II - Histórico

2.1. Introdução.

Como já foi dito, a motivação da presente tese surgiu dos projetos de tradução entre linguagens de programação usando transformações realizados no laboratório de métodos formais – LMF – do Departamento de Informática da PUC-Rio. Alguns desses trabalhos foram de grande importância para o desenvolvimento desta tese, pois nos permitiram amadurecer a compreensão do problema, identificar dificuldades do uso de transformações e procurar caminhos para a solução destas.

Esses trabalhos incluem o tradutor entre DDL e C++, desenvolvido para o projeto ARTS [Carvalho et al., 1997] [Martins et al., 1997], um sistema de tradução baseado em semântica, parte do projeto "Alquimia" [Haeberer et. al., 1993] [Garcia et al., 1997] e um projeto de tradutor de CHILL para C++ [Garcia e Guedes, 1999].

Iniciamos este capítulo apresentando a ferramenta de transformação TXL. Posteriormente apresentamos resumidamente os trabalhos desenvolvidos, destacando os problemas encontrados e como o entendimento e a busca de uma solução para os mesmos nos levou ao trabalho desenvolvido nesta tese. O restante deste capítulo está organizado da seguinte forma:

- A seção 2.2 apresenta a ferramenta de tradução TXL.
- A seção 2.3 mostra, em linhas gerais, a tradução de DDL para C++.
- A seção 2.4 apresenta uma visão geral do sistema de tradução baseado em semântica.
- A seção 2.5 expõe alguns pontos do projeto de tradução entre CHILL e C++, nos quais pode-se identificar o surgimento de idéias formalizadas posteriormente neste texto.

2.2. A ferramenta TXL.

A ferramenta TXL foi escolhida devido à sua linguagem elegante e documentação confiável [Cordy e Carmichael, 1993]. Esses aspectos foram considerados importantes após uma experiência problemática com a ferramenta TAMPR [Boyle, 1989]. Os trabalhos com a ferramenta TXL no LMF iniciaram-se em julho de 1995, com um grupo formado pelo Professor Edward Hermann Häusler, Luiz Carlos Guedes, Alex Garcia e Fernando Fonseca. Já naquele momento, pretendia-se utilizar a ferramenta no projeto “Alquimia” e na implementação de um tradutor de DDL para C++ no projeto “ARTS”.

Nessa fase inicial percebeu-se a grande flexibilidade da ferramenta de tradução TXL e a necessidade de se desenvolverem técnicas de programação em TXL para aumentar a produtividade da codificação de uma tradução (especialmente quando o desenvolvimento é feito em equipe), bem como bibliotecas de suporte ao tratamento do ambiente (tabela de símbolos).

Nesta seção apresentamos a ferramenta de tradução TXL e, em seguida, apresentamos as técnicas desenvolvidas.

2.2.1 O Funcionamento de TXL.

A execução de um programa TXL tem três fases, a saber:

- A fase de análise léxica e sintática, na qual é feita a análise léxica e sintática da entrada, produzindo uma árvore sintática. O TXL faz análise de qualquer linguagem livre de contexto. Como se sabe da teoria de compiladores, não pode existir analisador sintático eficiente (com complexidade linear) para todas as linguagens livres de contexto. A análise sintática do TXL é eficiente desde que a gramática analisada seja LL(1).

- A fase de transformação. Nessa fase, o TXL atua na árvore sintática, transformando-a de acordo com determinadas regras de transformação.
- A fase de desmontagem. Essa fase simplesmente desmonta a árvore sintática, percorrendo-a e gerando a saída formatada de acordo com símbolos especiais que podem ser colocados na gramática.

Um programa em TXL deve prover uma especificação sintática da linguagem de entrada e, também, a especificação do conjunto de transformações a ser executado, de modo que a análise sintática da entrada, bem como as transformações sobre a árvore sintática resultante, possam ser feitas. A seguir descreveremos estas duas partes da linguagem TXL.

A especificação da gramática.

A gramática da linguagem de entrada é especificada em uma notação proprietária do TXL. A figura 2.1 mostra o código TXL para uma gramática de expressões juntamente com as mesmas regras escritas em BNF. O símbolo não-terminal "**program**" representa o símbolo inicial, enquanto "**number**" é um símbolo terminal especial que representa constantes numéricas.

TXL	BNF

`define program`
`  [E]`
`end define`
`define E`
`  [T] + [E]  |`
`  [T]`
`end define`
`define T`
`  [F] * [T]  |`
`  [F]`
`end define`
`define F`
`  [number]  |`
`  ( [E] )`
`end define`
	`program ::= E`
	
	`E ::= T + E | T`
	
	`T ::= F * T  |  F`
	
	`F ::= number  |  ( E )`

Figura 2.1 Sintaxe de TXL – I.

As transformações.

Vamos apresentar a sintaxe de especificação das transformações em TXL através de um exemplo. O exemplo escolhido é implementar uma “calculadora” que avalie as expressões geradas pela gramática na figura 2.1. As regras de transformação são especificadas de forma intuitiva na figura 2.2.

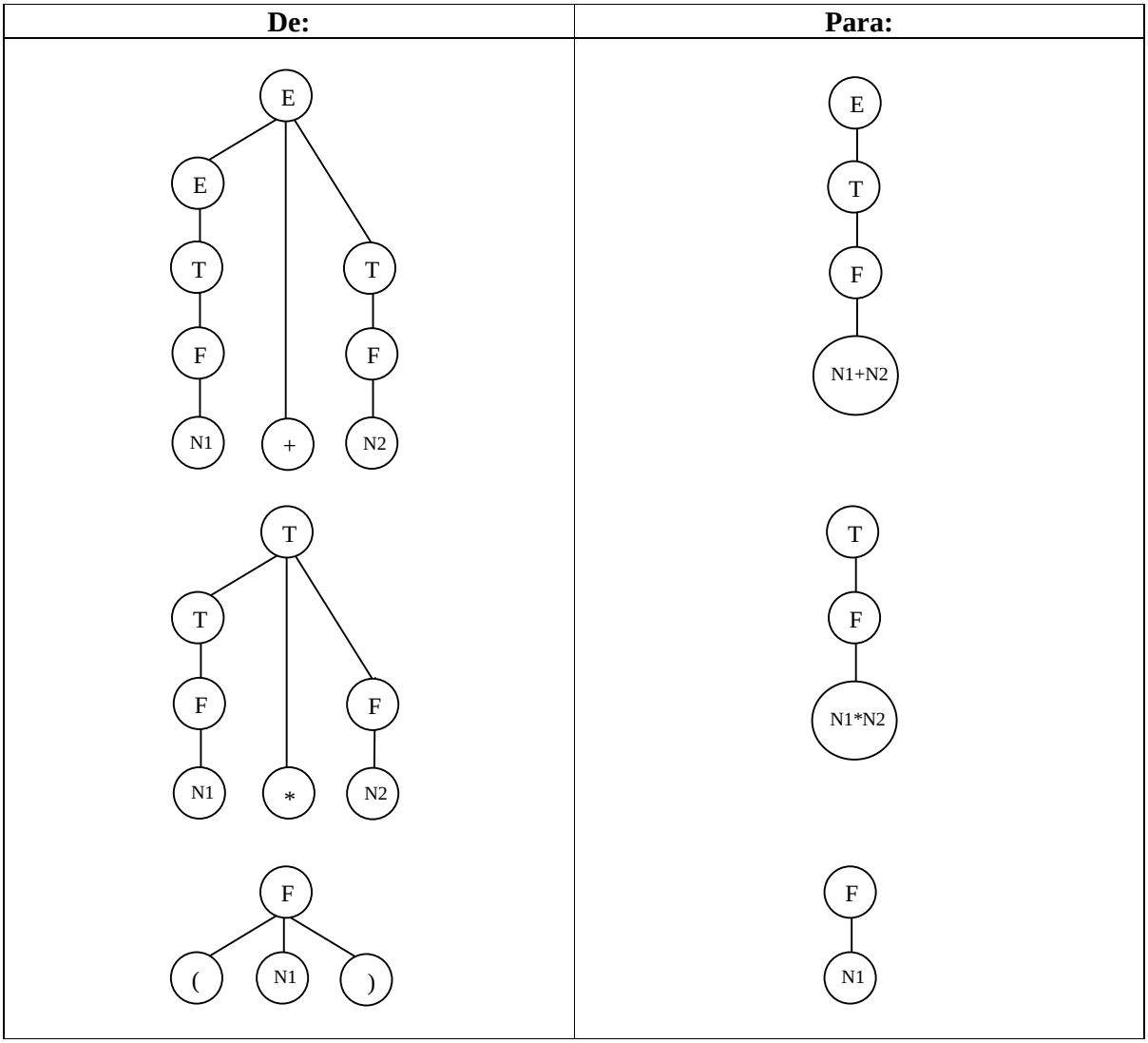

Figura 2.2 Especificação informal das transformações da “calculadora”.

Introduziremos agora a sintaxe de especificação das regras de transformação em TXL e codificaremos as regras acima nessa linguagem. Em TXL, a regra de transformação mais simples tem a forma:

```
rule nomedaregra
    replace tipo
        pattern
    by
        replacement
end rule
```

Figura 2.3 Sintaxe de uma regra em TXL.

A regra acima primeiramente tenta casar o *pattern* com alguma parte da árvore que é o escopo da regra. Em caso de sucesso, ela substitui o *pattern* pelo *replacement*. Essa operação é repetida até que não se possa mais casar o *pattern* no escopo da regra. Observe na primeira regra da figura 2.4 que na sintaxe do *pattern* existe a especificação de um não-terminal após a palavra reservada **replace**. Isso significa que o padrão procurado deve ser uma **árvore** de raiz "E" e folhas 'N1', '+' e 'N2' (as folhas também podem ser não terminais, isto é, as árvores podem representar uma derivação incompleta). O *replacement*, por sua vez, é uma árvore com a mesma raiz, "E", e com folha 'N3' (a soma de 'N1' e 'N2'), porque segundo a semântica do TXL o *replacement* representa uma árvore com a mesma raiz que o *pattern*. Caso o *pattern* ou o *replacement* não possam ser analisados sintaticamente para construir uma árvore com a raiz especificada o TXL acusará um "erro de sintaxe". A figura 2.4 mostra a codificação em TXL das transformações definidas informalmente na figura 2.2.

```
rule rAdd
    replace [E]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule

rule rMul
```

```
    replace [T]
        N1 [number] * N2 [number]
    by
        N1 [* N2]
end rule

rule rPar
    replace [F]
        ( N [number] )
    by
        N
end rule
```

Figura 2.4 Código TXL das transformações da “calculadora”.

Ainda em relação a figura 2.4, observe que os colchetes possuem uma semântica sobrecarregada. No *pattern* eles indicam o tipo de um símbolo, enquanto que no *replacement* representam a aplicação de uma função ou regra de transformação – nesse caso, a função soma – à árvore que os precedem. Essa árvore será o escopo da função aplicada.

O ponto de partida da execução de um programa TXL é uma regra especial de nome *mainRule*, que é disparada quando a execução se inicia e cujo escopo é a árvore de derivação de todo o programa de entrada. No caso da calculadora, essa regra (indicada na figura 2.5) invoca as demais regras até que não se possam aplicar mais transformações.

```
rule mainRule
    replace [E]
        E1 [E]
    construct E2 [E]
        E1 [rAdd] [rMul] [rPar]
    where not
        E2 [= E1]
    By
        E2
end rule
```

Figura 2.5 Regra *mainRule* para a calculadora.

2.2.2 Técnicas de programação de tradução usando-se TXL.

A ferramenta TXL é bastante versátil e tem sido usada para a implementação de variadas tarefas. Estamos particularmente interessados na utilização de TXL na implementação entre linguagens de programação. Nesse caso, temos que nos adaptar ao funcionamento da ferramenta que permite a especificação de apenas uma gramática, sobre a qual são executadas as transformações. Mostraremos a seguir três formas diferentes de se implementar a tradução de expressões infixas para expressões pósfixas.

Junção de gramáticas.

Esta é a forma mais direta de se traduzir de uma linguagem de programação para outra com TXL. No entanto, para se usar esse método, as gramáticas das linguagens de entrada e saída precisam ser similares. O método é ilustrado no exemplo a seguir. A gramática é exibida na figura 2.6, e as regras de transformação, na figura 2.7.

```
define program
      [E]
end define

define E
      [T] + [E] |
      [T] |
      [T] [E] +
end define

define T
      [F] * [T] |
      [F] |
      [F] [T] *
end define

define F
      [id] |
      ( [E] )
end define
```

Figura 2.6 Gramática para a tradução de expressões pelo método junção de gramáticas.

Observe que a gramática não possui a regra F→ E, que corresponderia à retirada dos parênteses. Tal regra tornaria a gramática recursiva à esquerda (e ambígua), o que tornaria o processo de transformação não-eficiente. Nesse caso, a retirada dos parênteses é deixada para um pós-processamento.

```
rule rAdd %troca soma por sua versão pósfixa
    replace [E]
        T1 [T] + E1 [E]
    by
        T1 E1 +
end rule

rule rMul %troca produto por versão pósfixa
    replace [T]
        F1 [F] * T1 [T]
    by
        F1 T1 *
end rule

rule mainRule %faz a iteracao das regras
    replace [E]
        E1 [E]
    construct E2 [E]
        E1 [rAdd] [rMul]
    where not
        E2 [= E1]
    by
        E2
end rule
```

Figura 2.7 Regras para a tradução de expressões pelo método junção de gramáticas.

Gramática achatada.

Outra forma de se traduzir entre duas linguagens usando TXL é desconsiderar a gramática da segunda linguagem (admitimos que as regras de transformação geram código sintaticamente correto), e ver um programa na segunda linguagem simplesmente como uma lista de *tokens*. Nesse caso, a gramática da linguagem de entrada é combinada com a gramática achatada da segunda linguagem, como mostra o exemplo da figura 2.8.

```
define program
    [E]  |
    : [repeat any_token]
end define

define E
    [T] + [E]  |
    [T]
end define

define T
    [F] * [T]  |
    [F]
end define

define F
    [id]  |
    ( [E] )
end define

define any_token
    +  |  *  |  id  |  (  | )
end define
```

Figura 2.8 Gramática para a tradução de expressões pelo método da gramática achatada.

Os dois pontos são usados para evitar que a gramática seja ambígua, eles precisam ser eliminados em um segundo passo. Nesse caso, são usadas funções no lugar de regras. A diferença é que, no caso de funções, exige-se que a raiz do padrão casado no *replacement* seja a raiz do escopo de aplicação da função (i.e., o padrão não é procurado nas sub-árvores) e a função executa apenas uma única substituição. A figura 2.9 exibe essas transformações. Nessa técnica, as funções são escritas de forma a percorrerem a árvore sintática enquanto geram o código de saída. A função “p” é uma função especial que coloca uma árvore no final de seu escopo, a qual deve ser uma árvore de raiz [repeat qualquer_token].

```
function mainRule
    replace [program]
        E1    [E]
    construct AUX [repeat qualquer_token]
    construct R1  [repeat qualquer_token]
        AUX [rE1 E1] [rE2 E1]
    by
        : R1
end function
```

```
function rE1 E0 [E]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct E0
        T1 [T] + E1 [E]
    by
        INI [rT1 T1] [rT2 T1] [rE1 E1] [rE2
E1] [p +]
end function

function rE2 E1 [E]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct E1
        T1 [T]
    by
        INI [rT1 T1] [rT2 T1]
end function

function rT1 T0 [T]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct T0
        F1 [F] * T1 [T]
    by
        INI [rF1 F1] [rF2 F1] [rT1 T1] [rT2
T1] [p *]
end function

function rT2 T1 [T]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct T1
        F1 [F]
    by
        INI [rF1 F1] [rF2 F1]
end function

function rF1 F1 [F]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct F1
        I1 [id]
    by
        INI [p I1]
end function

function rF2 F1 [F]
    replace [repeat qualquer_token]
        INI   [repeat qualquer_token]
    deconstruct F1
        ( E1 [E] )
    by
        INI [rE1 E1] [rE2 E1]
end function
```

Figura 2.9 Regras para a tradução de expressões pelo método da gramática achatada.

Gramática única.

Usando TXL, é possível traduzir entre duas linguagens apenas percorrendo a árvore sintática e executar a transformação identidade, enquanto o código de saída é gerado como um efeito colateral das regras de transformação. Nesse método, não existe qualquer consideração em relação à gramática da linguagem de saída; a gramática do programa fica reduzida à gramática da linguagem de entrada, como mostra a figura 2.10.

```
define program
    [E]
end define

define E
    [T] + [E] |
    [T]
end define

define T
    [F] * [T] |
    [F]
end define

define F
    [id] |
    ( [E] )
end define
```

Figura 2.10 Gramática para a tradução de expressões pelo método da gramática única.

Neste método, é preciso escrever o código de saída em um arquivo; caso contrário, ele seria perdido. A figura 2.11 exibe as transformações.

```
external function msg      M       [any]
external function abreArq  nomeArq [any]
external function fechaArq

function rE1
  replace [E]
    T1 [T] + E1 [E]
  by
    T1 [rT1] [rT2] + E1 [rE1] [rE2] [msg '+]
end function

function rE2
  replace [E]
```

```
    T1 [T]
  by
    T1 [rT1] [rT2]
end function

function rT1
  replace [T]
    F1 [F] * T1 [T]
  by
    F1 [rF1] [rF2] * T1 [rT1] [rT2] [msg '*]
end function

function rT2
  replace [T]
    F1 [F]
  by
    F1 [rF1] [rF2]
end function

function rF1
  replace [F]
    I1 [id]
  by
    I1 [msg I1]
end function

function rF2
  replace [F]
    ( E1 [E] )
  by
    ( E1 [rE1] [rE2] )
end function

function mainRule
  replace [program]
    E1    [E]
  by
    E1 [abreArq 'saida][rE1][rE2][fechaArq]
end function
```

Figura 2.11 Regras para a tradução de expressões pelo método da gramática única.

2.3. A Tradução de DDL para C++.

A tradução de DDL para C++ foi implementada no contexto do projeto ARTS (*Formal Approach to Real-Time Software*) [Carvalho et al., 1997]. Trata-se de um projeto de três anos com o objetivo de criar um ambiente para desenvolvimento de sistemas de tempo real orientados a objeto. A arquitetura básica do ARTS consiste de dois planos:

- No plano do usuário, o projeto do *software* é representado através de diagramas de relacionamento entre classes, de comunicação entre objetos, de estados e de cenários. Além dos diagramas, também é possível escrever o código dos métodos em DDL (*Design Description Language*). Ao final da modelagem, toda a informação contida nos desenhos é representada em código DDL, que pode ser traduzido automaticamente para C++.
- O plano formal tem como objetivo fornecer ferramentas para permitir testes ainda na fase de modelagem, garantindo a consistência do projeto. Por exemplo, os diagramas de cenário geram estados que são submetidos a um *model checker* para a verificação de certas propriedades.

A ferramenta TXL foi usada em diversos pontos do projeto, como por exemplo: para traduzir os diagramas de cenário em estados para o *model checker*, colocar a saída deste em um formato adequado para a visualização, "importar" bibliotecas C++, disponibilizando-as na ferramenta de desenho, entre outros. Uma vez que nesta tese estamos interessados na tradução entre linguagens de programação, examinaremos aqui a implementação do tradutor de DDL para C++.

Acreditamos que, durante a implementação do tradutor, conseguiu-se exercitar as funcionalidades da ferramenta TXL ao extremo, e comprovar a robustez de sua implementação. Verificamos que as transformações puderam realmente ser implementadas de forma modular. Desde o primeiro momento foi possível dividir o trabalho entre diferentes implementadores e integrá-los posteriormente acrescentando apenas uma chamada de regra de transformação.

2.3.1 Visão geral

DDL é uma linguagem orientada a objetos pura, de segunda geração. Um programa DDL (obtido a partir dos diagramas de especificação e do código dos métodos) contém

código para todas as classes definidas no projeto. A especificação da linguagem encontra-se em [Carvalho, 1996].

Em DDL, não é necessário que a declaração de uma classe preceda a criação de uma instância dessa classe. O mesmo não ocorre em C++: para se declarar uma instância da classe X, por exemplo, a declaração ou a definição da classe X – ou, ao menos, o seu protótipo – deve estar em algum ponto anterior do programa fonte. Do mesmo modo, a definição de um procedimento em DDL não precisa preceder a sua primeira chamada, enquanto que, em C++, pelo menos um protótipo (ou seja, uma declaração) deve aparecer no programa antes que um método ou função possa ser chamado.

Em DDL, não existem declarações de classe nem de procedimentos (i.e., protótipos). Toda informação relativa a uma classe consta de sua definição, no bloco de código entre as palavras reservadas CLASS e END CLASS, assim como toda informação relativa a um procedimento está contida na sua definição, entre as palavras reservadas PROCEDURE e END PROCEDURE.

Desta forma, para que um programa em DDL fosse traduzido para um programa em C++, a tradução foi implementada em vários passos, do seguinte modo:

- No **passo 0**, são geradas declarações de todas as classes.
- No **passo 1**, são geradas as definições de todas as classes, contendo as declarações dos atributos, as declarações do construtor, do destrutor e dos métodos de cada classe.
- No **passo 2**, são geradas as definições do construtor, do destrutor e dos métodos de cada classe.

A saída do passo 1 é tratada pelos programas **ordena** e **tabstate** que são responsáveis respectivamente por ordenar as definições das classes (levando em consideração a hierarquia de herança) e por gerar as definições das classes que

implementam estados. Finalmente, a saída é formatada para a obtenção de um programa legível.

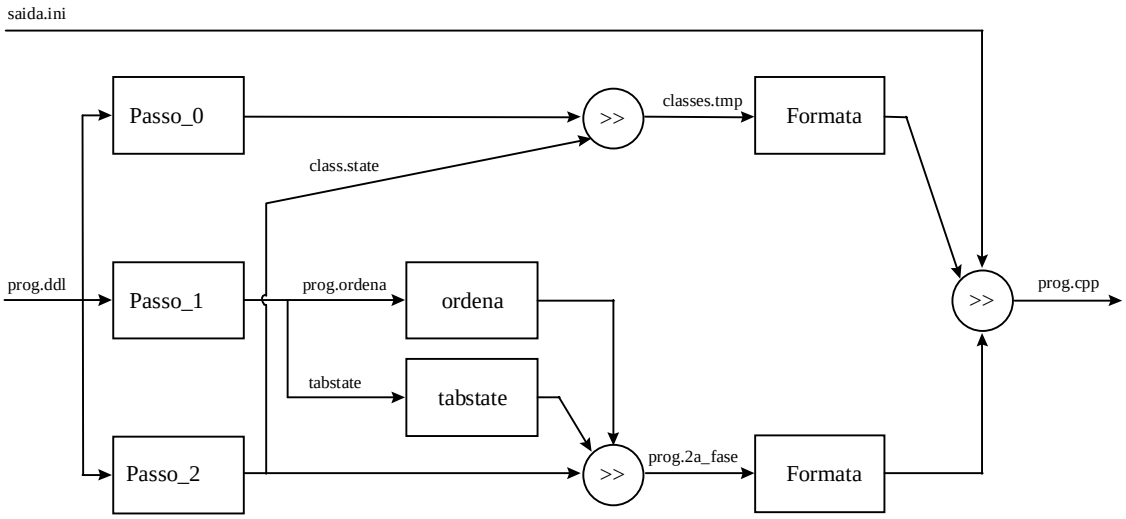

Figura 2.12 Visão Geral da Tradução de DDL para C++.

2.3.2 Dificuldades identificadas no projeto

DDL oferece algumas estruturas sofisticadas que não são disponíveis em C++ ou que têm uma semântica diferente nessa linguagem. Essas estruturas incluem: procedimentos, iteradores, corrotinas, tratadores de mensagens, tratadores de exceções no nível de classe, estados, classes genéricas, entre outras. Essas estruturas mereceram um cuidado especial na tradução, conforme documentado em [Martins et al., 1997]. No entanto, estamos mais interessados nas dificuldades inerentes da forma de utilização da tecnologia, entre elas podemos citar:

- **Nível de Abstração.** Mesmo utilizando técnicas modernas, como ferramentas de transformação, no lugar dos antigos compiladores, a codificação do tradutor entre linguagens precisa tratar diversos problemas de controle de fluxo e manutenção de tabelas de símbolos.

- **Semântica *ad-hoc*.** Para cada construção da linguagem de entrada, sua semântica precisa ser implementada na linguagem de saída, baseada nos conhecimentos de quem projeta a tradução. Isso torna difícil a construção de uma tradução correta, principalmente em função das interações entre a tradução de construções diferentes.
- **Ausência de critérios formais para se avaliar a tradução.** Assim como há registros de sucesso alcançado com migração automática de código, há também casos de fracasso, nos quais o sistema final não atende aos requisitos de performance e o código gerado não é legível. Não existe um critério formal para avaliar a qualidade do código gerado. Por exemplo: uma determinada tradução prejudica a manutenibilidade do código traduzido? Uma resposta formal a esta pergunta poderia alertar o projetista da tradução em questão de algumas dificuldades, oferecendo algum suporte para o projetista não precisar apoiar seu julgamento apenas em sua experiência.

Essas dificuldades motivaram o desenvolvimento de diversos trabalhos de pesquisa no Departamento de Informática da PUC-Rio que procuram solucioná-las. O desenvolvimento de técnicas de programação em TXL para a tradução entre programas, documentadas na seção 2.2.2 – e também em [Garcia et al., 1997a] – bem como o desenvolvimento de um *front-end* para a especificação de transformações [Felix e Haeusler, 1999] foram motivados pela primeira dificuldade. A implementação de um sistema de tradutor baseado em semântica [Garcia et al., 1997a] procurava soluções para as duas primeiras dificuldades enumeradas acima, permitir ao usuário especificar as transformações em um nível mais abstrato e, além disso, garantir a corretude das transformações. Finalmente, este trabalho, um estudo teórico sobre tradução entre linguagens de programação, foi motivado pela terceira dificuldade supracitada. O início

desta linha de trabalho foi a publicação “Towards a Categorical Model for Language Translation” [Garcia e Häusler, 1997].

2.4. Sistema de Tradução Baseado em Semântica.

Esse trabalho foi a continuação do projeto “Alquimia” [Haeberer et al., 1993]. O projeto se propõe a extrair uma especificação funcional do código na linguagem de entrada, por meio da semântica denotacional dessa linguagem e traduzi-la posteriormente para C++. Ambas as tarefas são implementadas usando-se ferramentas de transformação.

Foi implementado um protótipo da ferramenta que deu origem a diversas publicações [Garcia et. al., 1997][Garcia et. al., 1997a][Garcia et. al., 1997b]. Apresentamos aqui apenas uma visão geral do sistema; uma explicação mais detalhada pode ser encontrada nas referências acima.

2.4.1 Visão Geral

A idéia central do sistema consiste em usar a especificação denotacional da linguagem de entrada para extrair uma especificação funcional do programa de entrada, que, por sua vez, é traduzida para C++. A figura 2.13 mostra como se dá esse processo.

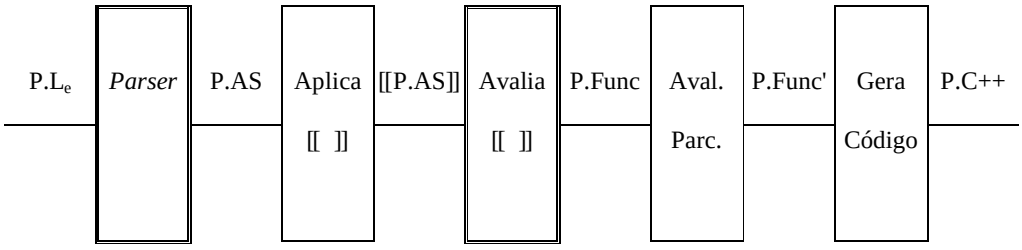

Figura 2.13 Visão da transformação do programa em L_e em programa equivalente em C++.

Na figura acima temos:

- P.L_e. Programa de entrada escrito na L_e.
- *Parser*. Responsável pela análise léxica e sintática do programa de entrada.

- P.AS. Árvore sintática produzida pelo *parser*.
- Aplica ⟦ ⟧. Aplica formalmente a função denotação.
- Avalia ⟦ ⟧. Transformações que avaliam a função denotação.
- P.Func. Especificação funcional do programa de entrada.
- Avaliação Parcial. Aplica reduções à especificação funcional. Expressões de tempo de compilação podem ser resolvidas nessa fase.
- P.Func'. Especificação funcional após aplicação da avaliação parcial.
- Gera Código. Gera código C++ a partir da especificação funcional.
- P.C++. Programa escrito em C++.

Cada passo descrito acima manipula código, portanto são adequados para uma implementação com ferramenta de transformação. Os passos simbolizados por um retângulo de borda simples são independentes da linguagem de entrada. O *parser* depende da especificação sintática de L_e e é gerado automaticamente a partir desta. O módulo avalia ⟦ ⟧ depende da especificação semântica de L_e e também é gerado automaticamente. Chamamos as transformações que geram esses passos automaticamente de metatransformações. As metatransformações sintáticas não despertam grande interesse. As metatransformações semânticas transformam uma regra semântica em uma redução na linguagem funcional, implementada em TXL. Por exemplo, a regra denotacional:

$$⟦ \text{CMD; CMDS} ⟧ (\gamma) = ⟦ \text{CMD} ⟧ \circ ⟦ \text{CMDS} ⟧ (\gamma)$$

é traduzida para a redução:

$$⟦ \text{CMD; CMDS} ⟧ (\gamma) \rhd ⟦ \text{CMD} ⟧ \circ ⟦ \text{CMDS} ⟧ (\gamma)$$

A figura 2.14 mostra um diagrama completo do sistema, explicitando a geração automática dos módulos dependentes da linguagem de entrada.

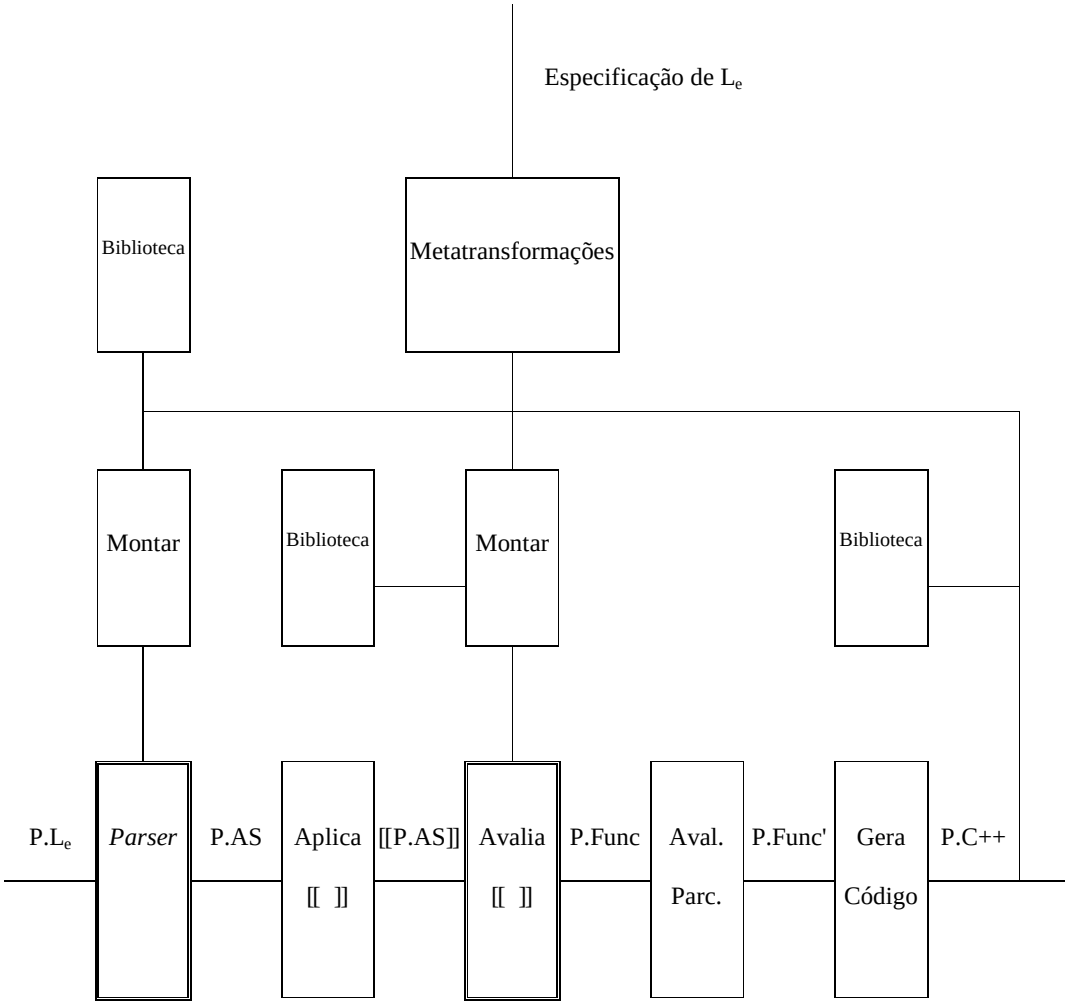

Figura 2.14 Visão completa do sistema de tradução baseado em semântica.

2.4.2 Dificuldades identificadas no projeto

No trabalho acima atingimos o objetivo de garantir a preservação da semântica do programa original. No entanto, não ficamos satisfeitos com a qualidade do código gerado. O programa final não era semelhante ao programa original e sua manutenção era difícil. Parecia-nos, intuitivamente, que o programa possuía outras informações relevantes ao problema que se propunha solucionar, além de sua semântica, que eram perdidas no processo. Essa impressão pode ser explicada pelos conceitos desenvolvidos nesta tese,

como o de semântica de árvores, que captura a intenção do programador e não apenas a extensão do programa.

2.5. Tradução de CHILL para C++.

O trabalho de projetar um tradutor de CHILL para C++ [Garcia e Guedes, 1999], iniciou-se no final de 1996; dessa forma, pôde se beneficiar da experiência adquirida nos projetos mencionados anteriormente.

2.5.1 Visão Geral

A maioria das aplicações existentes em CHILL [CCITT, 1997] roda em plataformas de hardware e software dedicadas. O rápido desenvolvimento dos equipamentos de telecomunicações tornou o investimento necessário para uma atualização constante destas plataformas não compensador, especialmente para pequenos sistemas privados. Assim, a indústria está se dirigindo para plataformas abertas. Além disso, a linguagem CHILL não é adequada para as novas tecnologias de desenvolvimento orientadas a objeto, e há escassez de programadores com experiência em CHILL. Desta forma, diversas empresas estão direcionando seus esforços de desenvolvimento para C++.

Uma das opções a serem consideradas na migração de um sistema para uma nova linguagem é a tradução automática dos programas fonte. Apesar do sucesso dessa abordagem estar razoavelmente disseminado, não é claro, em absoluto, que esse sucesso poderia ser repetido na tradução de CHILL para C++. Os sistemas de telecomunicação devem atender a requisitos **extremamente** rigorosos de consumo de tempo e de espaço de memória. Assim como a maioria das linguagens orientadas a domínio, CHILL possui construções feitas sob medida para ajudar os programadores a atenderem os requisitos de seu domínio. Linguagens de propósito geral podem não oferecer essas construções, o que é o caso na tradução de CHILL para C++.

Foram encontradas diversas dificuldades na tradução de CHILL para C++ como, por exemplo, os tipos conjunto, a compatibilidade de tipos, e a tradução de módulos. No entanto, a principal dificuldade encontrada foi a inicialização de variáveis em tempo de compilação. Apresentaremos, como exemplo, apenas a inicialização de *arrays*. Em C++, a única estrutura disponível para a inicialização de *arrays* em tempo de compilação é a enumeração dos valores iniciais do *arrays* entre chaves, enquanto CHILL oferece diversas possibilidades, entre elas a ilustrada na figura 2.15. É importante ressaltar que a tradução dessas inicializações para atribuições em tempo de execução é o suficiente para o sistema deixar de atender a seus rigorosos requisitos de consumo de tempo e de espaço.

A figura 2.15 mostra a tradução de um tipo *array* de T_SET e de uma variável desse tipo para C++. A tradução proposta busca preservar tanto a eficiência quanto a manutenibilidade do código gerado. Para se obter uma tradução legível para o acesso aos elementos de *arrays*, foi construído um *template* em C++. Este *template* oferece um operador de acesso a elementos − "[ ]" − que calcula *offsets*. Esse operador é implementado *inline*, de forma a não comprometer a eficiência.

Uma vez que as inicializações de *arrays* devem ser preservadas em tempo de compilação e as classes em C++ não suportam inicializações desse tipo, a tradução também gera um *array* usual de C++. Em tempo de compilação, é executada uma única atribuição de ponteiro para associar o *template* ao *array*.

CHILL	C++
NEWMODE T_ARRAY ARRAY (H'08:H'0F) T_SET; DCL V_ARRAY T_ARRAY INIT := (: (POS1): VAL1, (POS2): VAL2, (POS3): VAL3:);	typedef T_SET _T_ARRAY[1-0x08+0x0F]; typedef array<T_SET> T_ARRAY; T_SET _V_ARRAY = INIT_ARRAY(0x08, 0x0F, (POS1), VAL1, (POS2), VAL2, (POS3), VAL3); T_ARRAY V_ARRAY(0x08,0x0F,_V_ARRAY);

Figura 2.15 Exemplo de tradução de CHILL para C++.

A parte mais difícil é a inicialização do *array*. A estrutura de inicialização em CHILL significa: “inicializar a posição `POS1` com o valor `VAL1`”, e assim por diante. Caso essa estrutura fosse traduzida por uma inicialização usual em C++, teríamos algo do tipo: `T_SET _V_ARRAY = {0, 0, 0, VAL1, 0, 0, VAL2, VAL3} ;`. Esse código perderia o nome das posições onde os valores são atribuídos. Estes nomes são de suma importância para a manutenibilidade do programa, pois possuem um significado importante para o programador. Ao se alterar a constante `POS1` no programa traduzido, o programa não funcionará mais, pois deixou de existir uma parametrização do programa por esta constante.

Dessa forma, a tradução proposta na figura 2.15 sugere que a inicialização seja feita através de uma macro. No entanto, esta macro não pode ser implementada pelo pré-processador usual de C++, pois este não admite repetição (nem recursão). Esse mesmo problema foi encontrado em diversos pontos da tradução de CHILL para C++. Dessa forma, sugerimos a possibilidade de essas macros serem implementadas por uma ferramenta de transformação que estenda o poder do pré-processador de C++. Os detalhes podem ser encontrados em [Garcia e Guedes, 1999].

2.5.2 Dificuldades identificadas no projeto

Nesse projeto existiu uma preocupação intensa com a eficiência e com a manutenibilidade do código gerado e encontraram-se soluções criativas para a preservação das mesmas. Esse projeto em particular evidenciou que existem certas construções que são oferecidas por algumas linguagens, mas não por outras. Devido à experiência acumulada nos projetos anteriores, as soluções já se guiavam, intuitivamente, para traduções que preservassem a estrutura dos programas traduzidos. O passo seguinte seria formalizar e justificar esse procedimento, o que foi feito na presente tese.