UNIVERSIDADE
DE PERNAMBUCO

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Departamento de Sistemas e Computação

Pós-graduação em Engenharia da Computação

FXTL: UMA LINGUAGEM PARA TRANSFORMAÇÕES DE PROGRAMAS

Alexandre Alves dos Santos Junior

DISSERTAÇÃO DE MESTRADO

Recife
19 de fevereiro de 2009

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Departamento de Sistemas e Computação

Alexandre Alves dos Santos Junior

FXTL: UMA LINGUAGEM PARA TRANSFORMAÇÕES DE PROGRAMAS

Trabalho apresentado ao Programa de Pós-graduação em Engenharia da Computação do Departamento de Sistemas e Computação da Universidade de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenharia da Computação.

Orientador: *Prof. Dr. Luis Carlos de Sousa Menezes*
Co-orientador: *Prof. Dr. Márcio Lopes Cornélio*

Recife
19 de fevereiro de 2009

AGRADECIMENTOS

- Ao nosso Senhor Jesus Cristo por essa conquista;
- Aos meus pais, Alexandre Alves dos Santos e Vera Lúcia Monteiro da Silva Santos, pela educação, paciência e esforço para permitir que este sonho seja realizado;
- À minha irmã, Carolina Monteiro da Silva Santos, pela paciência e companheirismo fornecido durante todos estes anos;
- Aos meus orientadores, Prof. Dr. Luis Carlos de Sousa Menezes e Prof. Dr. Márcio Lopes Cornélio, pela paciência, dedicação, entusiasmo, apoio, e confiança fornecida durante todo este trabalho;
- Aos amigos Marcelo Moura, Mário Monteiro, César Augusto, Henrique Rebêlo, Diogo Pacheco, George Cabral e Petrônio Braga pela disponibilidade e conhecimentos compartilhados;
- Aos meus familiares e amigos pelos vários momentos de confraternização, descontração e força.

RESUMO

A reestruturação de programas é uma atividade extremamente importante no ciclo de desenvolvimento de software. Mudanças no projeto, melhorias no reuso de componentes e alterações de requisitos são exemplos de atividades que levam à necessidade de uma reestruturação. Essa reorganização deve ser realizada com o apoio de alguma ferramenta para evitar que erros sejam introduzidos durante o processo. Porém, sabemos que as ferramentas de desenvolvimento tradicionais possuem um conjunto restrito de transformações normalmente escritas utilizando APIs proprietárias e complexas, o que torna as personalizações difíceis de serem realizadas. Na prática, o usuário torna-se limitado às transformações predefinidas no ambiente utilizado.

Os sistemas de transformações foram propostos a fim de que o usuário possa escrever suas próprias transformações,bem como personalizar algumas das já existentes. De acordo com seu domínio de trabalho, eles podem ser classificados de duas formas: os genéricos, possíveis de ser aplicados a qualquer programa, independentemente da linguagem em que o programa foi escrito; e os específicos, que só podem ser aplicados a programas escritos em uma linguagem específica. Existem dois pontos que influenciam negativamente o uso dos sistemas genéricos em larga escala: o primeiro, devido ao fato de os sistemas genéricos possuírem uma sintaxe complexa e substancialmente diferente das linguagens de programação; o segundo, por eles não levarem em consideração as informações semânticas de um programa.

O objetivo deste trabalho é a definição de uma linguagem de transformação genérica cuja semântica é bastante similar às linguagens de programação imperativas, com o intuito de facilitar a escrita das transformações visto que o paradigma imperativo é bastante difundido entre os desenvolvedores. Além disso, tornar disponíveis as informações semânticas de um programa, de modo que possam ser utilizadas para realizar validações durante a transformação como, por exemplo, validações de tipo e escopo e, com isso, evitar que programas semanticamente incorretos sejam gerados.

Palavras-chave: fxtl, linguagem, transformação, refatoração, semântica.

ABSTRACT

Program restructuring is an inherent activity to software development. Changes in design decision, new requirements and reuse improvement are examples of motivations for a software restructuration. Such activity should be computer-aided to ensure that no errors are introduced during the transformation process. Unfortunately, the most used development tools provide a fixed set of program transformations written using complex APIs. In other words, the user is limited to the fixed set of transformations provided by the tool.

Transformation systems were proposed to give a user the possibility of writing and customizing transformations without difficulty. In accordance with its application domain, the transformation systems could be classified in two categories: the generic ones, which allow the application of transformations to programs whose syntax can be expressed in BNF notation; and the specific ones, which allows program transformations to be applied only to programs written in a specific programming language. Differently from specific systems, in which a transformation language is very similar to the programming language, in generic transformation systems there is a considerable difference between the programming language and the transformation language. Indeed, it is far easier for the user to understand the specific transformation languages than the generic ones, as he is already familiar with its syntax. Moreover, the generic systems do not take into consideration the semantic information of a program and, therefore, a program semantically incorrect may be produced.

The purpose of this work is the definition of a generic transformation language, whose semantics is very similar to those of imperative programming languages with the aim of making the writing of generic transformations easier, since the majority of developers already have experience with that paradigm. Moreover, the generic transformation language let developers consult the semantic information present in the program through the semantic support provided, in order to avoid the generation of semantically incorrect programs.

Keywords: fxtl, language, transformation, refactoring, semantic.

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

Reestruturação de programas é uma atividade inerente ao desenvolvimento de software. Mudanças na estrutura do projeto, melhorias no reuso de componentes, mudança de requisitos e até mesmo novos requisitos motivam a reestruturação de um software. Sabemos que é difícil isolar cada mudança no design de um projeto, então, inevitavelmente, a reorganização deste projeto torna-se necessária. Visto que a reestruturação realizada de forma manual é uma atividade muito cara e que tende a produzir erros [CB87], a reestruturação de software deve ser uma atividade realizada de forma automática, com apoio de alguma ferramenta para garantir que erros não serão introduzidos durante a transformação [GN93]. Processos de desenvolvimento como, por exemplo, Extreme Programming [Bec99], assumem que a transformação de um programa é uma atividade extremamente necessária e que deve estar continuamente ligada aos ciclos de desenvolvimento de um projeto de software.

As ferramentas de desenvolvimento tradicionais possuem um conjunto fixo de transformações escritas utilizando APIs proprietárias e complexas, tornando personalizações difíceis de serem realizadas. Na prática, o usuário está limitado às transformações predefinidas no ambiente utilizado. Em razão disso, uma nova geração de sistemas de transformações têm sido criadas, que permitem ao usuário escrever suas próprias transformações como também personalizar algumas das já existentes. Atualmente, existem duas classificações para os sistemas de transformações: os genéricos, possíveis de serem aplicados a programas escritos em qualquer linguagem de programação (desde que a sintaxe da linguagem utilizada possa ser descrita formalmente); e os específicos, que só podem ser aplicados a programas escritos em uma linguagem específica [CB01].

As ferramentas para transformações genéricas existentes, em sua maioria, necessitam de um profissional com bom conhecimento sobre a linguagem de transformação, visto que tais ferramentas possuem, em geral, uma sintaxe complexa e substancialmente diferente das linguagens alvo da transformação. Além disso, quando o desenvolvedor escreve suas transformações, é necessário garantir que modificações indesejadas não ocorram na semântica do programa. Considere uma transformação que altera o nome de uma variável. Essa transformação requer mais do que apenas alterar o nome de uma variável:

- toda ocorrência aplicada desta variável também deve ser alterada;
- deve verificar se o novo nome já não existe no escopo da variável alterada.

Essas verificações são fortemente dependentes da semântica da linguagem e obrigam o desenvolvedor a entender as regras de escopo da linguagem, e este conhecimento, precisa ser representado no código da transformação.

Para facilitar a escrita de transformações dependentes da semântica da linguagem, nós propomos um sistema de transformação formado por duas ferramentas de descrição:

- um analisador sintático e semântico, que recebe o código fonte do programa e, além de validá-lo, gera a árvore sintática abstrata do programa. Essa árvore possui informações produzidas pelo analisador semântico, como por exemplo, informações de tipo;

- uma linguagem de transformação que possui primitivas para buscar e modificar a árvore sintática abstrata do programa. Essas modificações podem acessar informações semânticas como, por exemplo, tipo e escopo, produzidas pelo analisador semântico.

O objetivo deste trabalho foi a definição de uma linguagem de transformação genérica cuja semântica é bastante similar às linguagens de programação imperativas, com o intuito de facilitar a escrita das transformações, visto que o paradigma imperativo é bastante difundido entre os desenvolvedores. Além disso, a linguagem definida disponibiliza ao desenvolvedor o acesso as informações semânticas de um programa, tornando possível a realização de validações durante as transformações, como, por exemplo, validações de tipo e escopo. Note que será necessário um analisador sintático e semântico para cada linguagem de programação utilizada. Dessa forma, será possível distinguir o que são construções semânticas válidas.

As definições sintáticas de uma linguagem devem ser fornecidas ao engenho de transformação antes de iniciar qualquer transformação. Através das definições, é possível identificar o que são construções válidas nesta linguagem.

Portanto, este trabalho disponibiliza ao desenvolvedor uma linguagem de transformação genérica que faz uso de construções semânticas para realizar reestruturação de programas. Por ser genérica, existe uma única sintaxe para realizar transformações em programas, independentemente da linguagem em que se está trabalhando. Com isso, o custo para realização de transformações em uma empresa deve cair, considerando que uma empresa trabalha com diversas linguagens de programação. Com os sistemas específicos, para cada linguagem utilizada existirá uma linguagem de transformação distinta, o que obriga o desenvolvedor a conhecer diversas linguagens de transformação.

Esta dissertação está estruturada da seguinte forma:

- O capítulo 2 apresenta inicialmente uma breve introdução sobre as ferramentas que fornecem apoio ao desenvolvedor na hora de realizar transformações de programas. Essas ferramentas buscam, além de automatizar o processo de transformação, a redução do número de erros produzidos durante uma transformação. Após a introdução, detalharemos as ferramentas que foram estudadas durante este trabalho;

- O capítulo 3 apresenta a linguagem de transformação definida neste trabalho, chamada de FxTL - Flexible Transformation Language, uma linguagem imperativa que dispõe de operações sobre a árvore sintática de um programa para realização de buscas, casamento de padrões e reescrita de termos. Além disso, FxTL provê suporte semântico ao desenvolvedor para executar validações durante a transformação, evitando assim que erros sejam introduzidos;

- O capítulo 4 apresenta uma visão geral sobre a implementação do sistema de transformação definido; será apresentada uma descrição sobre as entidades envolvidas no processo de transformação, o que é fornecido ao engenho de transformação e o que é obtido. Além disso, discutiremos como funciona o engenho de transformação, que atualmente é composto por dois interpretadores. E, por último, apresentaremos pontos importantes da implementação da linguagem, como, por exemplo, a linguagem de programação adotada, os padrões de codificação utilizados, as definições arquiteturais, entre outros;

- O capítulo 5 apresenta as provas de conceito executadas para validar a linguagem de transformação definida. Descreveremos em detalhe a aplicação de leis de refinamento e refatorações em programas escritos em um subonjunto da linguagem de programação Java [AG96], chamado de SubJava;

- O apêndice A apresenta a sintaxe abstrata de FxTL, definida usando a notação EBNF;

- O apêndice B apresenta a semântica estática de FxTL, escrita utilizando semântica denotacional [Mos90].

CAPÍTULO 2

SISTEMAS DE TRANSFORMAÇÃO

Sistemas de transformação são ferramentas que fornecem apoio ao desenvolvedor na hora de realizar alterações de código em softwares. Sabe-se que essas alterações, se realizadas de forma manual, podem gerar erros. Este capítulo apresenta três sistemas de transformação largamente conhecidos: Jackpot [Jac07], TXL [Cor04a] e ATL [ATL07]. Antes de apresentarmos em detalhe cada ferramenta, faremos uma breve introdução aos sistemas de transformação.

2.1 VISÃO GERAL

Para apresentarmos uma visão geral dos sistemas de transformação, iremos classificá-los de acordo com suas características:

- Reescrita de termos

  Os sistemas baseados puramente em reescrita de termos definem um conjunto de equações sobre termos em um formato algébrico. ASFandSDF [WN96] é um exemplo característico desse tipo de sistema, visto que é uma linguagem de reescrita com sintaxe de termos definida pelo usuário.

- Reescrita de árvore

  A reescrita de árvore é uma extensão para permitir a reescrita através de construções sintáticas distintas, definidas através de gramática livre de contexto. TXL é um exemplo característico desse tipo de linguagem, que dá suporte a definição de regras de transformação baseadas em termos de primeira ordem de uma gramática livre de contexto.

- Reescrita visual

  Os sistemas de reescrita que se baseiam em gráficos realizam transformações através de um conjunto de regras de reescrita visuais. Por exemplo, temos OPTIMIX [Chr03], que é uma linguagem de especificação projetada para especificar otimizações. Uma regra casa com um sub-gráfico de um gráfico podendo, por exemplo, adicionar ou apagar nós.

  Desde o princípio do nosso trabalho, sabíamos que não entraríamos no âmbito das transformações visuais. Em razão disso, não consideramos os sistemas desta área para nossos estudos de caso.

- Reescrita de modelos

É uma reescrita equivalente às de programas, porém, aplicadas à modelos. É necessária a definição de um modelo fonte e um destino, juntamente com seus metamodelos. Apesar de representar outra área de pesquisa, apresentaremos neste capítulo ATL, uma linguagem de transformação orientada a modelos que foi utilizada para compararmos com o nosso trabalho.

É importante destacar que os sistemas de transformações também podem ser classificados de acordo com seu domínio de trabalho:

- Sistema de transformação específico

  É um sistema que trabalha apenas com uma única linguagem de programação, por exemplo, Java. Por dispor de uma linguagem com sintaxe muito semelhante à que se está trabalhando, o desenvolvedor Java normalmente não enfrenta dificuldade em seu aprendizado, o que se torna uma vantagem. Porém, sabemos que será necessário um sistema de transformação distinto para cada linguagem utilizada, o que pode se tornar inviável, tendo em vista a dificuldade de uma pessoa conseguir trabalhar bem com diferentes linguagens de transformação, até porque existirão diferenças substanciais de uma para outra principalmente em suas notações. Jackpot é um exemplo de sistema específico que será estudado neste capítulo.

- Sistema de transformação genérico

  É um sistema capaz de trabalhar com programas escritos em diversas linguagens de programação utilizando uma única notação. Além disso, utiliza-se uma única ferramenta independentemente da linguagem em que se está trabalhando. Normalmente a sintaxe adotada em tais sistemas é um pouco mais complexa se comparada com os sistemas de transformação específicos visto que sua sintaxe é substancialmente diferente da que se está trabalhando. TXL é um exemplo de sistema de transformação genérico e será apresentado neste capítulo.

Após essa breve introdução sobre as características dos sistemas de transformação, apresentaremos nas próximas seções as ferramentas estudadas.

2.2 JACKPOT

Jackpot é um sistema de transformação simples, que realiza operações de casamento de padrões e substituições de termos em programas escritos em Java, utilizando o ambiente NetBeans [Net07]. Por possuir uma linguagem com sintaxe amigável, desenvolvedores conseguem escrever transformações sem a necessidade de conhecer profundamente a sua sintaxe. Não foi definida com a intenção de ser uma linguagem completa, possível de ser aplicada a qualquer tipo de transformação, mas o propósito dos autores foi simplificar os casos mais comuns [Jac08], como por exemplo, a eliminação de variáveis não utilizadas. As transformações são realizadas na árvore abstrata que representa o programa a ser transformado. Os elementos do programa são representados pelos nós da árvore.

Uma regra em Jackpot consiste de duas partes: um padrão e uma ação de substituição, separadas pelo operador `=>`. O padrão é uma expressão Java ou um comando que precisa ser encontrado, enquanto a ação de substituição define o código que será produzido.

Um exemplo de transformação em Jackpot seria a execução da regra `false => true;`, com a qual toda ocorrência de `false` existente no programa será substituída por `true`.

Outro conceito existente em Jackpot é o de meta-variável, onde uma determinada substituição afetará uma lista de comandos ou expressões com características compartilhadas. Por exemplo, `$a && false => false;` é um exemplo de uso de meta-variável, onde neste caso `$a` é uma meta-variável, pois casará com qualquer expressão que esteja sendo validada em conjunto com `false`.

Para os casos em que é desejado um critério maior na seleção dos nós casados existe o operador de restrição `::` seguido de uma expressão condicional. A transformação apenas ocorrerá para os nós que retornam verdadeiro na avaliação desta expressão. É importante observar que essas expressões condicionais podem ser inclusive expressões da própria linguagem Java, como por exemplo, o comando:

```
$C.enable() => $C.setEnabled(true) :: $C instanceof java.awt.Component;
```

A transformação apenas ocorrerá se `$C` for uma instância de `java.awt.Component`.

Por último, o conceito de meta-lista, que é muito similar à meta-variável, porém é definida entre '`$`', podendo representar, por exemplo, nenhum, um ou vários comandos. Por exemplo, o comando

```
{ $stmts$; $type $value = $expr; return $value; } => { $stmts$; return $expr; }
```

remove uma declaração de variável desnecessária. No padrão, a declaração é representada por `$type $value = $expr;`. Antes da declaração podem existir vários comandos, um comando ou até mesmo nenhum comando, independentemente disso, a variável será removida.

Para fins de demonstração da ferramenta descrita, utilizaremos como exemplo a remoção de variáveis desnecessárias. Como podemos ver na Figura 2.1, no próprio ambiente NetBeans escrevemos a transformação desejada, e aplicamos no projeto. Na janela de saída são mostrados os pontos que casaram com o formato descrito na transformação.

Na Figura 2.2, o lado esquerdo representa o código antes da transformação e o direito representa o código resultante da transformação.

O ambiente NetBeans ajuda o desenvolvedor na escrita e aplicação de suas transformações, inclusive distinguindo as palavras chaves da linguagem, visto que as mesmas possuem uma cor distinta das demais.

Apesar de ser um sistema de transformação específico para Java, Jackpot não conhece a semântica da linguagem, ou seja, vai contra uma das principais vantagens dos sistemas específicos, e por isso transformações semanticamente incorretas podem ser geradas a partir dele. Sistemas específicos normalmente conhecem a semântica da linguagem alvo e, portanto, não permitem que programas semanticamente incorretos sejam gerados. Por

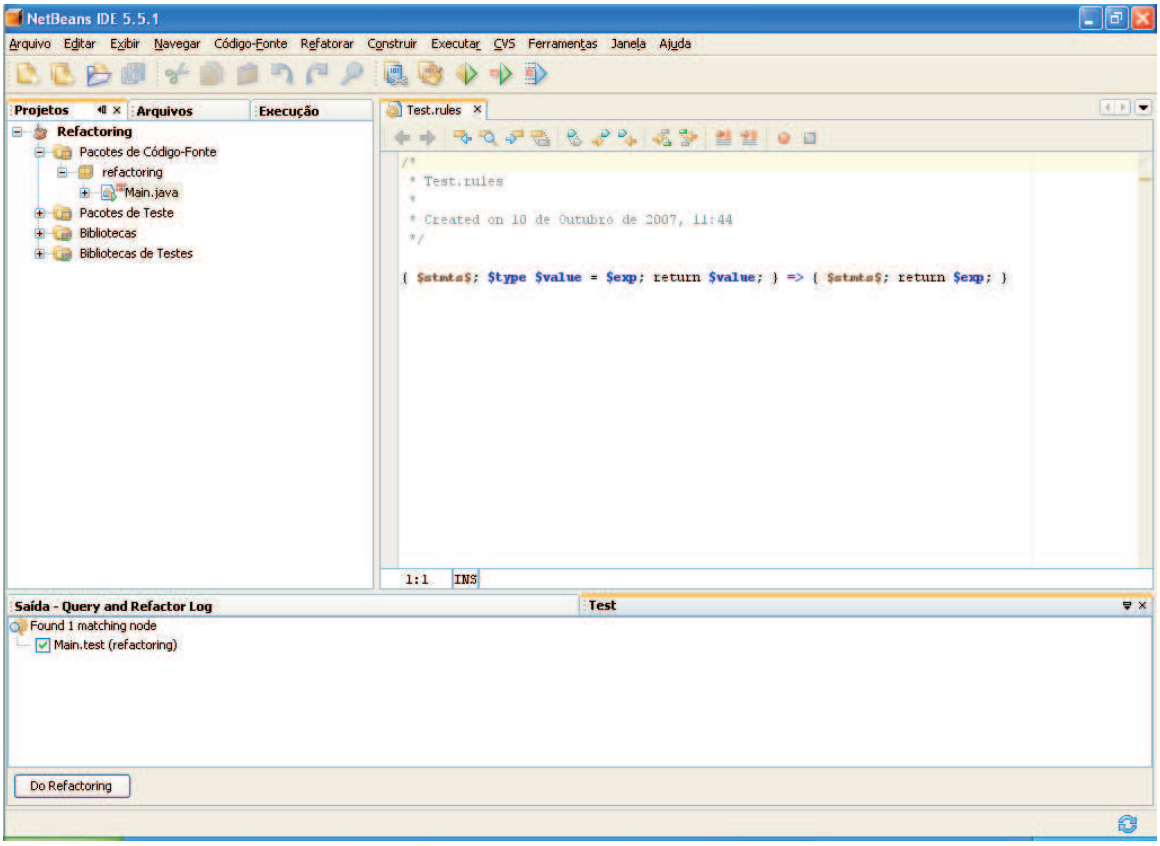

Figura 2.1 O ambiente NetBeans.

exemplo, o comando `false => 10;` é uma transformação possível de ser executada que gera como resultado um programa com erros de tipo. Além disso, por prezar pela simplicidade, Jackpot acaba perdendo em flexibilidade, visto que construções necessárias para realização de transformações complexas, como por exemplo, controle de fluxo, não existem na linguagem. Portanto, Jackpot é uma boa escolha para realização de transformações simples visto que elas são escritas sem dificuldades, até mesmo para um desenvolvedor inexperiente.

2.3 TXL

TXL é uma linguagem genérica definida especificamente para suportar análises de software e transformações de código. É uma linguagem madura, com mais de 15 anos de pesquisa concentrada em transformações baseadas em regras, para o desenvolvimento rápido de soluções para problemas complexos de computação. É uma linguagem funcional, capaz de realizar iterações e casamento de padrões durante as transformações.

Uma transformação em TXL é composta pela gramática da linguagem alvo da transformação e por um conjunto de regras de transformação. A gramática deve ser especificada usando o formato BNF. Os desenvolvedores de TXL tornaram disponíveis gramáticas para diversas linguagens de programação. Por exemplo, existem gramáticas disponíveis

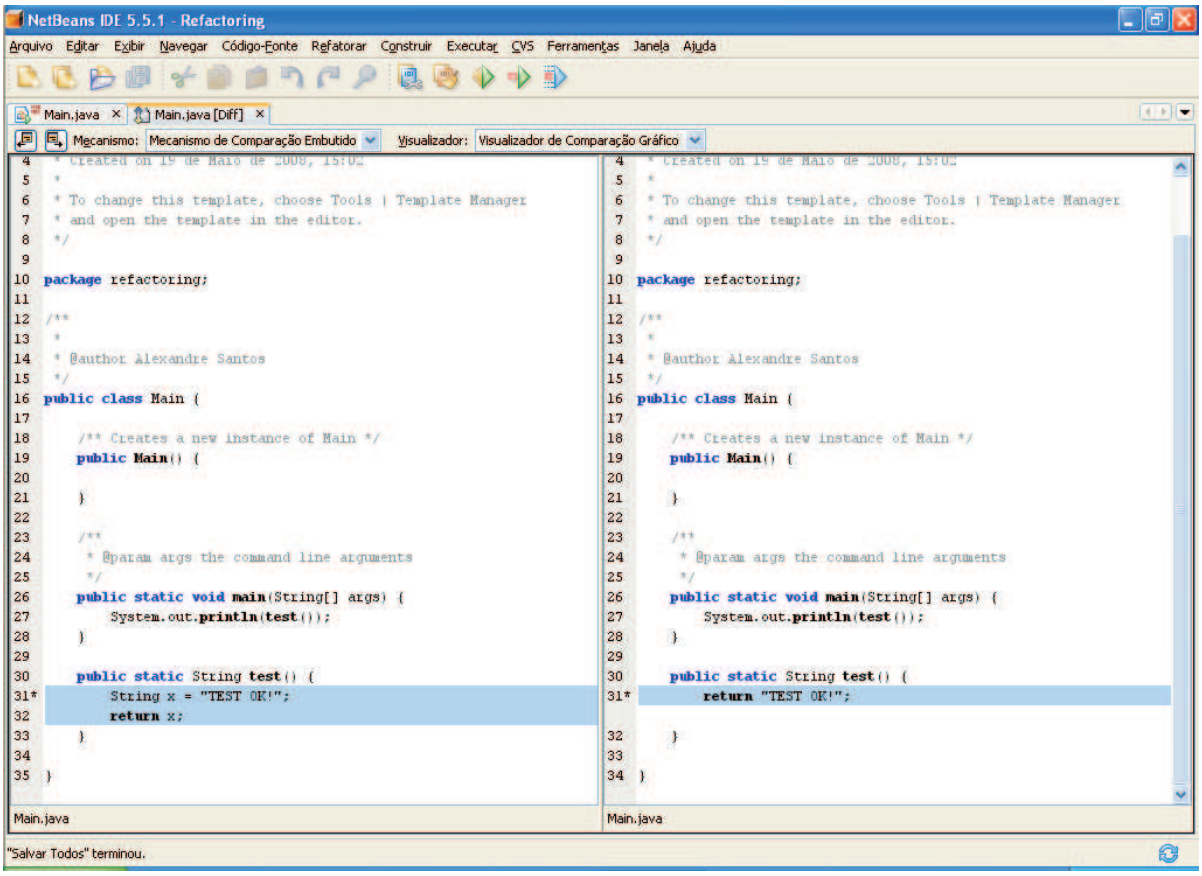

Figura 2.2 O antes e o depois da transformação.

para C, C++, Java, C#, Delphi, dentre outras linguagens. Dessa forma, desenvolvedores que trabalhem com essas linguagens não precisam definir suas gramáticas. As regras de transformação devem ser especificadas usando padrões e ações de substituição combinadas usando programação funcional. TXL possui uma estrutura funcional que disponibiliza escopo, abstração, parametrização e recursão usando regras de reescrita baseadas em Prolog, como busca de padrões, unificação de padrões e iteração implícita. Trabalha com a árvore abstrata do programa, árvore esta que fica oculta do usuário.

A execução de um programa TXL ocorre em três fases, como na maioria dos sistemas de transformações. Na primeira ocorre a construção da árvore sintática que representa o programa objeto. Em seguida, ocorre a etapa de transformação, onde as regras que foram escritas são aplicadas à árvore sintática. Por fim, a árvore resultante da transformação é convertida para o formato textual.

Todos os não terminais devem ser escritos entre colchetes. Tudo que for escrito sem os colchetes é considerado terminal. TXL possui uma lista de não-terminais internos que podem ser usados na definição da sua gramática, como por exemplo, `[number]`, que representa um número inteiro ou `[id]`, que representa um identificador. Porém, o desenvolvedor estará restrito às regras pré-estabelecidas a partir desses não-terminais internos, e se uma determinada linguagem de programação possuir uma pequena diferença, como

por exemplo, na definição de um identificador, a utilização desses *tokens* não será mais possível.

Na Figura 2.3, encontra-se uma regra escrita em TXL que funciona como uma calculadora que executa adições. A regra `resolveAddition` procura por expressões da forma `[number] + [number]` em toda a árvore sintática. A substituição ocorre aplicando a soma dos dois números através de funções internas da linguagem, tendo como resultado outra expressão. Os nomes `N1` e `N2`, que aparecem antes do não-terminal `[number]` na regra, capturam os valores correspondentes na árvore, para poderem ser usados na transformação.

```
01 rule resolveAddition
02   replace [expression]
03     N1 [number] + N2 [number]
04   by
05     N1 [+ N2]
06 end rule
```

Figura 2.3 Um exemplo de código em TXL

```
01 include "Java.Grm"
02
03 rule main
04   replace [repeat declaration_or_statement]
05     C1 [local_variable_declaration] C2 [statement]
06   deconstruct C1
07     M1 [repeat modifier] T1 [type_specifier] V1 [variable_declarators];
08     deconstruct V1
09       VN1 [variable_name] E1 [equals_variable_initializer]
10       deconstruct VN1
11         N1 [id]
12       deconstruct E1
13         '= I1 [variable_initializer]
14         deconstruct I1
15           EXP1 [expression]
16   deconstruct C2
17     'return V2 [id];
18   where N1 [= V2]
19   construct NewB [statement]
20     'return EXP1;
21   by
22     NewB
23 end rule
```

Figura 2.4 Código para realização da transformação.

Para fins de demonstração desta ferramenta, utilizaremos como exemplo a remoção de variáveis desnecessárias em um programa escrito em Java. Na Figura 2.4, exibimos o código da transformação escrito em TXL. A gramática utilizada foi retirada do repositório

de gramáticas existentes na página oficial [Lab08]. A regra `main` (linhas 03-23) precisa existir em qualquer transformação, ela é o ponto inicial para o início da execução. Utilizamos o operador `deconstruct`, responsável por desmembrar variáveis transformando-as em partes menores (regras internas), para conseguir alcançar a expressão utilizada na declaração da variável, visto que essa expressão será usada no comando `return`. Foi utilizada também a expressão `where` (linha 18), onde é verificado se o identificador utilizado no retorno do bloco é o mesmo identificador definido na declaração da variável. Se resultar em verdadeiro, é criado um novo comando a partir do operador `construct`, que irá substituir os comandos `C1` e `C2`.

```
01  package demo;
02
03  public class Main {
04
05    public static String test() {
06      String x = "TEST OK!";
07      return x;
08    }
09
10    public static void main(String[] args) {
11      System.out.println(test());
12    }
13
14  }
```

Figura 2.5 O programa antes da transformação.

```
01  package demo;
02
03  public class Main {
04
05    public static String test() {
06      return "TEST OK!";
07    }
08
09    public static void main(String[] args) {
10      System.out.println(test());
11    }
12
13  }
```

Figura 2.6 O programa depois da transformação.

As Figuras 2.5 e 2.6 apresentam o código do programa antes e depois da transformação respectivamente. Executando a ferramenta Txl.exe, obteremos como resultado o programa transformado. É possível também executar a ferramenta com a opção de gravar o resultado em arquivo obtendo o arquivo Java atualizado.

TXL não possui um ambiente de desenvolvimento, portanto, suas transformações podem ser escritas em qualquer editor de texto. Isto pesa contra o uso da ferramenta, pois não existe uma ferramenta que auxilia o usuário na escrita da transformação, como por exemplo, destacando palavras reservadas. A interação do usuário com a ferramenta é muito importante durante o desenvolvimento. Apesar disso, TXL demonstra ser um

sistema de transformação bastante maduro, possível de ser aplicado em programas escritos em diversas linguagens de programação, tendo como aliado o fato de já existirem várias gramáticas disponíveis em seu repositório. Porém, é pré-requisito para o desenvolvedor ter experiência no paradigma funcional.

2.4 ATL

Antes de explicarmos ATL, é de fundamental importância uma introdução sobre MDA (Model-Driven Architecture) [StOSSG08], visto que ATL segue as suas premissas. MDA é uma metodologia de desenvolvimento de software criada pela OMG (Object Management Group) [Gro08d], que dá suporte a todo o ciclo de desenvolvimento, ou seja, engloba todas as fases do desenvolvimento de um sistema. Sua principal característica é a importância dada aos modelos gerados nas diferentes etapas do processo de desenvolvimento e no rastreamento dos modelos de uma etapa para a etapa seguinte. O processo presente na MDA é composto por quatro etapas principais, onde ao final de cada etapa um modelo é gerado. A passagem de uma etapa para a etapa seguinte é realizada através de transformações de um modelo para outro, onde estas transformações possibilitam o rastreamento entre modelos.

ATL é uma linguagem de transformação entre modelos, especificada através de metamodelos e uma sintaxe concreta. A partir de ATL, é possível produzir um conjunto de modelos-destino a partir de modelos-fonte. É uma linguagem híbrida de programação declarativa e imperativa. A forma preferível de escrita para as transformações é utilizando a programação declarativa que simplifica o mapeamento entre os modelos fonte e destino. Entretanto, também é possível utilizar construções imperativas, principalmente nos mapeamentos que são difíceis de serem definidos de forma declarativa.

ATL possui um excelente ambiente de desenvolvimento que funciona como um plugin para a plataforma Eclipse. Através dele, são disponibilizadas ferramentas que auxiliam na escrita das transformações, como por exemplo, destacador de sintaxe e debugger.

Uma transformação ATL é composta de regras que definem como os elementos do modelo fonte são encontrados e utilizados para criar e inicializar os elementos do modelo de destino. O processo de transformação pode ser dividido em três partes:

1. O cabeçalho, que é usado para declarar informações gerais como o nome do módulo, os metamodelos fonte e destino, entre outras.

2. Os assistentes (sub-rotinas baseadas em OCL [WK99]), que são usados para evitar redundância de código.

3. As regras, que são o núcleo das transformações ATL. São elas que definem como os elementos de destino (baseados no metamodelo de destino) serão produzidos a partir dos elementos de origem (baseados no metamodelo de origem).

Todas as transformações são unidirecionais, ou seja, operam de modo somente-leitura nos modelos-fonte, produzindo modelos somente-escrita. Uma transformação bidirecional precisa ser implementada como um par de transformações: uma transformação para cada direção, onde ambos os modelos assumirão os dois papeis possíveis.

O desenvolvedor ATL precisa saber utilizar o EMF [Mer08], um *framework* para a geração de ferramentas baseadas em modelos, que não é muito simples e, normalmente, dificulta o aprendizado de ATL. Em razão disto, foi criada uma notação textual simplificada chamada de Kernel MetaMetaModel (KM3) [AGG07], com a qual o desenvolvedor pode criar e editar metamodelos e em seguida, transformá-los no formato Ecore, que é utilizado pelo EMF do Eclipse. A sintaxe de KM3 é semelhante à linguagem de programação Java.

Além desse conhecimento informado, para tornar possível uma transformação em um programa Java, será necessário primeiramente converter o programa para um formato que é reconhecido pelo ATL, como por exemplo, XMI [Gro08c], UML [Gro08b], entre outros. Essa conversão não é uma tarefa simples, porém, existe uma ferramenta chamada de Java Abstract Syntax Discovery Tool [GRO08a], que faz parte do projeto MoDisco, um conjunto de ferramentas que auxilia na engenharia reversa de programas [Pro08]. Através dessa ferramenta, é possível transformar um programa Java em um formato que segue o modelo XMI. Lembrando que esta ferramenta faz uso de um metamodelo para a linguagem Java. Por exemplo, a Figura 2.7 mostra o código fonte que será convertido para o formato XMI.

```
01  package demo;
02
03  public class Main {
04
05    public static String test() {
06      String x = "TEST OK!";
07      return x;
08    }
09
10    public static void main(String[] args) {
11      System.out.println (test());
12    }
13
14  }
```

Figura 2.7 Programa Java que será transformado.

Após a execução da ferramenta de engenharia reversa no código apresentado acima, teremos como resultado o código mostrado na Figura 2.8. Este já esta pronto para ser aplicado ao engenho de transformação.

Após esta etapa, forneceremos como entrada para o engenho de transformação três arquivos contendo respectivamente:

1. O código do programa Java convertido no formato XMI
2. O metamodelo da linguagem Java.
3. As transformações escritas em ATL (Figura 2.9).

Para fins de demonstração da ferramenta descrita nesta subseção, utilizaremos como exemplo a remoção de variáveis desnecessárias em um programa Java. Basicamente, estamos utilizando o modo de execução em refinamento, ou seja, onde é necessário especificar

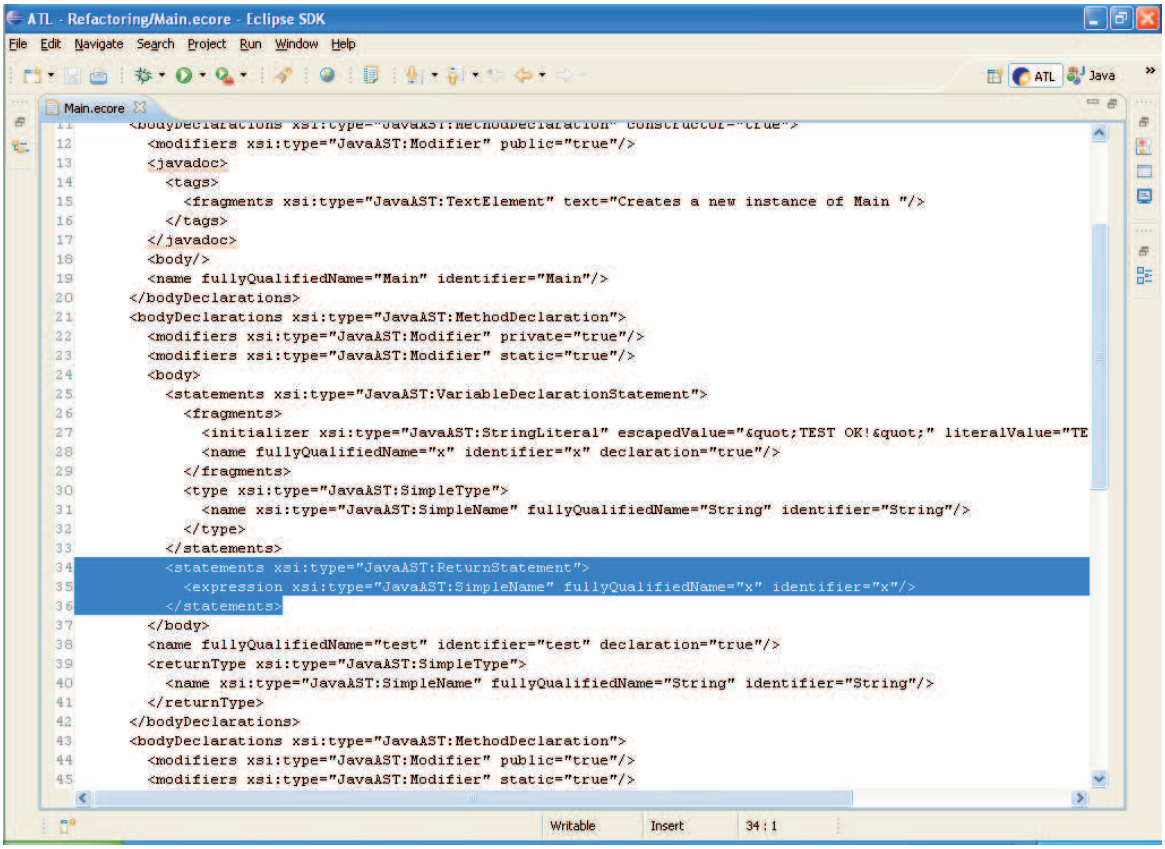

Figura 2.8 Programa convertido para o formato XMI.

apenas as mudanças que ocorrerão no modelo, sendo o restante copiado para o modelo resultante. Caso tivéssemos optado pelo modo de execução normal, teríamos que escrever regras para todos os termos presentes no modelo. A regra `BlockVars` 2.9 verifica se existe alguma declaração de variável dentro do corpo de um método. Caso exista, ela verifica se existe um retorno de função com o valor presente nesta variável. Se encontrado, a regra `ReturnStmt`, que transforma a expressão presente na variável declarada em um comando de retorno para a mesma, é chamada e este comando é substituído pelo comando de retorno antigo.

Após a execução da transformação, temos como resultado o modelo apresentado na Figura 2.10. Como podemos observar, a diferença entre o modelo antes e depois da transformação é apenas no comando de retorno (parte selecionada).

Apesar de ATL ser uma linguagem de transformação madura e possuir um excelente ambiente de desenvolvimento, as etapas do processo de transformação são muito complexas. Existem dependências de ferramentas externas no processo de transformação, principalmente em transformações *source-to-source*. Além disso, existe pouca documentação disponível na internet.

Primeiramente é necessária a transformação do programa para o modelo XMI, que não é uma atividade trivial. Além de requerer um tempo considerável, se realizada manualmente, erros podem ser introduzidos no modelo resultante. Podem-se usar ferramentas

```
module RemoveUnnecessaryVars; -- Module Template
create OUT : JavaAST refining IN : JavaAST;

rule RemoveUnnecessaryVars {
  from
    st: JavaAST!AST
  to
    ds: JavaAST!AST (compilationUnits <- st.compilationUnits)
}

rule BlockVars {
  from
    st: JavaAST!Block
  to
    ds: JavaAST!Block (
      statements <- let value : JavaAST!Expression = st.statements->
        select( a1 | a1.oclIsKindOf(JavaAST!VariableDeclarationStatement))->
             collect( a1 | a1.fragments)->flatten()->
               collect( a1 | a1.initializer)->flatten()
          in
            let name : JavaAST!SimpleName = st.statements->
              select( a1 | a1.oclIsKindOf(JavaAST!VariableDeclarationStatement))->
                collect( a1 | a1.fragments)->flatten()->collect( a1 | a1.name)->flatten()
              in
                let var : String = st.statements->
                  select( a1 | a1.oclIsKindOf(JavaAST!ReturnStatement))->
                       collect( a1 | a1.expression)->flatten()->
                         select( a1 | a1.oclIsKindOf(JavaAST!SimpleName))->
                              collect(a1 | a1.identifier)->flatten()
                    in
                       if (value.notEmpty() and
                         name->select(x | x.identifier = var.first()).notEmpty()) then
                           st.statements->
                      select( a1 | a1.oclIsKindOf(JavaAST!VariableDeclarationStatement))->
                        collect( a1 | a1.fragments)->flatten()->
                          collect( a1 | a1.initializer)->flatten()->
                               collect(m | thisModule.ReturnStmt(m))
                       else
                         st.statements
                       endif

          )
}

lazy rule ReturnStmt {
  from
    i1 : JavaAST!Expression
  to
    Return :JavaAST!ReturnStatement(expression <- i1)
}
```

Figura 2.9 As transformações escritas em ATL.

para automatizar esta fase, como fizemos, porém não existem ferramentas para muitas linguagens atualmente e, mesmo as que existem, ainda não estão maduras o suficiente para serem usadas em projetos de larga escala.

Segundo, a escrita da transformação é uma tarefa muito complexa. Até para transformações simples, é necessária a utilização de diversos comandos, o que resulta em muitas linhas de código. Portanto, transformações que são simples em diversas linguagens, quando escritas em ATL possuem um elevado grau de complexidade, até mesmo porque não existem muitas informações publicadas a respeito de transformações "source-to-source", ou seja, no nosso caso, "Java-to-Java".

Por último, a transformação de XMI para a linguagem de origem, que no nosso caso foi Java. É uma atividade que normalmente não é necessária em outras linguagens de transformação, porém em ATL é necessária. Em resumo, ATL é recomendada para transformações envolvendo modelos e não entre linguagens de programação.

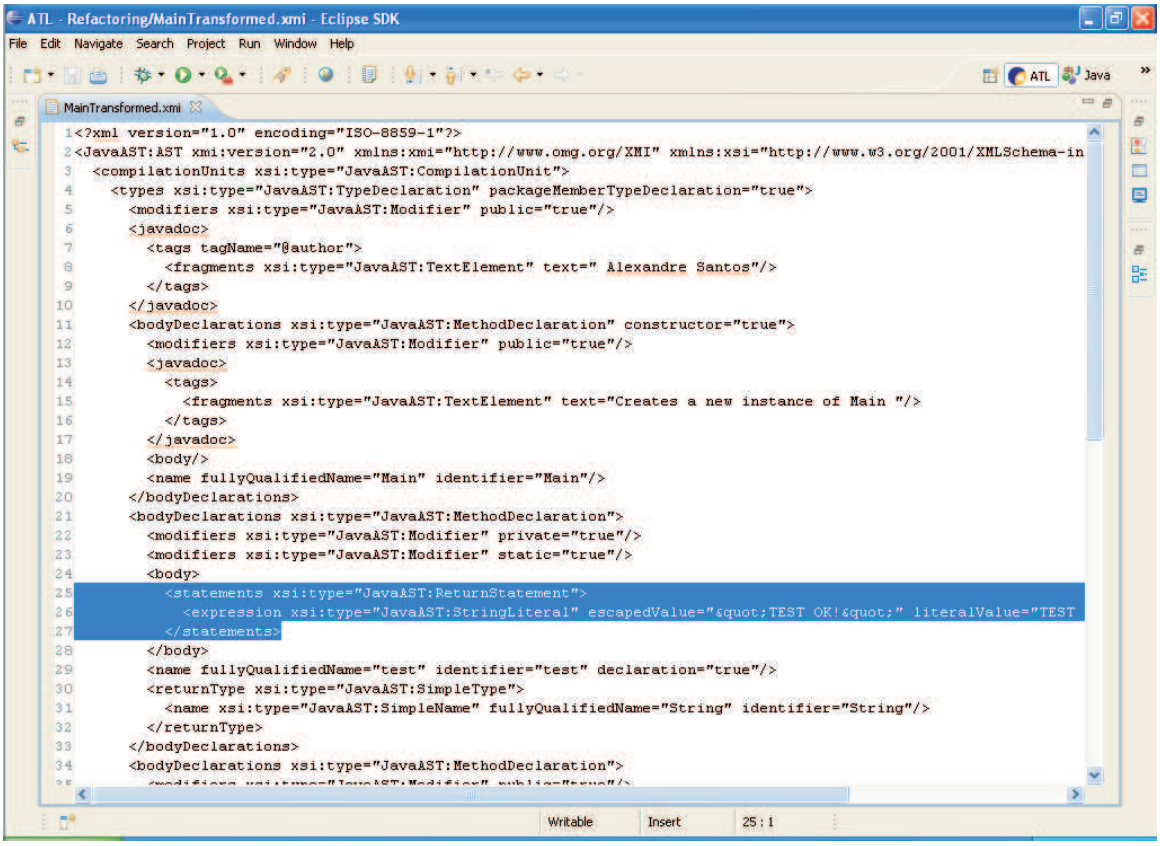

Figura 2.10 O programa no formato XMI após a transformação.

2.5 CONCLUSÕES

Neste capítulo apresentamos sistemas que fornecem apoio ao desenvolvedor na hora de realizar transformações de programas. Além de automatizar o processo de transformação, estas ferramentas podem evitar que erros sejam inseridos durante a transformação.

Jackpot é um sistema de transformação específico para Java bastante simples, que utiliza o ambiente de desenvolvimento NetBeans. Qualquer desenvolvedor familiarizado com Java consegue utilizá-lo sem dificuldades. As suas construções são muito simples, e por isso não é necessário um tempo considerável para seu aprendizado. É possível utilizar inclusive comandos condicionais de Java em suas construções.

Apesar de a maioria dos sistemas de transformação específicos conhecerem a semântica da linguagem em que se trabalha, Jackpot não apresenta essa característica. Transformações semanticamente incorretas são possíveis de serem geradas, por exemplo, programas com erros de tipos. Além disso, por prezar pela simplicidade, Jackpot acaba perdendo em flexibilidade, visto que operações complexas são mais difíceis de serem realizadas, visto que alguns comandos não existem na linguagem, por exemplo, controle de fluxo. Portanto, Jackpot é uma boa escolha para realização de transformações simples.

Já TXL, é uma linguagem de transformação funcional bastante madura, possível de ser aplicada em programas escritos em diversas linguagens de programação. TXL requer

como entrada a definição da sintaxe da linguagem em que se está trabalhando mas, por já possuir um repositório de gramáticas, não será necessário que o desenvolvedor defina uma para cada linguagem de programação utilizada.

TXL peca em relação ao ambiente de desenvolvimento, pois uma linguagem madura como ela deveria possuir um ambiente de desenvolvimento para melhorar a produtividade de seus usuários. Outra questão é que TXL não leva em consideração informações semânticas da linguagem e, portanto, programas semanticamente incorretos também podem ser gerados. Apesar disso, TXL é uma boa escolha para a realização de transformações complexas.

Por fim, ATL, uma linguagem de transformação bastante madura que possui um excelente ambiente de desenvolvimento. Apesar disso, as etapas existentes no processo de transformação são muito complexas. Existem dependências de ferramentas externas no processo de transformação, principalmente em transformações *source-to-source*. Além disso, existe pouca documentação disponível na internet. Por isso, ATL não é ideal para transformações *source-to-source*. Por ser uma linguagem orientada a modelos, é mais interessante a sua utilização em transformações envolvendo modelos distintos, o que seria bastante complicado ou até mesmo impossível de se fazer nas outras linguagens de transformação estudadas.

CAPÍTULO 3

A LINGUAGEM FXTL: FLEXIBLE TRANSFORMATION LANGUAGE

Foi visto no capítulo anterior que os sistemas de transformações genéricos possuem sintaxes complexas e distintas das linguagens de programação. Além disso, programas semanticamente incorretos podem ser gerados visto que os sistemas genéricos não possuem informações sobre a semântica do programa a ser transformado.

Este capítulo apresenta uma linguagem de transformação imperativa, dotada de construções sintáticas definidas para facilitar as operações sobre a árvore abstrata do programa. Essas construções fazem uso das definições sintáticas da linguagem de programação em que se está trabalhando, que deve ser fornecida à ferramenta antes da aplicação das transformações. Sabemos que a maioria das linguagens de transformações genéricas não levam em consideração a semântica do programa, por isso, uma das maiores motivações envolvida na definição de nossa linguagem foi prover acesso às informações semânticas de um programa. Com isso, validações semânticas podem ser realizadas durante uma transformação, evitando assim que programas semanticamente incorretos sejam gerados.

Portanto, apresentaremos a linguagem `Flexible Transformation Language`, também chamada de `FxTL`, uma linguagem de transformação simples, mas com poderosas construções para expressar transformações. FxTL é uma linguagem bastante intuitiva para programadores familiarizados com o paradigma imperativo. Seu objetivo é facilitar operações sobre a árvore sintática de um programa através do uso de funções de busca, casamento de padrões e reescrita de termos. Para descrever a linguagem em detalhe, nós apresentamos suas principais características em seções: tipos, expressões, comandos e declarações.

É importante destacar que os exemplos demonstrados neste capítulo foram escritos em um subconjunto da linguagem de programação Pascal [JW86], chamada de `SubPascal`.

3.1 TIPOS

Os tipos disponíveis em FxTL são: `Boolean`, `String`, `Integer`, `Node` e `NodeList`. Qualquer um destes tipos podem ser usados sobre os operadores disponíveis. As operações disponíveis para os valores do tipo `Boolean` são: `And`, `Or`, `Not`, `Equal to` e `Not Equal to`. Por exemplo, na Figura 3.1 mostramos um exemplo de algumas das operações disponíveis. Neste programa, `X` e `Y` terão os valores `2` e `4` associados, respectivamente. Logo em seguida, é verificado se `X` possui o valor `2` e `Y` um valor diferente de `2`. Como essa expressão é verdadeira, o valor `4` será atribuído a variável `X`.

Para os valores do tipo `String`, as seguintes operações estão disponíveis: `Equal to` (`==`), `Not Equal to` (`!=`) e Concatenação (`+`). Para exemplificar, na Figura 3.2 apre-

```
01 Start {
02   X = 2;
03   Y = 4;
04   If (X == 2 AND Y != 2) {
05     X = 4;
08   };
09 }
```

Figura 3.1 Exemplo de operações com valores booleanos.

sentamos o uso das operações de concatenação e `Equal to`. Primeiramente ocorre uma declaração implícita de `X`, que conterá a `String` "`Oi` ". Logo em seguida, é verificado se o valor associado a `X` é "`Oi` ", e como sabemos que avaliação resulta em verdadeiro, ocorrerá a concatenação do valor de `X` com o valor "`Mundo` ".

```
01 Start {
02   X = "Oi ";
03   If (X == "Oi ") {
04     X = X + "Mundo";
05   };
06 }
```

Figura 3.2 Exemplo de operações com strings.

Para valores do tipo `Integer`, as operações `Equal to` (`==`), `Not Equal to` (`!=`), `Less than` (`<`), `Greater than` (`>`), `Less or Equal to` (`<=`), `Greater or Equal to` (`>=`), Soma (`+`), Subtração (`-`), Multiplicação (`*`), Divisão (`/`) e Módulo (`%`) estão disponíveis. A Figura 3.3 demonstra algumas das operações disponíveis para valores inteiros. Neste caso, após a atribuição de `2` e `4` para as variáveis `X` e `Y` respectivamente, é verificado se o valor de `X` é menor que o valor de `Y`. Como sabemos que o resultado dessa avaliação é verdadeiro, ocorrerá a multiplicação entre `X` e `Y` e o resultado será atribuído a `X`.

Observe que os operadores `+`, `==` e `!=` são sobrecarregados, ou seja, o mesmo operador tem um comportamento distinto dependendo do tipo em que se está trabalhando.

```
01 Start {
02   X = 2;
03   Y = 4;
04   If (X< Y) {
05     X =X * Y;
08   };
09 }
```

Figura 3.3 Exemplo de operações com inteiros.

O tipo `Node` representa um ponto único de um programa, como por exemplo, a ocorrência de uma variável específica em um determinado programa escrito em SubPascal. Este ponto único é representado por um nó na árvore sintática abstrata do programa que está sendo transformado. Por exemplo, a variável global de nome `title` e tipo `String`

presente na Figura 3.4 é um exemplo de `Node` existente na árvore sintática do programa `temp`.

```
01 Program temp;
02   var title: String ;
03   Var rg: String;
04
05   begin
06      title := 'Just an example';
07      Writeln(title);
08   end.
```

Figura 3.4 Exemplo de um programa.

A definição de cada nó é baseada na sintaxe abstrata da linguagem fonte, que deve ser fornecida ao engenho de transformação de FxTL juntamente com o código da transformação. Estas definições precisam ser escritas seguindo o formato EBNF. Note que os tokens usados precisam ser definidos inicialmente usando o formato das expressões regulares da linguagem Java. Na Figura 3.5, existe um exemplo de definição sintática para SubPascal. Todos os exemplos mostrados neste capítulo seguem esta sintaxe.

```
TkId = \\p{Alpha}[\\w]*;
TkInteger = -?[0-9]+;
TkString = [\\p{ASCII}]*;

<Program> ::= ImpProgramDecl(<VarDecl>*,<FuncDecl>*,<ProcDecl>*,<Stmt>*)
<Exp> ::= IdUsage(TkId) | IntConst(TkInteger) | StringConst(TkString) | Sum(<Exp>,<Exp>)
<Stmt> ::= Assign(<Exp>,<Exp>) |WriteLn(<Exp>)
<VarDecl> ::= VarDecl (TkId,TkId)
<FuncDecl> ::= FuncDecl(TkId,TkId,<VarDecl>*,<VarDecl>*,<Stmt>*)
<ProcDecl> ::= ProcDecl(TkId,TkId,<VarDecl>*,<VarDecl>*,<Stmt>*)
```

Figura 3.5 Sintaxe abstrata de SubPascal.

`TkId`, `TkInteger`, `TkString` são exemplos de tokens. Eles são usados nas definições de nós para validar o formato dos valores primitivos. Em `VarDecl("String","title")`, temos uma declaração de variável que possui `String` como tipo e `title` como nome. Note que é possível utilizar nós encadeados, por exemplo, `Sum(IntConst("2"),IdUsage("x"))` representa a soma do número inteiro `2` e o valor associado à variável `x`, ou seja, a expressão `2+x`. Estes nós podem ser usados em operações de busca, casamentos de padrões e reescrita de termos. Estas operações serão discutidas em maior detalhe nas seções subseqüentes.

Por fim, `NodeList` representa uma lista de nós presente em um programa e pode ser, por exemplo, uma lista de declarações de variáveis globais, como podemos ver na Figura 3.4. Lembrando que neste caso específico, esta lista possui como elementos a variável de nome `title` e tipo `String` e a variável de nome `RG` e tipo `String`. Ou seja, a variável `title` é um nó e está contida dentro de uma lista de nós que contêm todas as declarações de variáveis globais.

É importante observar a diferença entre um nó e uma lista de nós. Os termos terminados com asterisco, como por exemplo `<VarDecl>*`, representam uma lista de nós. Esta lista precisa ser definida entre colchetes quando representar o argumento de um nó. Por exemplo,

```
ImpProgramDecl([VarDecl("string","x")],[],[],[Assign(IdUsage("x"),IntConst("10"))])
```

representa um programa escrito em SubPascal, que possui uma declaração de variável, nenhuma declaração de função e procedimento, e uma atribuição de `10` para `x`. Colchetes vazios representam uma lista vazia e `[VarDecl("string","x")]` representa uma lista de declarações de variável, neste caso com apenas um elemento. Entretanto, a definição `Assign` na Figura 3.5 utiliza apenas nós como argumento (ou seja, não aceita lista de nós). Por isso, `Assign(IdUsage("x"),IntConst("10"))` não utiliza colchetes.

3.2 EXPRESSÕES

As linguagens de transformação em sua maioria possuem expressões em comum com as linguagens de programação. Por serem comuns, essas expressões não precisam ser detalhadas. Apesar disso, várias já foram descritas na seção anterior. Nesta seção, apresentaremos apenas as expressões que possuem características peculiares, que agregam valor ao entendimento da linguagem.

3.2.1 Casamento de Padrões

O operador `is` lida com casamento de padrões. É responsável por testar se um nó possui um formato específico. Por exemplo, verificar se um nó representa uma declaração de variável. A sintaxe da expressão de casamento de padrão é:

```
Expression ::= Var "is" Node
```

sendo `Var` a variável que representa o nó a ser testado e `Node` que define o formato do nó.

Considere a situação em que é desejado verificar apenas o nome de uma variável, independentemente de seu tipo. O nó que representa uma declaração de variável em SubPascal é composto por dois filhos: o tipo e o nome da variável. Quando variáveis não declaradas são utilizadas na expressão de casamento de padrão, sabe-se que qualquer valor é aceitável naquela posição. Portanto, se desejamos verificar apenas o nome de uma variável, utilizaremos uma variável não declarada na posição referente ao seu tipo. Se a avaliação resultar em verdadeiro, a variável não declarada poderá ser usada dentro do bloco condicional encapsulando o valor do argumento em sua posição. Ou seja, no nosso exemplo, a variável não declarada conterá o valor do tipo da declaração. Porém, se a expressão de negação (`not`) for utilizada na operação de casamento de padrão, o caso verdadeiro será quando o nó não casar com o formato especificado. Portanto, as variáveis não poderão ser usadas visto que elas não terão valores associados. Na Figura 3.6, existe um exemplo que verifica se o nó `W` é uma declaração de variável que possui qualquer valor como tipo (no caso `X` é uma variável não declarada) e `title` como nome. Se resultar em verdadeiro, a variável `X` pode ser usada dentro do bloco condicional.

```
...
04  If (W is VarDecl(X, "title"))
05  {
06    ...
10  }
...
```

Figura 3.6 Exemplo da expressão de casamento de padrão.

3.2.2 A Cláusula Exist

O operador `exist` é similar ao operador `is`, porém, ao invés de testar se um nó possui um formato específico, verifica a existência de um nó com um formato específico dentro de um determinado escopo, de acordo com a definição de nó utilizada. Se pelo menos um nó é encontrado, o resultado é verdadeiro. Caso contrário, falso. A sintaxe da cláusula `exist` é:

```
Expression ::= "Exist" "(" Node ")" "in" Var
```

onde `Node` representa o formato do nó que se deseja encontrar, e `Var`, a variável que representa o escopo da pesquisa.

Nesta expressão, também é possível utilizar variáveis não declaradas na definição do nó, porém, ao contrário da expressão de casamento de padrão, independentemente de o resultado ser verdadeiro ou falso, não será possível a utilização destas variáveis dentro do bloco condicional. A expressão `exist` também pode ser usada juntamente com a expressão de negação (`not`). O código na Figura 3.7 verifica se existe uma declaração de variável global que possui qualquer valor como tipo (`X` é uma variável não declarada) e `title` como nome. Note que `W` representa o escopo global do programa. É possível substituir este nó por qualquer outro que represente um escopo. Por exemplo, para verificar se existe uma declaração de variável cujo nome é `title` dentro de uma função específica, é necessário utilizar a palavra reservada `in` seguida do nó que representa a declaração desta função, visto que não desejável obter, por exemplo, uma função que possui duas declarações de variável com o mesmo nome. Sem essa validação, este erro poderia ocorrer visto que uma variável poderia ser renomeada para um nome já existente no mesmo escopo. Ressaltamos que mesmo que exista uma declaração neste formato, a variável `X` não será visível dentro do bloco condicional.

```
...
04  foreach(W in ImpProgramDecl(A,B,C,D)) {
05    If (exist (VarDecl(X, "title")) in W)
06    {
07      ...
10    }
...
```

Figura 3.7 Exemplo da expressão Exist.

3.3 COMANDOS

FxTL possui comandos definidos para operar sobre árvores sintáticas abstratas. Algumas dessas operações são discutidas em detalhe ao longo desta seção.

3.3.1 Foreach

O comando `foreach` é iterativo e faz uso da definição de um nó para encontrar todos os nós que casam com um determinado formato. Se variáveis não declaradas forem usadas como parâmetros na definição do nó (representam qualquer valor), os valores correspondentes serão associados a elas, tornando possível a sua utilização durante as iterações. Note que cada iteração se dará sobre um diferente nó e, portanto, diferentes argumentos serão associados. A sintaxe do comando `foreach` é:

```
Command ::= "foreach" "(" Var "in" Node ")" "{" Commands "}"s
```

onde `Var` é a variável que apontará para o nó alvo da iteração; `Node` define o formato dos nós que serão casados e `Commands` representa os comandos que serão executados durante as iterações.

A Figura 3.8 mostra um exemplo de uso do comando `foreach`. Há uma iteração sobre todas as declarações de variáveis, independentemente do tipo (`X` é uma variável não declarada) e `title` como nome. Observe que todo nó que segue o formato usado na definição de nó será iterado e poderá ser acessado pela variável `W`. Além disto, a variável `X` possuirá o valor que representa o tipo do nó que está sendo iterado. Conseqüentemente, é possível realizar diferentes transformações em cada nó, visto que cada um será iterado por vez.

```
...
04  foreach  (W inVarDecl(X, "title"))
05  {
06    ...
10  };
...
```

Figura 3.8 Exemplo do comando Foreach.

É possível agrupar os nós que serão iterados através de `foreach` encadeados. Por exemplo, para acessar de forma agrupada todas as declarações de variáveis existentes em todas as funções de um programa, é necessário definir um `foreach` que itere sobre todas as funções existentes no sistema, e interno a ele, define-se outro `foreach` que itere sobre todas as declarações de variáveis da função iterada no momento. Em resumo, sobre cada função existente no programa, existirão iterações sobre todas as suas variáveis declaradas. O código para a realização deste exemplo é apresentado na Figura 3.9.

É importante notarmos a diferença entre os exemplos da Figura 3.8 e da Figura 3.9. No primeiro, todas as declarações que possuem qualquer tipo e `title` como nome estão sendo acessadas, independentemente de onde foram declaradas. Porém, no segundo caso,

estão sendo acessadas apenas as declarações de variáveis locais às funções. Portanto, esta diferença deve ser levada em consideração quando se esta escrevendo transformações para evitar que resultados inesperados aconteçam, como por exemplo, trocar o nome de variáveis globais quando se queria apenas trocar o nome de variáveis locais, ou até mesmo para otimizar suas transformações. Pode-se também fazer verificações de escopo para garantir a corretude das transformações, como veremos na Seção 3.5.

```
...
04  foreach (W in FuncDecl(ReturnType, FuncName, ParamsDecl, VarsDecl ,Stmts)) {
05    foreach(R in VarsDecl ) {
06      ...
07    }
08  }
...
```

Figura 3.9 Uso de foreach encadeado.

3.3.2 Reescrita de Termos

O comando de reescrita de termos substitui um nó por outro. A sua sintaxe base é:

```
Command ::= Var "=>" Node
```

onde `Var` é a variável que apontará para o nó alvo da transformação e `Node` que definirá o novo formato do nó.

Através da sintaxe base é possível realizar apenas a substituição de um nó específico por um novo nó. Na Figura 3.10, nós transformamos cada nó iterado (`W`) em um novo com mesmo tipo e nome concatenado com a string "`1`".

```
...
04  Foreach (W in VarDecl(X, Y))
05  {
06    W=>VarDecl(X, Y+"1");
07  };
...
```

Figura 3.10 Transformação de um nó em outro.

Sabemos que nem todas as transformações apenas substituem um nó por outro. Existe também a necessidade de adicionar nós em posições específicas de uma lista, como após uma declaração de variável, ou antes de uma atribuição de variável específica. Para isso, temos duas extensões da linguagem para tratar adições em uma lista. A primeira, lida com adições de um novo nó após um nó específico e adição de um novo nó no final de uma lista de nós. A sua sintaxe é:

```
Command ::= Var "=>" Var "+" Node
```

onde o primeiro e segundo `Var` representam a variável alvo da transformação, que apontará para um nó ou para uma lista de nós e `Node` que define o novo nó a ser adicionado.

A segunda, realiza adições de um novo nó antes de um nó específico e adições no início de uma lista de nós. A sua sintaxe é:

```
Command ::= Var "=>" Node "+" Var
```

onde o primeiro e segundo `Var` representam a variável alvo da transformação, que apontará para um nó ou para uma lista de nós e `Node` que define o novo nó a ser adicionado.

Porém, nenhuma das formas descritas acima lidam com remoção de nós. Para isso, temos a seguinte sintaxe:

```
Command ::= VarList "=>" VarList "-" Var
```

onde o primeiro e segundo `VarList` representam a variável que apontará para uma lista de nós e `Var` que representa a variável que apontará para o nó a ser removido da lista de nós.

Não é permitida a adição de nós antes ou depois de um nó único. Só é possível adicionar nós quando se está trabalhando com lista de nós, do contrário, um erro será gerado e a transformação será abortada. Os nós únicos são argumentos das definições de nós definidos sem o asterisco (veja Figura 3.5).

Na Figura 3.11, nós apresentamos o código para adicionar um novo nó antes do nó atual (alvo da iteração). Depois da execução, o número de variáveis declaradas será duplicado porque antes de cada declaração, nós adicionamos uma nova declaração de variável de mesmo tipo e nome concatenado com a string "1".

```
...
04  Foreach (W in VarDecl(X, Y))
05  {
06    W=>VarDecl(X, Y+"1") + W;
07  };
...
```

Figura 3.11 Adicionando um nó antes do nó iterado.

Escrevendo `W => W + VarDecl(X, Y+"1")`, um novo nó será adicionado depois do nó atual. Este caso é muito similar ao anterior mostrado na Figura 3.11, porém, aqui o nó é adicionado depois do nó alvo da iteração.

Figura 3.12 apresenta um exemplo de transformação que adiciona um novo nó no início de uma lista de nós. Depois da execução, uma nova variável global é adicionada no início da lista de variáveis globais. A lista de variáveis globais `Vars` é utilizada na definição de nó no comando `foreach`, e, como `Vars` é uma variável não declarada, podemos utilizá-la dentro do escopo do `foreach` com seus respectivos valores.

Escrevendo `Vars => Vars + VarDecl("string", "newGlobalVar")`, um novo nó será adicionado no final da lista de variáveis globais. Este caso é muito similar ao anterior

```
...
04  start {
05    foreach(W in ImpProgramDecl(Vars,Funcs,Procs,Stmts))
06    {
07      Vars => VarDecl("string", "newGlobalVar") + Vars;
08    };
09  }
...
```

Figura 3.12 Adicionando um nó no início de uma lista de nós.

mostrado na Figura 3.12, porém, aqui o nó é adicionado no final da lista de nós (variáveis globais).

Figura 3.13 apresenta um exemplo onde ocorre a remoção de um nó de uma lista de nós. Após a execução, a variável `x` de tipo `string` é removida da lista de variáveis globais.

```
01  Start {
02    foreach(W in ImpProgramDecl (Vars,Funcs,Procs,Stmts )) {
03      foreach(X in  Vars) {
04        if (X is VarDecl("string", "x")) {
05          Vars => Vars  - X;
06        };
07      };
08    };
09  }
```

Figura 3.13 Removendo um nó de uma lista de nós.

3.4 DECLARAÇÕES

A linguagem de transformação FxTL possui três tipos de declarações: declaração do ponto inicial do programa, declaração de variável e declaração de procedimento.

3.4.1 Ponto Inicial do Programa

A declaração do ponto inicial delimita o ponto de execução do programa. Ou seja, a parte executável da transformação estará definida neste bloco. Sem este bloco, o programa será tratado como uma biblioteca não podendo ser executado, apenas importado para reuso de procedimentos definidos. Um programa deverá possuir apenas uma declaração do ponto inicial do programa. Caso contrário, ocorrerá um erro e a transformação não será iniciada. Figura 3.14 mostra um exemplo de declaração do ponto inicial do programa.

Na linha 04, a palavra reservada `Start` é seguida por um parênteses esquerdo representando o inicio do ponto inicial do programa. Tudo entre o parênteses esquerdo e o direito será executado. Note que os procedimentos declarados poderão ser chamados dentro deste bloco.

```
...
04  Start {
...
08      ChangeName(X);
...
10  }
```

Figura 3.14 Declaração do ponto inicial do programa.

3.4.2 Variáveis

Uma variável em FxTL é declarada implicitamente através da sua primeira ocorrência em um comando, e seu escopo é o bloco onde ela ocorre. Na Figura 3.15 mostramos como introduzir uma variável através de sua ocorrência no lado esquerdo de uma atribuição. Nesta figura, nós introduzimos a variável `X` com seu valor inicial dado pela expressão `2`, cujo o tipo é `Integer`.

```
01  Start {
02    X=2; // X is an Integer
03    ...
08  }
```

Figura 3.15 Declaração de variável em uma atribuição.

Porém, se uma variável não declarada for utilizada no lado direito de uma operação de atribuição (Figura 3.16), um erro será gerado informando que variáveis não declaradas só podem ser utilizadas no lado esquerdo de uma operação de atribuição.

```
01  Start {
02    X=Z; // Z is undeclared
03    ...
08  }
```

Figura 3.16 Uso de variável não declarada de forma errônea.

3.4.3 Procedimentos

FxTL suporta procedimentos para facilitar a tarefa de escrita das transformações visto que cada etapa da transformação pode ser implementada como um procedimento. Os procedimentos podem ser recursivos desde que exista uma condição de parada. Caso contrário, ocorrerá um loop infinito. A partir de um procedimento, é possível chamar qualquer outro procedimento. O mecanismo de passagem de parâmetros adotado é por referência, o qual permite que o parâmetro formal seja ligado diretamente ao argumento. Na Figura 3.17, um exemplo de declaração de procedimento e de chamada à procedimento é demonstrado.

Na linha 01, o procedimento `ChangeName` é declarado recebendo como parâmetro uma lista de nós (`W`). O tipo desse parâmetro é `NodeList`, mas outros tipos também poderiam

ser usados. Dentro do procedimento, qualquer operação pode ser executada inclusive uma chamada recursiva ao próprio procedimento desde que exista uma condição de parada. Caso contrário, uma recursão infinita irá ocorrer.

Na linha 09, existe uma chamada ao procedimento `ChangeName`, passando `X` como argumento. Lembrando que `X` neste caso é uma variável declarada representando uma lista de nós. Caso contrário, um erro seria gerado informando que uma variável não declarada foi utilizada ou que uma variável de tipo distinto do esperado foi utilizada.

```
01 Procedure  ChangeName(W: NodeList) {
...
04 }
...
09     ChangeName(X);
...
```

Figura 3.17 Exemplo de declaração e chamada à procedimento.

3.5 SUPORTE SEMÂNTICO PARA TRANSFORMAÇÕES

A semântica de um programa possui informações importantes que precisam ser levadas em consideração em uma transformação para que resultados inesperados sejam evitados. Com essas informações, o desenvolvedor pode escrever transformações que geram programas corretos com mais facilidade. A geração da árvore abstrata do programa passa por duas fases: a análise sintática e a análise semântica. É na segunda fase que as informações semânticas do programa são escritas nos nós. Note que para cada linguagem de programação utilizada, será necessário um analisador sintático e semântico para decorar os nós existentes na árvore abstrata do programa.

Portanto, é possível consultar as informações semânticas de um nó utilizando o suporte semântico disponível para FxTL. Observe que após uma transformação, FxTL executa uma chamada ao analisador semântico, que automaticamente atualiza todas as informações semânticas dos elementos de um programa. Esta seção apresenta funções para lidar com escopo e tipo dos elementos de um programa.

3.5.1 Escopo

O escopo de uma declaração é a parte do programa onde a declaração é efetiva [Wat04]. Em FxTL, o escopo de uma declaração é representado por um nó, que é o bloco onde sucede a ocorrência de ligamento. Toda ocorrência aplicada deste identificador possui como escopo o mesmo nó. A função `Scope` que trabalha com o escopo dos elementos de um programa tem a seguinte assinatura:

```
Node Scope(Node arg)
```

A função `Scope` recupera as informações de escopo associada a um nó. Se um nó que não é nem uma ocorrência aplicada nem uma ocorrência de ligamento for utilizado como argumento desta função, uma exceção será levantada.

É importante entendermos o funcionamento da função `Scope` profundamente. Para

isso, na Figura 3.18 existe o exemplo de um programa escrito em SubPascal. Na linha 2 existe a ocorrência de ligamento da variável `title`, cujo escopo é representado pelo nó que encapsula o programa `temp`. Ou seja, se executarmos a função `Scope` passando como parâmetro o nó que representa a ocorrência de ligamento da variável `title`, obteremos como resultado o nó que representa o programa `temp`.

Observe que a variável `title` utilizada na linha 07 não refere-se à mesma utilizada na linha 12. A primeira possui como escopo o nó que representa a declaração da função `Test`; a segunda possui como escopo o nó que representa o programa `temp`. Se executarmos a função `Scope` para ambos os casos, teremos como resultado escopos distintos.

```
01 Program temp;
02  var title: String;
03  var rg: String;
04
05  function Test(title: String): String;
06  begin
07    Result := title +  rg;
08  end
09
10  begin
11     rg := '01010101'
12     title := 'Just an example';
13     Writeln(Test(title));
14  end.
```

Figura 3.18 Exemplo de um Programa escrito em SubPascal.

Na Figura 3.19, a função `Scope` (linha 04) é utilizada para verificar se o ambiente associado ao nó `X` (ocorrência de ligamento) e o ambiente recebido pelo nó `Y` (ocorrência aplicada) representam o mesmo nó. Essa validação é importante para garantir que apenas as ocorrências aplicadas de `x` que possuem o mesmo escopo serão transformadas. Caso contrário, ocorrências aplicadas de `x` com um escopo diferente também seriam renomeadas, o que não é o intuito da transformação. O objetivo seria renomear para `title` apenas as ocorrências aplicadas da variável de tipo `string` e nome `x`.

```
01  Start {
02   foreach(X in VarDecl("string","x")) {
03     foreach(Y in IdUsage("x")) {
04       if(Scope(X) == Scope(Y)) {
05         Y => IdUsage("title");
06       };
07     };
08     X => VarDecl("string", " title");
09   };
10  }
```

Figura 3.19 Exemplo da função de escopo.

3.5.2 Tipo

Na maioria das linguagens de programação, expressões são associadas a um tipo. Nestas linguagens, expressões podem ser utilizadas como argumento da função `Type`. Note que inspecionar o tipo de algum nó que não possui tipo associado gera uma exceção e a transformação é abortada. A função `Type` que trabalha com o tipo dos elementos de um programa possui a seguinte assinatura:

```
String Type(Node arg)
```

A função `Type` retorna uma `String` que representa o tipo do nó passado como argumento. Essa abordagem também é seguida por LUA [IdFF96].

Na Figura 3.20, a função `Type` (linha 04) é usada para verificar se o nó `X` (ocorrência de ligação) e o nó `Y` (ocorrência aplicada) possuem o mesmo tipo. Se verdadeiro, significa que existe uma ocorrência aplicada e uma ocorrência de ligação de mesmo tipo, e conseqüentemente, a operação de reescrita de termo não será executada. Caso contrário, a variável `count` não será incrementada, visto que não existe ocorrência aplicada da variável iterada, e, portanto, a reescrita de termo será executada modificando o tipo da variável de `string` para `Integer`.

```
01  Start {
02    Count = 0;
02    foreach(X in VarDecl("string","x")) {
03      foreach(Y in IdUsage("x")) {
04        if (Type(X) == Type(Y)) {
05          Count = Count + 1;
06        };
07      };
08      If(Count == 0) {
09        X => VarDecl("Integer", "x");
10      };
11    };
12  }
```

Figura 3.20 Exemplo da função de tipo.

Note que FxTL não está preparada para trabalhar com herança e polimorfismo. Em trabalhos futuros, operadores podem ser desenvolvidos para preencher esta lacuna.

3.6 UM EXEMPLO DE TRANSFORMAÇÃO

Com o intuito de demonstrar a aplicabilidade dos operadores de FxTL, nós apresentamos um exemplo de transformação aplicado em um programa escrito em SubPascal.

Nosso programa exemplo possui uma variável `x` que é utilizada em uma atribuição, e uma chamada ao comando `writeln`. O propósito é trocar o nome da variável de `x` para `title` por questões de legibilidade. As informações semânticas precisam ser levadas em consideração durante a transformação visto que apenas as ocorrências aplicadas de mesmo escopo deverão ser renomeadas, portanto, as funções semânticas introduzidas na Seção 3.5 são usadas nesta transformação. Figura 3.21 apresenta na caixa esquerda o programa original; o código transformado aparece na caixa do lado direito.

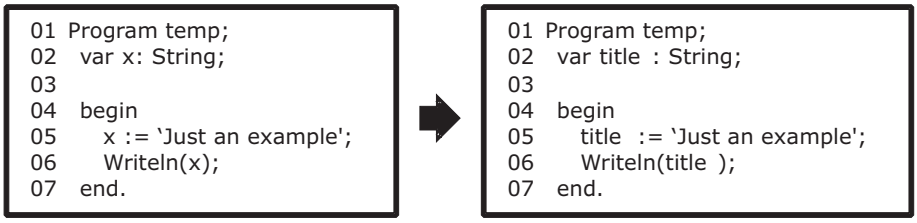

Figura 3.21 Exemplo de transformação.

Na Figura 3.22, nós apresentamos o código responsável pela transformação escrito em FxTL, de acordo com a sintaxe abstrata mostrada na Figura 3.5. O código procura por uma declaração de variável que possui `x` como nome e `String` como tipo. Se este nó for encontrado, o sistema verifica se não existe uma declaração de variável com qualquer tipo e nome `title` no mesmo escopo do nó encontrado. Se não existir, todo uso de `x` será substituído pelo uso de `title` e a declaração de `x` será substituída pela declaração de `title`.

```
01  Start {
02    foreach(W in ImpProgramDecl(A,B,C,D)) {
03      foreach(X in VarDecl("string","x")) {
04        if(not exist(VarDecl(ANY, " title")) in W) {
05          foreach(Y in IdUsage("x")) {
06            if(Scope(X) == Scope(Y)) {
07              Y =>IdUsage("title");
08            };
09          };
10          X => VarDecl("string", " title");
11        };
12      };
13    }
14  }
```

Figura 3.22 O código da transformação.

Os comandos dentro do `foreach` externo (linhas 02-13) são executados apenas uma vez, apenas para tornar possível a utilização do nó `W`, que representa o programa principal (o escopo das variáveis globais). O `foreach` (linhas 03-12) iterara sobre os nós que seguem o formato determinado pela definição de nó `VarDecl("string", "x")`, que significa todas as variáveis declaradas com qualquer tipo e `x` como nome. Note que dentro do escopo do `foreach`, cada nó pode ser acessado pela variável `X`.

O comando `IF` faz uso da expressão `exist` (linhas 04-11) precedido pela expressão de negação (`not`). Neste caso, ele verifica se não existe uma declaração de variável de qualquer tipo que tenha `title` como nome no escopo `W`. Observe que a variável `ANY`, usada como primeiro argumento na definição de nó, é uma variável não declarada. Na linha 06, é verificado se o ambiente associado ao nó `X` (ocorrência de ligação) e o ambiente recebido pelo nó `Y` (ocorrência aplicada) representam o mesmo nó. Se verdadeiro, a reescrita de termos é executada. Caso contrário, nada acontece. O comando de reescrita de termos (linha 07) transforma todo o uso de `x` em um uso de `title`. Ele também é executado

na linha 10, onde transforma a declaração de variável de nome `x` e tipo `String` em uma declaração de variável de tipo `String` e nome `title`.

3.7 CONCLUSÕES

Sabemos que a maioria dos sistemas de transformações genéricos possuem uma sintaxe complexa e distinta da linguagem de programação em que se está trabalhando. Além disso, os sistemas genéricos, por trabalhar com diversas linguagens de programação, a semântica dos programas normalmente não é conhecida, impossibilitando assim que o desenvolvedor realize validações semânticas em sua transformação, o que dificulta a escrita das transformações, e pode resultar em erros.

Neste capítulo apresentamos uma linguagem de transformação imperativa que dispõe de operações sobre a árvore sintática de um programa para realização de buscas, casamento de padrões, reescrita de termos, etc. As suas principais características foram classificadas em: tipos, expressões, comandos e declarações. Além de possuir uma sintaxe simples, FxTL possui construções que fornecem acesso as informações semânticas de um programa, podendo ser usadas para realizar validações durante a transformação. Essas informações facilitam a tarefa do desenvolvedor na hora de escrever suas transformações, evitando assim, que programas semanticamente incorretos sejam gerados.

Portanto, temos uma linguagem de transformação simples, mas com poderosas construções para expressar transformações. Além de fazer uso das definições sintáticas de uma linguagem, FxTL também provê suporte semântico ao desenvolvedor, garantindo assim facilidade e segurança na escrita de suas transformações.

CAPÍTULO 4

A IMPLEMENTAÇÃO DE FXTL

Este capítulo apresenta uma visão geral do sistema de transformação definido. Nele, serão discutidas as decisões arquiteturais, as ferramentas de apoio utilizadas, entre outras. A linguagem de programação adotada para o seu desenvolvimento foi Java, por ser robusta, bastante difundida no mercado e gratuita. Para o gerador de parser, escolhemos o `javacc` [Kod04] [Col07] por possuir um gerador automático de árvore sintática e aceitar construções léxicas e sintáticas em um mesmo arquivo. Além disso, é bastante difundido no mercado e possui plugin para o ambiente de desenvolvimento que utilizamos (Eclipse [Ecl07]).

Nosso trabalho está inserido dentro de um contexto maior para a construção de um sistema de transformação genérico. Basicamente, somos responsáveis pelo desenvolvimento do engenho de transformação. Este é formando por dois interpretadores: um para a linguagem de programação utilizada e outro para a linguagem de transformação definida. Entraremos em detalhes sobre cada componente do sistema de transformação nas próximas seções.

4.1 VISÃO GERAL

O sistema de transformação definido é capaz de realizar transformações em qualquer programa, independentemente da linguagem em que este foi escrito, desde que sejam fornecidas as definições sintáticas da linguagem utilizada. Na Figura 4.1 podemos visualizar uma representação de todo o processo de transformação.

O sistema de transformação recebe como entrada um programa, as definições sintáticas da linguagem em que o programa foi escrito e as transformações codificadas em FxTL. A primeira etapa é gerar a árvore sintática abstrata do programa a partir do analisador da linguagem fonte para que, em seguida, o analisador semântico entre em vigor, e enriqueça a árvore com informações semânticas. Após essa fase de análise, o engenho de transformação executa as transformações escritas em FxTL, gerando como resultado uma árvore atualizada. Após cada operação de reescrita, o analisador semântico é executado para validar e gerar as novas informações semânticas do programa. Para podermos obter um texto legível do programa transformado, executamos o pretty printer sobre a árvore transformada. Se ocorrer algum erro ao longo da execução do engenho de transformação, a execução do sistema é abortada e a árvore do programa retorna ao seu estado inicial.

4.2 ENGENHO DE TRANSFORMAÇÃO

O engenho de FxTL é formado por dois interpretadores: o primeiro para reconhecimento das definições sintáticas da linguagem; e o segundo para validação das construções de FxTL. Além de executar as transformações, o engenho de transformação também realiza

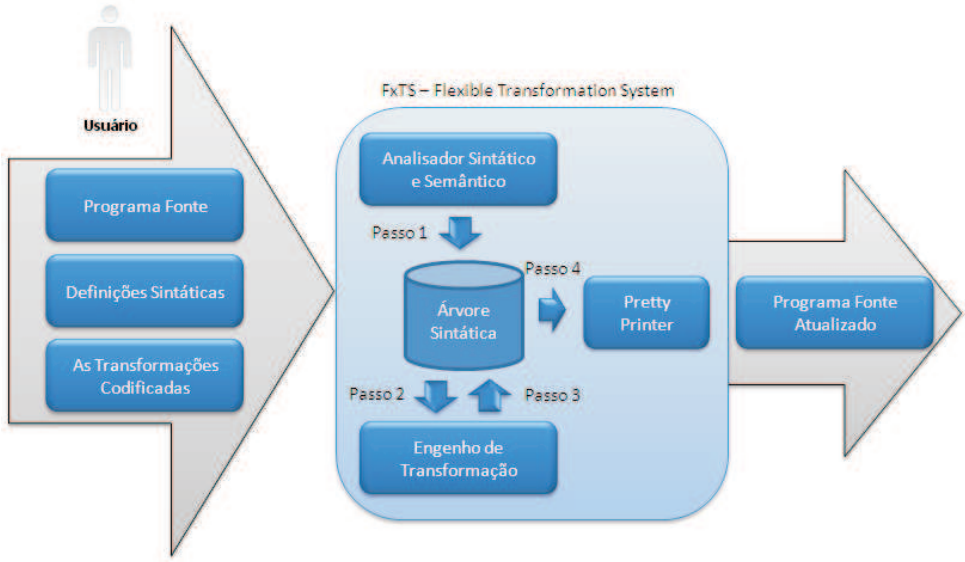

Figura 4.1 Visão Geral do Sistema de Transformação.

validações sintáticas sobre o programa. Note que ambos os interpretadores utilizam analisadores que foram gerados a partir do JavaCC utilizando as configurações descritas abaixo:

- `IGNORE_CASE=true` - O case dos tokens processados é ignorado. Ou seja, é indiferente para o analisador se o token está em maiusculo ou minusculo.
- `JDK_VERSION="1.5"` - A versão do JDK utilizada é a 1.5.
- `MULTI=true` - Os nós gerados pelo analisador herdam de um nó base.
- `NODE_EXTENDS="BaseNode"` - Disponibilizando uma implementação base, as alterações desejadas não precisarão ser realizadas a cada compilação. Todos os nós herdaram de "BaseNode".
- `VISITOR=true` - O padrão visitor será aplicado em todos os nós gerados. Ou seja, cada nó terá um método `jjtAccept()` que formalizará a visita.
- `VISITOR_EXCEPTION="Exception"` - Define a exceção que pode ser gerada durante uma visita.
- `STATIC=false` - Define que o analisador gerado será estático.

Portanto, os analisadores usado pelos interpretadores são estáticos, utilizam o padrão visitor, e herdam do nó "BaseNode". Além disso, sabemos que exceções podem ser geradas durante uma visita, que os tokens não diferenciam maiúsculo e minúsculo, e que o JDK utilizado é o 1.5.

A figura 4.2 mostrada abaixo será utilizada nas seções posteriores para compreendermos melhor a responsabilidade de cada interpretador.

```
TkId = \\p{Alpha}[\\w]*;
TkInteger = -?[0-9]+;
TkString = [\\p{ASCII}]*;

<Program> ::= ImpProgramDecl(<VarDecl>*,<FuncDecl>*,<ProcDecl>*,<Stmt>*)
<Exp> ::= IdUsage(TkId) | IntConst(TkInteger) | StringConst(TkString) | Sum(<Exp>,<Exp>)
<Stmt> ::= Assign(<Exp>,<Exp>) |WriteLn(<Exp>)
<VarDecl> ::= VarDecl (TkId,TkId)
<FuncDecl> ::= FuncDecl(TkId,TkId,<VarDecl>*,<VarDecl>*,<Stmt>*)
<ProcDecl> ::= ProcDecl(TkId,TkId,<VarDecl>*,<VarDecl>*,<Stmt>*)
```

Figura 4.2 Sintaxe abstrata de SubPascal.

4.2.1 Interpretador para as definições sintáticas

As construções sintáticas da linguagem são formadas por tokens, classes sintáticas e termos. Essas definições seguem o formato BNF. Os tokens são definidos através de expressões regulares da linguagem de programação Java. Para validarmos se um valor é válido, precisamos verificar se a expressão regular que define este token reconhece o valor utilizado. Portanto, se utilizarmos um valor que não é reconhecido pela expressão regular, um erro será gerado e a transformação será abortada. Na Figura 4.2, `TkId`, `TkInteger` e `TkString` são exemplos de tokens, e `\\p{Alpha}[\\w]*`, `-?[0-9]+` e `[\\p{ASCII}]*` são exemplos de expressões regulares.

Os termos representam operações da linguagem, como por exemplo, expressões e comandos. Portanto, só poderá existir em uma transformação operações que foram definidas sintaticamente. Se uma operação não definida for encontrada, um erro será gerado e a transformação será abortada. Na Figura 4.2, `IdUsage(TkId)` é um exemplo de termo.

Por último, as classes sintáticas. Elas são formadas por um conjunto de termos. Em uma operação de reescrita, um nó só poderá ser substituído por outro de uma mesma classe sintática. Caso contrário, um erro será gerado e a transformação será abortada. Portanto, não é possível transformar um nó de uma classe sintática B em um nó de uma classe sintática C. Na Figura 4.2, `<Program>` representa uma classe sintática.

4.2.2 Interpretador para a linguagem de transformação

Após a interpretação de toda a definição sintática de uma linguagem, é possível iniciar a interpretação dos comandos responsáveis por transformar um programa. Este interpretador reconhece toda a sintaxe de FxTL, e portanto, comandos mal formados são detectados durante sua execução. Se qualquer erro sintático for detectado, a transformação será abortada.

Para uma análise completa da sintaxe e semântica de FxTL, veja o apêndice A e o apêndice B.

4.3 ARQUITETURA

A Figura 4.3 demonstra os pacotes existentes no sistema. Em `javacc` encontra-se tanto o arquivo de definição léxica e sintática para o `JavaCC`, como também as classes que são geradas automaticamente pelo seu compilador. Já em `core`, encontra-se o processador e o transformador da linguagem, como também o validador sintático e semântico. Em `exceptions` estão todas as exceções personalizadas do sistema. Temos também `facts`, onde se encontram as classes que definem o modelo para a árvore abstrata de um programa. o como o próprio nome diz, encontram-se todas as exceções personalizadas do sistema. Em `util` estão os arquivos que são utilizados por diversas partes do sistema, como por exemplo, as constantes. Por último, `gui`, onde se encontra a implementação da interface gráfica do sistema.

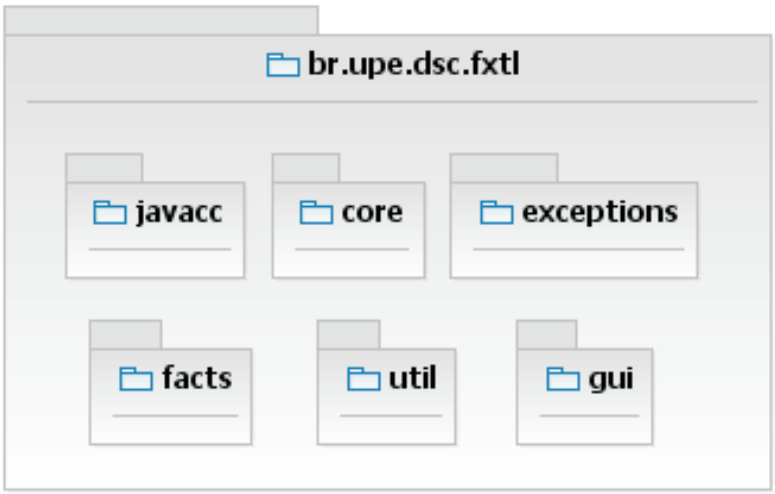

Figura 4.3 Diagrama de Pacotes.

Dentre as classes que compõem o sistema, as mais importantes estão relacionadas na Figura 4.4 demonstrando como ocorre o relacionamento entre elas. A classe Processor age sobre a árvore abstrata do programa, e ao identificar algum tipo de transformação, repassa esta atividade para a classe Transformer, que realiza a transformação. Após a transformação ser realizada, a classe Validator é chamada com a finalidade de verificar se a transformação ocorrida está sintaticamente correta. Caso não seja, uma exceção é gerada e transformação é abortada.

Para o melhor entendimento da operação de reescrita, vide a Figura 10. A função `transform` da classe `Transformer` é responsável em realizar a operação base de reescrita. A partir da figura, é possível visualizar a criação de um fato que recebe como argumentos o identificador sintático e uma lista de filhos (linha 15). Logo em seguida, ocorre a validação sintática, onde o validador recebe como parâmetro o novo e o antigo fato (linha 16). Note que erros sintáticos podem ocorrer, como por exemplo, a classe sintática do novo fato é diferente da classe sintática do antigo fato, e portanto, uma `ValidationException` será gerada. Caso nenhum problema seja encontrado durante a validação, o antigo fato será

substituído pelo novo fato (linha 17). Lembre-se que existem extensões para as operações de reescrita, e portanto, demonstramos aqui apenas um caso.

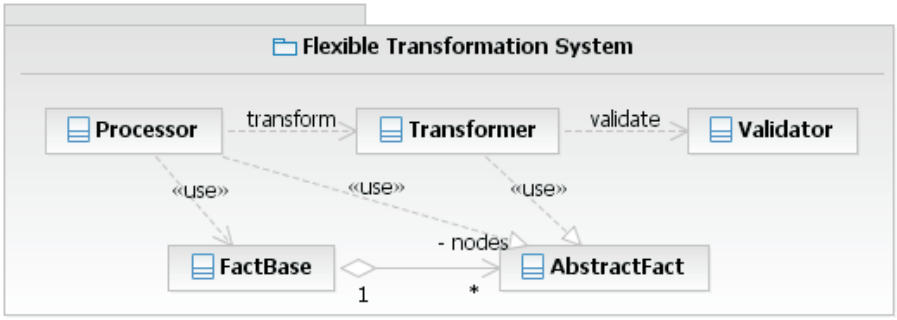

Figura 4.4 Diagrama de Classes.

```
...
12  public  static void transform( AbstractFact listFact, String name,
13   Object[] childs) throws  ValidationException {
14
15       Fact newFact  = new  Fact(name, childs);
16       FxTLSyntaxValidator.validate(newFact ,  listFact.getFact());
17       listFact.setFact(newFact);
18   }
...
```

Figura 4.5 Fragmento de código da classe FxTLTransform.

Para o funcionamento correto da transformação, a definição do modelo que representa a árvore abstrata de um programa é uma atividade de extrema importância, pois a árvore será gerada a partir de um analisador desenvolvido por terceiros, e, portanto, precisa ser compatível com o modelo definido. A Figura 4.6 representa o modelo definido.

Um nó é representado pela classe `Fact`. Porém, este nó pode ser único ou estar contido em uma lista de nós, representado pelas classes `UniqueFact` e `ListFact` respectivamente. É a partir deste modelo que podemos distinguir o que representa uma lista de nós, que podem ser percorridas ou um único nó, que pode ser alvo de uma transformação. Para simplificar o entendimento, podemos exemplificar com Pascal, o comando `return` possui apenas um parâmetro, que neste caso seria representado pelo `UniqueFact`. Já uma declaração de função possui uma lista de parâmetros, uma lista de variáveis locais, e uma lista de comandos. Cada uma dessas listas, seriam representadas pelo `ListFact`.

A Figura 4.7 demonstra a representação de um programa na memória. Apesar de não estar presente na figura, sabe-se que todos os nós estão decorados com suas respectivas informações (sintáticas e semânticas). A árvore apresentada é referente ao código fonte do programa da Figura 4.8.

4.4 CONCLUSÕES

Neste capítulo apresentamos o engenho de transformação que foi desenvolvido neste trabalho. Esse engenho faz parte de um esforço maior para a construção de um sistema

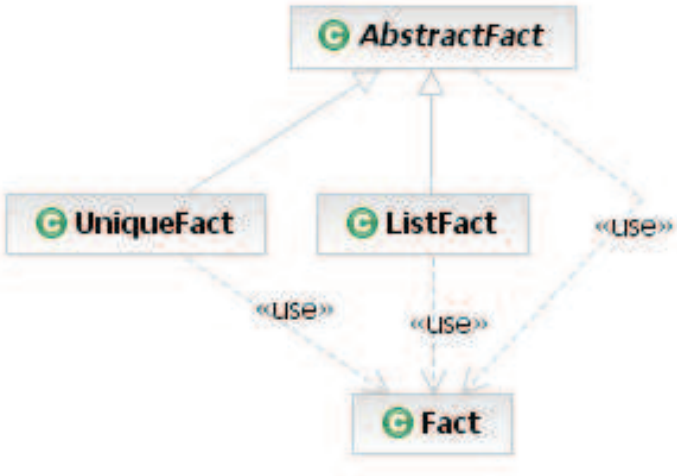

Figura 4.6 Modelo para Árvore Abstrata.

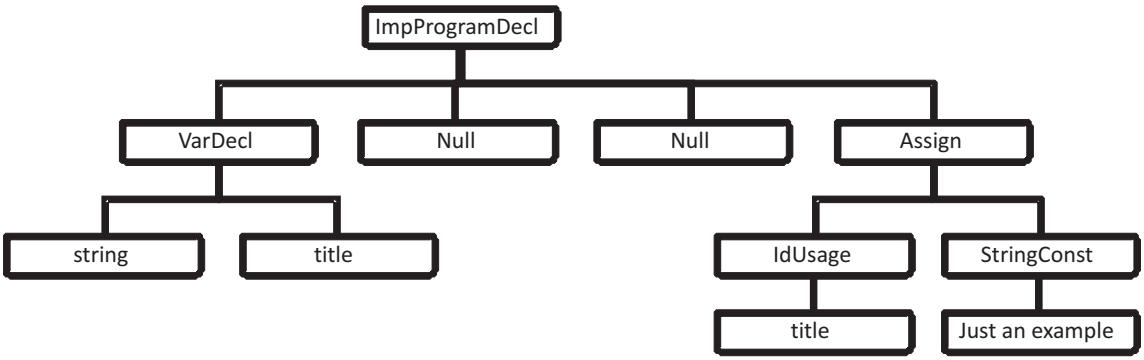

Figura 4.7 Representação de um programa na memória.

de transformação genérico. Apresentamos também as decisões tomadas para a sua construção, como linguagem de programação adotada e gerador de parser escolhido.

O engenho de transformação é formado por dois interpretadores, onde o primeiro é responsável por reconhecer todas as construções sintáticas da linguagem de programação utilizada, sabendo reconhecer o que são construções válidas através de seu formato. O segundo, é responsável por executar propriamente as transformações. Este conhece a sintaxe e semântica da linguagem FxTL.

Enfim, o objetivo deste capítulo foi aprofundar o leitor no sistema de transformação definido, apresentando além de características da implementação, definições arquiteturais, podendo assim, o leitor ter uma ideal real dos componentes existentes. Além disso, demonstra que o resultado obtido não tem valor apenas teórico, visto que temos uma ferramenta desenvolvida.

```
01 Program temp;
02   var title : String;
03
04   begin
05      title  := 'Just an example';
06   end.
```

Figura 4.8 Código fonte do programa.

CAPÍTULO 5

PROVA DE CONCEITO: TRANSFORMAÇÕES EM PROGRAMAS JAVA

Este capítulo apresenta as provas de conceito executados com o intuito de validar e demonstrar a linguagem FxTL. Descreveremos em detalhe a aplicação de algumas leis de refinamento [Cor04b], como também refatorações de programas escritos em um subconjunto da linguagem de programação Java, chamada de SubJava.

A primeira atividade necessária é a definição da gramática da linguagem em que se esta trabalhando. No nosso caso, SubJava. Esta gramática precisou ser definida usando a notação BNF. Com essas definições, o transformador consegue validar as classes sintáticas utilizadas, como também o formato dos tokens, entre outras coisas. Após a apresentação da gramática, começaremos com o estudo em detalhe de cada transformação realizada.

5.1 DEFINIÇÃO DA GRAMÁTICA

O engenho de transformação de FxTL precisa receber como entrada a gramática da linguagem que você está trabalhando antes da aplicação de qualquer transformação. Portanto, a primeira atividade é definir a gramática da linguagem. Na Figura 5.1, apresentamos a gramática de Subjava.

Consideramos um programa SubJava como um conjunto de pacotes representado pela função de nó `OOPProgramDecl(<PackageDecl>*)`, onde `<PackageDecl>*` representa um conjunto de zero ou n pacotes. Um pacote possui nome, lista de importações representada por `<ImportDecl>*` e uma declaração de tipo `<TypeDecl>`. Uma declaração de tipo pode ser, por exemplo, uma classe, ou uma interface. Veremos na próxima seção a aplicação de transformações, como por exemplo, leis de refinamento em programas escritos em Java.

5.2 TRANSFORMAÇÃO 1: REMOÇÃO DE ATRIBUIÇÃO DESNECESSÁRIA

A atribuição de uma expressão a ela mesma não altera o estado de um programa. Portanto, a remoção desta atribuição resultará em um programa equivalente. Figura 5.2 apresenta o código original na caixa à esquerda e o código transformado na caixa à direita.

Na Figura 5.3, nós apresentamos o código escrito em FxTL seguindo o formato das definições sintáticas demonstrado na Figura 5.1. O código procura por todas as declarações de métodos e dentre cada método encontrado, itera sobre seus comandos. Se um desses comandos for uma operação de atribuição (`Assign`), e a expressão do lado esquerdo for igual à operação do lado direito, este comando será removido da lista de comandos. Lembrando que a operação de igualdade verifica todas as informações presente no nó (escopo, tipo, etc.).

```
TkId = "\\p{Alpha}[\\w]*";
TkInteger = "-?[0-9]+";
TkString = "[\\p{ASCII}]*";

<Program> ::= OOPProgramDecl(<PackageDecl>*)
<PackageDecl> ::= PackageDecl(TkString, <ImportDecl>*, <TypeDecl>)
<ImportDecl> ::= ImportDecl(TkString)
<TypeDecl> ::= ClassDecl(TkString, TkString, <ExtendsDecl>*, <ImplementsDecl>*,
<StaticInitBlock>*, <TypeDecl>*, <ConstructorDecl>*, <VarDecl>*, <MethodDecl>*)
                    | InterfaceDecl (TkString, <ExtendsDecl>*, <InterfaceMethodDecl>*)
<ExtendsDecl> ::= ExtendsDecl(TkString)
<ImplementsDecl> ::= ImplementsDecl(TkString)
<InterfaceMethodDecl> ::= InterfaceMethodDecl(TkId,TkId,TkId,<ParamDecl>*,<ThrowsDecl>*)
<StaticInitBlock> ::= StaticInitBlock(<Stmt>*)
<ConstructorDecl> ::= ConstructorDecl(TkId,TkId,<ParamDecl>*,<ThrowsDecl>*,<Stmt>*)
<MethodDecl> ::= MethodDecl(TkId,TkId,TkId,<ParamDecl>*,<ThrowsDecl>*,<Stmt>*)
<ThrowsDecl> ::= ThrowsDecl(TkId)
<ParamDecl> ::= ParamDecl(TkId,TkId)
<VarDecl> ::= VarDecl(TkId,TkId,TkId) | VarDeclInit(TkId,TkId,TkId,<Exp>)
<Exp> ::= IdUsage(TkId) | IntConst(TkInteger) | StringConst(TkString) | Sum(<Exp>,<Exp>)
<Stmt> ::= Assign(<Exp>,<Exp>) | StmtVarDecl(TkId,TkId) | StmtVarDeclInit(TkId,TkId,<Exp>)
                    | Return (<Exp>) | BlockStmt(<Stmt>*)
```

Figura 5.1 Sintaxe abstrata de SubJava.

5.3 TRANSFORMAÇÃO 2: REMOÇÃO DE VARIÁVEL DESNECESSÁRIA

Se um método possui como retorno uma variável local que é inicializada, porém, não é modificada ao longo de sua execução. Esta variável pode ser removida, e o retorno do método passa a ser diretamente a expressão que inicializava a variável. Esta transformação resultará em um programa equivalente. Figura 5.4 apresenta na caixa a esquerda o programa retornando uma variável local inicializada e na caixa direita o programa retornando diretamente a expressão.

Na Figura 5.5, apresentamos o código da transformação. Ele procura por todas as declarações de métodos e dentre cada método encontrado, itera sobre seus comandos. Se um desses comandos for uma declaração com inicialização, ele verifica se existe um retorno de método com essa variável. Se existir, verifica se o escopo dessa variável usada é realmente referente à variável local, e se verdadeiro, o retorno do método passa a ser a expressão diretamente, e logo em seguida, remove a declaração dessa variável local.

5.4 TRANSFORMAÇÃO 3: REMOÇÃO DE VARIÁVEL LOCAL NÃO USADA

Se uma variável local é declarada, porém não é utilizada em nenhum ponto do programa, esta variável pode ser removida sem alterar o estado do programa. Portanto, o resultado será um programa equivalente. Figura 5.6 apresenta na caixa esquerda um programa com uma declaração de variável local que não é utilizada e na caixa direita um programa equivalente sem a variável local.

Na Figura 5.7, apresentamos o código para a remoção de variáveis locais não usadas.

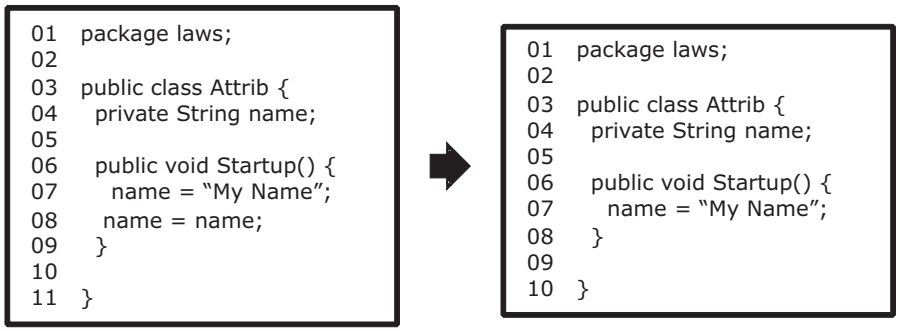

Figura 5.2 Aplicação da remoção de atribuição desnecessária.

```
01 Start {
02  foreach (X in MethodDecl(Modifiers,T1,Name,Extends,Implements,Stmts)) {
03   foreach (Y in Stmts) {
04    if (Y is Assign(A, B)) {
05     if A == B {
06      Stmts => Stmts - Y;
07     };
08    };
09   };
10  };
11 }
```

Figura 5.3 Código para remoção de atribuição desnecessária.

Primeiramente ele procura por todas as declarações de métodos e dentre cada método encontrado, itera sobre seus comandos. Se um desses comandos for uma declaração, logo em seguida, é verificado se não existe uso dessa variável. Se não existir, a declaração é removida. Observe que caso exista algum uso dessa variável, nada acontece com a mesma.

5.5 TRANSFORMAÇÃO 4: ALTERAÇÃO DO NOME DE UMA VARIÁVEL

Na alteração do nome de uma variável deve-se ter cuidado com a consistência do programa, visto que todas as ocorrências dessa variável também precisam ser alteradas. Note que se existir ocorrências de outras variáveis de mesmo nome e diferente escopo, estas não deverão ser modificadas. Esta transformação produz um programa equivalente como resultado. Figura 5.8 apresenta na caixa esquerda um programa com uma declaração de variável global `x` e na caixa direita o programa teve sua variável global renomeada de `x` para `name`.

Na Figura 5.9, apresentamos o código para a alteração do nome de uma variável global. Primeiramente ele procura por todas as declarações de variáveis globais que possuem `x` como nome com qualquer tipo e modificador. Dentre as declarações encontradas, procura por todas as ocorrências das variáveis que possuem `x` como nome, e sobre cada ocorrência encontrada, verifica se o escopo é referente ao da variável global que está sendo iterada (`X`). Se sim, verifica se não existe uma declaração de variável com nome `name` no mesmo

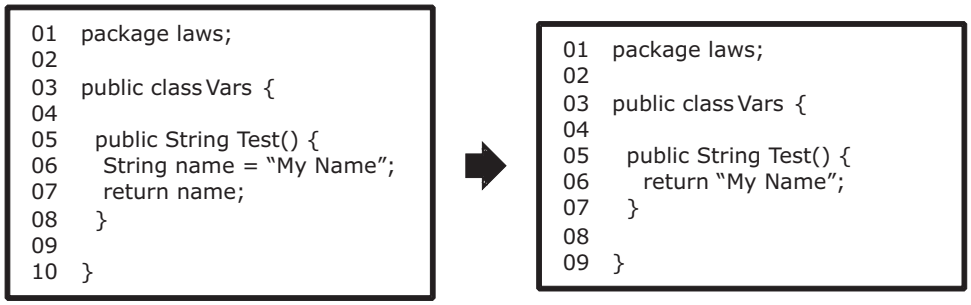

Figura 5.4 Aplicação da remoção de variável desnecessária.

```
01 Start {
02  foreach (X in MethodDecl(Modifiers,T1,Name,Extends,Implements,Stmts)) {
03   foreach (Y in Stmts) {
04    if (Y is StmtVarDeclInit(T2, N, Exp1)) {
05     foreach(Z in Stmts) {
06      if (Z is Return (Exp2)) {
07       if (Exp2 is IdUsage(N) and Scope(Exp2) == Scope(Y)) {
08        Z => Return(Exp1);
09        Stmts => Stmts - Y;
10       };
11      };
12     };
13    };
14   };
15  };
16 }
```

Figura 5.5 Código para remoção de variável desnecessária.

escopo da variável global que está sendo iterada. Se não existir, troca para `name` as ocorrências da variável `x`. E por último, altera o nome da variável global de `x` para `name`. Em resumo, teremos a alteração da variável global e todas as suas ocorrências de `x` para `name`.

5.6 TRANSFORMAÇÃO 5: REMOÇÃO DO COMANDO IF COM EXPRESSÃO FALSA

Um comando `IF` sem a cláusula `ELSE` e que possui como expressão uma constante `false` nunca terá seus comandos executados visto que os comandos são executados apenas quando o resultado da avaliação é verdadeiro. Portanto, o comando `IF` pode ser removido sem nenhuma alteração no estado do programa. Esta transformação resultará em um programa equivalente. Figura 5.10 apresenta na caixa esquerda o programa com uma expressão condicional falsa e na caixa direita o programa após a remoção da expressão condicional.

Na Figura 5.11, apresentamos o código para a remoção do comando `IF` com expressão falsa. Primeiramente ele procura por todas as declarações de métodos. Dentre as

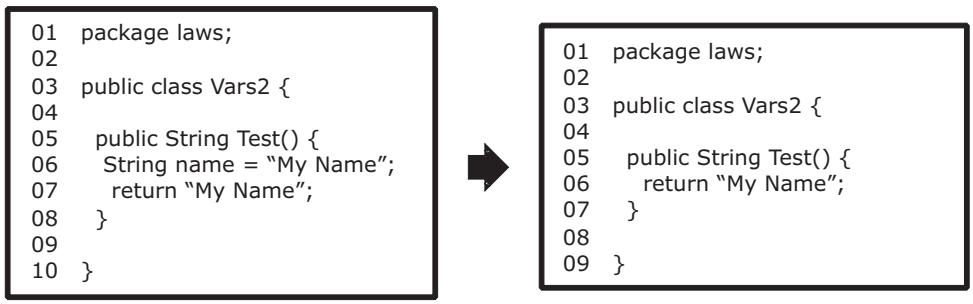

Figura 5.6 Aplicação da remoção de variável local não usada.

```
01 Start {
02  foreach(X in MethodDecl(Modifiers,T1,Name,Extends,Implements,Stmts)) {
03   foreach(Y in Stmts) {
04    if (Y is StmtVarDecl(T2, N)) {
05     if (not exist(IdUsage(N)) in X) {
06      Stmts => Stmts - Y;
07     };
08    } else {
09     if (Y is StmtVarDeclInit(T2, N, Exp1)) {
10      if (not exist(IdUsage(N)) in X) {
11       Stmts => Stmts - Y;
12      };
13     };
14    };
15   };
16  };
17 }
```

Figura 5.7 Código para remoção de variável local não usada.

declarações encontradas, itera sobre todos os seus comandos verificando se existe algum comando `IF` que possui como expressão a constante `false`, ou seja, com uma condição que nunca será satisfeita. Caso encontre, este comando será removido da lista de comandos do método iterado. Em resumo, teremos como resultado um código mais limpo visto que comandos `IF` com expressões falsas serão removidos.

5.7 TRANSFORMAÇÃO 6: REMOÇÃO DO COMANDO IF COM EXPRESSÃO VERDADEIRA

Um comando `IF` que possui como expressão uma constante `true` sempre terá seus comandos executados visto que o resultado da avaliação condicional sempre será verdadeiro. Portanto, podemos retirar os comandos do bloco condicional visto que eles sempre serão executados, e logo em seguida, remover o comando `IF` sem nenhuma alteração no estado do programa. Esta transformação resultará em um programa equivalente. Figura 5.12 apresenta na caixa esquerda o programa com uma expressão condicional verdadeira e na caixa direita o programa após a transformação.

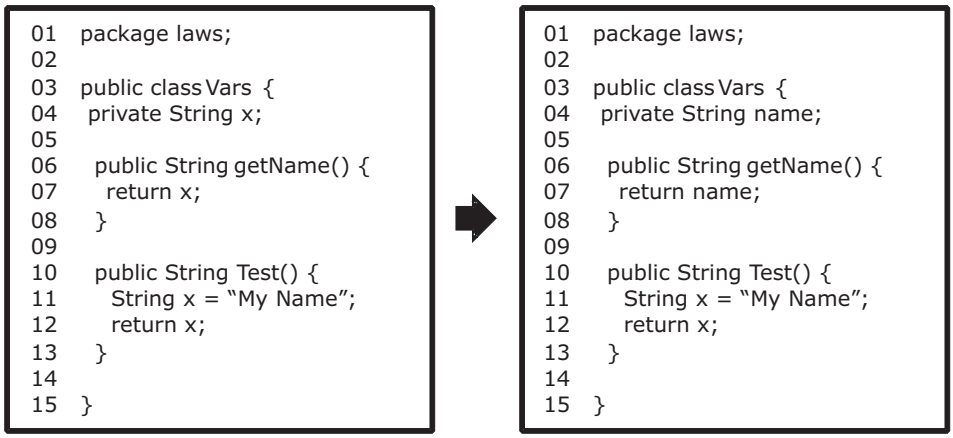

Figura 5.8 Aplicação do alterar nome de variável.

```
01 Start {
02   foreach(X in VarDecl(Modifiers,T1,"x")) {
03     foreach(Y in IdUsage(Name)) {
04       if (Scope (Y) ==Scope(X)) {
05         S = Scope(X);
06         if (not exist(VarDecl(ANYModifier, ANYType, "name")) in S) {
07           Y => IdUsage("name");
08         };
09       };
10     };
11     X => VarDecl(Modifiers,T1,"name");
12   };
13 }
```

Figura 5.9 Código para alteração do nome de uma variável.

Na Figura 5.13, apresentamos o código responsável em retirar do comando `IF` os comandos que sempre seriam executados, visto que a expressão condicional sempre será verdadeira. Primeiramente ele procura por todas as declarações de métodos. Dentre as declarações encontradas, itera sobre todos os seus comandos verificando se existe algum comando `IF` que possui como expressão a constante `true`, ou seja, com uma condição que sempre será satisfeita. Caso encontre, a variável `IfStmts` apontará para os comandos que sempre seriam executados. Logo em seguida, a variável `Y`, que refere-se ao comando `IF`, será transformada em um bloco de comandos que conterá os comandos apontado por `IfStmts`. Em resumo, teremos como resultado a remoção de um bloco condicional desnecessário, visto que sempre os comandos internos a ele seriam executados.

5.8 CONCLUSÕES

Neste capítulo apresentamos transformações que serviram como prova de conceito da linguagem de transformação definida.

Algumas das transformações escritas neste capítulo foram escritas também para os

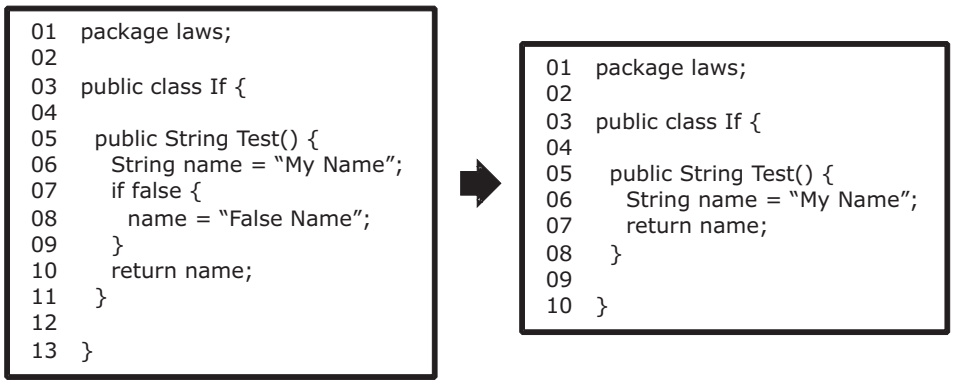

Figura 5.10 Aplicação do remover comando IF com expressão falsa.

```
01 Start {
02  foreach (X in MethodDecl(Modifiers,T1,Name,Extends,Implements,Stmts)) {
03   foreach (Y in Stmts) {
04    if (Y is IfCondition(FalseConst(), IfStmts)) {
05     Stmts => Stmts - Y;
06    };
07   };
08  };
09 }
```

Figura 5.11 Código para remoção de um comando IF com expressão falsa.

sistemas de transformação estudados. Apesar de FxTL não ser tão simples como Jackpot, que é uma linguagem de transformação específica para Java, ainda assim FxTL pode ser considerada uma linguagem de transformação simples. Se compararmos com TXL e ATL, as transformações foram escritas em um menor número de linhas e gastando um tempo consideravelmente menor. Além disso, FxTL provê suporte semântico ao desenvolvedor, o que nenhuma das ferramentas estudadas fornecem.

Podemos afirmar que FxTL não é mais uma linguagem que fornece apoio ao desenvolvedor na hora de realizar transformações de programas. FxTL é uma linguagem que foi definida buscando simplicidade e similaridade com o paradigma imperativo, ou seja, desenvolvedores habituados com linguagens a partir do paradigma imperativo não terão dificuldades em seu entendimento. Além disso, sabemos que os sistemas de transformação genéricos em sua maioria não reconhecem as informações semânticas de um programa. Portanto, tivemos como maior motivação, prover um mecanismo para fornecer acesso às informações semânticas de um programa durante uma transformação.

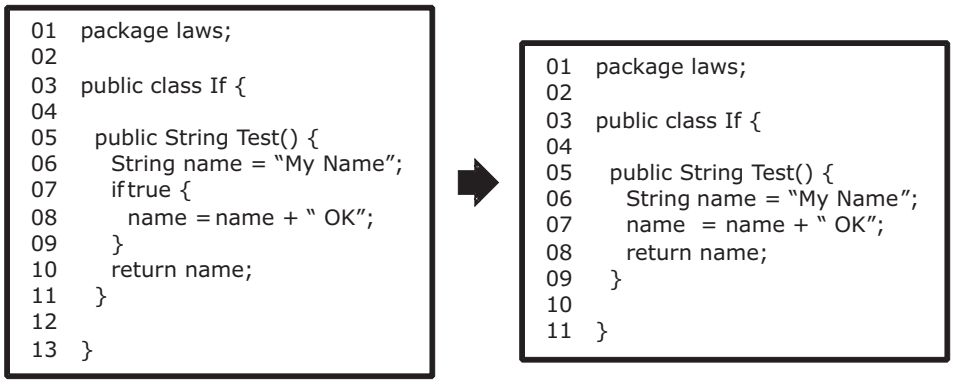

Figura 5.12 Aplicação do remover comando IF com expressão verdadeira.

```
01 Start {
02   foreach (X in MethodDecl(Modifiers,T1,Name,Extends,Implements,Stmts)) {
03     foreach (Y in Stmts) {
04       if (Y is IfCondition(TrueConst(), IfStmts)) {
05         Y => BlockStmt(IfStmts);
06       };
07     };
08   };
09 }
```

Figura 5.13 Código para remoção de um comando IF com expressão verdadeira.

CAPÍTULO 6

CONCLUSÕES

Sabemos que reestruturação de programas é uma atividade inerente para o desenvolvimento de software e que se realizada de forma manual, erros poderão ser introduzidos. Por isso, necessita-se de ferramentas para automatizar este processo. Porém, as ferramentas de desenvolvimento tradicionais possuem um conjunto restrito de transformações, e personalizações são muito difíceis de serem realizadas visto que utilizam APIs proprietárias e complexas. Por isso, a utilização de um sistema de transformação é de fundamental importância em qualquer projeto, visto que qualquer usuário poderá escrever suas próprias transformações ou inclusive personalizar algumas das já existentes.

Observamos que os sistemas de transformações podem ser classificados de duas formas, de acordo com seu domínio de trabalho: os genéricos, possíveis de serem aplicados a qualquer programa, independentemente da linguagem em que o programa foi escrito (desde que sua sintaxe possa ser descrita em BNF); e os específicos, que só podem ser aplicados a programas escritos em uma linguagem específica. O problema é que a maioria dos sistemas de transformação genéricos requer um profissional com conhecimento aprofundado, visto que suas linguagens, em geral, possuem uma sintaxe substancialmente diferente das linguagens de programação. Além disso, é necessário garantir que as modificações realizadas não produzam alterações indesejadas na semântica do programa. Como os sistemas genéricos em sua maioria não provêem acesso às informações semânticas de um programa, um programa semanticamente incorreto pode ser gerado por uma transformação. Essas formam as principais motivações levadas em consideração na definição da linguagem de transformação proposta nesta dissertação. O paradigma adotado foi o imperativo, visto que este é bastante difundido entre os desenvolvedores (vide C, Pascal), e foi criado um mecanismo de suporte às informações semânticas de um programa, tornando possível a realização de validações, como por exemplo, de tipo e escopo.

Note que através de FxTL, é possível realizar transformações em qualquer programa escrito, desde que a linguagem de programação utilizada possa ser descrita em BNF. Como a maioria das linguagens de programação existente existentes podem ser descritas através da notação BNF, o campo de aplicação de FxTL é bastante amplo. Entretanto, essas definições precisam ser fornecidas ao interpretador para que as transformações possam ser executadas. Como ainda não existem definições completas disponíveis no mercado, as primeiras definições precisam ser escritas integralmente para posteriormente, serem publicadas e reutilizadas pelos usuários. Esta atividade será comentada na seção de trabalhos futuros.

Portanto, temos uma linguagem de transformação genérica com semântica imperativa, que faz uso de construções semânticas para realizar transformações sem introduzir erros semânticos. Por ser genérica, existe uma única sintaxe para realizar transformações em programas, independentemente da linguagem em que se está trabalhando. Com isso,

o custo para realização de transformações em uma empresa deve cair, considerando que uma empresa trabalha com diversas linguagens de programação. Este trabalho também produziu uma implementação de um sistema de transformação genérico testado em programas escritos em diferentes linguagens de programação. Nesta dissertação, as linguagens de programação utilizadas foram `Pascal` e `Java`.

É importante lembrar que este trabalho faz parte de um esforço maior para a construção de um sistema de transformação genérico, chamado de FxTS - Flexible Transformation System. Concluímos aqui, a definição da linguagem de transformação, chamada de FxTL - Flexible Transformation Language, e a implementação do engenho de transformação responsável em aplicar e validar as transformações escritas.

6.1 TRABALHOS RELACIONADOS

No capítulo 2, apresentamos em detalhe três sistemas de transformações juntamente com suas características. Nesta seção, abordaremos suas vantagens e desvantagens em comparação com a linguagem de transformação definida neste trabalho.

6.1.1 Jackpot

Sabemos que Jackpot é um sistema de transformação projetado para realizar as transformações mais comuns de forma simples. Existe um preço a ser pago por essa simplicidade, e por isso, transformações complexas são difíceis de serem escritas, e dependendo do caso, até impossível, se, por exemplo, construções para controle de fluxo forem necessárias.

É importante destacar que qualquer usuário, até mesmo aqueles que não possuem experiências com sistemas de transformações, conseguem escrever código em Jackpot. Essa é uma de suas principais vantagens, porém, a semântica da linguagem de programação alvo não é levada em consideração durante suas transformações. Ou seja, uma das principais vantagens dos sistemas de transformações genéricos não existe em Jackpot.

O ambiente de desenvolvimento utilizado por Jackpot ajuda bastante o usuário na hora de escrever transformações. Palavras chaves são destacadas, o usuário consegue visualizar o programa resultante após a transformação, entre outras coisas, que terminam aumentando a produtividade do desenvolvedor. Atualmente, FxTL não possui um ambiente de desenvolvimento, e portanto, as suas transformações podem ser escritas em qualquer editor de texto.

A simplicidade existente em FxTL foi inspirada em parte por Jackpot, onde procuramos facilitar a escrita de transformações simples, mas sem extinguir construções importantes para o desenvolvimento de transformações complexas. Além disso, FxTL é uma linguagem de transformação genérica que recebe como entrada construções semânticas da linguagem alvo, podendo assim realizar validações semânticas durante as transformações.

6.1.2 TXL

TXL é uma linguagem de transformação funcional, com mais de 15 anos de pesquisa, que se baseia em regras para o desenvolvimento rápido de soluções complexas. É genérica, ou seja, desde que as definições sintáticas da que se deseja trabalhar sejam definidas,

é possível realizar qualquer tipo de transformação. Além disso, existe um repositório de gramáticas, onde você encontra definições para as linguagens de programação mais comuns no mercado.

Em geral, TXL é uma boa escolha para a realização de transformações, porém, exige que o desenvolvedor tenha experiência com o paradigma funcional (paradigma que não é comum na indústria). Isso motivou a escolha do paradigma imperativo em nosso trabalho, visto que este é bastante difundido na indústria, e, além disso, praticamente todas as universidades o ensinam. Portanto, os usuários deverão ter mais facilidade em seu aprendizado.

Visualizando o resultado obtido após a transformação, sabemos que é importante realizar validações semânticas durante o processo de transformação, e por isso, podemos destacar FxTL em relação a TXL. As informações semânticas de um programa não são levadas em consideração por TXL, e, portanto, deverá existir um trabalho maior para garantir que programas semanticamente corretos sejam produzidos.

Em relação ao ambiente de desenvolvimento, TXL também não possui um ambiente de desenvolvimento, e, portanto, suas transformações podem ser escritas em qualquer editor de texto, da mesma forma como FxTL. Como trabalho futuro, devemos ter o desenvolvimento de um plugin para Eclipse que fornecerá apoio ao desenvolvedor na escrita e aplicação de suas transformações.

6.1.3 ATL

ATL é uma linguagem de transformação orientada a modelos. A partir de ATL, é possível produzir um conjunto de modelos a partir de outros modelos. As suas transformações podem ser escritas de forma declarativa e imperativa. A forma preferível é utilizando a programação declarativa que simplifica o mapeamento entre os modelos fonte e destino. Entretanto, também é possível utilizar construções imperativas, principalmente nos mapeamentos que são difíceis de serem definidos de forma declarativa.

O ambiente de desenvolvimento utilizado por ATL é o Eclipse. Existe um plugin que fornece auxílio na escrita das transformações, o que melhora bastante a produtividade do usuário. Apesar disso, ATL é uma linguagem bastante complexa que requer que diversas etapas sejam realizadas antes da aplicação de uma transformação. Acreditamos que o seu uso ideal seria para transformações entre modelos distintos. Por isso, para transformações source-to-source, acreditamos que existem melhores opções disponíveis além de FxTL, como por exemplo, TXL.

Sabemos que FxTL não foi definida para competir com os sistemas de transformações orientado a modelos. Por isso, transformações que visam produzir um programa escrito em "A"a partir de um programa escrito em B não podem ser realizadas em FxTL. Para isso, aconselhamos ATL. Como podemos ver, ATL e FxTL não possuem o mesmo foco, e portanto, não disputam o mesmo mercado.

6.2 TRABALHOS FUTUROS

Como trabalho futuro, teremos a definição de um analisador sintático e semântico para Java, que, a partir de um código fonte escrito em Java, será capaz de gerar a árvore sintática no formato que o sistema proposto utiliza. Esta árvore será decorada com as informações sintáticas e semânticas do programa. Além disso, será necessária a integração entre dos analisadores com o engenho de transformação desenvolvido neste trabalho, para assim, concluir a criação do sistema de transformação genérico, chamado de FxTS - Flexible Transformation System.

Por fim, estudamos a hipótese de desenvolver um plugin para Eclipse, almejando o aumento da produtividade na escrita e aplicação de transformações. Atualmente, para a realização dos testes, utilizamos uma árvore sintática definida manualmente.

REFERÊNCIAS BIBLIOGRÁFICAS

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996. 1

[AGG07] INRIA ATLAS Group and LINA RESEARCH GROUP. Kernel metametamodel. Technical report, Disponível em: http://www.eclipse.org/gmt/am3/km3/doc/KernelMetaMetaModel.pdf, Acesso em abril de 2007. 2.4

[ATL07] ATL: Atlas Transformation Language. Technical report, Disponível em: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf, Acesso em abril de 2007. 2

[Bar84] Hengt Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984. Second Revised Edition.

[Bec99] Kent Beck. Extreme programming explained: Embrace change. *Addison-Wesley*, 1999. 1

[CB87] J. S. Collofello and J. J. Buck. Software quality assurance for maintenance. *IEEE Software*, 4(5):46–51, 1987. 1

[CB01] Fernando Castor and Paulo Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programmig Languages*, pages 236–251, Curitiba, PR, Brasil, May 2001. 1

[Cha05] Robert N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, 2005.

[Chr03] Alexander Christoph. Graph rewrite systems for software design transformations. *Lecture Notes in Computer Science*, 2591:76–86, Jan 2003. 2.1

[Col07] CollabNet, Disponível em: http://javacc.dev.java.net. *Java Compiler Compiler*, Acesso em abril de 2007. 4

[Cor04a] James R. Cordy. TXL - A language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, December 2004. 2

[Cor04b] Márcio Cornélio. *Refactoring as Formal Refinements*. Ph.D. thesis, Universidade Federal de Pernambuco, Recife, PE, Brazil, 2004. 5

[Ecl07] The Eclipse Foundation, Disponível em: http://www.eclipse.org. *Eclipse.org home*, Acesso em abril de 2007. 4

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993. 1

[GRO08a] ATLAS INRIA RESEARCH GROUP. Java abstract syntax discovery tool, Acesso em abril de 2008. Disponível em: http://www.eclipse.org/gmt/modisco/toolBox/JavaAbstractSyntax/. 2.4

[Gro08b] Object Management Group. Introduction to omgś unified modeling language (uml). Technical report, Disponível em: http://www.omg.org/gettingstarted/what_is_uml.htm, Acesso em abril de 2008. 2.4

[Gro08c] Object Management Group. Mof 2.0/xmi mapping, version 2.1.1. Technical report, Disponível em: http://www.omg.org/docs/formal/07-12-02.pdf, Acesso em abril de 2008. 2.4

[Gro08d] Object Management Group. The object management group, Acesso em abril de 2008. Disponível em: http://www.omg.org/. 2.4

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. 3.5.2

[Jac07] Jackpot research project, Acesso em julho de 2007. Disponível em: http://jackpot.netbeans.org. 2

[Jac08] Jackpot rule language, Acesso em maio de 2008. Disponível em: http://jackpot.netbeans.org/docs/rule-language.html. 2.2

[JW86] Kathleen Jensen and Niklaus Wirth. *Pascal user manual and report.* Springer-Verlag, New York, NY, USA, third edition, 1986. 3

[Kod04] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javaCC tutorial. *IEEE Software*, 21(4):70–77, 2004. 4

[Kra03] Jerry Krasner. Embedded software development issues and challenges. Technical report, Embedded Market Forecasters whitepaper, http://www.embeddedforecast.com, 2003.

[Lab08] Software Technology Laboratory. Txl world, Acesso em abril de 2008. Disponível em: http://www.txl.ca/nresources.html. 2.3

[Lar07] Craig Larman. *Utilizando UML e Padrões.* Bookman, 3 edition, 2007.

[Mer08] Ed Merks. Eclipse modeling framework, Acesso em abril de 2008. Disponível em: http://www.eclipse.org/modeling/emf/. 2.4

[Mos90] Peter Mosses. *Denotational Semantics*, volume B. MIT Press, Cambridge, MA, USA, third edition, 1990. 1

[Net07] Welcome to netbeans, Acesso em julho de 2007. Disponível em: http://www.netbeans.org. 2.2

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[Pre06] Roger S. Pressman. *Engenharia de Software*. McGraw-Hill, 6 edition, 2006.

[Pro07] ESTIMATE Professional. Technical report, Software Productivity Centre, Software disponível em: http://www.spc.ca/, Acesso em 13 de julho de 2007.

[Pro08] ModelPlex European Integrated Project. Modisco home page, Acesso em abril de 2008. Disponível em: http://www.eclipse.org/gmt/modisco/. 2.4

[Put78] Lawrence H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transaction on Software Engineering,*, 4(4):345–361, July 1978.

[set07] PROMISE Data sets. Technical report, PROMISE Data, Software disponível em: http://promisedata.org/, Acesso em 13 de julho de 2007.

[Sim99] Carlos Alberto Simões. Sistemática de métricas, qualidade e produtividade. Technical report, BFPUG, http://www.bfpug.com.br/Artigos/sistematica_metricas_simoes.htm, 1999.

[SMC07] Alexandre Alves Santos, Luis C. Souza Menezes, and Marcio Lopes Cornelio. Flexible transformation language. In *1st Workshop on Refactoring Tools held in conjunction with 21st European Conference on Object-Oriented Programming*, pages 9–10, Berlim, Germany, July 2007.

[StOSSG08] Richard Soley and the OMG Staff Strategy Group. Model driven architecture. Technical report, Object Management Group, Disponível em: http://www.omg.org/ soley/mda.html, Acesso em abril de 2008. 2.4

[Sys07] Softstar Systems. Technical report, Costar, Software disponível em: http://www.softstarsystems.com/, Acesso em 13 de julho de 2007.

[UCI07] UCI. Technical report, Machine Learning Repository, Software disponível em: http://www.ics.uci.edu/ mlearn/MLRepository.html, Acesso em 13 de julho de 2007.

[Wat04] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004. 3.5.1

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, second edition, 1999. 2

[WN96] M. Wirsing and M. Nivat. Industrial applications of asfand sdf. *Lecture Notes in Computer Science*, 1101:9–18, 1996. 2.1

[WW97] Y. Wang and I. Witten. Inducing model trees for continuous classes. European Conference on Machine Learning, April 1997.

[Xpe07] Cost Xpert. Technical report, Cost Xpert Group, Software disponível em: http://www.costxpert.com/en/index.html, Acesso em 13 de julho de 2007.

APÊNDICE A

SINTAXE ABSTRATA

Neste apêndice nós apresentamos a sintaxe abstrata da linguagem de transformação FxTL. Para a sua definição, foi utilizado a EBNF. A partir desta sintaxe, o usuário tem uma idéia formal de todos os comandos e expressões da linguagem.

```
Program ::= [Procedure] ''Start" "{" Command "}"

Procedure ::= "Procedure" Identifier "("FormalParameter*"){"Command"}"
            | Procedure Procedure ;

FormalParameter ::= Identifier ":" Type

Type ::= String | Integer | Boolean | NodeType | NodeListType

Command ::= Identifier "=" Expression
            | "If(" Expression ") {"Command"}" ["else {"Command"}"]
            | Command ";" Command
            | "Foreach" "(" Identifier "in" Node ")" "{"Command"}"
            | Identifier "=>" Node
            | Identifier "=>" Identifier "+" Node
            | Identifier "=>" Node "+" Identifier
            | Identifier "=>" Identifier "-" Node
            | Identifier "(" [ActualParameter] ")" ;
ActualParameter ::= Expression | Expression "," Expression

Node ::= Identifier "(" [Fact] ")"

Fact ::= Node | Fact "," Fact

Expression ::= Identifier "is" Node
            | "Exist" "(" Node ")" "in" Identifier
            | "Not" Expression
            | "Scope" "(" Identifier ")"
            | "Type" "(" Identifier ")"
            | Expression "==" Expression
            | Expression "!=" Expression
            | Expression ">" Expression
            | Expression "<" Expression
            | Expression ">=" Expression
            | Expression "<=" Expression
            | Expression "+" Expression
```

```
| Expression "-" Expression
| Expression "/" Expression
| Expression "*" Expression
| Expression "%" Expression
| Numeral
| true
| false
| Identifier ;
```

APÊNDICE B

SEMÂNTICA DA LINGUAGEM

Neste apêndice nós apresentamos a semântica estática de FxTL. Para a sua definição, utilizamos semântica denotacional. A partir dela, o usuário poderá saber o resultado de uma transformação sem executá-la, visto que todas as funções existentes estão definidas aqui. Além de demonstrarmos formalmente a implementação da semântica, iremos discuti-la em detalhes para um facilitar o entendimento do leitor. Abaixo, seguem as definições básicas da semântica de FxtL:

```
Top = Location

String = Character*

Childs = (str String + node Node)*

UniqueNode = Childs x Integer x String x String x Node x String

ListNode = Childs x Integer x String x String x Node x String x
                                                    Variable x Variable

Node = (unique UniqueNode + list ListNode + £)

Value = truth-value Truth-Value + integer Integer + string String
                + node Node + nodeList NodeList

Storable = Value

Argument = value Value + variable Location

Variable = variable Location

Procedure = Argument* -> Store -> Top -> Store x Top

Bindable = value Value + variable Location + procedure Procedure

empty-store : Store

allocate : Store -> Store x Location

update : Store x Location x Storable -> Store

fetch : Store x Location -> Storable
```

```
empty-environ : Environ

bind : Identifier x Bindable -> Environ

overlay : Environ x Environ -> Environ

find : Environ x Identifier -> maybe Bindable
```

Nas próximas seções encontram-se as implementações das funções semânticas da linguagem.

B.1 DECLARAÇÕES

```
run : Program -> (Sto -> Sto')
run [P1 "Start {" C1 "}"] top =
  let env = empty-environ,
      sto = empty-store in
          let (env', sto') = elaborate P1 env sto top in
              let (env'', sto'') = execute C1 env' sto' top in
                  (sto'')
```

A função `run` representa a execução de um programa. Ele recebe como entrada uma memória e após a execução do programa, uma memória alterada é gerada como resultado.

```
bind-parameter : FormalParameter -> Store ->
                                    (Argument -> Environ x Store)
bind-parameter [I ":" T] (variable loc) sto =
  (bind(I, variable loc), sto)
bind-parameter [I ":" T] (value val) sto =
  let (sto', loc) = allocate(sto) in
    let env =  bind(I, variable loc) in
      (env, update(sto', loc, val))

bind-parameters : FormalParameter* -> Argument* -> Store ->
                                    (Environ x Store)
bind-parameters [] [] sto =
   (empty-environ, sto)
bind-parameters [X] [Y] sto =
   bind-parameter X Y sto
bind-parameters (X.Y) (A.B) sto =
   let (env, sto') = bind-parameter X A sto in
     let (env', sto'') = bind-parameters Y B sto' in
       (overlay(env', env'), sto'')
```

A função `bind-parameter` realiza a ligação entre o parâmetro formal e o argumento. Lembre-se que esta trata apenas um argumento, e não uma lista de argumentos. Já a função `bind-parameters` trata uma lista de argumentos. Portanto, cada argumento é tratado individualmente e delegado para função `bind-parameter`.

```
give-argument : ActualParameter -> (Environ -> Store -> Argument*)
give-argument [E] env sto =
  value(evaluate E env sto)
give-argument ["var" I] env sto =
  let variable loc = find(env, I) in
    variable loc
```

A função `give-argument` avalia os valores passados como argumento.

```
elaborate : Procedure -> (Environ -> Store -> Environ x Store)
elaborate [Procedure" I "(" FP ") {" C1 "}"] env sto =
  let proc arg sto' top =
    let env' = overlay(bind(I, procedure proc), env) in
      let (parenv, sto'') = bind-parameter FP arg sto' in
        execute C1 (overlay(parenv,env')) sto'' top in
          (bind(I, procedure proc), sto)

elaborate ["Procedure" I "(" ") {" C "}"] env sto =
  let proc arg sto' top =
    let env' = overlay(bind(I, procedure proc), env) in
      execute C (overlay(env',env)) sto' top
    in
      (bind(I, procedure proc), sto)

elaborate [P1 P2] env sto =
  let (env', sto') = elaborate P1 env sto in
    elaborate P2 env' sto'
```

Acima, vimos duas declarações de procedimentos: a primeira com parâmetros; a segunda sem parâmetros. Dessa forma, quando um procedimento for declarado, uma das duas funções semânticas que descrevem declarações de procedimentos será utilizada. Por último, vimos declarações seqüenciais, onde poderemos ter várias declarações de procedimentos.

B.2 COMANDOS

```
execute : Command -> (Environ -> Store -> Top -> Environ x Store)
execute [I "(" AP ")"] env sto top =
  let procedure proc = find(env, I) in
    let arg = give-argument AP env sto in
      (env', proc arg sto top)

execute [I "()"] env sto top =
  let procedure proc = find(env, I) in
    (env', proc [] sto top)
```

Temos acima duas chamadas a procedimentos: a primeira com argumentos; a segunda sem argumentos.

```
execute ["If(" E ") {" C1 "}"] env sto top =
  if (evaluate E env sto) = (truth-value true)
    then execute C1 env sto top
  else
    (env, sto)

execute ["If(" E ") {" C1 "}" "else {" C2 "}"] env sto top =
  if (evaluate E env sto) = (truth-value true)
    then execute C1 env sto top
    else execute C2 env sto top

execute [C1; C2] env sto =
  let (env', sto') = execute C1 env sto top in
    execute C2  env' sto' top
```

Acima, temos as funções semânticas para comandos condicionais e seqüenciais.

```
execute [I "=" E] env sto top =
  let val = evaluate E env sto in
    if (find(env, I) = fail)
      then let (sto', loc) = allocate(sto) in
        let env' =  bind(I, variable loc) in
          (env', update(sto', loc, val))
    else
      let variable loc = find(env, I) in
        (env,update(sto, loc, val))
```

A função semântica acima é responsável pela declaração implícita da linguagem. Se o identificador utilizado não foi declarado, é alocado um espaço na memória, e é realizado o binding. Se a variável já existe, ela é atualizada com o valor da expressão utilizada.

```
execute [I "=>" node] env sto top =
  let variable loc = find(env, I) in
    (env, update(sto, loc, node))
```

Acima, temos a função semântica que trata a sintaxe base da reescrita de termos. Simplesmente, atualiza a memória com o novo nó definido.

```
execute [I "=>" I "+" node] env sto top =
  let variable loc = find(env, I) in
    let Node oldNode = fetch(sto, loc)  In
      let (childs, size, idSintatico, classe,
scope, type, next, prior) = oldNode in
        let (childsNew, sizeNew, idSintaticoNew,
classeNew, scopeNew, typeNew, nextNew, priorNew) = node in
          let lastNode = fetch(sto, next) in
            let (childsLast, sizeLast, idSintaticoLast,
classeLast, scopeLast, typeLast, nextLast, priorLast) = lastNode in
              let sto' = update(sto, priorLast, node) in
                let sto'' = update(sto', nextNew, lastNode) in
```

```
                let sto''' = update(sto'', next, node) in
                  (env, update(sto'''', priorNew, oldNode))
```

Acima, temos a adição de um novo nó após o nó atual. Primeiramente, busca-se a posição de memória onde o nó `I` está posicionado. Logo em seguida, os dois nós são desmembrados (`I` e o novo nó), e busca-se a posição de memória do próximo nó referente ao nó `I`, e o desmembra. Para facilitar a compreensão, iremos chamá-lo de `K`. Por fim, atualiza-se a memória fazendo a referência para o nó anterior de `K` apontar para o novo nó, a referência para o nó anterior do novo nó apontar para `I` e referência do próximo nó apontar para `K`. A referência para o nó posterior de `I` apontará para o novo nó. Essas atualizações são realizadas através de updates de memória.

```
execute [I "=>" node "+" I] env sto top =
  let variable loc = find(env, I) in
    let Node oldNode = fetch(sto, loc)  In
      let (childsNew, sizeNew, idSintaticoNew, classeNew, scopeNew,
typeNew, nextNew, priorNew) = node in
        let (childs, size, idSintatico, classe, scope, type, next,
prior) = oldNode in
          let priorNode = fetch(sto, prior) in
            let (childsFirst, sizeFirst, idSintaticoFirst,
classeFirst,scopeFirst, typeFirst, nextFirst, priorFirst) = priorNode
              in let sto' = update(sto, nextFirst, node) in
                let sto'' = update(sto', priorNew, priorNode) in
                  let sto''' = update(sto'', nextNew, oldNode) in
                    (env, update(sto''', prior, node))
```

Acima, temos a adição de um novo nó antes do nó atual. Da mesma forma, busca-se primeiramente a posição de memória onde o nó `I` está posicionado. Após o desmembramento dos dois nós (`I` e o novo nó), busca-se o nó anterior a `I`, e o desmembra. Para facilitarmos a compreensão, chamaremos este nó de `K`. Logo em seguida, atualiza-se a referência para o próximo elemento de `K`, fazendo-o apontar para o novo nó. Além disso, faz-se a referência para o nó anterior do novo nó apontar para `K` e a referência para o próximo nó apontar para `I`. Por último, a referência para o anterior de `I` passará a apontar para o novo nó.

```
execute [I "=>" I "-" node] env sto top =
  let variable loc = find(env, I) in
    let Node oldNode = fetch(sto, loc)  In
      let find-recursive nod1 =
        if node == nod1 then
          let (childsFinal, sizeFinal, idSintaticoFinal, classeFinal,
scopeFinal, typeFinal, nextFinal, priorFinal) = nod1 in
            let priorNode = fetch(sto, priorFinal) in
              let (childs, size, idSintatico, classe, scope, type,
next, prior) = priorNode in
                let newNextNode= fetch(sto, nextfinal) in
                  let (childsNew, sizeNew, idSintaticoNew, classeNew,
```

```
scopeNew, typeNew, nextNew, priorNew) = newNextNode in
                    let sto' = update(sto, next, newNextNode) in
                      (env, update(sto', priorNew, priorNode)
        else
          let (childs, size, idSintatico, classe, scope, type,
next, prior) = nod1 in
            let val = fetch(sto, next) in
              if (val == undefined) then
                (env, sto)
              else
                find-recursive val
      in
        find-recursive oldNode
```

A função semântica acima é responsável pela remoção de um nó em uma lista de nós. Primeiramente, busca-se a posição onde o nó `I` está posicionado. Logo em seguida, define-se uma função recursiva que verifica se dois nós são equivalentes: o nó passado para remoção e o atual elemento da lista. Se não forem iguais, a função recursiva é chamada novamente passando o próximo elemento da lista como argumento. Caso nenhum elemento case com o nó passado, o comando será encerrado sem modificações. Porém, se o nó for encontrado, ele será removido, o elemento anterior apontará para seu próximo, e o seu próximo apontará para seu anterior. Essas atualizações são realizadas através de updates de memória.

```
execute ["Foreach" "(" I "in" Node ")" "{" C1 "}"] env sto top =
  if (find(env, I) = fail) then
    let (sto', loc) = allocate(sto) in
      let env' =  bind(I, variable loc) in
        let list = mountList Node env' sto' top in
          let execute-foreach nodes env sto top =
            if listNodes = [] then
              (env,sto)
            else
              let (cab, calda) = nodes in
                let sto' = update(sto, loc, cab) in
                  let (env', sto'') = execute C1 env sto' top in
                    execute-foreach calda env' sto'' top
           in
             execute-foreach list env' sto' top
  else
    fail
```

Acima, temos a função semântica para o comando iterativo. Este fará uma busca por todos os nós que casam com um determinado formato, e sobre o resultado obtido, comandos serão executados. Se `n` elementos forem retornados como resultado, ocorrerá `n` iterações, onde em cada iteração sabe-se qual o nó iterado no momento. Primeiramente, verifica se a variável `I` já não está declarada. Se declarada, o comando falhará. Caso não exista, esta é declarada. Logo em seguida, a função auxiliar `mountList`, responsável por

montar a lista de nós que casam com um determinado formato de nó é chamada. Com o resultado desta função, uma função recursiva é definida e chamada. Caso não existam mais elementos nesta lista, a recursão é encerrada. Caso contrário, a recursão continua, e, sobre cada iteração, os comandos definidos são executados.

B.3 EXPRESSÕES

```
evaluate : Expression -> (Environ -> Store -> Value)
evaluate [I] env sto =
  coerce(sto, find(env, I))

evaluate [I "is" Node] env sto =
  let variable loc = find(env, I) in
    let Node oldNode = fetch(sto, loc)  In
      let (childs, size, idSintatico, classe, scope, type,
next, prior) = oldNode in
        let (childsTest, sizeTest, idSintaticoTest, classeTest,
scopeTest, typeTest, nextTest, priorTest) = Node in
          if ((idSintatico = IdSintaticoTest) and (size = sizeTest))
then
            verifyChild childs childsTest
          else
            (truth-value false)
```

Primeiramente, temos o cabeçalho da função semântica que define a avaliação de expressões da linguagem.

A avaliação de um identificador é realizada através da função auxiliar de coerção `"coerce"`. Se `I` for um valor, este valor é retornado diretamente. Se for uma variável, um `fetch` é realizado na memória.

A segunda função semântica é a de casamento de padrão. Ela verifica se um nó segue um determinado formato. Primeiramente, é verificado se o identificador sintático e o número de filhos são iguais entre ambos os nós. Se sim, a função auxiliar `verifyChild` é chamada. Caso contrário, o valor `false` é retornado da função.

```
evaluate ["Exist" "(" Node ")" "in" I] env sto =
  let variable loc = find(env, I) in
    let Node oldNode = fetch(sto, loc)  In
      let (childs, size, idSintatico, classe, scope, type,
next, prior) = oldNode in
        existInChilds childs Node
```

A função semântica `exist` verifica se existe um determinado nó no escopo `I`. Para sua realização, a função auxiliar `existInChilds` é chamada passando como argumento a lista com todos os filhos de `I`.

```
evaluate ["Type" "(" I ")"] env sto =
  let variable loc = find(env, I) in
    let Node node = fetch(sto, loc)  In
```

```
      let (childs, size, idSintatico, classe, scope, type,
next, prior) = node in
        type

evaluate ["Scope" "(" I ")"] env sto =
  let variable loc = find(env, I) in
    let Node node = fetch(sto, loc)  In
      let (childs, size, idSintatico, classe, scope, type,
next, prior) = node in
        scope
```

As funções semânticas acima disponibilizam o suporte semântico em FxTL. Através delas, os usuários conseguem obter informações relativas a escopo e tipo. Ambas as informações estão presentes na estrutura de um nó. Portanto, a execução da função `type`, por exemplo, é simplesmente retornar o valor de `type` presente no nó. O mesmo para a função `scope`.

B.4 FUNÇÕES AUXILIARES

```
coerce : Store x Bindable -> Value
coerce sto,value val =
  val
coerce sto,variable loc =
  fetch(sto, loc)

isString : Childs -> Truth-Value
isString str =
  Truth-Value true
isString node =
  Truth-Value false

IsNodeList : Node -> Truth-Value
isNodeList  unique =
  Truth-Value false
isNodeList list =
  Truth-Value true
```

A função de coerção verifica se o que foi passado é um valor uma referência, e retorna um valor. Se o argumento passado for um valor, este valor é retornado diretamente. Caso contrário, realiza um `fetch` na memória, e logo em seguida, retorna o resultado do `fetch`.

A função `isString` verifica se o argumento passado é uma string ou não.

A função `isNodeList` verifica se o argumento passado é uma lista de nós ou não.

```
verifyChilds : Childs -> Childs -> Truth-Value
verifyChilds C1 C2 =
  if C1 = [] then
    if C2 = [] then
      Truth-Value true
```

```
    Else
      Truth-Value false
  Else
    Let (cab, calda) = C1 in
      Let (cab2, calda2) = C2 in
        If isString cab = Truth-Value true then
          If cab = "ANY" then
            verifyChilds calda calda2
          else
            if isString cab2 = Truth-Value true then
              if cab = cab2 then
                verifyChilds calda calda2
              else
                Truth-Value false
            else
              Truth-Value false
        else
          if isString cab2 = Truth-Value true then
            Truth-Value false
          else
            let (childs, size, idSintatico, classe, scope, type,
next, prior) = cab in
              let (childsChk, sizeChk, idSintaticoChk, classeChk,
scopeChk, typeChk, nextChk, priorChk) = cab2 in
                let result = verifyChilds childs childsChk in
                  if result = Truth-Value true then
                    verifyChilds calda calda2
                  else
                    Truth-Value false
```

A função semântica `verifyChilds` recebe como argumento duas listas de filhos para comparar se são equivalentes. Durante as verificações, é levado em consideração se o filtro "ANY"foi passado como argumento, visto que neste caso, qualquer valor é considerado equivalente na outra lista. Os comandos de verificações são executados sobre cada elemento da lista, e caso estes elementos possuam filhos, a função também é executada sobre seus filhos. A condição de parada é quando uma lista vazia é recebida como argumento, ou quando uma das verificações resulta em falso.

```
existInChilds : Childs -> Node -> Truth-Value
existInChilds C1 N1 =
  if C1 = [] then
    Truth-Value false
  Else
    Let (cab, calda) = C1 in
      If isString cab = Truth-Value false then
        let (childs, size, idSintatico, classe, scope, type,
next, prior) = cab in
          let (childsTest, sizeTest, idSintaticoTest, classeTest,
```

```
scopeTest, typeTest, nextTest, priorTest) = N1 in
            if ((idSintatico = IdSintaticoTest) and (size = sizeTest))
then
              if ((verifyChilds childs childsTest) =
                    Truth-Value false) then
                if ((existInChilds childs N1) = Truth-Value false)
                then
                  existInChilds calda N1
                else
                  Truth-Value true
              else
                Truth-Value true
            else
              if ((existInChilds childs N1) = Truth-Value false) then
                existInChilds calda N1
              else
                Truth-Value true
      Else
        existInChilds calda N1
```

A função semântica `existInChilds` recebe como argumento uma lista de filhos e o formato do nó que é desejado encontrar. Essa lista será percorrida recursivamente em busca do nó desejado, e caso exista, o valor verdadeiro será retornado. As condições de paradas são quando a função recebe como argumento uma lista vazia, ou o nó desejado é encontrado.

```
mountList : Node -> Environ -> Store -> Top -> Node*
mountList Node env sto top =
  if (isNodeList top = Truth-Value false) then
    let (childs, size, idSintatico, classe, scope, type) = top in
      if (isNodeList Node = Truth-Value false) then
        let (childsNode, sizeNode, idSintaticoNode, classeNode,
scopeNode, typeNode) = Node in
          if ((idSintatico = IdSintaticoNode) and (size = sizeNode))
then
            if (verifyChild childs childsNode = Truth-Value true then
              top + mountList Node env sto childs
            else
              mountList Node env sto childs
          else
            mountList Node env sto childs
      else
        let (childsNode, sizeNode, idSintaticoNode, classeNode,
scopeNode, typeNode, nextNode, priorNode) = Node in
          if ((idSintatico = IdSintaticoNode) and (size = sizeNode))
then
            if (verifyChild childs childsNode = Truth-Value true then
              top + mountList Node env sto childs
```

```
            else
              mountList Node env sto childs
          else
            mountList Node env sto childs
  else
    let (childs, size, idSintatico, classe, scope, type, next, prior)
= top in
      if (isNodeList Node = Truth-Value false) then
        let (childsNode, sizeNode, idSintaticoNode, classeNode,
scopeNode,typeNode) = Node in
          if ((idSintatico = IdSintaticoNode) and (size = sizeNode))
          then
            if (verifyChild childs childsNode = Truth-Value true then
              top + mountList Node env sto next +
mountList Node env sto childs
            else
              mountList Node env sto next +
                        mountList Node env sto childs
          else
            mountList Node env sto next +
                        mountList Node env sto childs
      else
        let (childsNode, sizeNode, idSintaticoNode, classeNode,
scopeNode,typeNode, nextNode, priorNode) = Node in
          if ((idSintatico = IdSintaticoNode) and (size = sizeNode))
          then
            if (verifyChild childs childsNode = Truth-Value true then
              top + mountList Node env sto next +
mountList Node env sto childs
            else
              mountList Node env sto next +
                          mountList Node env sto childs
          else
            mountList Node env sto next +
                        mountList Node env sto childs
```

A função semântica `mountList` verifica todos os elementos de um programa em busca daqueles que casam com um determinado nó. Os elementos casados são adicionados em uma lista que será retornada no final da busca. Essa função é utilizada no comando `foreach`, onde, basicamente, ocorrerão iterações sobre nós que casam com um determinado formato.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo