I - Introdução

1.1. Posicionamento

Esta tese não seria possível sem o modelo bem-sucedido de aproximação entre academia e indústria desenvolvido no Departamento de Informática da PUC-Rio através de seus diversos laboratórios. No Laboratório de Métodos Formais (LMF) desenvolveram-se diversos projetos para a tradução entre linguagens de programação usando-se técnicas de transformação.

Em um primeiro momento, houve uma aplicação de técnicas acadêmicas (transformações) em projetos junto à indústria. Posteriormente, a experiência gerada por estes projetos realimentou a academia com novas idéias para a pesquisa, dando origem a diversas publicações nesta área [Carvalho, 1996] [Carvalho et al., 1997] [Cortés et al., 1999] [Felix e Haeusler, 1999] [Garcia e Haeusler, 1997] [Garcia e Haeusler, 2000] [Garcia et al., 1997] [Garcia et al., 1997a] [Garcia et al., 1997b] [Garcia e Guedes, 1999] [Haeberer et. al., 1993] [Martins et al., 1997].

Nesses projetos adquiriu-se considerável experiência prática na implementação de tradução entre linguagens de programação por meio de técnicas de transformação. Essa experiência inclui o desenvolvimento de técnicas de programação usando a ferramenta de transformação TXL [Cordy e Carmichael, 1993], o desenvolvimento de uma metodologia para projeto e implementação de transformações e, principalmente, o surgimento de uma compreensão intuitiva das dificuldades de geração de um código eficiente e manutenível.

Na seção 2.1 desta tese documentamos as técnicas de programação mencionadas, enquanto o restante do capítulo II traça um histórico dos projetos realizados e de como a

busca de soluções para os problemas que se apresentaram neles levaram a pesquisa no departamento a novas direções, gerando as publicações supracitadas.

Durante o desenvolvimento desses projetos ficou claro que o projeto de traduções entre linguagens de programação exigia um considerável esforço criativo por parte dos projetistas e que não existia nenhum critério formal para ajudá-los a avaliar a qualidade dessas traduções. Os projetistas seguiam a sua experiência e intuição a fim de projetar traduções que preservassem atributos como: confiabilidade, manutenibilidade e eficiência dos programas originais.

A partir desta constatação, iniciamos o presente trabalho visando a executar um programa de dois pontos:

- Criar um modelo para tradução entre linguagens de programação;
- Utilizá-lo para formalizar nossa compreensão intuitiva do que seja uma boa tradução.

Para tanto, partimos de um modelo de R.F.C. Walters para linguagens livres de contexto [Walters, 1989], a fim de propor um modelo de tradução entre elas e, finalmente, propor um critério que tal tradução deva satisfazer para preservar a manutenibilidade do programa traduzido. A fim de se estabelecer esse critério, fez-se necessária a introdução de semântica para os modelos categóricos de linguagens livres de contexto. Mostramos através de exemplos que a semântica dos morfismos (que são modelos de trechos de derivação) está relacionada com a preservação da manutenibilidade dos programas.

Acreditamos que a técnica desenvolvida, além de se aplicar ao problema estudado nesta tese, traz um *insight* importante em relação à semântica de linguagens de programação e à forma pela qual a intenção do programador está relacionada com a estrutura sintática de um programa.

1.2. Tradução entre Linguagens de Programação

Nesta seção apresentamos técnicas conhecidas para a implementação de traduções entre linguagens de programação. Para padronizarmos a terminologia, de agora em diante chamaremos a linguagem de entrada, da qual é feita a tradução, de L_e. A linguagem de saída, para a qual é feita a tradução, chamaremos de L_s. Os requisitos gerais de um tradutor entre linguagens de programação são:

- Produzir um código em L_s que preserve a semântica do código original em L_e.
- O novo programa não deve consumir memória e tempo de processamento desnecessariamente, i.e., deve fazê-lo na mesma medida que o programa original.
- O código em L_s deve preservar a manutenibilidade do código original em L_e. Ou seja, o código produzido deve preservar a legibilidade, a modularidade, etc., dentro dos limites impostos por L_s.

Apresentamos em seguida duas técnicas de tradução automática de programas: compilação e transformação. Os critérios desenvolvidos nesta tese independem da técnica utilizada para se implementar a tradução.

1.2.1 Compilação

O processo usual de compilação é uma tradução entre linguagens de programação. Na verdade, o nome mais apropriado para um compilador seria *tradutor*. A figura 2.1 representa esquematicamente o funcionamento de um compilador.

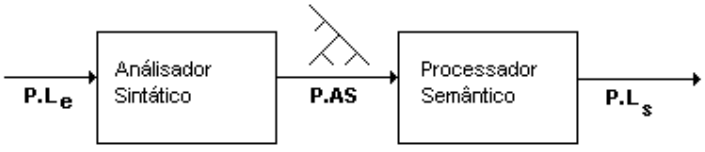

Figura 1.1 Processo de compilação.

- $P.L_e$. Programa escrito na linguagem de entrada.
- P.AS. Árvore de análise sintática do programa.
- $P.L_s$. Programa escrito na linguagem de saída.

A especificação sintática (gramática de L_e) é usada para criar o analisador sintático, enquanto o processador semântico, que implementa o significado de $P.L_e$, é construído manualmente a partir de uma especificação semântica informal de L_s. Hoje em dia, as técnicas de construção de compiladores estão maduras. Uma das linhas de desenvolvimento das pesquisas a partir de compiladores são as ferramentas de transformação.

Quando usamos um compilador para fazer migração de código, estamos diante de uma situação especial. Neste caso, L_s é, geralmente, uma linguagem de alto nível, o que torna mais fácil a implementação do significado das construções de L_e em L_s (ao menos no caso em que L_e e L_s são linguagens semelhantes). Além disso, o programa original é um programa em funcionamento, o que nos permite abandonar as verificações estáticas como checagem de escopo e tipo. Com base nessas premissas, o processador semântico é reduzido a um manipulador de árvores, i.e., um programa que troca padrões em uma árvore sintática. Veremos que as ferramentas de transformação se adequam perfeitamente à execução deste tipo de manipulação.

1.2.2 Transformação

As ferramentas de transformação são basicamente ferramentas para a manipulação de árvores de derivação. Sua aplicação à tradução entre linguagens de programação permite uma aproximação do código da tradução da forma com o qual a tradução é geralmente concebida, a saber: através de exemplos de tradução para cada estrutura da linguagem de entrada. Nesse sentido, citamos [Félix e Haeusler, 1999] como um avanço ainda maior nessa direção.

A figura 1.2 ilustra o uso de transformações para traduzir programas entre L_e e L_s. Neste caso a “transformação” executa trocas de determinados padrões na árvore sintática por outros padrões, de acordo com o especificado nas chamadas “regras de transformação”. A seção 2.2 aborda com mais atenção e detalhe a ferramenta de transformação TXL.

Figura 1.2 Tradução usando transformações.

- $P.L_e$. Programa escrito em L_e.
- P.AS. Árvore sintática do programa $P.L_e$.
- P.AS'. Árvore sintática do programa em L_s.
- $P.L_s$. Programa escrito na linguagem de saída.

1.2.3 Ferramentas de transformação

Na última década, foram implementadas diversas ferramentas para a construção de compiladores que oferecem recursos para manipulação de árvores, em especial, para casamento e substituição de padrões em árvores. Essas ferramentas de transformação sugerem um método de programação, o paradigma transformacional. Entre as principais ferramentas encontram-se:

- **Refine**. Ferramenta comercial produzida pela Reasoning Systems [REFINE, 1992].
- **Kimwitu**. Ferramenta para construção de programas que usam árvores e termos como as principais estruturas de dados.
- **Optran**. Um gerador de compiladores com um poderoso suporte para gramática de atributos, casamento de padrões e transformações.

- **Rigal**. Uma linguagem para construção de compiladores. Suas principais estruturas de dados são átomos, listas e árvores.
- **Memphis**. A ferramenta Memphis Tree Builder & Tree Walker permite a definição e o processamento de árvores sintáticas abstratas. É implementada através de um pré-processador que estende C/C++. Permite a declaração de tipos de dados no estilo gramatical e oferece o comando *match* que os processa usando casamento de padrões em árvores.
- **Popart**. Um sistema com uma linguagem para definição de *parsers* e regras de reescrita.
- **Smart**. Uma extensão do ANSI C para algoritmos em grafos e árvores.
- **TM**. Um pré-processador de *templates* que gera código para uma linguagem especificada.
- **Tampr**. Um sistema escrito em Lisp. A maior parte das aplicações publicadas aborda a migração de Lisp para Fortran [Boyle, 1989].
- **Draco-PUC**. Uma ferramenta poderosa desenvolvida no departamento de informática da PUC do Rio de Janeiro. Seu objetivo é implementar o paradigma Draco para construção de programas [Leite et al., 1994].
- **Txl**. Esta ferramenta foi escolhida para a implementação das transformações encontradas nesta tese devido à sua linguagem elegante e documentação confiável.

1.2.4 Aplicações

Ferramentas de transformações de programas são adequadas para se resolver qualquer problema que possa ser facilmente modelado através de alterações estruturais sobre uma *string* (programa) em uma linguagem cuja sintaxe é formalizável por uma gramática livre de contexto. Inúmeros problemas na área de informática e, em particular,

de linguagens de programação, se enquadram nesta descrição. Entre as áreas às quais as ferramentas de transformação têm sido aplicadas encontram-se:

- Tradutores.
- Interpretadores.
- Prototipação rápida de características novas e dirigidas a domínio e dialetos em linguagens já existentes [Cordy et. al., 1991].
- Análise de código-fonte e recuperação de projetos.
- Reestruturação de programas e remodularização.
- Metaprogramação e reuso de *software* [Cordy e Shukla, 1992].
- Otimização e paralelização de código.
- Simplificação de fórmulas lógicas.
- Transformação de esquemas de um modelo de dados em esquemas equivalentes em outro modelo [Abu-Hamdeh et. al., 1994].
- Comparação estrutural de programas.

1.3. Contribuições

Iniciamos o estudo teórico sobre tradução procurando um modelo categórico de tradução, o qual nos parecia mais fácil de manipular do que as definições antigas usuais [Aho e Ullman, 1972] e [Aho e Ullman, 1972b]. Uma vez atingido esse objetivo, procuramos responder o que seria uma *boa* tradução. Uma possível abordagem seria dizer que uma *boa* tradução é aquela que leva bons programas em bons programas.

Ghezzi cita como atributos de um “bom programa” corretude, confiabilidade, robustez, performance, amigabilidade, verificabilidade, manutenibilidade, reusabilidade, portabilidade, entendibilidade e interoperabilidade, entre outros [Ghezzi et. al., 1991].

Veloso é mais econômico e cita qualidades um pouco mais ortogonais: confiabilidade, manutenibilidade e eficiência [Veloso, 1987].

Entre os atributos acima, concentramo-nos na manutenibilidade, pois, de acordo com a nossa experiência em projeto de traduções, parece-nos especialmente difícil de se preservar. Veloso também apresenta uma definição de manutenibilidade que nos será útil: “Um programa é dito manutenível quando se pode alterá-lo com relativa facilidade para adaptá-lo a mudanças” [Veloso, 1987, pág. 25].

Na presente tese, apresentamos um critério formal, que sustentamos ser necessário para a preservação da manutenibilidade (atributo informal). Desenvolvemos nossos argumentos através de exemplos, nos quais mostramos que dados programas que “podem ser alterados com relativa facilidade” para atender a determinadas mudanças, traduções que atendem ao critério proposto produzem programas que ainda “podem ser alterados com relativa facilidade”, enquanto que traduções que não atendem ao critério produzem programas que não satisfazem esta condição.

Acreditamos que a técnica desenvolvida, juntamente com os exemplos apresentados, trazem um *insight* importante em relação à semântica de linguagens de programação e à forma com a qual a intenção do programador está relacionada com a estrutura sintática de um programa.

1.4. Apresentação

Nesta tese, apresentamos sumariamente os conceitos usuais de gramáticas e linguagens, pois, apesar de serem amplamente difundidos, são absolutamente necessários para a compreensão dos modelos; a terminologia é usada extensivamente durante todo o texto. Infelizmente, devido à extensão e à profundidade do assunto, não foi possível fazer o mesmo com os conceitos de teoria das categorias utilizados. Admitimos que o leitor esteja

familiarizado com os conceitos de categoria, funtor, transformação natural e adjunção. Como uma introdução à teoria das categorias suficiente para a compreensão desta tese, sugerimos [Walters, 1991], que apresenta as idéias apresentadas no capítulo III desta tese. A referência clássica é [Mac Lane, 1971]. [Arbib e Manes, 1974] também é uma ótima introdução. Para a leitura dos exemplos de tradução, principalmente no capítulo 7, é necessário um conhecimento básico de Pascal, C e Fortran. O restante desta tese está organizado da seguinte forma:

- O capítulo II apresenta um histórico dos trabalhos desenvolvidos no LMF que motivaram o desenvolvimento desta tese.
- O capítulo III apresenta um modelo categórico de gramáticas e linguagens livre de contexto. Este modelo é a base do restante do trabalho.
- O capítulo IV insere o conceito de semântica de linguagens de programação no contexto do modelo categórico apresentado, definindo *semântica de árvores*. Este conceito estende o conceito usual de semântica para trechos de derivação, o que será usado posteriormente para se fazer conexão com a idéia de *intenção* do programador.
- Em seguida, no capítulo V, apresentamos argumentos de que a preservação da semântica de árvores por uma tradução é condição necessária para a preservação da manutenibilidade dos programas.
- O capítulo VI define formalmente tradução entre linguagens e preservação de semântica de árvores, conceitos estes já utilizados informalmente no capítulo anterior.
- O capítulo VII faz uma breve análise de alguns tradutores existentes no mercado sob a ótica das idéias desenvolvidas nos capítulos anteriores.
- Finalmente, o capítulo VIII apresenta conclusões e sugestões para trabalhos futuros.