightarrow{\mathcal{E}} v} \quad v = v_1 + v_2$$

$$\frac{\langle \mathbf{e}_1, m\rangle \xrightarrow{\mathcal{E}} v_1 \quad \langle \mathbf{e}_2, m\rangle \xrightarrow{\mathcal{E}} v_2}{\langle \mathbf{e}_1 * \mathbf{e}_2, m\rangle \xrightarrow{\mathcal{E}} v} \quad v = v_1 \times v_2$$

para $\mathbf{e}_1, \mathbf{e}_2 \in$ Exp; $\mathbf{n} \in$ Num; $\mathbf{id} \in$ Id; $v, v_1, v_2 \in$ Int e $m \in$ Memória, onde $\overline{\mathbf{n}}$ denota a interpretação de **n** no domínio Int.

No exemplo, fica claro como as definições semânticas são estruturais, no sentido que a semântica de um objeto é obtida a partir da semântica de seus subcomponentes. A SIS também permite fazer definições recursivas sem usar ponto fixo (ao contrário da semântica denotacional). Correspondente a este tipo de definições temos um princípio de (prova por) indução mais geral que o princípio de indução estrutural [Ast89]. Usando o mesmo é possível ter definições semânticas, não estruturais e recursivas na sintaxe, do tipo:

$$\frac{\langle \mathbf{b}, m\rangle \xrightarrow{\mathcal{B}} \mathit{false}}{\langle \mathbf{while\ b\ do\ c\ od}, m\rangle \xrightarrow{\mathcal{C}} m}$$

$$\frac{\langle \mathbf{b}, m\rangle \xrightarrow{\mathcal{B}} \mathit{true} \quad \langle \mathbf{c}, m\rangle \xrightarrow{\mathcal{C}} m' \quad \langle \mathbf{while\ b\ do\ c\ od}, m'\rangle \xrightarrow{\mathcal{C}} m''}{\langle \mathbf{while\ b\ do\ c\ od}, m\rangle \xrightarrow{\mathcal{C}} m''}$$

onde o objeto sintático (neste caso o comando **while**) cuja semântica está sendo definida aparece tanto nas premissas como na conclusão. Obviamente, definições não recursivas podem ser feitas para este tipo de objeto sintático como as encontradas em [Ast89].

2.2 Semântica Denotacional

Semântica denotacional (SD) [Sch86,Mos89b] está baseada na teoria dos domínios, um ambiente computacional para funções e dados, desenvolvido inicialmente por D. Scott. O princípio é o de atribuir modelos matemáticos a linguagens de programação. Modelos são construídos a partir de entidades matemáticas básicas, tais como funções, tuplas, etc..

Numa descrição denotacional, atribuímos um objeto semântico a toda frase da linguagem, o qual chamamos a *denotação* da frase. A denotação de frases compostas só depende da denotação das suas subfrases, o que é chamado de *composicionalidade*. Os espaços matemáticos usados para denotar objetos das linguagens são chamados *domínios semânticos*.

Mediante *funções semânticas* mapeamos frases da linguagem para suas denotações. Desta forma a semântica de uma linguagem pode ser especificada definindo uma função semântica para cada tipo de frase. Estas funções são definidas indutivamente, escrevendo uma *equação semântica* para cada produção da sintaxe abstrata da linguagem. Tais equações têm a forma

$$\mathcal{F}[\text{Op } (\text{fr}_1,\ldots,\text{fr}_n)] = g(\mathcal{F}_1[\text{fr}_1],\ldots,\mathcal{F}_n[\text{fr}_n])$$

que podem ser vistas como definições de homomorfismos [GTW78] por indução estrutural.

Exemplo: Expressões aritméticas com variáveis, como para a SIS.

- Sintaxe Abstrata:

$$\mathbf{e} ::= \mathbf{e}_1 + \mathbf{e}_2 \mid \mathbf{e}_1 * \mathbf{e}_2 \mid \mathbf{n} \mid \mathbf{id}$$

onde $\mathbf{e}, \mathbf{e}_1, \mathbf{e}_2 \in$ Exp; $\mathbf{n} \in$ Num; $\mathbf{id} \in$ Id.

- Semântica

Memória = (Id → Int)

$\mathcal{E}$: Exp → (Memória → Int)

$\mathcal{N}$: Num → Int

$$\mathcal{E}[\![\mathbf{e}_1 + \mathbf{e}_2]\!] = \lambda m.\ \mathcal{E}[\![\mathbf{e}_1]\!]m + \mathcal{E}[\![\mathbf{e}_2]\!]m$$
$$\mathcal{E}[\![\mathbf{e}_1 * \mathbf{e}_2]\!] = \lambda m.\ \mathcal{E}[\![\mathbf{e}_1]\!]m \times \mathcal{E}[\![\mathbf{e}_2]\!]m$$
$$\mathcal{E}[\![\mathbf{id}]\!] = \lambda m.\ m(\mathbf{id})$$
$$\mathcal{E}[\![\mathbf{n}]\!] = \lambda m.\ \mathcal{N}[\![\mathbf{n}]\!]$$
$$\mathcal{N}[\![\mathbf{n}]\!] = \text{Não especificado.}$$

Tal como para SIS, na SD as definições são estruturais, como é o caso do exemplo anterior. Definições recursivas na sintaxe não são permitidas neste formalismo pelas seguintes razões:

- o significado de uma frase composta só depende do significado de suas subfrases próprias.
- especificações recursivas podem não definir univocamente uma função, ou seja várias funções podem satisfazer uma mesma definição recursiva.
- uma definição recursiva pode ser vista como uma descrição operacional do comportamento da função ao computar seus valores.

A SD trabalha com valores matemáticos e não com descrições operacionais, além de usar somente expressões de função que univocamente determinam um valor. Para uma discussão profunda destes aspectos o leitor pode se referir a [Sch86]. Para dar o significado a construções sintáticas recursivas e iterativas, a SD usa uma teoria conhecida como *semântica do menor ponto fixo* [Sch86,Pau87]. Esta teoria permite fazer definições novamente estruturais mediante o uso do operador de ponto fixo *fix* (entendendo que o significado de uma função especificada recursivamente é o limite de uma determinada seqüência de funções). Um princípio de indução de ponto fixo permite provar propriedades sobre funções definidas com o uso do operador *fix*. Vejamos como fica a definição da semântica do comando **while** usando uma definição por ponto fixo (sendo interessante que o leitor compare esta definição com a dada em forma recursiva na SIS).

$\mathcal{B}$: Bexp → (Memória → Bool)

$\mathcal{C}$: Comando → (Memória → Memória)

$\mathcal{C}[\![\textbf{while b do c od}]\!] = fix\ \mathrm{F}$

onde $\mathrm{F} : (\text{Memória} \to \text{Memória}) \to (\text{Memória} \to \text{Memória})$

$\mathrm{F} = \lambda f \cdot \lambda m \cdot \text{if } \mathcal{B}[\![\textbf{b}]\!]m \text{ then } f(\mathcal{C}[\![\textbf{c}]\!]m) \text{ else } m$

Esta definição diz que a semântica do comando **while** é o ponto fixo do funcional F, ou seja, $\mathcal{C}[\![\textbf{while b do c od}]\!]$ é uma função g : Memória → Memória que cumpre a especificação recursiva $g = \mathrm{F}\ g$. Vejamos como é possível chegar a este resultado. A semântica do comando **while** pode ser descrita mediante a seguinte equação recursiva:

$$\mathcal{C}[\![\textbf{while b do c od}]\!] = \lambda m \cdot \text{if } \mathcal{B}[\![\textbf{b}]\!]m \text{ then } \mathcal{C}[\![\textbf{while b do c od}]\!](\mathcal{C}[\![\textbf{c}]\!]m) \text{ else } m$$

que pode ser escrita como

$$\mathcal{C}[\![\textbf{while b do c od}]\!] = (\lambda f \cdot \lambda m \cdot \text{if } \mathcal{B}[\![\textbf{b}]\!]m \text{ then } f(\mathcal{C}[\![\textbf{c}]\!]m) \text{ else } m)\ (\mathcal{C}[\![\textbf{while b do c od}]\!])$$

concluindo então que

$$\mathcal{C}[\![\textbf{while b do c od}]\!] = \mathrm{F}\ (\mathcal{C}[\![\textbf{while b do c od}]\!])$$

ou seja $\mathcal{C}[\![\textbf{while b do c od}]\!] = fix\ \mathrm{F}$.

A descrição semântica de uma linguagem de programação pode ser vista também como uma especificação funcional dos requerimentos para a implementação da linguagem. Existem diversos métodos que permitem derivar compiladores ou interpretadores a partir da especificação denotacional de linguagens.

2.3 SIS vs. SD

Como foi mencionado na apresentação da SIS, as definições mediante sistemas "big-step" podem ser canonicamente convertidas em definições denotacionais e vice-versa. Claramente esta relação vem do fato que as definições estruturais por sistemas "big-step" possuem uma forma composicional ou, em outras palavras, respeitam o princípio de *composicionalidade* definido na SD. É assim que a função semântica $\mathcal{E}$ definida na SIS do exemplo pode ser definida usando indução estrutural (ou formalmente como um homomorfismo) como:

$$\mathcal{E}[\![\mathbf{e}_1 + \mathbf{e}_2]\!]m = \mathcal{E}[\![\mathbf{e}_1]\!]m + \mathcal{E}[\![\mathbf{e}_2]\!]m$$
$$\mathcal{E}[\![\mathbf{e}_1 * \mathbf{e}_2]\!]m = \mathcal{E}[\![\mathbf{e}_1]\!]m \times \mathcal{E}[\![\mathbf{e}_2]\!]m$$
$$\mathcal{E}[\![\mathbf{id}]\!]m = m(\mathbf{id})$$
$$\mathcal{E}[\![\mathbf{n}]\!]m = \overline{\mathbf{n}},$$

que não é nada mais que a definição de $\mathcal{E}$ através de SD. Ou seja, a diferença entre definições semânticas equivalentes em SIS e SD é originada do estilo: SD define um homomorfismo em forma estrutural, enquanto que SIS define uma função (ou relação) também em forma estrutural mas por um sistema lógico de dedução.

Da mesma forma pode se ver que uma definição mediante SD pode ser canonicamente convertida numa equivalente mediante um sistema big-step. Para maiores detalhes e exemplos, o leitor pode se referir a [Ast89].

Uma questão de nota neste ponto é a das definições recursivas. Como já visto, na SIS é possível fazer definições não estruturais, recursivas na sintaxe, enquanto que na SD estas não são permitidas, usando em seu lugar definições mediante ponto fixo. É então que, para o caso de

definições recursivas como no comando **while**, a conversão canônica de SIS para SD resulta numa definição que não pode ser vista como denotacional, mas sim como uma definição operacional usando uma lógica equacional (ou seja, um estilo funcional). No caso do comando **while** esta definição fica da seguinte forma:

$$C[\textbf{while b do c od}]m = \text{if } B[\text{b}]m \text{ then } C[\textbf{while b do c od}](C[\text{c}]m) \text{ else } m$$

onde $C[\textbf{while b do c od}]$ não é mais que o ponto fixo da equação do **while** definida mediante SD. Na Seção 4 veremos como este tipo de definição operacional, mediante lógica equacional, pode ser usado na prova de propriedades sem fazer uso de ponto fixo, e como isto pode auxiliar na solução do problema da tradução fonte-a-fonte.

3 Meta-regras para a Manipulação de SIS

Em lógica equacional, existem certas regras que permitem fazer manipulação das equações, e o resultado da sua aplicação é a definição de novas equações. Isto acontece, por exemplo, em muitas das metodologias atuais de transformação de programas. A idéia é que, por exemplo, ao invés de provar que as duas definições de uma certa função f são equivalentes, deriva-se uma das definições de f a partir da outra, mediante o uso de uma série de regras de transformação que preservam a semântica. Em geral o objetivo destas técnicas de transformação é chegar a definições mais eficientes das funçõesem pauta.

Como veremos posteriormente, o uso de regras de transformação bem conhecidas como *fold/unfold* vai possibilitar a prova de propriedades em definições operacionais da semântica de linguagens usando lógica equacional.

Numa abordagem similar à de lógica equacional, nosso trabalho visa definir um conjunto de regras (que a partir deste instante passaremos a chamar de *meta-regras*) que permita manipular cláusulas (formalmente *regras* de inferência) de uma definição mediante SIS, de forma a obter novas regras derivadas. O objetivo principal é obter regras da SIS para objetos sintáticos compostos (ou seja, termos da sintaxe abstrata nos quais instamos variáveis por outros termos que possivelmente contém variáveis).

Iniciaremos o estudo de meta-regras fazendo uma revisão da notação usada para as regras das SIS. Uma regra da SIS, segundo [Ast89], tem a seguinte forma:

$$\frac{P_1 \ldots P_n}{\text{Conclusão}} \quad \text{Cond}$$

onde as P_i são denominadas "premissas sintáticas", por serem aplicadas sobre objetos sintáticos, enquanto que as condições em Cond são "premissas semânticas", por só conterem objetos semânticos. As premissas sintáticas avaliam as sub-frases do objeto sintático sendo especificado na regra. As premissas semânticas avaliam as condições para a aplicação da regra ou simplesmente fazem a ligação de uma variável a uma expressão, tendo a seguinte sintaxe informal:

- $v \triangleq op(t_1,\ldots,t_n)$
  onde op é uma operação dos domínios semânticos e $t_1,\ldots,t_n$ são termos semânticos. v é uma variável semântica à qual está-se fazendo uma ligação para ser usada tanto em alguma premissa sintática (como entrada), na conclusão (como saída) ou em outra premissa semântica.

- $op(t_1,\ldots,t_n)$
  onde op é uma operação dos domínios semânticos, com co-domínio Bool.

A conclusão simplesmente define uma relação entre as variáveis semânticas, de entrada e saída, e o objeto sintático sendo especificado.

Introduzimos na sintaxe das regras uma operação OR nas premissas (como usado por exemplo em [BL84]), a qual permite declarar numa regra só a semântica de um objeto sintático que com a notação original usaria mais de uma regra, como é o caso do comando **while** mostrado anteriormente, na Seção 2.

Com estas modificações a sintaxe abstrata das regras da SIS tem a seguinte forma:

$R \in$ Regra_SIS	$Esem \in$ Expressão_semântica
$P \in$ Premissas	$Vsem \in$ Variável_semântica
$C \in$ Conclusão	$vi \in$ Variável_semântica_entrada
$Psint \in$ Premissas_sintáticas	$vo \in$ Variável_semântica_saida
$Psem \in$ Premissas_semânticas	$Vsint \in$ Variável_sintática
$Rsint \in$ Relação_sintática	$OPsint \in$ Operação_sintática
$Rsem \in$ Relação_semântica	$OPsem \in$ Operação_semântica
$Esint \in$ Expressão_sintática	$NR \in$ Nome_Relação

$$R ::= \frac{P}{C}$$

$$P ::= P_1 \textbf{ OR } P_2 \mid (Psint, Psem)$$

$$Psint ::= \{Rsint_1, \ldots, Rsint_n\}$$

$$Psem ::= \{Rsem_1, \ldots, Rsem_n\}$$

$$C ::= \langle Esint, vi_1, \ldots, vi_n \rangle \xrightarrow{NR} \langle vo_1, \ldots, vo_m \rangle$$

$$Rsint ::= \langle Esint, Esem_1, \ldots, Esem_n \rangle \xrightarrow{NR} \langle vo_1, \ldots, vo_m \rangle$$

$$Rsem ::= Vsem \hat{=} Esem \mid Esem$$

$$Esint ::= OPsint(Esint_1, \ldots, Esint_n) \mid OPsint \mid Vsint$$

$$Esem ::= OPsem(Esem_1, \ldots, Esem_n) \mid OPsem \mid Vsem$$

3.1 As Meta-regras

A seguir apresentamos as principais meta-regras definidas e utilizadas no projeto. Para um estudo mais exaustivo, o leitor deve se referir a [Par90]

Renomeação

Esta meta-regra permite renomear as variáveis das regras. Meta-regras similares a esta existem no λ-cálculo [Bar81] e em sistemas de semântica natural [BH88], por exemplo.

$$\text{(Renomeação)} \quad \frac{\text{Regra}}{\{y \,/\, x\}\text{Regra}} \qquad \text{Tipo}(x) = \text{Tipo}(y)$$

Substituição

Usando esta meta-regra podemos fazer substituição de variáveis por termos, para o caso de variáveis sintáticas.

$$\text{(Substituição)} \quad \frac{\text{Regra}}{[t \,/\, x]\text{Regra}} \qquad \begin{array}{l} x \in \text{VarSint(Regra)} \\ \text{Tipo}(x) = \text{Tipo}(t) \end{array}$$

Aplicação de conclusão

Usando esta meta_regra podemos dar semântica a objetos sintáticos compostos, pois ela permite combinar a semântica dos sub-componentes.

$$\text{(Aplic-conc)}\quad \frac{\dfrac{P_1}{C_1} \qquad \dfrac{(\{C_1\} \cup Psi_2,\ Pse_2)\ \text{OR}\ P}{C}}{\dfrac{[P_1\ \text{AND}\ (Psi_2.Pse_2)]\ \text{OR}\ P}{C}}$$

onde a operação AND sobre as premissas não é parte da notação "padrão" da SIS. Esta operação pode ser definida indutivamente como:

$[(Psi_1.Pse_1)\ \text{AND}\ (Psi_2.Pse_2)] \equiv (Psi_1 \cup Psi_2.Pse_1 \cup Pse_2)$

$[(P_1\ \text{OR}\ P_2)\ \text{AND}\ (Psi.Pse)] \equiv [P_1\ \text{AND}\ (Psi.Pse)]\ \text{OR}\ [P_2\ \text{AND}\ (Psi.Pse)]$

Aplicação de premissas

Nesta meta-regra se resolve a substituição de premissas (numa regra) por conclusões (de outra regra, onde as premissas também aparecem).

$$\text{(Aplic-prem)}\quad \frac{\dfrac{(Psi_1.Pse_1)}{C_1} \qquad \dfrac{(Psi_1 \cup Psi_2\ ,\ Pse_1 \cup Pse_2)\ \text{OR}\ P}{C}}{\dfrac{(\{C_1\} \cup Psi_2.Pse_2)\ \text{OR}\ P}{C}}$$

3.2 Um Sistema Formal para as Meta-regras

Um *sistema formal* é composto por uma linguagem (que é o conjunto das fórmulas bem formadas (fbf) do sistema) e um conjunto de regras inferência que permitem derivar fbf a partir de outras fbf. Existem certas regras de inferência que não contêm premissas, os *aziomas* (ou de outro ponto de vista fbf postuladas válidas).

Para o nosso caso, o sistema formal está definido por: as possíveis regras das SIS como as fbf, as meta-regras como o conjunto das regras de inferência, e as regras da definição SIS da particular linguagem que se está dando a semântica como os axiomas. Por exemplo, para o caso da definição da semântica de expressões aritméticas com variáveis da Seção 2, o sistema formal teria as 4 regras semânticas como axiomas.

3.3 Relação com Lógica Equacional

Como já mencionamos, em lógica equacional existem certas regras de transformação que possibilitam a obtenção de novas equações a partir de outras dadas. Uma metodologia de transformação bem conhecida é o chamado método *fold-unfold* [Dar82], descrito pelas seguintes regras básicas:

- *Definição*: acrescentar uma nova equação cujo lado esquerdo não é uma instância de nenhuma das outras.
- *Instanciação*: substituir um valor numa equação.
- *Unfold*: se `e = e1` e `f = f1` são duas equações, e há uma ocorrência de uma instância de `e` em `f1`, substituir `e` por `e1` em `f1`, obtendo `f2`, e acrescentar a equação `f = f2`.
- *Fold*: se `e = e1` e `f = f1` são duas equações, e há uma ocorrência de uma instância de `e1` em `f1`, substituir `e1` por `e` em `f1`, obtendo `f2`, e acrescentar a equação `f = f2`.

Vejamos agora um exemplo de como estas regras podem ser usadas para obter a semântica de comandos compostos numa linguagem de programação simples, com a seguinte sintaxe (parcial):

$$\begin{array}{lll} c & ::= & c_1 \,;\, c_2 \mid \textbf{while b do c} \\ e & ::= & e_1 + e_2 \mid e_1 * e_2 \mid n \mid id \\ b & ::= & e_1 = e_2 \mid \textbf{not b} \end{array}$$

onde $c, c_1, c_2 \in Com$; $e, e_1, e_2 \in Exp$; $b \in Bexp$; $id \in Id$.

Damos as equações semânticas, em forma operacional, só dos objetos sintáticos que são usados no exemplo de derivação.

$$C[\![c_1 \,;\, c_2]\!]m = C[\![c_2]\!](C[\![c_1]\!]m) \tag{1}$$

$$C[\![\textbf{while b do c od}]\!]m = \text{if } \mathcal{B}[\![b]\!]m \text{ then } C[\![\textbf{while b do c od}]\!](C[\![c]\!]m) \text{ else } m \tag{2}$$

$$\mathcal{B}[\![\textbf{not b}]\!]m = \neg\, (\mathcal{B}[\![b]\!]m) \tag{3}$$

A derivação a seguir tem como objetivo achar uma equação semântica recursiva para o objeto sintático composto c ; **while not b do c od**.

$C[\![c \,;\, \textbf{while not b do c od}]\!]m$

$$\begin{array}{lll} = & C[\![\textbf{while not b do c od}]\!](C[\![c]\!]m) & \text{(1,Instanc.)} \\ = & \text{if } \mathcal{B}[\![\textbf{not b}]\!](C[\![c]\!]m) \text{ then } C[\![\textbf{while not b do c od}]\!](C[\![c]\!](C[\![c]\!]m)) \text{ else } C[\![c]\!]m & \text{(2,Unfold)} \\ = & \text{if } \neg\, \mathcal{B}[\![b]\!](C[\![c]\!]m) \text{ then } C[\![\textbf{while not b do c od}]\!](C[\![c]\!](C[\![c]\!]m)) \text{ else } C[\![c]\!]m & \text{(3,Unfold)} \\ = & \text{if } \neg\, \mathcal{B}[\![b]\!](C[\![c]\!]m) \text{ then } C[\![c \,;\, \textbf{while not b do c od}]\!](C[\![c]\!]m) \text{ else } C[\![c]\!]m & \text{(2,Fold)} \end{array}$$

Uma abordagem similar para a dedução de definições semânticas para objetos compostos mediante o uso de regras de transformação foi usado para o caso de definições SIS em [PM90]. Nesse trabalho é apresentada uma derivação da regra semântica para o mesmo objeto sintático que neste exemplo, com o uso das meta-regras. Fazendo uma análise comparativa das meta-regras com as regras do método *fold-unfold* aqui descritas, podemos apreciar uma grande similaridade entre elas, como é descrito a seguir:

(1) A meta-regra **Aplicação de conclusão** permite substituir a conclusão C de uma regra R por suas premissas se C aparece como parte das premissas de outra regra R_1, obtendo assim uma nova regra R_2. Se comparamos o comportamento desta meta-regra com o da regra **Unfold**, podemos ver que é extremamente similar.

(2) A meta-regra **Aplicação de premissas** permite, de forma contrária à meta-regra anterior, substituir as premissas de uma regra R por sua conclusão se elas aparecem como parte das premissas de outra regra R_1, obtendo assim uma nova regra R_2. Se comparamos o comportamento desta meta-regra com o da regra **Fold** podemos ver que também é extremamente similar.

(3) A meta-regra **Substituição** pode ser vista como um caso particular da regra de **Instanciação**, dado que só pode se aplicar a variáveis sintáticas.

(4) A meta-regra **Renomeação** está relacionada com a α-conversão do λ-cálculo.

De fato, as meta-regras que definimos não são mais que o equivalente às regras equacionais no sistema da SIS. Uma forma possível de verificar esta equivalência é mediante o uso da conversão canônica entre semânticas definidas em sistemas SIS ("big-step") e lógica equacional. Ou seja, verificar de alguma forma que:

Meta-regra r = Conversão$_{EQ \to SIS}$(*Regra*(Conversão$_{SIS \to EQ}$ r))

onde *Meta-regra* e *Regra* são as regras que se quer provar equivalentes e r é o conjunto de regras da SIS sobre as quais estamos aplicando *Meta-regra*.

4 Prova de Propriedades

A prova de propriedades é de suma importância na validação de especificações semânticas, dado que estas, em geral, modelam conceitos informais, pelo que é impossível provar que as especificações cumprem os requisitos. Mostraremos posteriormente como a prova de propriedades pode ser de grande ajuda na resolução do problema de tradução fonte-a-fonte.

Tanto no caso de lógica equacional como da SIS, as regras de manipulação são de grande ajuda para a prova de propriedades das linguagens sendo definidas.

Na SD é usada em geral a indução de ponto fixo para a prova de propriedades que envolvem definições recursivas dadas mediante o uso do operador *fix*. No caso da SIS, usamos indução generalizada para casos de definições recursivas. Algum princípio de indução equivalente ao princípio de indução generalizada tem que ser usado para as definições operacionais mediante lógica equacional.

Mas, em vários casos de definições recursivas mediante SIS ou lógica equacional, temos visto que não é necessário o uso de tais princípios de indução para realizar provas, sendo simplesmente um problema de transformação de expressões. Mostraremos isto mediante um exemplo simples.

Suponha que acrescentamos o comando **repeat** aos comandos já existentes da linguagem definida na Seção anterior, ou seja:

c ::= $\cdots$ | **repeat** c **until** b

com sua semântica operacional habitual:

C[**repeat** c **until** b]m = if B[b](C[c]m) then C[c]m else C[**repeat** c **until** b](C[c]m)

Na Seção anterior derivamos uma equação recursiva para a semântica do objeto sintático composto c ; **while not** b **do** c **od**. Usando esta equação e a da semântica do comando **repeat** podemos provar a propriedade:

$\forall m$:Memória· C[**repeat** c **until** b]m = C[c ; **while not** b **do** c **od**]m

Para isto primeiro fazemos as seguintes nomeações, as quais deixam em evidência a relação entre as duas expressões:

$f = C$[c ; **while not** b **do** c **od**]
$h = C$[**repeat** c **until** b]
$g = C$[c]
$b = B$[b]

Temos então que:

$f\ m$ = if $\neg\ b(g\ m)$ then $f(g\ m)$ else $g\ m$

$h\ m$ = if $b(g\ m)$ then $g\ m$ else $h(g\ m)$

Mas por propriedade do if_then_else temos:

$f\ m$ = if $b(g\ m)$ then $g\ m$ else $f(g\ m)$

com o que vemos que as funções f e h são a mesma a menos de nome (ou que as equações que definem f e h podem ser unificadas, aspecto discutido na próxima seção). O importante aqui é notar que a prova de uma propriedade reduziu-se à manipulação das expressões que formam parte do predicado da prova mediante o uso das regras de transformação.

Provas envolvendo objetos sintáticos mais complexas têm sido feitas usando a metodologia do exemplo, mostrando na maioria dos casos sua adequabilidade, a menos da introdução de algumas variantes que omitimos aqui por limitações de espaço (ver [Par90]).

Com isto não estamos querendo descartar as provas por indução, mas mostrar que há casos em que as mesmas não são estritamente necessárias. Na próxima seção discutimos a razão da busca de uma metodologia de prova sem o uso de indução e sim com o uso de regras de transformação.

5 Tradução Fonte-a-Fonte e Semântica Formal

O processo da tradução de programas, de forma geral, consiste em converter programas escritos numa linguagem origem em programas, *semanticamente equivalentes*, numa linguagem destino.

$$\mathcal{T}: L_{origem} \longrightarrow L_{destino}$$

tal que

$$\forall\, p_o \in L_{origem},\, p_d \in L_{destino} \cdot \mathcal{T}(p_o) = p_d \Rightarrow \mathit{Semântica}(p_o) \equiv \mathit{Semântica}(p_d)$$

Uma aplicação importante da tradução de programas é a tradução fonte-a-fonte, onde programas são traduzidos geralmente de uma linguagem possivelmente obsoleta para outra linguagem onde a manutenção seja mais fácil e assistida. Um outro caso bem mais comum de tradução de programas é a compilação, onde programas escritos numa linguagem de alto nível são traduzidos para uma linguagem de máquina. Enquanto que a tradução fonte-a-fonte tem como objetivo a legibilidade e manutenção dos programas gerados, a compilação tem como objetivo a eficiência.

Na abordagem tradicional, a construção de um tradutor é feita para cada par de linguagens, o que pode ser muito caro como também tedioso em alguns casos.

Neste trabalho consideramos o problema da tradução fonte-a-fonte entre linguagens de programação visando a geração de tradutores. Nossa abordagem é construir o tradutor de forma (semi-)automática a partir das definições das duas linguagens que tomam parte na tradução, especialmente para classes de linguagens que poderíamos chamar "semelhantes" (do ponto de vista do nível de abstração e do modelo semântico das linguagens). Assim chegamos à formulação já apresentada do problema da tradução.

Em [MCP89] foi apresentada a primeira abordagem ao problema, que usa a SIS como método para descrever as linguagens, ou seja, o meta-compilador deve deduzir as regras de tradução a partir das SIS das duas linguagens. Ali foi apresentado o conceito de *unificação* de SIS. Isto é, a unificação dos objetos semanticamente equivalentes das duas linguagens, ou seja, a unificação das relações que formam parte das SIS. Um algoritmo de unificação é descrito, e foi desenvolvido na base de certas restrições às SIS das linguagens. O algoritmo permite fazer unificação para os casos em que uma tradução "um-a-um" pode ser feita entre as linguagens envolvidas, entendendo por tradução "um-a-um" aquela que faz corresponder a cada construtor da linguagem origem um construtor da linguagem destino. Esta restrição é contornada mediante a definição de regras adicionais na SIS da linguagem destino para definir a semântica de *esqueletos* sintáticos os quais são semanticamente equivalentes a objetos sintáticos da linguagem origem que não podem ser diretamente unificados. Um protótipo de um *Meta-Compilador* foi desenvolvido em Standard ML [Wik87] permitindo a geração semi-automática de regras de compilação.

Duas regras $Regra_o$ e $Regra_d$ unificam quando:

(i) As condições das regras são equivalentes;

(ii) As conclusões das regras são unificáveis;

(iii) Existe uma bijeção f entre as premissas sintáticas das duas regras, tal que, $f(p_o) = p_d \Rightarrow p_o$ e p_d são unificáveis e

(iv) Existe a mesma ligação de variáveis nas duas regras.

A verificação de (i) requer um provador de teoremas. A unificação de conclusões e premissas é similar. Basicamente se unificam os termos que tem estrutura similar.

Posteriormente, em [PM90], foi apresentado um conjunto de meta-regras e as idéias que levaram à sua concepção. A abordagem baseia-se na tentativa de incrementar o poder do algoritmo de unificação anteriormente mencionado, principalmente para os casos em que uma tradução "um-a-um" entre as linguagens não é possível. Nestes casos, propomos que as meta-regras sejam usadas na linguagem destino, de forma de derivar as regras de objetos compostos cuja semântica seja equivalente à de construtores da linguagem origem que não tem um construtor correspondente na linguagem destino.

A busca de uma metodologia de prova para o caso de descrições semânticas operacionais se baseia no fato que podemos ver o processo de unificação como o de provar propriedades (*unificar*) que relacionam fórmulas (*regras semânticas*) de teorias diferentes (as SIS das linguagens origem e destino). De forma natural, esta metodologia se aplica para o caso de buscar correspondentes semânticos a construtores não unificados da linguagem origem, em que se faz necessário a obtenção de regras derivadas para objetos sintáticos compostos na linguagem destino.

Os resultados referentes à relação entre SIS e lógica equacional mostram que as tentativas de solução do problema da tradução visto como um problema de unificação pode ser desenvolvida em qualquer um destes formalismos com uma abordagem uniforme. Ainda mais, não é de descartar que outros formalismos, como por exemplo Semântica Natural [Kah87] (do estilo de SIS), possam ser incluídos dentro desta abordagem.

6 Operadores Modulares de Semântica

É de se esperar que para linguagens com relativa complexidade semântica a dificuldade de unificação esteja muito grande. Por esse motivo procuramos obter operadores que permitam abstrair o comportamento das definições semânticas que frequentemente aparecem. Um trabalho neste estilo já existe para a SD, descrito em [Mos89b], que chega a definir outro formalismo para a especificação semântica de linguagens (*Action Semantics* [Mos89a]) baseado em SD.

A idéia do uso destes operadores é que, na medida que eles consigam abstrair comportamentos padrão, as definições semânticas passam a ser instâncias destes operadores. O mais importante deste fato é que a unificação de regras passa a ser feita no nível de operadores, simplificando então o processo.

Operadores de comportamento estão sendo definidos para a SIS. Como exemplo simples, apresentamos aqui um operador que captura o comportamento de definições semânticas para expressões aritméticas, booleanas, etc. (no caso em que estas não tem efeitos colaterais). Este operador foi chamado de *binário-homogêneo-paralelo* por avaliar paralelamente duas sub-frases do mesmo tipo que a frase composta da conclusão. A sintaxe de declaração de operadores é informal pois ainda está em estudo.

OP Binario_par_hom $[op,\mathcal{R},g](t_1,t_2,a,b,b_1,b_2)$

$$\frac{\langle t_1,a\rangle \xrightarrow{\pi} b_1 \quad \langle t_2,a\rangle \xrightarrow{\pi} b_2}{\langle t_1 \text{ op } t_2,a\rangle \xrightarrow{\pi} b} \text{ WHERE } b = g(b_1,b_2)$$

END Binario_par_hom;

Daí temos que:

Semântica(e_1 + e_2) = Binario_par_hom[+,$\mathcal{E}$,*add*](e_1,e_2,m,v_1,v_2,v)

Semântica(be_1 **or** be_2) = Binario_par_hom[**or**,$\mathcal{B}$,$\vee$](be_1,be_2,m,bv_1,bv_2,bv)

onde

$\mathcal{E}$: Exp → (Memória → Int)

$\mathcal{B}$: Bexp → (Memória → Bool)

$$\frac{\langle e_1,m\rangle \xrightarrow{\mathcal{E}} v_1 \quad \langle e_2,m\rangle \xrightarrow{\mathcal{E}} v_2}{\langle e_1 + e_2,m\rangle \xrightarrow{\mathcal{E}} v} \qquad v = add(v_1,v_2)$$

$$\frac{\langle be_1,m\rangle \xrightarrow{\mathcal{B}} bv_1 \quad \langle be_2,m\rangle \xrightarrow{\mathcal{B}} bv_2}{\langle be_1 \textbf{ or } be_2,m\rangle \xrightarrow{\mathcal{B}} bv} \qquad bv = bv_1 \vee bv_2$$

7 Conclusões

Mostramos como diferentes métodos de especificação semântica, com estilos e fundamentos diferentes podem ser tratados de maneira uniforme, no escopo do problema da tradução fonte-a-fonte. Desta forma conseguimos identificar meta-regras para manipular regras da SIS como regras de sistemas de transformação para lógica equacional.

Vimos como estas regras de transformação podem assistir na prova de propriedades, e como certo tipo de propriedades não precisa de indução para sua prova. Isto tem fundamental importância para a unificação de SIS em que propriedades têm que ser provadas relacionando objetos de teorias diferentes.

Os operadores modulares podem ser uma nova solução para o problema de unificação, na medida que permitem abstrair o comportamento de um número amplo de regras. Desta forma o processo de unificação pode ter como primeira etapa a tentativa de unificação de operadores. Outra proposta em análise é a definição de módulos de descrição semântica que agrupem conjuntos de regras, possivelmente sobre o mesmo domínio sintático. Por exemplo, podemos ver que a semântica de expressões aritméticas é, em certa medida, um padrão em muitas linguagens (com variações quase só na sintaxe).

Muito esforço tem que ser feito ainda para conseguir levar esta linha de trabalho a um estágio que permita o desenvolvimento "real" de tradutores entre linguagens complexas. Abordagens como a aqui tratada permitem um desenvolvimento formal de compiladores com a vantagem neste caso que, verificada a correção do algoritmo de unificação e geração de regras, os compiladores gerados pelo meta-compilador são corretos por construção.

Referências

[Ast89] E. Astesiano. Operational Semantics. In *Lecture Notes of the State of the Art Seminar on Formal Description of Programming Concepts.*, IFIP TC2 WG 2.2, Petrópolis, 1989.

[Bar81] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[BH88] R. Burstall and F. Honsell. *A Natural Deduction Treatment of Operational Semantics.* Technical Report ECS-LFCS-88-69, Department of Computer Science, University of Edinburgh, 1988.

[BL84] R. Burstall and B. Lampson. A Kernel Language for Abstract Data Types and Modules. In *Semantics of Data Types, Lecture Notes in Computer Science, Vol. 173*, Springer-Verlag, 1984.

[Dar82] J. Darlington. Program Transformation. In P. Henderson J. Darlington and D. Turner, editors, *Functional Programming and its Applications*, Cambridge University Press, Cambridge, 1982.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An Initial Approach to the Specification, Correctness and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology*, Prentice Hall, 1978.

[Kah87] G. Kahn. Natural Semantics. In *STACS'87, Lecture Notes in Computer Science, Vol. 247*, Springer-Verlag, 1987.

[MCP89] S. Meira, A. L. Cavalcanti, and A. Pardo. On the Use of Structured Inductive Semantics in Source-to-Source Translation of Programming Languages. 1989. Submitted to Information Processing Letters, North-Holland.

[Mos89a] P. D. Mosses. Action Semantics, Notas de Curso, Aarhus University, Dinamarca. 1989.

[Mos89b] P. D. Mosses. Denotational Semantics. In *Lecture Notes of the State of the Art Seminar on Formal Description of Programming Concepts*, IFIP TC2 WG 2.2, Petrópolis, 1989.

[Par90] A. Pardo. Tese de Mestrado, Departamento de Informática, Universidade Federal de Pernambuco. 1990. (Em preparo).

[Pau87] L. C. Paulson. *Logic and Computation — Interactive Proof with Cambridge LCF*. Volume 2 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1987.

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

[PM90] A. Pardo and S. Meira. Tradução Fonte-a-Fonte de Linguagens de Programação Baseada em Semântica Formal. In *Anais do X Congresso da SBC*, Sociedade Brasileira de Computação, Vitória, 1990.

[Sch86] D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. Allyn and Bacon, Inc., Boston Mass., 1986.

[Wik87] Å. Wikström. *Functional Programming using Standard ML*. Prentice Hall International, UK, 1987.