V - Tradução e Preservação da Manutenibilidade

5.1. Introdução

O objetivo deste capítulo é apresentar argumentos de que a preservação da semântica de árvores significa a preservação da intenção do programador e de que essa é uma condição necessária à preservação da manutenibilidade do programa traduzido. A definição formal de preservação de semântica de árvores é apresentada somente no próximo capítulo; apresentamos os exemplos neste capítulo como uma motivação para o posterior desenvolvimento formal do conceito.

É desnecessário dizer que outros fatores também podem influenciar na manutenibilidade do programa traduzido, tais como a escolha dos identificadores, a sintaxe concreta da linguagem de saída, o ambiente no qual será feita a manutenção do programa, a preservação dos comentários, entre outros.

5.2. Exemplos reais de tradução

No intuito de ilustrar as vantagens de se preservar a semântica de árvores de derivação, apresentamos nesta seção alguns exemplos encontrados em problema reais de tradução entre linguagens de programação. Na verdade, esse conceito de semântica surgiu da observação atenta das dificuldades encontradas no processo de tradução destes exemplos.

Exemplo 5.1. A figura 5.1 apresenta um trecho de programa PASCAL que usa o comando **with** e uma possível tradução desse trecho de programa para C++. Este problema foi encontrado na tradução de CHILL para C++ [Garcia e Guedes, 1999].

A proposta de tradução consiste em substituir cada acesso a variável abreviado pelo comando **with** por sua forma não-abreviada. Essa tradução é extremamente simples e, a menos de efeitos colaterais no cálculo da posição de memória abreviada, preserva a semântica no sentido usual.

Observando o código em C++, constatamos que caso o endereço abreviado seja alterado, essa mudança se refletiria em mudanças em diversos pontos no programa C++. Obviamente, essas mudanças podem tornar-se arbitrariamente difíceis em exemplos apropriados. Lembramos aqui o critério mencionado na seção 1.3: “Um programa é dito manutenível quando se pode alterá-lo com relativa facilidade para adaptá-lo a mudanças” [Veloso, 1987, pag. 25]. O ponto importante é que, de acordo com este critério, houve uma degradação na manutenibilidade do programa.

O mesmo motivo que provoca a perda de manutenibilidade, a repetição de código, faz com que a tradução **não** preserve a semântica de árvores. Observe que no programa original há um morfismo em $L_{\square}G_{PASCAL}$ de assinatura `VAR_ACCESS`$\rightarrow$`CMD` ao passo que não há um morfismo de assinatura correspondente em $L_{\square}G_{C++}$.

PASCAL	C++
...	...
WITH DISC[I].PROF DO	
BEGIN	
HORA := HORA + 4;	DISC[I].PROF.HORA = DISC[I].PROF.HORA + 4;
...	...
AULAGRAD := TRUE;	DISC[I].PROF. AULAGRAD = true;

END	
...	

Figura 5.1 Tradução do comando **with** preservando a extensão.

Exemplo 5.2. Na figura 5.2, voltamos ao exemplo anterior, apresentando uma nova proposta de tradução: gerar um novo identificador, que será um ponteiro ao qual é atribuído o endereço abreviado.

Esta nova tradução elimina o problema da repetição de código, evitando a conseqüente degradação da manutenibilidade e garantindo a preservação da semântica de árvores de derivação. Curiosamente, o código gerado pelo compilador C++ neste caso é análogo ao código gerado pelo compilador PASCAL para o comando **with**.

PASCAL	C++
...	...
WITH DISC[I].PROF DO	{ tprof * p = & DISC[I].PROF
BEGIN	
HORA := HORA + 4;	p->HORA = p->HORA + 4;
...	...
AULAGRAD := TRUE;	p-> AULAGRAD = true;
END	}
...	

Figura 5.2 Tradução do comando **with** preservando a intenção.

Exemplo 5.3. Em [Garcia e Guedes, 1999], os autores apresentam uma proposta de tradução de CHILL para C++ preservando a eficiência e a manutenibilidade do código. É dada especial atenção ao problema da inicialização de variáveis, pois C++ não implementa as mesmas estruturas de inicialização de variáveis que CHILL.

A figura 5.3 apresenta uma tradução que não preserva a semântica de árvores. Observe que a manutenibilidade do código é comprometida, pois muitas vezes a posição do *array* que é inicializada é uma constante com um nome, que possui um significado para o programador. Uma vez que a inicialização é traduzida para C++ esta informação "se perde". Se a constante for alterada, o programador terá que alterar também a inicialização da variável. A classe genérica ARRAY implementa o ajuste dos limites do *array* através de métodos *inline*. O *array* _V_ARRAY pode ser inicializado diretamente na área de dados, deixando para tempo de execução apenas o construtor da classe genérica.

CHILL	C++
DCL	
V_ARRAY ARRAY(H'03:H'0F) T_SET	T_SET _V_ARRAY[1-0x03+0x0F] =
INIT := (:	{ 0, 0, 0, 0, 0, 0, 0, VAL1,
(H'09):VAL1,	VAL2, 0, 0, 0, VAL3, 0 }
(H'0A):VAL2,	
(H'0E):VAL3,	ARRAY <T_SET>
:);	V_ARRAY(0x03,0x0F,_V_ARRAY);

Figura 5.3 Tradução da inicialização de **array** de CHILL.

A figura 5.4 apresenta uma alternativa que preserva a semântica de árvores. Neste caso, a manutenibilidade do programa é preservada. No entanto, a inicialização acontece apenas em tempo de execução, o que compromete a eficiência do programa. No artigo mencionado, os autores propõem uma extensão do pré-processador de C++ para resolver esse problema.

CHILL	C++
DCL	
V_ARRAY ARRAY(H'03:H'0F) T_SET	ARRAY <T_SET>
INIT := (:	V_ARRAY(0x03,0x0F,_V_ARRAY,
(H'09):VAL1,	0x09,VAL1,
(H'0A):VAL2,	0x0A,VAL2,
(H'0E):VAL3,	0x0E,VAL3
:);	);

Figura 5.4 Tradução da inicialização de **array** preservando a semântica de árvores.

Exemplo 5.4. A figura 5.5 apresenta um trecho de programa DDL que utiliza um tratador de exceções a nível de classe e uma possível tradução deste para DDL sem tratadores de exceção a nível de classe [Cortés et al., 1999]. Esse também foi um problema real encontrado no projeto ARTS [Martins et al., 1997], quando traduzindo de DDL para C++. Existem outras linguagens orientadas a objeto (como Eiffel, Guide e Lore) [Beder et. al, 1997] que também possuem tratadores de exceção no nível de classe e que encontrariam o mesmo tipo de problema ao serem traduzidas para uma linguagem sem tratadores no nível de classe, como C++ e Java. A tradução proposta claramente não preserva a semântica de árvores.

DDL	DDL Simplificada
CLASS A;	CLASS A;
...	...
EHANDLER E4;	
BEGIN	
<CMDS1>	
END EHANDLER	
END CLASS	END CLASS
CLASS B INHERITS A;	CLASS B INHERITS A;
...	...
EHANDLER E3;	
BEGIN	
<CMDS2>	
END EHANDLER	
PROCEDURE M1;	PROCEDURE M1;
BEGIN	BEGIN
<CMDS3>	<CMDS3>
EXCEPT	EXCEPT
WHEN E1 THEN <CMDS4>;	WHEN E1 THEN <CMDS4>;
	WHEN E3 THEN <CMDS2>;
	WHEN E4 THEN <CMDS1>;
END EXCEPT	END EXCEPT
END PROCEDURE	END PROCEDURE

END CLASS	END CLASS

Figura 5.5 Tradução do tratador de exceções a nível de classe em DDL.

5.3 Outras aplicações para o conceito de semântica de árvores

Nesta seção ilustramos a utilidade do conceito de semântica de árvores exibindo outras potenciais aplicações do mesmo. Os exemplos apenas apontam caminhos que podem vir a ser explorados, sendo assim, devem ser entendidos como indicações para trabalhos futuros.

5.3.1 Avaliar a expressividade de uma linguagem

Na seção anterior, apresentamos exemplos reais de tradução de determinadas estruturas e, em diversos casos, construímos traduções que preservam a semântica de árvores. No entanto, é fácil ver que isso nem sempre é possível. Por exemplo, se estamos traduzindo de uma linguagem que possui a função `sqr(_)` (quadrado), para outra que não possua tal função e que também não permita ao usuário construir funções, seríamos obrigados a traduzir `sqr(`E_1`)` para E_1`*`E_1.

Essa impossibilidade aumenta a utilidade do conceito de semântica de árvores. Além de o usarmos para qualificar traduções, podemos também usá-lo para dizer se uma linguagem *expressa* ou não uma determinada construção, i.e., se possui uma árvore de derivação com a mesma semântica daquela construção.

Esse problema foi abordado anteriormente, sem sucesso, por Baliga e Schende [Baliga e Schende, 1994], pois segundo sua definição todas as linguagens recursivas expressam todas as estruturas de controle. Halstead [Halstead, 1977], também sem sucesso [Hamer e Frewin, 1982], definiu um conceito semelhante: o *nível* de uma linguagem.

Exemplo 5.5. As estruturas de controle **while** e **for** não aumentam a expressividade da linguagem C, pois podem ser trivialmente traduzidas para os comandos **if** e **goto**, preservando a semântica de árvores, como ilustrado na figura 5.6. Por outro lado, o comando **finally** aumenta a expressividade da linguagem Java, pois a eliminação deste comando implicaria em repetição de código, impossibilitando a preservação da semântica de árvores, como ilustrado na figura 5.7. A tradução ilustrada preserva a semântica no sentido usual caso `Cmd3` e `Cmd4` não levantem exceções.

C	C sem **while** e **for**
`...`	`...`
`while` `(exp) {`	`L2:` **`if`** `(!exp)` **`goto`** `L1;`
`  Cmd;`	`  Cmd;`
`}`	**`goto`** `L2;`
	`L1:`
`...`	`...`

Figura 5.6 Tradução do comando **while** usando **if** e **goto**.

JAVA	JAVA sem **finally**
...	...
Try {	**try** {
Cmd1;	Cmd1;
If (exp1) return;	If (exp1) {
	Cmd4;
	return; }
Cmd2;	Cmd2;
} **catch** (Exception e) {	} **catch** (Exception e) {
Cmd3;	Cmd3;
} **finally** {	}
Cmd4;	Cmd4;
}	
...	...

Figura 5.7 Tradução do comando **finally**.

5.3.2 Comparar programas

O exemplo a seguir é de Dijkstra [Dahl et. al., 1972, pag. 24]. Considere os dois trechos de programa abaixo, onde assumimos que a expressão booleana B2 não é afetada pela execução dos comandos C1 e C2:

if B2 **then**
while B1 **do** C1
else
while B1 **do** C2
	while B1 **do**
	If B2 **then** C1
	else C2

Figura 5.8 Exemplo de comparação entre programas devido a Dijkstra.

Ao exibir o exemplo acima, Dijkstra comenta: “Eu posso estabelecer a equivalência da saída destes dois programas, mas não posso considerá-los equivalentes em nenhum outro sentido útil. Eu sou obrigado a admitir que os dois programas são 'difíceis de comparar'. Inicialmente esta conclusão me aborreceu muito. Posteriormente eu cresci o suficiente para aceitar esta incomparabilidade como coisas da vida e, portanto, como um dos principais motivos pelo qual eu considero a escolha entre estas duas alternativas como uma importante decisão de projeto, que não deve ser tomada sem uma reflexão cuidadosa.”

Evidentemente, ambos os trechos de código possuem a mesma extensão, portanto a mesma semântica usual; mas diferentes intenções, portanto, diferentes semânticas de árvore. Além disso, o critério de um programa mais manutenível ter que se adaptar mais facilmente a mudanças nos dá uma indicação de como comparar os dois programas. Em um cenário onde acreditamos que as duas ocorrências de B1 no primeiro programa possam mudar, porém continuando iguais entre si, o segundo programa será mais manutenível. Entretanto, se acreditamos que seja mais provável que as ocorrências de B1 no primeiro programa mudem sem continuar iguais entre si, neste caso o primeiro programa será mais manutenível.

Esses possíveis cenários dependem do entendimento do problema por parte do programador, e não estão disponíveis para um tradutor automático. Portanto, num processo de tradução automático, o melhor que pode ser feito é preservar a semântica de

árvores do programa original, preservando assim sua manutenibilidade. É claro que, no caso de se possuir alguma informação sobre o domínio de aplicação do programa, pode-se chegar à conclusão de que é necessário melhorar a estrutura do programa; nesse caso estaríamos fazendo reengenharia. Dijkstra também advogava este ponto de vista como fica clara no seguinte comentário [Dahl et. al., 1972, pag. 41]:

"A estrutura do programa deve ser projetada de forma a antecipar suas adaptações e modificações. Nosso programa deve não só refletir (através de sua estrutura) nosso entendimento dele, mas também deve ser claro, a partir de sua estrutura, que tipo de adaptações podem ser feitas suavemente."

5.3.3 Estudar padrões de projeto

A semântica de árvores pode ser aplicada as estruturas sintáticas mais altas de forma razoavelmente independente de linguagem, bastando que se convencionem certas estruturas que devam estar presentes em determinada classe de linguagens, juntamente com seus respectivos significados. Além disso, essa aplicação pode ser feita através da sintaxe abstrata e não da sintaxe concreta. Por exemplo, pode-se convencionar que toda a linguagem orientada a objetos deve possuir classe, herança, membros e métodos com uma determinada sintaxe abstrata e uma determinada semântica.

Isso torna possível mostrar a semântica de certas árvores parciais que surgem em determinados padrões de projeto, o que ajuda a justificá-los e até mesmo a entendê-los melhor.