Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Trabalho de Graduação

TIPOS PARA UMA LINGUAGEM DE TRANSFORMAÇÃO

Alexandra Barreto Assad de Barros

Orientador: Paulo Henrique Monteiro Borba

Recife
Setembro de 2004

A meus pais, minha irmã e meu amor.

AGRADECIMENTOS

Agradeço a meus pais e a minha irmã pelo constante apoio, carinho e compreensão. Por todo o amor e dedicação que recebi durante toda a vida.

Um agradecimento especial ao meu namorado que mesmo estando longe nunca negou ajuda. Por ter suportado o meu mau humor e me dado todo o apoio. Enfim, por ser essa pessoa tão maravilhosa e amada.

Ao meu orientador Paulo Borba ("O Rei") por ter me proporcionado um incrível aprendizado durante minha iniciação científica. Pela confiança, pelas repreensões e por toda a sua paciência, dedicação e sabedoria.

A todos os professores que contribuíram para minha formação acadêmica, profissional e também para minha formação como pessoa.

A minha amiga Adeline que de última hora pôde revisar este trabalho.

Aos meus amigos da faculdade, em especial, Cynthia, Felipe, Ivan, Julio, Mauro, Marcus, Paulo, Popis e Rafael, pelo companheirismo e por me ajudarem a ser uma pessoa melhor.

Ao meu cachorro, Papito, que veio para alegrar a todos em minha casa numa hora de necessidade.

Meus sinceros agradecimentos a todos que, direta ou indiretamente, contribuíram para este trabalho.

Obrigada!

Amo os prazos. Amo a pressão surda que geram quando se aproximam.

—DOUGLAS ADAMS

RESUMO

Transformação de programas é uma técnica poderosa no suporte a diversas atividades de engenharia de software. Transformações são normalmente definidas em uma meta-linguagem e têm como alvo programas escritos em uma linguagem objeto. A atividade de especificar transformações de programas em uma meta-linguagem é conhecida como meta-programação. Apesar da meta-programação ser uma abordagem poderosa para transformação de programas, ela é notoriamente passível a erros, já que a maioria das meta-linguagens não possui um sistema de tipos. Sistemas de tipos podem adicionar poder e regularidade às linguagens. Na programação, os tipos podem esclarecer várias questões assim como impor certas limitações que irão ajudar a reforçar a corretude dos programas.

O objetivo deste trabalho é definir um sistema de tipos para JaTS - uma linguagem para especificação de transformações de programas Java. Na implementação atual da linguagem, a noção de tipos já existe, mas foi definida apenas de maneira implícita e informal. É necessário especificar o sistema de tipos de JaTS com precisão para garantir, parcialmente, a corretude dos programas gerados pelas transformações.

Começamos discutindo a importância de um sistema de tipos e o conceito de linguagens de transformação. Em seguida, apresentamos os problemas presentes na linguagem JaTS e o sistema de tipos formal que foi definido para solucionar esses problemas.

Palavras-chave: sistemas de tipos, linguagens de transformação e geração de código.

SUMÁRIO

Capítulo 1—Introdução 1

1.1 Linguagens de Transformação 2
1.2 Trabalho Proposto 3
1.3 Visão Geral 3

Capítulo 2—JaTS - Java Transformation System 4

2.1 Variáveis JaTS 5
2.2 Casamento e Substituição Opcionais 6
2.3 Casamento de Declarações 7
2.4 Declarações Executáveis 9
2.5 Declarações Iterativas 11
2.6 Pré-condições 12
2.7 Declarações Condicionais 12

Capítulo 3—Problemas na Linguagem JaTS 14

3.1 Declaração de Variáveis JaTS 14
3.2 Type, Name e Identifier 15
3.3 Verificação de Tipos Java 17
3.4 Outros Erros 19

Capítulo 4—O Sistema de Tipos da linguagem JaTS 22

4.1 Sintaxe 22
4.2 Regras de Subtipo 26
4.3 Contexto Γ 26
4.4 Regras de Tipo 27
4.4.1 Erros Detectados 35

Capítulo 5—Implementação 37

5.1 Nova Arquitetura de JaTS 38
5.2 Arquitetura do Verificador de Tipos 39
5.3 Tecnologias Utilizadas 39
5.3.1 AspectJ 40
5.3.2 BCEL 40

Capítulo 6—Considerações Finais 41

Capítulo 7—Trabalhos Futuros 42

Apêndice A—Tipos de Variáveis JaTS 44

Apêndice B—Sistema de Tipos de JaTS 46

B.1 Sintaxe 46
B.1.1 Identificadores 46
B.1.2 Expressões 46
B.1.3 Declaraçõoes 47
B.2 Regras de Subtipo 48
B.3 Regras de Tipo 48

CAPÍTULO 1

INTRODUÇÃO

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program.

—LUCCA CARDELLI (Type Systems)

Um sistema de tipos é uma coleção de regras de tipos para uma linguagem de programação. As regras de tipos determinam condições para a correta execução das declarações de um programa. Quando uma linguagem de programação está associada a um sistema de tipos, dizemos que ela é uma linguagem de programação tipada. Sistemas de tipos podem adicionar poder e regularidade às linguagens. Na programação, os tipos podem esclarecer várias questões assim como impor certas limitações que irão ajudar a reforçar a corretude dos programas. Ou seja, quando a linguagem está bem projetada e verificação de tipos pode ser realizada de forma automática, podemos garantir que certas classes de erros não irão ocorrer em tempo de execução.

Cardelli mostra que o universo não-tipado de objetos computacionais foi naturalmente se decompondo em subconjuntos de comportamento uniforme e tais subconjuntos podem ser referenciados como tipos [CW85]. Sendo assim, um tipo provê uma capa protetora que esconde a representação interna dos dados e determina a forma como os objetos devem se comunicar entre si. Objetos de um tipo possuem, então, uma representação que respeita determinadas propriedades, o que facilita a execução de determinadas operações sobre esses objetos. Por exemplo, todos os inteiros possuem o mesmo comportamento, pois permitem que o mesmo conjunto de operações aja sobre eles.

Violar o sistema de tipos significa poder manipular os dados de forma não-intencional, o que pode causar resultados desastrosos no programa. Para evitar violações no sistema de tipos, é imposta aos programas uma estrutura de tipos estática, na qual tipos são associados a variáveis, operadores e métodos ou funções. Um sistema de inferência de tipos é usado para determinar os tipos das expressões quando pouca ou nenhuma informação sobre o tipo é dada explicitamente.

Uma linguagem de programação é estaticamente tipada quando o tipo de cada expressão pode ser determinado em tempo de compilação. A tipagem estática permite que os erros de tipos sejam descobertos mais cedo, além de impor uma disciplina nos programadores que acaba tornando os programas mais estruturados e fáceis de entender. No entanto, linguagens com checagem de tipos puramente estática perdem muita flexibilidade e expressividade. Sendo assim, algumas linguagens, como Java, possuem também checagem de tipos dinâmica, onde os tipos de algumas expressões só podem ser determinados em tempo de execução.

Resumindo, podemos tomar o ponto de vista de uma metodologia de software e dizer que tipos são essenciais para a evolução ordenada de grandes sistemas de software [Car91].

Porém, nós estamos mais interessados na confiabilidade que um sistema de tipos bem projetado fornece a uma linguagem de programação, principalmente quando entramos no contexto de linguagens de transformação, onde a corretude dos programas é de vital importância.

1.1 LINGUAGENS DE TRANSFORMAÇÃO

A transformação de programas é uma técnica poderosa no suporte a atividades de engenharia de software como: *refactoring* [Fow00, Opd92], desenvolvimento formal de software [BS00, Mor90], geração de código [FH99] e tradução de linguagens [FH99]. De fato, as aplicações de transformações de programas possuem grande importância, mas seu uso em projetos reais, de grande escala, não é possível sem automação. O suporte de ferramentas é vital para a aplicação de transformações de programas, pois aumenta a produtividade e diminui a chance de introduzir erros, o que ocorre muitas vezes quando se realiza uma tarefa tediosa e repetitiva.

Para implementar transformações de programas, precisamos saber como representá-las em uma meta-linguagem. Uma meta-linguagem é uma linguagem de programação na qual construímos programas para representar programas objeto. Por linguagem objeto entendemos a linguagem utilizada para construir os programas onde serão aplicadas as transformações (Java e C# são exemplos de linguagens objeto). Sendo assim, a meta-linguagem será usada para implementar as transformações.

As meta-linguagens oferecem uma abordagem poderosa para a construção de programas, permitindo que o programador escreva código que possa gerar código durante a execução da transformação. Porém, a meta-programação (ou seja, a programação em uma meta-linguagem), especialmente em contextos não tipados é notoriamente passível a erros. Alguns artigos contribuíram para a formalização de abordagens para a meta-programação tipada e descreveram como implementar transformações de programas que possam representar o sistema de tipos da linguagem objeto [CX03b, CX03a]. Porém, a maioria das meta-linguagens existentes ainda não possuem um sistema de tipos.

Meta-linguagens que não possuem um sistema de tipos não podem refletir as informações sobre os tipos de um programa objeto na representação da própria meta-linguagem. Além disso, como foi dito na seção anterior, elas não oferecem confiabilidade e regularidade, permitindo que os programadores introduzam uma grande quantidade de erros. No entanto, com uma meta-linguagem tipada, podemos garantir que grande parte dos programas gerados estarão corretos. Eis o principal motivo para criarmos um sistema de tipos rigoroso para as meta-linguagens.

Várias ferramentas para transformação de programas já foram implementadas. Castor e Borba [CB01] definiram JaTS - uma meta-linguagem para especificação de transformações de programas Java, que foi posteriormente implementada por um sistema de mesmo nome [COS+01]. Esse sistema, específico para a linguagem Java, traz alguns benefícios em relação a outros sistemas de transformações de programas, pelo fato de definir uma linguagem para especificar transformações cuja sintaxe é um superconjunto da sintaxe de Java. Isto diminui o *gap* semântico entre a linguagem objeto e a meta-linguagem. Porém, JaTS não é uma linguagem orientada a objetos, pois leva em con-

sideração somente parte da semântica de Java. No entanto, mesmo essa pequena parte torna possível especificar transformações muito mais complexas do que se somente a sintaxe fosse levada em consideração. A existência dessa linguagem e de um sistema capaz de dar suporte a ela também é uma vantagem em relação a outros sistemas, pois o programador não fica limitado a um conjunto pré-definido de transformações.

Apesar das vantagens apresentadas pela linguagem JaTS, ela ainda não possui um sistema de tipos bem definido. Isso torna possível a implementação de transformações com erros de tipos que não são detectados nem em tempo de compilação nem em tempo de execução. Sendo assim, um programador pode facilmente criar uma transformação que gere código Java incorreto. Conseqüentemente, é necessário definir um sistema de tipos para JaTS, a fim de permitir que erros na transformação (o meta-programa) sejam detectados.

1.2 TRABALHO PROPOSTO

Diante do cenário apresentado, o objetivo do trabalho é definir um sistema de tipos para JaTS. Na implementação atual da linguagem, a noção de tipos já existe, mas foi definida apenas de maneira implícita e informal. É necessário especificar o sistema de tipos de JaTS com precisão e garantir que a implementação da linguagem está em conformidade com o sistema de tipos definido. O trabalho irá abordar apenas um subconjunto significativo da linguagem JaTS, criando um sistema de tipos para esse subconjunto e permitindo que ele possa ser estendido posteriormente para abordar os demais aspectos da linguagem. Para modelar formalmente esse sistema, serão usadas regras de tipos e um cálculo minimal para Java [IPW99].

A principal contribuição deste trabalho é a criação de um sistema de tipos para parte da linguagem JaTS, impedindo a criação da algumas transformações que gerem código Java incorreto, e a extensão da implementação de JaTS para suportar parte desse sistema de tipos e localizar erros em tempo de compilação.

1.3 VISÃO GERAL

Este documento está organizado da seguinte maneira. Primeiro apresentamos uma descrição informal da sintaxe atual e da semântica de JaTS através de exemplos simples, mas expressivos. A seguir, descrevemos todos os problemas encontrados na linguagem após ter sido realizada uma série de testes. Diversos exemplos de ambigüidades e de geração de código incorreto serão mostrados. No capítulo 4 é apresentado o sistema de tipos da linguagem, detalhando as regras de tipos elaboradas. A descrição da implementação de parte do sistema de tipos encontra-se no capítulo 5. Os Capítulos 6 e 7 apresentam algumas considerações finais e sugestões para trabalhos futuros, respectivamente.

CAPÍTULO 2

JATS - JAVA TRANSFORMATION SYSTEM

Podemos escrever transformações JaTS utilizando a linguagem Java mais as construções próprias de JaTS. O objetivo dessas construções é permitir o casamento de tipos (classes ou interfaces) e a especificação dos novos tipos que serão gerados. A fim de que possamos, posteriormente, abordar os problemas da linguagem JaTS com maior clareza, esse capítulo irá descrever a sintaxe e a semântica de suas construções e como é realizado o casamento de tipos.

Uma transformação em JaTS é composta por uma pré-condição, um conjunto de *templates* fonte e um conjunto de *templates* destino e é aplicada a um conjunto de arquivos fonte Java. Não há relação entre o número de *templates* presentes em cada lado da transformação (fonte ou destino), a única restrição é que o número de *templates* fonte deve ser igual ao número de programas fonte Java aos quais a transformação será aplicada. Cada um dos *templates* pode consistir de uma ou mais declarações de tipos escritas em JaTS. Por questões de simplicidade, consideramos apenas transformações com um único *template* fonte e um único *template* destino, cada um desses contendo somente uma declaração de tipo.

O *template* fonte da transformação é casado com o tipo Java sendo transformado, conseqüentemente, ambos devem possuir estruturas sintáticas similares. O *template* destino, por sua vez, define o tipo que será produzido pela transformação e como o códigio será gerado. Resumindo, o primeiro padrão descreve o padrão a ser casado e o segundo padrão descreve como o primeiro será reescrito [RJ99]. A aplicação de uma transformação JaTS é realizada em três fases: *parsing*, transformação e *pretty-printing*. Nosso foco é na segunda fase, transformação, pois as demais fases são comuns a quase todos os compiladores e sistemas de transformação, sendo bem conhecidas e entendidas [AUS86].

A fase de transformação pode ser sub-dividida em três etapas: casamento, substituição e processamento. Na primeira fase, a árvore sintática do *template* fonte da transformação é casada com a árvore sintática do tipo fonte em Java sendo transformado. Um nó do *template* fonte casa com um nó do tipo fonte em Java em duas ocasiões: caso eles sejam idênticos ou caso o nó do *template* fonte seja uma variável JaTS. Na etapa de casamento, é produzido um mapeamento de variáveis para os valores com as quais foram casadas. Este mapeamento é chamado *mapeamento resultado* do casamento. Na segunda etapa, substituição, as ocorrências de variáveis JaTS no *template* destino da transformação são substituídas pelos valores aos quais foram mapeadas na fase de casamento. A última etapa, processamento, consiste em executar algumas estruturas de JaTS no *template* destino da transformação. As variáveis JaTS e as estruturas executáveis são descritas nas próximas seções.

2.1 VARIÁVEIS JATS

As variáveis JaTS são as construções mais simples da linguagem e consistem apenas de um identificador Java precedido do caractere '#'. Elas são usadas como *placeholders* nas transformações. A Figura 2.1 apresenta um exemplo de transformação utilizando uma variável JaTS.

Template Fonte	**Template Destino**
`class #C extends Object{}`	`class #C{}`

Figura 2.1. Transformação - Variáveis JaTS.

O *template* fonte dessa transformação casa com uma classe vazia, sem modificador, que estenda explicitamente a classe `Object` e não implemente nenhuma interface. Sendo assim, a variável `#C` é usada como um *placeholder* para o nome da classe. Quando o tipo fonte Java a ser transformado possui essas características, essa transformação gera uma classe similar, com o mesmo nome e diferindo da original apenas por não estender `Object` explicitamente. Uma aplicação dessa transformação é dada na Figura 2.2 onde o fonte Java é a classe a ser transformada e código gerado é o resultado da transformação.

Programa Fonte	**Programa Gerado**
`class Pessoa extends Object{}`	`class Pessoa{}`

Figura 2.2. Aplicação da transformação.

Variáveis podem ser declaradas como tendo um tipo específico, correspondendo a uma das construções sintáticas de Java. Em muitos casos, como no exemplo anterior, isso não é necessário uma vez que a variável aparece em um local onde não há dúvidas sobre o tipo de estrutura com a qual ela será casada. Há situações, porém, em que é necessário declarar o tipo de algumas variáveis, a fim de especificar corretamente a semântica pretendida para a transformação. Na Figura 2.3, a variável `#md` foi declarada como sendo do tipo `MethodDeclaration`. Isso significa que a variável `#md` deve ser casada com uma única declaração de método e não com uma declaração de atributo, por exemplo. Conseqüentemente, o *template* fonte dessa transformação não casaria com a seguinte classe Java:

```
public class Pessoa extends Object{
  public String nome;
}
```

Uma vez que a classe contém apenas uma declaração de atributo, para haver o casamento poderíamos definir o *template* fonte descrito na Figura 2.4.

Template Fonte

```
class #C extends Object{
   MethodDeclaration:#md;
}
```

Template Destino

```
class #C{
   MethodDeclaration:#md;
}
```

Figura 2.3. Declaração de variável explicitando o tipo.

Template Fonte

```
class #C extends Object{
   FieldDeclaration:#fd;
}
```

Figura 2.4. Variável `#fd` casa com uma declaração de atributo.

Durante a aplicação de uma transformação, na etapa em que ocorre o casamento entre o *template* fonte e o fonte Java, é possível que uma variável que já tenha sido mapeada a um valor seja casada novamente. Um casamento assim só será bem-sucedido se o valor ao qual a variável já está mapeada for idêntico àquele com a qual ela estiver sendo casada. Se essa condição não for satisfeita, o casamento falha. Também é possível, ao utilizar mais de uma vez a mesma variável, atribuir a ela tipos diferentes em cada utilização. A única real restrição é que o valor da variável seja sempre o mesmo quando houver o casamento. Problemas que podem ocorrer devido a isso serão abordados no próximo capítulo.

2.2 CASAMENTO E SUBSTITUIÇÃO OPCIONAIS

Outra construção útil de JaTS nos permite especificar o casamento de estruturas opcionais. Se uma estrutura no *template* fonte da transformação é opcional, ela será casada com a declaração correspondente no tipo fonte Java, se existir uma. Caso contrário, a parte opcional será simplesmente ignorada pelo processo de casamento. Porém, se a variável opcional for instanciada em algum outro lugar ela será gerada no programa destino (código Java gerado pela transformação). Por exemplo, na Figura 2.5, o *template* fonte da transformação casa com qualquer classe que estenda `Object` e que implemente ou não uma lista de interfaces. Ou seja, a cláusula `implements` do *template* fonte da transformação é declarada opcional. Isso é indicado pelos símbolos "`#[`" e "`]#`". Sendo assim, o *template* fonte dessa transformação pode ser casado com as duas seguintes classes:

```
class Pessoa extends Object{}

class CadastroPessoa extends Object implements ICadastroPessoa{}
```

Essa característica de JaTS possibilita a construção de transformações mais genéricas,

Template Fonte

```
class #C extends Object #[implements NameList:#IFS]#{}
```

Template Destino

```
class #C #[implements NameList:#IFS]#{}
```

Figura 2.5. Casamento opcional.

uma vez que detalhes irrelevantes, no contexto da transformação, podem ser abstraídos. No exemplo da Figura 2.5, o objetivo é retirar a declaração `extends Object` e não importa se a classe implementa ou não uma lista de interfaces.

Estruturas opcionais também podem aparecer no *template* destino da transformação. Assim como no *template* destino da transformação da Figura 2.5 que especifica que a cláusula `implements` só será parte da classe produzida pela transformação se algum valor for mapeado à variável `#IFS` no mapeamento resultado do casamento. Caso contrário a cláusula `implements` é simplesmente ignorada. É importante mencionar que, se todas as ocorrências de uma variável no *template* fonte da transformação estão contidas em estruturas opcionais, o mesmo deve ser verdade para ocorrências dessa variável no *template* destino da transformação.

2.3 CASAMENTO DE DECLARAÇÕES

Variáveis JaTS não precisam ser casadas apenas com estruturas simples como nome de classe ou uma única declaração de atributo. Elas também podem ser casadas com estruturas mais complexas como um conjunto de declarações de atributos ou com estruturas mais específicas como o nome de um atributo.

Na transformação apresentada na Figura 2.6, o *template* fonte espera que o tipo fonte Java a ser casado tenha as seguintes características: estenda a classe `Object`, implemente ou não uma lista de interfaces, tenha uma declaração de atributo com modificador `private` e tenha no mínimo alguma outra declaração de atributo. As variáveis `#t` e `#n` casam, respectivamente, com o tipo e o identificador de um atributo, já a variável `#fd` casa com toda a declaração de outro atributo. Note que a classe pode ter muitas declarações de atributos a mais, já que elas seriam casadas com `#FDS`. Caso a classe não tivesse nenhuma outra declaração a variável `#FDS` seria um conjunto vazio. Lembrando que as posições em que `#t` e `#n` são declaradas já dizem quais serão os tipos dessas variáveis, o que torna opcional a declaração na forma `Type:#t` e `Identifier:#n`. Figura 2.7 apresenta a aplicação dessa transformação em um tipo Java.

Além de remover a cláusula `extends` de uma declaração de classe, vemos pela Figura 2.7 que essa transformação remove um atributo privado declarado no corpo da classe fonte, contanto que a classe tenha as características mencionadas anteriormente. Uma vez que as declarações dos atributos `idade` e `sexo` casam com a variável `#FDS`, assumimos

Template Fonte

```
class #C extends Object #[implements NameList:#IFS]#{
   FieldDeclarationSet:#FDS;
   private #t #n;
   FieldDeclaration:#fd;
}
```

Template Destino

```
class #C extends Object #[implements NameList:#IFS]#{
   FieldDeclaration:#fd;
   FieldDeclarationSet:#FDS;
}
```

Figura 2.6. Casamento de declarações.

Programa Fonte

```
class Pessoa extends Object {
   private String nome;
   private Endereco endereco;
   private int idade;
   private char sexo;
}
```

Programa Gerado

```
class Pessoa{
   private Endereco endereco;
   private int idade;
   private char sexo;

}
```

Figura 2.7. Aplicação da transformação.

que o usuário escolhe casar `#t` com o tipo e `#n` com o identificador do atributo `nome` e `#fd` com a declaração do atributo `endereco`. De fato, em algumas situações, o usuário precisa fornecer argumentos indicando como o casamento deve proceder, caso contrário, o resultado de aplicar a transformação não pode ser previsto.

Semanticamente, o resultado seria não-determinístico, mas, na prática, uma implementação particular de JaTS teria um comportamento padrão que produziria algum dos resultados possíveis de acordo com a semântica de JaTS. Por exemplo, a implementação atual de JaTS tem o seguinte comportamento padrão: primeiro,declarações de atributo no *template* fonte da transformação são casados com declarações de atributo na classe fonte. No exemplo anterior, a declaração "`private #t #n;`" é casada com a primeira declaração de atributo compatível, na classe fonte, "`private String nome;`". Depois variáveis do tipo FieldDeclaration são casadas com declarações de atributo que ainda não tenham sido casadas. No o exemplo anterior, a variável `#fd` é casada com a próxima declaração de atributo não casada, "`private Endereco endereco;`". Finalmente, se

houver uma variável do tipo FieldDeclarationSet no *template* fonte da transformação, ela casará com as declarações de atributo que ainda não tiverem sido casadas na classe fonte. O mesmo procedimento se aplica para o casamento de declarações de construtor e método.

Adiantamos aqui que o subconjunto da linguagem JaTS selecionada pelo nosso trabalho não leva em conta a escolha do usuário sobre o valor com o qual uma variável será casada. Por uma questão de simplificação e pelo fato dessa escolha não ser feita na linguagem JaTS e sim no sistema que foi implementado para ela, assumimos o comportamento padrão.

2.4 DECLARAÇÕES EXECUTÁVEIS

Nos exemplos mostrados até agora, os *templates* destino das transformações continham apenas declarações também apareciam nos *templates* fonte. Porém, poderíamos ter, código java adicional nos *templates* destino, o que acarretaria na geração desse código. Além disso, existem declarações, que aparecem apenas no *template* destino, que usam informações das declarações originais, presentes no template fonte, mas não são necessariamente idênticas a elas. Estas declarações estão apresentadas nesta seção.

Com o objetivo de suportar essa extração ou modificação de declarações originais, JaTS define construções chamadas Declarações Executáveis. Elas podem aparecer em qualquer lugar onde uma variável também possa aparecer, mas apenas no *template* destino de transformações. Freqüentemente, essas declarações podem executar métodos em objetos que representam nós na árvore sintática sendo transformada; o que ocorre, portanto, é um encapsulamento de código Java que deve ser executado com o objetivo de gerar declarações ou, de maneira geral, construções Java válidas.

Declarações executáveis aparecem em transformações JaTS entre os símbolos "`#<`" e "`>#`". Há dois tipos de declarações executáveis. O primeiro é chamado *Declaração de Extração de Informação* como por exemplo: `#< #a.getName() >#`. Caso a variável `#a` estivesse casada a uma declaração de atributo, essa declaração produziria como resultado o nome do atributo, que nada mais é do que um objeto retornado pela invocação de um método. Os efeitos da invocação do método sobre o objeto que a executa não são importantes, apenas o resultado da declaração de método mais externa importa.

Figura 2.8 apresenta o *template* destino de uma transformação. Suponha que a variável `#a` esteja casada com a declaração de atributo "`private String nome`" de uma classe `Pessoa`. A execução das declarações executáveis produziria a seguinte classe:

```
class Pessoa{
  private String nome;
  public String nome() {
    return nome;
  }
}
```

Os valores associados a variáveis JaTS são vistos como objetos que têm um conjunto

Template Destino

```
class Pessoa{
  FieldDeclaration:#a;
  public #< #a.getType() >#  #< #a.getName() >#() {
    return #< #a.getName() >#;
  }
}
```

Figura 2.8. *Template* destino com declarações de extração de informação.

Template Destino

```
class Pessoa{
  FieldDeclaration:#< #result = #a ::  #result.removeAcessModifier();
                      #result.addModifier(‘‘protected’’);
                      #result.addModifier(‘‘volatile’’) >#
}
```

Figura 2.9. *Template* destino com declarações de modificação de informação.

de métodos que podem ser invocados dentro de declarações executáveis. Para declarações de atributo temos, entre outros, os métodos `getType()` e `getName()`. Os métodos em uma declaração de extração de informação são executados do mais interno para o mais externo, assim como métodos Java seriam. O valor retornado pelo mais externo é o resultado da declaração de extração de informação. Depois da execução, ele substitui a declaração executável.

Enquanto declarações de extração de informação obtêm informação das declarações originais, o segundo tipo de declaração executável modifica declarações originais. Neste caso, fazemos uso de uma variável temporária para armazenar um clone do objeto sobre o qual desejamos invocar um método. É possível também especificar mais de uma ação a ser executada sobre o objeto, bastando, para tanto, separar cada expressão por um “`;`”. Declarações executáveis desse tipo são chamadas *Declarações de Modificação de Informação*.

Dado o *template* destino da Figura 2.9 e assumindo que `#a` está mapeada ao mesmo atributo “`private String nome`”, a transformação produziria a classe:

```
class Pessoa{
  protected volatile String nome;
}
```

A declaração “`#result = #a`” antes do operador “`::`” indica que a variável `#result` será mapeada a uma cópia do valor mapeado a `#a`. Os métodos no corpo de uma

declaração de modificação de informação modificam o valor mapeado a `#result`. Depois de sua execução, o valor modificado é retornado como o resultado da declaração executável.

2.5 DECLARAÇÕES ITERATIVAS

As declarações iterativas de JaTS são usadas com o fim de especificar transformações que iteram sobre conjuntos de declarações realizando algum tipo de operação sobre seus elementos. Essas transformações geram declarações que seguem um determinado padrão, diferindo apenas pelas informações obtidas de cada elemento. Similarmente às declarações executáveis, declarações iterativas podem aparecer apenas no *template* destino das transformações.

O conjunto de declarações sobre o qual se quer iterar pode ser expresso através de uma variável JaTS que denote um conjunto, ou de uma expressão executável, cujo valor de retorno seja um conjunto. Além do conjunto de declarações, uma declaração iterativa possui ainda dois elementos: uma variável JaTS que assume os valores dos elementos do conjunto e um corpo de declarações.

Declarações iterativas podem estar presentes dentro do corpo de classes e interfaces e também dentro do corpo de métodos e construtores. O código gerado por uma declaração iterativa deve, necessariamente, ser válido no escopo em que esta aparece. Um exemplo clássico de aplicação é a geração de métodos de acesso aos atributos de uma classe, como mostrado na Figura 2.10. Logo em seguida apresentamos um exemplo de aplicação dessa transformação na Figura 2.11.

Template Fonte

```
class #C{
  FieldDeclarationSet:#atts;
}
```

Template Destino

```
class #C{
  forall #a in #atts {
    #< #result = #a ::  #result.removeModifier(‘‘public’’);
                        #result.addModifier(‘‘private’’) >#
    public #< #a.getType()># #< #a.getName().addPrefix("get") ># (){
      return this.#< #a.getName() >#;
    }
  }
}
```

Figura 2.10. Transformação com declaração iterativa.

Programa Fonte

```
class Conta{

   public String numero;

   public double saldo;

}
```

Código Gerado

```
class Conta{
   private String numero;
   private double saldo;
   public double getNumero(){
      return this.numero;
   }
   public double getSaldo(){
      return this.saldo;
   }
}
```

Figura 2.11. Aplicação da transformação.

2.6 PRÉ-CONDIÇÕES

Algumas transformações podem ser aplicadas apenas se determinadas pré-condições forem satisfeitas. Este é o caso de transformações usadas para implementar *refactorings* e refinamentos de programas.

Uma pré-condição em JaTS aparece no início do *template* destino e é especificada como uma expressão delimitada pelos símbolos "`#<`" e "`>#`" precedida pela palavra chave `precondition`. O valor retornado pela expressão deve, necessariamente, ser do tipo booleano. Quase todos os operadores lógicos, aritméticos e relacionais de Java podem ser utilizados, exceto aqueles que realizam atribuições, tais como "`+=`", "`++`". Um exemplo simples de pré-condição para o *template* destino da Figura 2.10 poderia ser:

```
precondition #< #atts.size() > 0 >#;
```

2.7 DECLARAÇÕES CONDICIONAIS

A geração de código em JaTS pode ser condicionada à satisfação de determinadas condições. O funcionamento das declarações condicionais é similar ao de um comando `if-then-else` de Java. Essas declarações delimitam blocos de código que são processados apenas se uma determinada condição for satisfeita. Diferente das pré-condições, cláusulas opcionais se referem a contextos menores dentro de um programa e podem permitir a geração de código alternativo, no caso em que a condição não é satisfeita.

Na condição de uma declaração opcional podem ser usados todos os operadores permitidos para as pré-condições. Assim como as declarações executáveis e iterativas, declarações condicionais podem aparecer apenas no *template* destino da transformação.

Um exemplo de transformação contendo uma declaração condicional pode ser visto na Figura 2.12. Essa transformação produz uma classe Java na qual todos os atributos que antes eram públicos passam a ser privados. Além disso, para cada um desses atributos

é gerado um método que retorna o valor do atributo. Note que o *template* destino dessa transformação é bastante similar ao da Figura 2.10, porém, nesse último, caso houvesse algum atributo sem o modificador "`public`" ocorreria um erro. Com uma declaração condicional esse tipo de erro pode ser evitado.

Template Fonte

```
class #C{
  FieldDeclarationSet:#FDS;
  MethodDeclarationSet:#MDS;
}
```

Template Destino

```
class #C{

  forall #a in #FDS {
    #if (#< #a.hasModifier(‘‘public’’) >#) {

      #< #result = #a ::  #result.removeModifier(‘‘public’’);
                          #result.addModifier(‘‘private’’);

      public #< #a.getType()># #< #a.getName().addPrefix("get") ># (){
        return this.#< #a.getName() >#;
      }

    }
    else {
      FieldDeclaration:#a;
    }
  }

  MethodDeclarationSet:#MDS;
}
```

Figura 2.12. Transformação com declaração condicional.

CAPÍTULO 3

PROBLEMAS NA LINGUAGEM JATS

Até o momento, JaTS possui um sistema de tipos informal e mesmo as poucas regras que foram implementadas não estão organizadas de maneira adequada. Como vimos no Capítulo 1, definir um sistema de tipos ajuda a evitar erros e, no caso de uma linguagem de transformação, um sistema de tipos pode garantir que o código Java gerado esteja correto. Antes de defini-lo, vários testes foram executados a fim de encontrar problemas na linguagem, entre eles, quais erros ela não consegue detectar.

Com os testes realizados, encontramos vários casos em que é possível criar transformações contendo erros de tipos, gerando código Java incorreto. Em todos os casos de erros, era possível gerar código correto apenas se programas fonte Java específicos fossem dados como entrada. Esse fato leva a transformação a depender do programa fonte para gerar código correto. Com nosso sistema de tipos, pretendemos eliminar parte dessa dependência. Vale notar que todos os testes foram realizados assumindo que o programa fonte Java dado como entrada da transformação era um programa correto. Dizemos que os erros de tipos encontrados na linguagem são todos erros dinâmicos, pois ocorrem na hora em que a transformação é executada. Os testes realizados e os erros encontrados foram agrupados em conjuntos para um melhor entendimento. Cada conjunto é descrito em detalhes nas próximas seções.

Outro problema encontrado na linguagem diz respeito às declarações de variáveis JaTS. A maneira como a linguagem trata essas declarações pode gerar muita confusão. Em alguns casos, fica impossível para o programador determinar o tipo de uma variável. Esse problema é abordado com mais detalhes na seção a seguir.

3.1 DECLARAÇÃO DE VARIÁVEIS JATS

Como visto no capítulo anterior, para algumas variáveis é opcional a declaração explícita de seu tipo, pois ele pode ser inferido a partir de sua localização. Como o programador deve saber que tipo de variável é necessário em cada contexto, é fácil olhar a transformação e deduzir os tipos das variáveis declaradas. Na Figura 3.1 podemos dizer com segurança que a variável `#C` é do tipo `Identifier`, a variável `#t` é do tipo `Type` e a variável `#a` também é do tipo `Identifier`.

Foi visto no Capítulo 2 que uma variável JaTS pode ser usada mais de uma vez, com apenas uma única restrição para o uso: os valores atribuídos em cada casamento devem ser idênticos. Sendo assim, é possível criar *templates* fonte como os da Figura 3.2, onde usamos duas vezes a mesma variável `#C`. Note que os *templates* A e B são idênticos, pois a declaração de tipo é opcional no caso das variáveis `#C` e `#a` e sempre que um programa fonte Java casar com o *template* fonte A ele também casará com o *template* fonte B.

O problema encontrado nos *templates* fonte da Figura 3.2 diz respeito ao uso da variável `#C`: é impossível dizer qual tipo será mapeado à variável `#C`. No caso do *template*

Template Fonte

```
public class #C {
  private #t #a;
}
```

Figura 3.1. A declaração explícita de tipos é opcional.

Template Fonte A

```
public class #C {
  public #C #a;
}
```

Template Fonte B

```
public class Identifier:#C {
  public Type:#C #a;
}
```

Figura 3.2. Utilizando mais de uma vez a variável `#C`.

fonte B isso ainda é mais grave, pois existem duas declarações para a mesma variável, e cada declaração possui um tipo diferente. Isso dá a entender que essa variável realmente possa assumir mais de um tipo, o que não ocorre de fato. Através da realização dos testes, vimos que o tipo atribuído às variáveis utilizadas mais de uma vez é último que fora declarado. No caso da variável `#C`, é atribuído `Type`. O mesmo problema também ocorre no *template* destino, onde poderíamos declarar explicitamente tipos diferentes a uma mesma variável.

Sabemos que uma mesma variável JaTS pode ser usada mais de uma vez se os valores atribuídos em cada casamento forem idênticos. Como apenas variáveis dos tipos `Type`, `Name` e `Identifier` podem assumir valores iguais, os teste apontaram ocorrências do problema descrito acima apenas quando tratamos de variáveis desses tipos.

Não saber o tipo de uma variável pode gerar problemas quando utilizamos declarações executáveis, onde métodos são invocados a partir delas. Além do mais, existem métodos diferentes para cada tipo de variável. Todo esse problema poderia ser evitado se fosse permitida apenas uma única declaração explícita de tipo para cada variável, pois ela assumiria o tipo declarado. Por questões de simplicidade, e partindo do pressuposto de que as variáveis possuem um contexto global, a Figura 3.4 mostra, através de um exemplo, uma possível solução para o problema. Utilizaremos essa solução para escrever as declarações de variáveis JaTS na definição do sistema de tipos.

3.2 TYPE, NAME E IDENTIFIER

Na implementação atual da linguagem JaTS, variáveis que possuem os tipos `Type`, `Name` e `Identifier` são totalmente compatíveis. Isto é, onde podemos declarar variáveis do tipo `Type`, podemos declarar variáveis do tipo `Name` e `Identifier`, e o mesmo ocorre para outras combinações. Essa característica ocasiona muitos erros e torna a transformação fortemente dependente do programa fonte Java dado como entrada. Um teste

Template Fonte

```
jvar
  Identifier:#C;
  Identifier:#a;
  Type:#t;
in
public class #C {
  public #t #a;
}
```

Figura 3.3. Sintaxe alternativa.

simples foi realizado para mostrar um dos erros que podem ocorrer quando tratamos com variáveis dos tipos citados. O teste consiste da transformação da Figura 3.4 e das duas classes descritas abaixo. Na Figura 3.4 temos uma transformação simples que gera uma nova classe com o nome da superclasse do programa fonte Java que foi casado.

Template Fonte

```
class Identifier:#C extends Name:#D{}
```

Template Destino

```
class #D {}
```

Figura 3.4. Erro envolvendo `Name` e `Identifier`.

No caso do programa fonte Java dado como entrada e casado ao *template* fonte da transformação fosse a classe abaixo:

```
class Aluno extends Pessoa{}
```

o código gerado estaria correto, pois seria gerada uma classe com a declaração: `class Pessoa`. Porém, caso o programa fonte Java dado como entrada tivesse a seguinte declaração de superclasse: "`extends pacote.pessoa.Pessoa`", seria gerada a classe abaixo que está sintaticamente errada em Java:

```
class pacote.pessoa.Pessoa{}
```

Na linguagem JaTS, está definido que: `Identifier` é toda String válida em Java, `Name` é todo nome formado por identificadores separados por um ponto, como na declaração de `import`, e `Type` são os nomes de tipos, incluindo a declaração de *array*. Como o nome da superclasse pode ser escrito na forma de *fully qualified name*, as variáveis que casam com o nome da superclasse são do tipo `Name`. Na declaração do tipo de um atributo, como podemos ter um *array*, a variável que casaria com o tipo do atributo seria do tipo `Type`, e assim por diante. Contudo, apesar dessa diferença entre `Type`, `Name` e

`Identifier`, algumas vezes, as variáveis desses tipos podem assumir valores iguais. Por isso a linguagem permite que esses tipos de variáveis sejam permutadas.

Outra transformação que pode acarretar o mesmo tipo de problema é apresentada na Figura 3.5. Caso a variável `#t` seja casada a um nome de tipo como `Pessoa` ou `Conta`, nenhum erro ocorre. Porém, caso `#t` seja casada a um tipo primitivo, ou um tipo escrito na sua forma *fully qualified name* ou mesmo um tipo *array*, será gerado código incorreto. Uma aplicação para essa transformação está na Figura 3.6, onde o programa fonte contém um atributo, `contas`, do tipo *array* de contas, `Conta[]`. Como a transformação irá gerar uma classe do tipo do atributo do programa fonte, o código gerado não obedecerá a sintaxe de Java.

Template Fonte

```
class Identifier:#C{
  private Type:#t Identifier:#a;
}
```

Template Destino

```
class #t {}
```

Figura 3.5. Erro envolvendo `Type` e `Identifier`.

Programa Fonte

```
class Banco{
  private Conta[] contas;
}
```

Código Gerado

```
class Conta[] {}
```

Figura 3.6. Exemplo de código incorreto.

Devido a esse problema, uma grande liberdade é dada aos programadores JaTS, tornando fácil a escrita de transformações que gerem código incorreto. Nosso sistema de tipos provê uma solução para esse problema a partir da criação de tipos novos para a linguagem JaTS e da adição de novas restrições quanto a permutação dessas variáveis. Maiores detalhes são descritos no Capítulo 4.

3.3 VERIFICAÇÃO DE TIPOS JAVA

Esse conjunto de erros é sem dúvida o mais interessante e o mais difícil de se tratar, além de ser o mais extenso. O sistema de tipos, que está descrito no Capítulo 4, provê uma solução parcial para os erros que serão descritos aqui. Porém, muito trabalho ainda precisa ser feito a fim de evitar esse conjunto de erros.

Não existe na implementação da linguagem JaTS qualquer suporte à verificação de tipos Java. Ou seja, é possível escrever no *template* destino código Java incorreto e esse código será gerado normalmente. A transformação da Figura 3.7 demonstra esse tipo de erro. Para qualquer programa fonte dado como entrada, sempre será gerada uma classe

contendo o método `getCodigo()` declarado no *template* destino. O método está incorreto, pois possui a cláusula `return` mesmo tendo `void` como tipo de retorno.

Template Fonte

```
class Identifier:#C{
  private String codigo;
}
```

Template Destino

```
class #C {
  private String codigo;
  public void getCodigo(){
    return codigo;
  }
}
```

Figura 3.7. Erro na verificação de tipos Java.

Para solucionar o problema, basta fazer a verificação de tipos em tempo de compilação, antes de haver o casamento, pois, sempre será gerado código incorreto, independente do programa fonte dado como entrada. Porém, se tivermos uma transformação como a da Figura 3.8 essa solução não se aplica. A transformação mantém o atributo e o conjunto original de métodos do programa fonte e cria um novo método que dobra o valor do atributo. Note que ela só funciona se o programa fonte dado como entrada possuir atributo do tipo `int`. Ou seja, a variável `#t` deve casar com o tipo `int`. Caso o atributo seja do tipo String, por exemplo, será gerado código Java incorreto. Como a transformação depende do programa fonte para gerar código correto, a verificação de tipos deverá ser feita em tempo de execução, ou seja, durante a aplicação da transformação no programa fonte (somente durante a execução da transformação possuímos informações sobre o programa fonte). Por enquanto, o sistema de tipos que definimos não possui meios para tratar desse problema.

Template Fonte

```
class Identifier:#C{
  private #t a;
  MethodDeclarationSet:#mds;
}
```

Template Destino

```
class #C {
  private #t a;
  public int getDobro(){
    int b = 2*a;
    return b;
  }
  MethodDeclarationSet:#mds;
}
```

Figura 3.8. Geração de erro depende do programa fonte.

Quando construímos transformações com declarações executáveis o problema é o mesmo: a transformação dependerá de determinados programas fonte para gerar código

correto. O *template* destino da transformação da Figura 3.9 possui três declarações executáveis: a primeira retorna o nome do método composto do prefixo `set` mais o nome do atributo, a segunda `#<#fd.getType()>#` retorna o tipo do atributo, e a terceira, retorna o nome do atributo. O problema dessa transformação encontra-se na condição da cláusula `if`, definida no *template* destino. A condição compara o parâmetro `a` ao valor `null`. Como o tipo do parâmetro é determinado pelo tipo do atributo, essa transformação irá gerar código incorreto caso o tipo do parâmetro for primitivo. A aplicação dessa transformação descrita na Figura 3.10 demonstra esse erro. Novamente, a transformação dependerá programa fonte e para capturar todos os erros é necessária uma verificação de tipos dinâmica (em tempo de execução).

Template Fonte

```
class Identifier:#C{
  FieldDeclaration:#fd;
}
```

Template Destino

```
class #C {
  FieldDeclaration:#fd;
  public void #<#fd.getName().addPrefix("set")>#(#<#fd.getType()>#  a){
    if (a != null) {
      #<#fd.getName()># = a;
    }
  }
}
```

Figura 3.9. Geração de erro depende do programa fonte.

3.4 OUTROS ERROS

Dois outros erros que não se enquadram em nenhum grupo de erros específicos foram detectados a partir dos testes realizados. O primeiro deles trata da lista de modificadores de uma variável. Sabemos que um atributo possui uma lista de modificadores diferente da de uma variável local, porém, a linguagem JaTS, não trata essa diferença. Podemos usar livremente a declaração de um atributo no lugar da declaração de uma variável. Aplicamos a transformação da Figura 3.11 no programa fonte contendo uma classe com a seguinte declaração de atributo: `private String nome`. Vimos que foi gerado o método `met()` e dentro dele, assim como descrito no *template* destino, foi colocada a declaração desse atributo gerando código incorreto a seguir:

Programa Fonte

```
class Conta{
  private int numero;
}
```

Código Gerado

```
class Conta{
  private int numero;
  public void setNumero(int a){
    if (a != null) {
      numero = a;
    }
  }
}
```

Figura 3.10. O código gerado é incorreto.

```
public void met(){ private String nome; }
```

Template Fonte

```
class Identifier:#C{
  FieldDeclaration:#fd;
}
```

Template Destino

```
class #C{
  public void met(){
    #fd;
  }
}
```

Figura 3.11. Problema envolvendo lista de modificadores.

O segundo erro encontrado pode ocorrer no caso de uma falta de atenção por parte do programador (de fato, é por esse motivo que ocorrem a maioria dos erros durante a programação). Na transformação da Figura 3.12, a variável `#l1` casa com uma lista de interfaces e a variável `#l2` casa com uma lista de exceções. Ambas são do tipo `NameList`, o que indica que posso usar a variável `#l1` no lugar da variável `#l2` e vice-versa. Exatamente isso está sendo feito no *template* destino. Como a linguagem não detecta esse tipo de erro, o código gerado terá uma classe que implementa uma lista de exceções e um método que pode lançar um conjunto de interfaces. O programador só se dará conta do erro que cometeu quando estiver usando o código gerado. Para evitar que seja gerado esse tipo de código incorreto, podemos definir tipos diferentes para a lista de interfaces e para a lista de exceções, como `InterfaceNameList` e `ExceptionNameList`.

Template Fonte

```
class Identifier:#C implements NameList:#l1{
  public void met() throws NameList:#l2{
    Block:#b;
  }
}
```

Template Destino

```
class #C implements #l2 {
  public void met() throws #l1{
    Block:#b;
  }
}
```

Figura 3.12. Problema envolvendo `NameList`.

CAPÍTULO 4

O SISTEMA DE TIPOS DA LINGUAGEM JATS

Após identificar os erros mais freqüentes presentes na linguagem JaTS, apresentamos uma formalização do seu sistema de tipos. Como foi discutido anteriormente, um sistema de tipos provê regularidade e poder à linguagem, define o que é um programa bem tipado e garante a corretude desse programa. O sistema de tipos formal de uma linguagem de programação é uma caracterização matemática do sistema de tipos informal descrito nos manuais dessa linguagem. Uma vez que um sistema de tipos é formalizado, pode-se tentar provar um teorema que declara que *todo programa bem tipado se comporta corretamente*. Caso o teorema seja válido, podemos dizer que o sistema de tipos é correto [Car04].

Na implementação atual de JaTS a noção de tipos já existe, mas foi definida de maneira implícita e informal. O sistema de tipos aqui definido se refere apenas à linguagem JaTS e visa garantir que a definição de uma transformação não apresenta ambigüidades. Com esse sistema de tipos, também visamos garantir, embora parcialmente, que o código gerado pela transformação é correto. A definição de regras de tipos que garantam por completo a corretude do código Java gerado está além do escopo deste trabalho.

Pretendemos, com a formalização do sistema de tipos de JaTS, impedir a ocorrência de alguns erros descritos no Capítulo 3. Essa formalização também irá guiar a implementação do sistema de tipos, tornando-a mais coerente e concisa, pois as regras podem ser facilmente mapeadas em código.

A Seção 4.1 descreve o primeiro passo para formalizar o sistema de tipos: descrever a sua sintaxe. Por questões de simplicidade, consideramos apenas um subconjunto significativo da linguagem JaTS e não todas as suas construções. Após descrita a sintaxe, prosseguimos para a definição das regras nas seções seguintes. As regras de tipo descrevem uma relação *possui-tipo* da forma $A \triangleright T$ entre termos A e tipos T (também definimos algumas relações *subtipo-de* na forma $A <: B$ entre tipos). O conjunto de todas as regras de tipos, e subtipo, que descrevemos formam o sistema de tipos de JaTS.

4.1 SINTAXE

A Figura 4.1 apresenta as definições dos identificadores da linguagem JaTS, através de BNFs. Identificadores podem representar: nomes de classes, de atributos, de parâmetros, de métodos e de tipos. Inicialmente, a linguagem possuía apenas três tipos de identificadores : `Type`, `Name` e `Identifier`. Como vimos na Seção 3.2, existe um sério problema quando tratamos com variáveis desses tipos. Para impor uma maior restrição sobre a permutação dessas variáveis, criamos o tipos `TypeIdentifier`, que corresponde ao nome de uma classe, e `TypeName`, que corresponde ao nome da superclasse e pode assumir a forma *fully qualified name*. Após descrever as regras de subtipo, abordaremos este assunto com mais detalhes.

```
Identifier      ::=  JVariable | Word | ExecutableDeclaration
JVariable       ::=  '#'Word
Word            ::=  Letter (Letter | Digit)*
Name            ::=  Identifier | Name'.'Identifier
Type            ::=  PrimitiveType | TypeName('[]')*
                     | ExecutableDeclaration
TypeName        ::=  TypeIdentifier | Name'.'TypeIdentifier
TypeIdentifier  ::=  Identifier
PrimitiveType   ::=  'boolean' | 'char' | 'int'
```

Figura 4.1. Identificadores

```
Expression   ::=  MethodCall | FieldAccess | Identifier
MethodCall   ::=  (Expression'.'Identifier | Identifier)
                  '('(Expression | ε)')'
FieldAccess  ::=  Identifier'.'Identifier
```

Figura 4.2. Expressões

A Figura 4.2 apresenta a BNF para as expressões que podem aparecer nas declarações de um *template* JaTS. Por questões de simplicidade, uma chamada de método recebe apenas um parâmetro. Seria simples estender essa definição para que um método recebesse uma lista de parâmetros, já que as regras seriam bastante similares. Optamos por não fazê-lo apenas para manter a definição do sistema de tipos de JaTS o mais concisa possível.

A Figura 4.3 apresenta a BNF para todas as declarações que podem aparecer em uma transformação JaTS. A construção JaTSLeftHandSide corresponde ao *template* fonte da transformação, antes chamado de lado esquerdo. JaTSRightHandSide corresponde ao *template* destino. Entre as construções JaTS apresentadas no Capítulo 2, a sintaxe engloba declarações de variáveis, declarações iterativas, declarações executáveis, apenas as de extração de informação, e declarações opcionais, apenas na declaração de superclasse.

Um exemplo de transformação que obedece a essa sintaxe está apresentada na Figura 4.4. O *template* fonte deve casar com um programa fonte que estenda ou não a classe `Object`, tenha um atributo na forma `private int codigo`, tenha ou não mais declarações de atributos e tenha um método `get()` e um método `set()` para o atributo `codigo`. O *template* destino remove a declaração de superclasse, se ela existir, remove o atributo `codigo`, e seus métodos de acesso, e insere um método `get()` para cada um dos outros atributos. É importante entender essa transformação, pois posteriormente iremos aplicar nela as regras de tipos.

```
JaTSLeftHandSide        ::=  (ε | JVarDeclarationSet) ClassDeclaration
JaTSRightHandSide       ::=  ClassDeclaration
JVarDeclarationSet      ::=  'jvar' JVarDeclaration
                             (JVarDeclaration)* 'in'
JVarDeclaration         ::=  JVariable ':'  TypeIdentifier ';'
ClassDeclaration        ::=  'class' TypeIdentifier
                             (ε | ParentDeclaration)
                             '{' ClassBodyDeclarations '}'
ParentDeclaration       ::=  'extends' TypeName
                             | '#['`extends' TypeName`]#'
ClassBodyDeclarations   ::=  Declaration (Declaration)*
Declaration             ::=  FieldDeclaration
                             | MethodDeclaration
                             | ForAllDeclaration
FieldDeclaration        ::=  Type Identifier`;'
MethodDeclaration       ::=  Type Identifier '('FormalParameter`)'
                             '{'Block`}'
FormalParameter         ::=  Type Identifier
Block                   ::=  (Name '=' Expression`;')*
                             (ε | 'return' Expression`;')
ForAllDeclaration       ::=  'forall' JVariable 'in' JVariable
                             '{' ForAllBlock '}'
ForAllBlock             ::=  ForAllBlockDeclaration
                             (ForAllBlockDeclaration)*
ForAllBlockDeclaration  ::=  FieldDeclaration
                             | MethodDeclaration
                             | ExecutableDeclaration
ExecutableDeclaration   ::=  #<JVariable`.'MethodCall ('.'MethodCall)*>#
```

Figura 4.3. Declarações

Template Fonte

```
jvar
  TypeIdentifier:#C;
  FieldDeclarationSet:#atts;
  Identifier:#p;
in
class #C #[extends Object]#{

  int cod;
  #atts;

  int getCod(){
    return this.cod;
  }

  void setCod(int #p){
    this.cod = #p;
  }

}
```

Template Destino

```
class #C{

  int cod;
  #atts;

  void mudarCod(String str){
    this.cod = str;
  }

  forall #a in #atts {

    #< #a.getType()># #< #a.getName().addPrefix("get") ># (){
      return this.#< #a.getName() >#;
    }

  }

}
```

Figura 4.4. Transformação JaTS.

4.2 REGRAS DE SUBTIPO

Para solucionar o problema descrito na Seção 3.2 definimos regras de subtipos entre os tipos `TypeIdentifier`, `TypeName` e `Type` e entre os tipos `Identifier` e `Name`. A criação dos tipos `TypeIdentifier` e `TypeName`, e com as regras de tipos apresentadas na Figura 4.5 torna inválida a transformação da Figura 3.5. As regras de subtipo e os novos tipos foram necessários, pois os identificadores, dependendo de sua localização, assumem características diferentes. Antes, os tipos não representavam bem as características e, além disso, a relação entre eles não estava definida.

```
Identifier <:  Name

TypeIdentifier <:  TypeName <:  Type
```

Figura 4.5. Regras de subtipo.

A primeira regra de subtipo diz que o tipo `Identifier` é subtipo de `Name`, ou seja, podemos colocar uma variável JaTS do tipo `Identifier` em uma posição que espera uma variável do tipo `Name`. O mesmo se aplica à segunda regra de subtipo. Isso impede a ocorrência de problemas como os da Figura 3.4, pois agora o nome da classe deve casar com uma variável do tipo `TypeIdentifier` e o nome da superclasse com uma variável do tipo `TypeName`. O erro descrito na Figura 3.4 não ocorrerá mais, pois não é permitido colocar uma variável do tipo `TypeName` em um lugar que espera uma variável `TypeIdentifier`. O *template* fonte da transformação da Figura 3.4 deverá ser escrito como mostra a Figura 4.6. Já o *template* destino é inválido e não passaria pelo verificador de tipos.

Template Fonte

```
jvar
  TypeIdentifier:#C;
  TypeName:#D;
in
class #C extends #D{ }
```

Figura 4.6. Template fonte corrigido.

4.3 CONTEXTO Γ

As regras de tipo não podem ser formalizadas sem antes introduzir outro elemento fundamental que não está refletido na sintaxe da linguagem: ambientes, ou contextos, de tipagem estática. Eles são usados para gravar os tipos de variáveis livres durante o processamento do programa; estão bastante ligados às tabelas de símbolos de um compilador durante a fase de verificação de tipos. As regras de tipos são sempre formuladas levando em consideração o ambiente onde está o fragmento de programa sendo analisado.

Por exemplo, a relação *possui-tipo*, $A \triangleright T$, está associada a um contexto Γ que contém informações sobre as variáveis livres de A e T. A relação completa é escrita na forma $\Gamma \vdash A \triangleright T$ (lê-se: A possui tipo T no contexto Γ).

Para o sistema de tipos da linguagem JaTS, foi definido um contexto Γ bastante parecido com o descrito acima. Como JaTS é uma meta-linguagem, ao invés de conter mapeamentos de variáveis para tipos, o contexto Γ contém mapeamentos de identificadores para tipos. Tais identificadores são: nomes de tipos primitivos, métodos, atributos e variáveis JaTS. Antes de iniciar a verificação de tipos da transformação, devemos construir o Estado Inicial de Γ. Para isso, inserimos em Γ todos os tipos primitivos como identificadores e os associamos ao tipo `Type` e em seguida, as variáveis JaTS declaradas no início da transformação são inseridas e associadas a seus respectivos tipos. Nomes de métodos, atributos e nomes de parâmetros, se não forem variáveis JaTS, são inseridos em Γ associados ao tipo `Identifier`. Não é necessário inserirmos nomes de classe, superclasse, tipo de método, atributo, parâmetro ou tipo de retorno do método em Γ, pois eles podem assumir a forma de qualquer `String` válida em Java e como já foi feita a análise sintática, garantimos que estão corretos. Além do mais, a verificação de tipos Java dentro da transformação, em junção com a verificação das construções de JaTS (Seção 3.3), ainda não armazena os tipos Java das declarações e está fora do escopo deste trabalho.

Alguns identificadores do contexto Γ possuem um escopo de classe. Em Γ, tais identificadores são representador por um par `(C, id)`, onde `id` representa o identificador e `C` a classe onde ele está. As variáveis JaTS declaradas no início da transformação e os tipos primitivos, possuem contexto global e não estão contidos em um par.

Utilizamos a transformação da Figura 4.4 para descrever como um contexto Γ se comporta durante a verificação. A Figura 4.7 contém o Estado Inicial de Γ e seus demais estados. Primeiramente, os *templates* são percorridos a fim construir o estado inicial de Γ - assim como foi descrito no parágrafo anterior. Após a construção, inicia-se a verificação de tipos dos *templates*. Γ permanece Estado A durante a verificação do método `getCod()` e também, posteriormente, durante a verificação do método `setCod()`. Neste estado, o identificador `this` é inserido em Γ associado ao tipo `Identifier`, pois ele pode ser usado para acessar um atributo ou método da classe. A verificação segue e durante a checagem do método `mudarCod()`, presente no *template* destino, Γ permanece no Estado B. Além de conter o indetificador `this`, durante este estado, Γ também contém o parâmetro `str` associado ao tipo `Identifier`. Por fim, durante a verificação da declaração iterativa - `forall` - Γ permanece no Estado C, onde foi inserida a variável `#a` associada ao tipo `FieldDeclaration`, pois são os elementos do conjunto representado por `#atts`.

4.4 REGRAS DE TIPO

Com o objetivo de simplificar as regras de tipos, é necessário estabelecer algumas restrições: nomes de atributos e métodos devem ser diferentes; os parâmetros não podem possuir o mesmo identificador que um atributo ou método, porém, podem repetir os identificadores entre si; o programa fonte dado com entrada da transformação é um programa que compila em Java; não estamos tratando a declaração de variáveis locais.

Uma transformação JaTS possui, em um mesmo contexto, variáveis JaTS ou con-

Estado Inicial

`boolean`	↦	`Type`
`char`	↦	`Type`
`int`	↦	`Type`
`#C`	↦	`TypeIdentifier`
`#atts`	↦	`FieldDeclarationSet`
`#p`	↦	`FormalParameter`
`(#C, cod)`	↦	`Identifier`
`(#C, getCod)`	↦	`Identifier`
`(#C, setCod)`	↦	`Identifier`
`(#C, mudarCod)`	↦	`Identifier`

Estado A

Estado Inicial ∪

{ `(#C, this)` ↦ `Identifier` }

Estado B

Estado Inicial ∪

{ `(#C, str)` ↦ `Identifier`,

`(#C, this)` ↦ `Identifier` }

Estado C

Estado Inicial ∪

{`(#C,#a)` ↦ `FieldDeclaration`}

Figura 4.7. Estados do contexto Γ.

struções Java. Por exemplo, no lugar de um atributo podemos ter uma variável JaTS do tipo `FieldDeclaration` ou uma declaração de atributo que esteja de acordo com a linguagem Java. Nas regras de tipo, quando queremos remeter a ambas representações escrevemos na forma de uma *meta-meta-variável*. Elas estão diferenciadas nas regras por palavras escritas im itálico.

Regras de tipo validam julgamentos tendo como base julgamentos já conhecidos como válidos. Julgamentos possuem a forma $\Gamma \vdash \Im$, onde $\Im$ é uma asserção, por exemplo, $A \triangleright T$. Abaixo segue a forma geral de uma regra de tipo:

$$\frac{\Gamma_1 \vdash \Im_1 \; ... \; \Gamma_n \vdash \Im_n}{\Gamma \vdash \Im} \qquad \text{(Nome da regra)}$$

Cada regra de tipo contém um conjunto de premissas sobre um linha horizontal com uma única conclusão abaixo da linha. Quando todas as premissas são satisfeitas, a conclusão é válida; o número de premissas pode ser zero. Cada regra possui um nome.

A coleção de todos os identificadores declarados em Γ, inclusive os que não possuem um contexto global, é indicada nas regras por `dom(Γ)`, o domínio de Γ. Para obter o tipo de um identificador no contexto Γ escrevemos $\Gamma(id)$. Caso o identificador possua tipo `T` então, $\Gamma(id)$ `= T`. Se o identificador estiver dentro do contexto de uma classe obtemos seu tipo da seguinte maneira: $\Gamma(\texttt{C})(id)$, onde `C` é o nome da classe em que está o identificar `id`. A Figura 4.8 mostra as regras referentes à tipagem de um identificador.

Para entender melhor o funcionamento das regras de tipos, vamos aplicá-las à transformação da Figura 4.4. Como sabemos uma transformação é composta por dois *templates* e para verificar a corretude de ambos, criamos duas regras de tipos. Elas estão descritas na Figura 4.9. Começaremos explicando detalhadamente como ocorre a verificação de tipos

$$\frac{\Gamma(id) = \mathtt{T}}{\Gamma \vdash id \triangleright \mathtt{T}} \quad (\textsc{Id})$$

$$\frac{\Gamma(C)(id) = \mathtt{T}}{\Gamma, C \vdash id \triangleright \mathtt{T}} \quad (\textsc{CId})$$

Figura 4.8. Tipo de um identificador.

$$\frac{\Gamma \vdash jvSet \triangleright \mathtt{JVarDeclarationSet} \qquad cd \triangleright \mathtt{ClassDeclaration}}{\Gamma \vdash jvSet\ cd \triangleright \mathtt{JaTSLeftHandSide}} \quad (\textsc{LSProg})$$

$$\frac{\Gamma \vdash cd \triangleright \mathtt{ClassDeclaration}}{\Gamma \vdash cd \triangleright \mathtt{JaTSRightHandSide}} \quad (\textsc{RSProg})$$

Figura 4.9. `JaTSLeftHandSide` e `JaTSRightHandSide`.

do *template* fonte. É importante notar que, no momento em que iniciamos a verificação, o contexto Γ está no Estado Inicial da Figura 4.7.

Para garantir que o *template* fonte da transformação esteja bem tipado, ele deve obedecer a regra LSProg. Podemos ver, pela regra, que para uma construção ser do tipo `JaTSLeftHandSide` ela deve possuir a forma "*jvSet cd*", onde *jvSet* é uma meta-meta-variável que possui o tipo `JVarDeclarationSet` e *cd* é uma meta-meta-variável que possui o tipo `ClassDeclaration`. Observando o *template* fonte da transformação, vemos que ele contém declarações de variáveis JaTS. Esse conjunto de declarações, desde a palavra `jvar` até `in`, é associado à meta-meta-variável *jvSet* da regra LSProg. Já a declaração de classe é associada à meta-meta-variável *cd*.

Agora, precisamos definir duas regras que irão inferir o tipo de *jvSet* e o tipo de *cd*. Lembrando que, para a transformação estar correta, essas meta-meta-variáveis devem possuir os tipos `JVarDeclarationSet` e `ClassDeclaration`, respectivamente. Começamos tentando mostrar que a variável *jvSet* é do tipo `JVarDeclarationSet`. Para isso, existem as regras descritas na Figura 4.10, onde $\overline{\mathtt{x}}$ representa uma seqüência de elementos que possuem o mesmo tipo de x. A regra JVarDecSet estabelece que para uma construção ser do tipo `JVarDeclarationSet` ela deve possuir a forma "`jvar` $\overline{var}$ `in`", onde *var* é do tipo *JVarDeclaration*. Para garantir que *var* realmente possui o tipo *JVarDeclaration*, devemos aplicar a regra JVar. Observando o *template* fonte da transformação, concluímos que até o momento ele obedece às regras de tipo. Porém, devemos checar agora se a meta-meta-variável *cd* é do tipo `ClassDeclaration`.

$$\frac{\Gamma \vdash var \triangleright \texttt{JVarDeclaration}}{\Gamma \vdash \texttt{jvar } \overline{var} \texttt{ in} \triangleright \texttt{JVarDeclarationSet}} \quad (\textsc{JVarDecSet})$$

$$\frac{\Gamma \vdash v \triangleright \texttt{T}}{\Gamma \vdash \texttt{T} : v; \triangleright \texttt{JVarDeclaration}} \quad (\textsc{JVar})$$

Figura 4.10. `JVarDeclarationSet` e `JVarDeclaration`.

$$\frac{\Gamma, T \vdash \mathit{classBody} \triangleright \texttt{ClassBodyDeclarations} \quad \Gamma \vdash T \in \texttt{dom}(\Gamma) \Rightarrow T \triangleright \texttt{TypeIdentifier}}{\Gamma \vdash \texttt{class } T\{\mathit{classBody}\} \triangleright \texttt{ClassDeclaration}} \quad (\textsc{ClassDec})$$

$$\frac{\Gamma, T \vdash \mathit{classBody} \triangleright \texttt{ClassBodyDeclarations} \quad \Gamma \vdash (T \in \texttt{dom}(\Gamma) \Rightarrow T \triangleright \texttt{TypeIdentifier}) \qquad \mathit{parentDec} \triangleright \texttt{ParentDeclaration}}{\Gamma \vdash \texttt{class } T \ \mathit{parentDec}\{\mathit{classBody}\} \triangleright \texttt{ClassDeclaration}} \quad (\textsc{ClassParentDec})$$

Figura 4.11. `ClassDeclaration`.

Duas regras verificam se uma declaração de classe está bem tipada, ou seja, se a partir dela podemos inferir o tipo `ClassDeclaration`. Elas estão descritas na Figura 4.11. Como a declaração de superclasse é opcional, foram necessárias essas duas regras: uma assumindo que há declaração de superclasse e outra assumindo o contrário. A regra ClassDec diz que para uma construção ser do tipo `ClassDeclaration` ela precisa ser da forma "`class` T $\{\mathit{classBody}\}$", onde a meta-meta-variável T deve ser do tipo `TypeIdentifier` apenas se ela pertencer ao domínio de Γ, caso contrário, T é qualquer String válida em Java, e *classBody* deve ser do tipo `ClassBodyDeclarations`. Note que, para efetuar a checagem do corpo da classe, seu nome é passado para o contexto, pois quando nos referimos a métodos e atributos, precisamos saber a que classe pertencem.

A transformação que estamos trabalhando (4.4), enquadra-se na regra ClassParentDec, pois ela possui uma declaração de superclasse envolvida por uma declaração opcional de JaTS. Sendo assim, devemos checar se a declaração opcional é válida e possui o tipo `ParentDeclaration`. Para tal usamos a regra OptDec da Figura 4.12. Essa regra diz que para uma declaração opcional possuir um tipo `T`, a expressão que ela envolve deve possuir o mesmo tipo `T`. Visto isso, precisamos garantir que a expressão `extends Object`, que está envolvida pela declaração opcional no *template* fonte da transformação, é do tipo `ParentDeclaration`. Para fazer essa verificação usamos a regra ParentDec. Como o identificador `Object` não está no contexto de Γ e é uma `String` válida de Java, concluímos

$$\frac{\Gamma \vdash T \in \texttt{dom}(\Gamma) \Rightarrow T \triangleright \texttt{TypeName}}{\Gamma \vdash \texttt{extends } T \triangleright \texttt{ParentDeclaration}} \quad (\textsc{ParentDec})$$

$$\frac{\Gamma \vdash e \triangleright \texttt{T}}{\Gamma \vdash \#[e]\# \triangleright \texttt{T}} \quad (\textsc{OptDec})$$

Figura 4.12. `ParentDeclaration` e `OptionalDeclaration`.

$$\frac{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{FieldDeclaration}}{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{ClassBodyDeclarations}} \quad (\textsc{DecsFD})$$

$$\frac{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{MethodDeclaration}}{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{ClassBodyDeclarations}} \quad (\textsc{DecsMD})$$

$$\frac{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{ForAllDeclaration}}{\Gamma, \texttt{C} \vdash fd \triangleright \texttt{ClassBodyDeclarations}} \quad (\textsc{DecsFA})$$

$$\frac{\Gamma, \texttt{C} \vdash d' \triangleright \texttt{ClassBodyDeclarations} \qquad d \triangleright \texttt{ClassBodyDeclarations}}{\Gamma, \texttt{C} \vdash d'\ d \triangleright \texttt{ClassBodyDeclarations}} \quad (\textsc{Decs})$$

Figura 4.13. `ClassBodyDeclarations`.

que a transformação possui uma declaração de superclasse válida.

Para completar a checagem da classe, precisamos verificar a corretude do corpo da mesma através das regras contidas na Figura 4.13. De maneira geral, essas regras definem que uma construção é do tipo `ClassBodyDeclarations` se ela for composta por um conjunto de `FieldDeclaration`, um conjunto de `MethodDeclaration` e um conjunto de `ForAllDeclaration`. Para cada declaração dentro do corpo da classe, devemos garantir que ela possui um desses tipos.

Para checar os atributos da classe utilizamos a regra descrita na Figura 4.14. No *template* fonte da transformação temos um atributo, "`int cod`". Como `int` está no domínio de Γ mapeando em `Type` e `cod` está no domínio de Γ mapeando em `Identifier`, as premissas da regra FieldDec são verdadeiras, garantindo que a declaração de atributo "`int cod`" é válida.

A próxima declaração presente na transformação é a variável JaTS `#atts`. Essa

$$\frac{\Gamma, \mathtt{C} \vdash t \in \mathtt{dom}(\Gamma) \Rightarrow t \triangleright \mathtt{Type} \wedge t \neq \mathtt{void} \quad \Gamma, \mathtt{C} \vdash id \triangleright \mathtt{Identifier}}{\Gamma, \mathtt{C} \vdash t\ id; \triangleright \mathtt{FieldDeclaration}} \quad \text{(FieldDec)}$$

Figura 4.14. `FieldDeclaration`.

$$\frac{\Gamma \vdash t \triangleright \mathtt{T}}{\Gamma \vdash \mathtt{t} \triangleright \mathtt{T}^*} \quad \text{(SetsUnit)}$$

$$\frac{\Gamma \vdash t' \triangleright \mathtt{T} \qquad t \triangleright \mathtt{T}^*}{\Gamma \vdash \mathtt{t'\ t} \triangleright \mathtt{T}^*} \quad \text{(Sets)}$$

Figura 4.15. `Sets`.

variável representa um conjunto de elementos, para verificar se ela é uma declaração válida para o corpo da classe, devemos aplicar as regras descritas na Figura 4.15. $\mathtt{T}^*$ representa um conjunto de tipos `T` que é válido apenas se todos os elementos forem do tipo `T`. A variável `#atts` está associada em Γ ao tipo `FieldDeclarationSet`, o que já garante que ela possui um tipo $\mathtt{T}^*$. Sendo assim, ela corresponde a um conjunto de declarações de atributos, `FieldDeclaration`, o que valida a ocorrência de `#atts`.

Devemos agora verificar o método `getCod()`. A verificação dos métodos de uma classe é feita seguindo a regra MethodDec descrita na Figura 4.16. Para a declaração de método ser válida ela deve possuir a forma "*ret met* `(`*t id*`)` `{`*block*`}`", onde *ret* é o tipo de retorno e caso esteja contido em Γ deve ser do tipo `Type`, *met* é o nome de método e deve ser do tipo `Identifier`, "*t id*" é o parâmetro do método e deve ser do tipo `FormalParameter` em Γ e *block* representa o corpo do método e deve ser do tipo `Block`. É importante notar que, o contexto em que é verificado o corpo do método não é apenas Γ e C - classe atual - , mas sim Γ e C mais o identificador `this` mapeando em `Identifier` e o identificador *id* mapeando em Identifier. Como a assinatura do método obedece a regra, vamos partir agora para a checagem de seu corpo. Quando garantirmos que o corpo é bem tipado, garantimos que todas as premissas da regra MethodDec são verdadeiras, tornando o método `getCod()` bem tipado.

A título de observação, durante a checagem do corpo do método `getCod()`, o contexto Γ está no Estado A da Figura 4.7.

Para verificar o corpo de um método seguimos as regras descritas na Figura 4.17. De maneira geral, o corpo de um método pode conter: um conjunto de atribuições, onde uma expressão é atribuída a um identificador ou a um atributo, e apenas uma cláusula `return` seguida de uma expressão. O corpo do método sendo analisado, `getCod()`, possui apenas uma declaração de retorno, `return this.cod;`. Para determinar se essa declaração obedece às regras de tipos, devemos verificar se `this.cod` é uma expressão válida. Para tal

$$\frac{\Gamma, \mathtt{C} \oplus (\mathtt{this} \mapsto \mathtt{Indentifer}, id \mapsto \mathtt{Identifier}) \vdash block \triangleright \mathtt{Block} \qquad \Gamma, \mathtt{C} \vdash (ret \in \mathtt{dom}(\Gamma) \Rightarrow ret \triangleright \mathtt{Type}) \qquad met \triangleright \mathtt{Identifier} \qquad t\ id \triangleright \mathtt{FormalParameter}}{\Gamma, \mathtt{C} \vdash ret\ met(t\ id)\{block\} \triangleright \mathtt{MethodDeclaration}}\ (\textsc{MethodDec})$$

Figura 4.16. `MethodDeclaration`.

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression}}{\Gamma, \mathtt{C} \vdash (id \triangleright \mathtt{Identifier} \wedge id \neq \mathtt{this}) \vee id \triangleright \mathtt{FieldAccess}}\ (\textsc{BlockExp})$$

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression}}{\Gamma, \mathtt{C} \vdash \mathtt{return}\ exp; \triangleright \mathtt{Block}}\ (\textsc{BlockRet})$$

$$\frac{\Gamma, \mathtt{C} \vdash b' \triangleright \mathtt{Block} \qquad b \triangleright \mathtt{Block}}{\Gamma, \mathtt{C} \vdash b'\ b \triangleright \mathtt{Block}}\ (\textsc{Block})$$

Figura 4.17. `Block`.

utilizamos a regra Exp descrita na Figura 4.18. Essa regra diz que uma expressão pode ser uma chamada de método ou um acesso a atributo. As regras que validam chamadas de métodos e acessos a atributos estão descritas nas Figuras 4.19 e 4.20, respectivamente.

Para uma chamada de método ser válida ela pode possuir duas formas: "*id*`()`" ou "*exp.id*()", onde *id* é do tipo `Identifier` e *exp* é d tipo Expression. Para um acesso a atributo ser válido ele deve possuir a forma "*v.i*", onde *v* e *i* são ambos do tipo `Identifier`. Como a expressão que estamos checando, `this.cod`, é uma declaração de acesso a atributo, a checagem do corpo do método está terminada. Podemos voltar para a regra BlockRet e ver que todas as premissas foram satisfeitas, garantindo que o corpo é bem tipado.

A verificação do método `setCod(int #p)` é feita de maneira semelhante à verificação do método `getCod()`. Porém, como esse método possui um parâmetro, devemos garantir que o mesmo é bem tipado. A verificação do parâmetro do método é bastante simples, sendo realizada através da regra FormParam da Figura 4.21. Essa regra define

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{MethodCall} \vee exp \triangleright \mathtt{FieldAccess}}{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression}}\ (\textsc{Exp})$$

Figura 4.18. `Expression`.

$$\frac{\Gamma \vdash id \triangleright \texttt{Identifier}}{\Gamma \vdash id() \triangleright \texttt{MethodCall}} \quad (\textsc{MetCall})$$

$$\frac{\Gamma \vdash exp \triangleright \texttt{Expression} \qquad id \triangleright \texttt{Identifier}}{\Gamma \vdash exp.id() \triangleright \texttt{MethodCall}} \quad (\textsc{MetCallExp})$$

Figura 4.19. `MethodCall`.

$$\frac{\Gamma \vdash v, i \triangleright \texttt{Identifier} \wedge i \neq \texttt{this}}{\Gamma \vdash v.i \triangleright \texttt{FieldAccess}} \quad (\textsc{FAccess})$$

Figura 4.20. `FieldAccess`.

uma declaração de parâmetro como sendo uma construção na forma "*t id*", onde *t*, caso pertença ao domínio de Γ e *C*, deve possuir o tipo `Type`, caso contrário, *t* pode ser qualquer `String` Java válida, contando que seja diferente de `void`, e *id*, caso pertença ao domínio de Γ e *C*, deve possuir o tipo `Indentifier`.

Após finalizada a checagem do *template* fonte da transformação, vamos checar apenas a declaração iterativa e as declarações executáveis do *template* destino. As demais declarações do *template* destino são quase idênticas as declarações do *template* fonte e a regras são aplicadas de maneira bastante similar.

Para verificarmos a validade de uma declaração iterativa, utilizamos a regra ForAll descrita na Figura 4.22. Uma declaração iterativa é válida se ela possuir a forma "`forall` *u* `in` *s*{*decs*}", onde *s* é um conjunto de elementos, *u* é o tipo desses elementos e *decs* é do tipo tt ForAllBlock, pois representa as declarações contidas no bloco `forall`. É válido notar que *u* será inserido no contexto Γ e *C* associada a seu tipo durante a checagem do bloco `forall`. Analisando a transformação, vemos que, como a variável `#atts` é do tipo FiledDeclarationSet, a variável `#a` será inserida em Γ mapeando em `FieldDeclaration`.

A regra da Figura 4.23 define o que é um bloco `forall` válido. De maneira geral, um bloco `forall` pode ser composto por declarações de atributo ou declarações de métodos. Analisando a transformação, vemos que a declaração contida no bloco `forall` é uma declaração de método. Porém, essa declaração de método contém declarações executáveis

$$\frac{\begin{array}{c}\Gamma, \texttt{C} \vdash t \in \texttt{dom}(\Gamma) \Rightarrow t \triangleright \texttt{Type} \wedge t \neq \texttt{void} \\ \Gamma, \texttt{C} \vdash id \in \texttt{dom}(\Gamma) \Rightarrow id \triangleright \texttt{Identifier}\end{array}}{\Gamma, \texttt{C} \vdash t\ id \triangleright \texttt{FormalParameter}} \quad (\textsc{FormParam})$$

Figura 4.21. `FormalParameter`.

$$\frac{\Gamma, \mathtt{C} \vdash s \triangleright \mathtt{T}^* \qquad \Gamma, \mathtt{C} \oplus (u \mapsto \mathtt{T}) \vdash decs \triangleright \mathtt{ForAllBlock}}{\Gamma \vdash \mathtt{forall}\ u\ \mathtt{in}\ s\{decs\} \triangleright \mathtt{ForAllDeclaration}} \quad (\textsc{ForAll})$$

Figura 4.22. `ForAllDeclaration`.

$$\frac{\Gamma \vdash fd \triangleright \mathtt{FieldDeclaration}}{\Gamma \vdash fd \triangleright \mathtt{ForAllBlock}} \quad (\textsc{ForAllBlockFD})$$

$$\frac{\Gamma \vdash fd \triangleright \mathtt{MethodDeclaration}}{\Gamma \vdash fd \triangleright \mathtt{ForAllBlock}} \quad (\textsc{ForAllBlockMD})$$

$$\frac{\Gamma \vdash d' \triangleright \mathtt{ForAllBlock} \qquad d \triangleright \mathtt{ForAllBlock}}{\Gamma \vdash d', d \triangleright \mathtt{ForAllBlock}} \quad (\textsc{ForAllDecs})$$

Figura 4.23. `ForAllBlock`.

no lugar do tipo de retorno, do nome do método e da expressão da cláusula `return`. Sendo assim, devemos verificar se cada uma dessas expressões executáveis é válida. Para tal utilizamos a regra ExecExp descrita na Figura 4.24. A premissa dessa regra é diferente das demais, pois a verificação de expressão dentro da declaração executável deve ser feita apenas através do sistema de tipos de Java, o que está demonstrado pelo símbolo $\models$. Deve ser feita uma pequena modificação em Γ para poder aplicar a verificação de tipos Java, essa mudança é feita pela função $f(\Gamma)$.

Após aplicadas todas as regras de tipo podemos garantir que a transformação da Figura 3.4 está bem tipada não irá gerar código Java incorreto. Na seção a seguir vamos descrever alguns casos em que o sistema de tipos detecta erros nas transformações.

4.4.1 Erros Detectados

A transformação da Figura 4.25 demonstra alguns dos erros que as regras de tipos podem detectar. Por exemplo, a variável `#C` não pode ser usada no lugar do nome de uma classe, pois em Γ ela está associada ao tipo `TypeName`. Esse erro seria detectado pela

$$\frac{\mathrm{f}(\Gamma) \models exp \triangleright T}{\Gamma \vdash \# < exp > \# \triangleright \mathtt{T}} \quad (\textsc{ExecExp})$$

Figura 4.24. `Tipo da declaração executável.`

Template Fonte

```
jvar
  TypeName:#C;
  TypeIdentifier:#D;
  FieldDeclaration:#a;
in
class #C extends #D {

  #a;

}
```

Template Destino

```
class #D{

  #a;

  #<#a.getType()># get(){
    return #a;
  }

}
```

Figura 4.25. `Transformação com erros.`

regra ClassParentDec, pois ela estabelece que o identificador correspondendo ao nome da superclasse, se estiver contido em Γ deve estar associado ao tipo `TypeIdentifier`. Outro erro, agora detectado pela regra BlockRet, refere-se à declaração de retorno do método `get()`. Para a declaração ser válida é necessário termos uma expressão logo após a palavra chave `return`, porém, a variável `#a`, no contexto Γ está associada ao tipos `FieldDeclaration` e não ao tipo `Expression`.

CAPÍTULO 5

IMPLEMENTAÇÃO

Com o objetivo de verificar a aplicabilidade das regras de tipos e garantir a corretude das transformações, estamos implementando o sistema de tipos da linguagem JaTS a partir da formalização descrita no Capítulo 4.

A arquitetura da implementação do sistema de tipos é composta de dois módulos principais: um verificador de tipos para Java e um verificador de tipos para JaTS, que está sendo desenvolvido a partir do primeiro. Como a linguagem JaTS é um superconjunto da linguagem Java, primeiro implementamos o verificador de tipos para Java. Utilizando esse verificador, podemos garantir, por exemplo, que o programa fonte dado como entrada da transformação está bem tipado, antes de aplicarmos a transformação.

A Figura 5.1 descreve os relacionamentos entre os dois módulos do sistema e como eles agem sobre o programa fonte e sobre os *templates*. O verificador de tipos para Java realiza a checagem do programa fonte, o verificador de tipos para JaTS realiza a checagem das transformações e utiliza o verificador de tipos para Java.

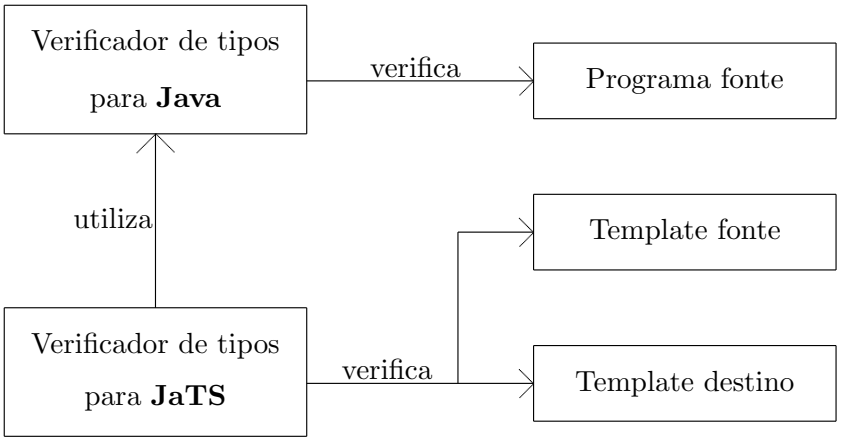

Figura 5.1. Relacionamentos.

O fluxo de execução de JaTS foi modificado para que fosse inserida a fase análise semântica, onde é feita a verificação de tipos. Antes, as entradas (programa fonte, *template* fonte e *template* destino) sofriam um processo de *parsing* e eram passadas diretamente para a fase de transformação. Em seguida, era realizado o processo de *pretty-printing* e gerado o código Java. Agora, após passar pelo processo de *parsing* a entrada passa pela fase de análise semântica antes de entrar na fase de transformação, constituída pelas etapas de casamento, substituição e processamento. O novo fluxo de eventos está demonstrado na Figura 5.2.

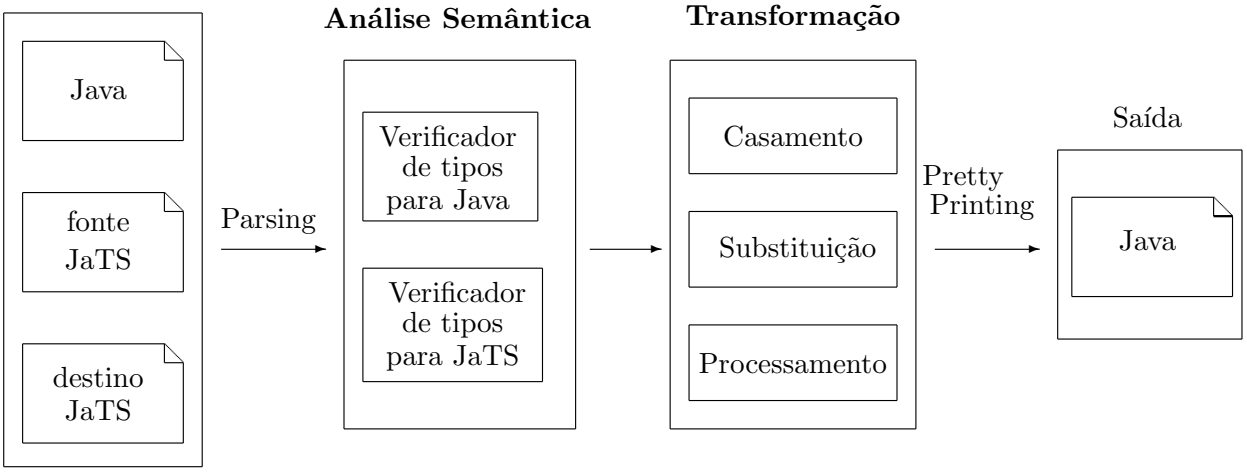

Figura 5.2. Fluxo de eventos.

Na Seção 5.1, mostramos onde o verificador de tipos se encaixa na arquitetura de JaTS e descreveremos a mesma brevemente. A seguir, na Seção 5.2, descrevemos apenas a arquitetura do verificador de tipos para Java, pois seu desenvolvimento já está completo. Por fim descrevemos tecnologias importantes utilizadas na implementação.

5.1 NOVA ARQUITETURA DE JATS

A arquitetura do sistema JaTS (Figura 5.3) é composta de dois componentes principais: `JaTSIO` e `JaTSEngine`. Um subsistema que realiza a verificação de tipos para Java, `JaTSTypeChecker`, foi incorporado ao componente `JaTSEngine`.

O módulo `JaTSIO` é composto por em dois subsistemas: `JaTSParser` e `JaTSPrinter`. O primeiro é responsável pela análise sintática e pela geração das árvores sintáticas de cada padrão de entrada (programa fonte, *template* fonte e *template* destino). O segundo, `JaTSPrinter`, realiza a impressão da árvore gerada pela transformação.

`JaTSEngine`, era apenas responsável pela aplicação das transformações, agora também é responsável pela análise semântica dos padrões de entrada. O `JaTSEngine` agora é composto de quatro subsistemas: `JaTSTypeChecker`, `JaTSMatcher`, `JaTSReplacer` e `JaTSProcessor`. O `JaTSTypeChecker` realiza a verificação de tipos do programa fonte. E, futuramente, realizará a verificação de tipos completa dos *templates*. O `JaTSMatcher` tenta realizar o casamento da árvore sintática de tipos JaTS no *template* fonte da transformação com a árvore sintática dos tipos do programa fonte. Caso seja bem sucedido, gera o mapeamento resultado composto de pares (variável JaTS, valor) construídos durante o casamento. O `JaTSReplacer` percorre as árvores sintáticas do *template* destino realizando substituindo as ocorrências de variáveis JaTS pelos valores a elas mapeados. No final, o `JaTSProcessor` realiza o processamento de pré-condições, declarações executáveis e declarações iterativas.

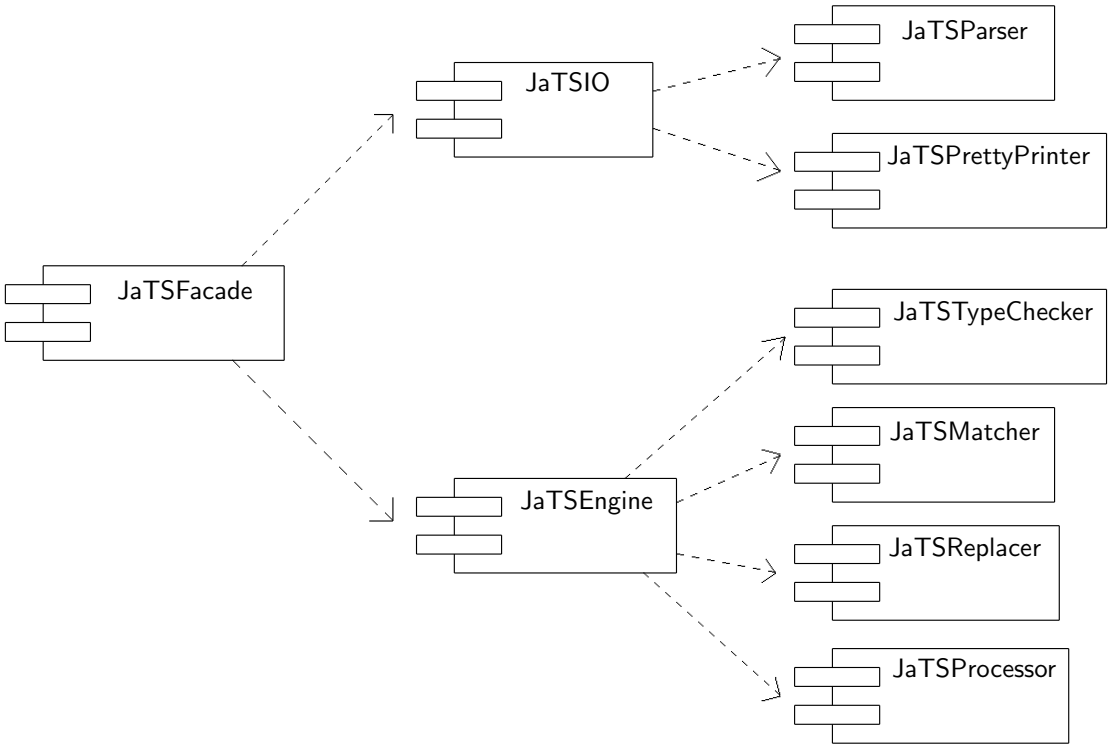

Figura 5.3. Arquitetura.

5.2 ARQUITETURA DO VERIFICADOR DE TIPOS

O subsistema `JaTSTypeChecker` é composto por três pacotes: `typeChecker.loader`, `typeChecker.errorHandler` e `typeChecker`. O primeiro contém as classes que implementam a verificação de tipos. Uma delas, `Enviroment` implementa a tabela de símbolos e contém informações sobre os tipos Java que serão analisados. Esse pacote também contém um aspecto `TypeCheckingAspect`. Quando descrevermos as tecnologias que estão sendo utilizadas, esse aspecto ficará mais claro. O pacote `typeChecler.errorHandler` contém as classes que realizam o tratamento de erros e o pacote `typeChecker.loader` contém as classes que carregam arquivos .class. Esse último pacote é necessário, pois precisamos obter informações de arquivos `.class` em bibliotecas, como métodos e atributos que o programa sendo checado esteja utilizando. Esse pacote foi implementado utilizando uma biblioteca chamada BCEL. A Seção 5.3.2 a descreve. A Figura 5.4 contém os pacotes do sub-sitema JaTSTypeChecker.

5.3 TECNOLOGIAS UTILIZADAS

As seções a seguir tratam de importantes tecnologias que contribuíram para o desenvolvimento do trabalho.

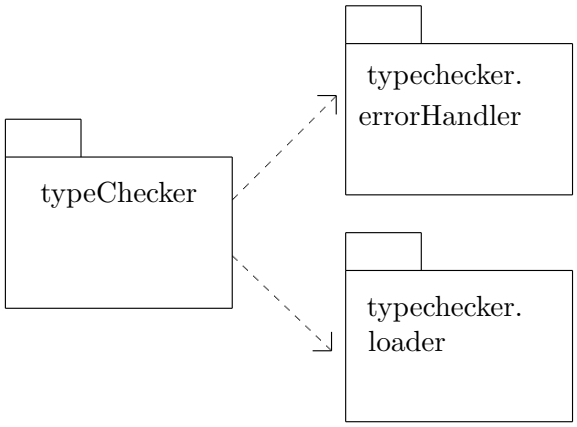

Figura 5.4. Verificador de tipos.

5.3.1 AspectJ

AspectJ [Lad03] é uma extensão de Java orientada a aspectos. A programação orientada a aspectos (AOP [K+97]) procura solucionar algumas ineficiências da orientação a objetos, como o entrelaçamento e espalhamento de código com diferentes propósitos. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção dos sistemas extremamente difícil. AOP auxilia na modularização de funcionalidade que afetam diferentes partes do sistema, chamadas preocupações ortogonais (crosscutting concern). Além do mais, AOP aumenta a coesão e melhora a extensibilidade do programa [SB02].

Usar aspectos tornou-se uma abordagem bastante intuitiva para a implementar a verificação de tipos. Uma maneira bem tradicional de implementar verificação de tipos em linguagens orientadas a objetos é usar o padrão de projeto Visitor[GHJV95]. Acontece que, de acordo com Hannemann [HK02], há benefícios em se implementar o padrão de projeto Visitor usando aspectos, pois é promovida uma melhor separação de interesses. Ao invés de usarmos um Visitor, criamos o aspecto `TypeCheckingAspect`. Esse aspecto insere em cada nó da árvore o método `typeCheck()`, sem mexer na implementação dos nós, o que forneceu a separação de interesses no nosso caso.

5.3.2 BCEL

O BCEL (Byte Code Engineering Library) provê aos seus usuários uma maneira conveniente de analisar, criar e manipular arquivos compilados (*bytecodes*) da linguagem Java [bce]. "O BCEL provê recursos poderosos para a manipulação de arquivos `.class` e é usado com sucesso em diversos projetos acadêmicos e comerciais [jbo].

Como precisávamos obter informações a partir arquivos compilados (arquivos com a extensão `.class`) presentes nas bibliotecas de Java, utilizamos o BCEL para implementar parte do verificador de tipos para Java. Especificamente, o BCEL foi utilizado para implementar o pacote `typeChecker.loader`. O BCEL é usado para ler tipos armazenados como arquivos `.class`. Uma vez lidos, esses tipos são convertidos da representação usada pelo BCEL para a que é empregada por JaTS. Se tivéssemos optado por não utilizá-lo, certamente o trabalho teria se tornado muitas vezes mais complexo.

CAPÍTULO 6

CONSIDERAÇÕES FINAIS

Neste trabalho vimos que um sistema de tipos provê poder e regularidade às linguagens. Vimos que a linguagem JaTS, por possuir apenas uma noção de tipos definida de maneira implícita e informal, apresenta diversos problemas. A fim de poder identificar esses problemas, realizamos vários testes com a linguagem. Descrevemos os problemas na sintaxe e os erros encontrados durante os testes. Mostramos diversos exemplos de transformações válidas que geravam código Java incorreto.

A fim de solucionar parte dos problemas da linguagem, definimos um sistema de tipos formal para um subconjunto significativo de JaTS. Com esse sistema de tipos, podemos garantir parcialmente que uma transformação bem tipada não gera código Java incorreto. Apresentamos, e explicamos brevemente, a sintaxe e as regras de tipos e subtipo que constituem o sistema de tipos formalizado.

Descrevemos parte da implementação da linguagem e algumas tecnologias importantes que foram utilizadas. A maioria das regras de tipos ainda não foram implementadas, porém, como já temos o sistema de tipos formalizado, podemos facilmente mapear as regras em código.

Sem o sistema de tipos definido neste trabalho, a implementação da verificação de tipos da linguagem JaTS não poderia ser feita de maneira concisa. Várias questões importantes surgiram durante a criação das regras que seriam esquecidas se partíssemos direto para a implementação. Essa definição formal do sistema de tipos da linguagem JaTS é de crucial importância para a implementação da verificação de tipos.

CAPÍTULO 7

TRABALHOS FUTUROS

O sistema de tipos definido neste trabalho ainda não possui meios para solucionar o problema descrito na Seção 3.3. Ele ignora grande parte da semântica de Java presente na transformação, checando apenas a sintaxe e as declarações da linguagem JaTS. Como trabalho futuro, pretendemos estender o sistema de tipos a fim de incluir a verificação da semântica de Java presente na transformação. Para tal, precisaríamos modificar primeiramente a construção do contexto Γ, pois informações extras precisam ser armazenadas. Para a transformação da Figura 4.4 (reproduzida na Figura 7.2), deveríamos ter como Estado A de Γ algo parecido com o que está descrito na Figura 7.1. Os identificadores deveriam estar associados não somente a um tipo JaTS, mas sim a um par `(P, Q)`, onde `P` é o tipo Java do identificador, caso possua tipo um Java, e `Q` é o tipo JaTS. Algumas variáveis JaTS possuem uma "?" no lugar do tipo Java. Isso ocorre pois o valor que essa variável pode assumir só será preenchido durante a aplicação da transformação.

Estado A

`boolean`	↦	`( , Type)`
`char`	↦	`( , Type)`
`int`	↦	`( , Type)`
`#C`	↦	`(?, TypeIdentifier)`
`#atts`	↦	`(?, FieldDeclarationSet)`
`#p`	↦	`(int, FormalParameter)`
`(#C, cod)`	↦	`(int, Identifier)`
`(#C, getCod)`	↦	`(int, Identifier)`
`(#C, setCod)`	↦	`(void, Identifier)`
`(#C, mudarCod)`	↦	`(void, Identifier)`
`(#C, this)`	↦	`(#C, Identifier)`

Figura 7.1. Novo contexto Γ.

Outro trabalho inclui a continuação do desenvolvimento do sistema de tipos da linguagem. É necessário implementar todas as regras de tipos definidas neste trabalho e as modificações que surgirão quando concluirmos o trabalho descrito acima.

Template Fonte

```
jvar
  TypeIdentifier:#C;
  FieldDeclarationSet:#atts;
  Identifier:#p;
in
class #C #[extends Object]#{

  int cod;
  #atts;

  int getCod(){
    return this.cod;
  }

  void setCod(int #p){
    this.cod = #p;
  }

}
```

Template Destino

```
class #C{

  int cod;
  #atts;

  void mudarCod(String str){
    this.cod = str;
  }

  forall #a in #atts {

    #< #a.getType()># #< #a.getName().addPrefix("get") ># (){
      return this.#< #a.getName() >#;
    }

  }

}
```

Figura 7.2. Transformação JaTS.

APÊNDICE A

TIPOS DE VARIÁVEIS JATS

As variáveis JaTS podem ser declaradas como pertencendo a um determinado tipo. Nesta seção descrevemos brevemente os tipos possíveis para as variáveis JaTS.

Tipo	**Descrição**
`ConstructorDeclaration`	Uma declaração de construtor.
`ConstructorDeclarationSet`	Um conjunto de declarações de construtor.
`Expression`	Qualquer expressão de Java.
`ExpressionList`	Uma lista de expressões. Normalmente corresponde a uma lista de argumentos.
`FieldDeclaration`	Uma declaraçãoo de atributo.
`FieldDeclarationSet`	Um conjunto de declarações de atributo.
`Identifier`	Um identificador. Uma sequência de caracteres iniciada por uma letra ou pelos símolos '_' e '$'.
`ImportDeclaration`	Uma declaração de importação.
`ImportDeclarationSet`	Um conjunto de declarações de importação.
`MethodDeclaration`	Uma declaração de método.
`MethodDeclarationSet`	Um conjunto de declarações de método.
`ModifierList`	Uma lista de modificadores.
`Name`	Uma nome. Uma lista de identificadores.
`NameList`	Uma lista de nomes. Normalmente nomes de tipos como nas cláusulas throws, extends e implements.
`PackageDeclaration`	Uma declaração de pacote.
`Parameter`	Um parâmetro de um método ou construtor.
`ParameterList`	A lista de parâmetros de um método ou construtor.

Tipo	**Descrição**
`Type`	O tipo de uma entidade. Pode ser, por exemplo, o tipo de retorno de um método, o tipo de um atributo, o tipo de um operador de `cast`, etc.
`VariableDeclarator`	Um declarador de variável. Corresponde ao identificador de uma variável e sua inicialização opcional.

APÊNDICE B

SISTEMA DE TIPOS DE JATS

Aqui encontram-se reunida toda a formalização do sistema de tipos de JaTS: a sintaxe utilizada, as regras de subtipo e as regras de tipo.

B.1 SINTAXE

B.1.1 Identificadores

```
Identifier      ::=  JVariable | Word | ExecutableDeclaration
JVariable       ::=  '#'Word
Word            ::=  Letter (Letter | Digit)*
Name            ::=  Identifier | Name'.'Identifier
Type            ::=  PrimitiveType | TypeName('[]')*
                     | ExecutableDeclaration
TypeName        ::=  TypeIdentifier | Name'.'TypeIdentifier
TypeIdentifier  ::=  Identifier
PrimitiveType   ::=  'boolean' | 'char' | 'int'
```

B.1.2 Expressões

```
Expression   ::=  MethodCall | FieldAccess | Identifier
MethodCall   ::=  (Expression'.'Identifier | Identifier)
                  '('(Expression | ε)')'
FieldAccess  ::=  Identifier'.'Identifier
```

B.1.3 Declaraçõoes

```
JaTSLeftHandSide        ::=  (ε | JVarDeclarationSet) ClassDeclaration
JaTSRightHandSide       ::=  ClassDeclaration
JVarDeclarationSet      ::=  'jvar' JVarDeclaration
                             (JVarDeclaration)* 'in'
JVarDeclaration         ::=  JVariable ':'  TypeIdentifier ';'
ClassDeclaration        ::=  'class' TypeIdentifier
                             (ε | ParentDeclaration)
                             '{' ClassBodyDeclarations '}'
ParentDeclaration       ::=  'extends' TypeName
                             | '#['`extends' TypeName`]#'
ClassBodyDeclarations   ::=  Declaration (Declaration)*
Declaration             ::=  FieldDeclaration
                             | MethodDeclaration
                             | ForAllDeclaration
FieldDeclaration        ::=  Type Identifier`;'
MethodDeclaration       ::=  Type Identifier '('FormalParameter`)'
                             '{'Block`}'
FormalParameter         ::=  Type Identifier
Block                   ::=  (Name '=' Expression`;')*
                             (ε | 'return' Expression`;')
ForAllDeclaration       ::=  'forall' JVariable 'in' JVariable
                             '{' ForAllBlock '}'
ForAllBlock             ::=  ForAllBlockDeclaration
                             (ForAllBlockDeclaration)*
ForAllBlockDeclaration  ::=  FieldDeclaration
                             | MethodDeclaration
                             | ExecutableDeclaration
ExecutableDeclaration   ::=  #< JVariable`.'MethodCall ('.'MethodCall)* >#
```

B.2 REGRAS DE SUBTIPO

`Identifier <: Name`

`TypeIdentifier <: TypeName <: Type`

B.3 REGRAS DE TIPO

$$\frac{\Gamma(id) = \mathtt{T}}{\Gamma \vdash id \triangleright \mathtt{T}} \quad \text{(ID)}$$

$$\frac{\Gamma(C)(id) = \mathtt{T}}{\Gamma, C \vdash id \triangleright \mathtt{T}} \quad \text{(CID)}$$

$$\frac{\Gamma \vdash jvSet \triangleright \mathtt{JVarDeclarationSet} \qquad cd \triangleright \mathtt{ClassDeclaration}}{\Gamma \vdash jvSet\ cd \triangleright \mathtt{JaTSLeftHandSide}} \quad \text{(LSPROG)}$$

$$\frac{\Gamma \vdash cd \triangleright \mathtt{ClassDeclaration}}{\Gamma \vdash cd \triangleright \mathtt{JaTSRightHandSide}} \quad \text{(RSPROG)}$$

$$\frac{\Gamma \vdash var \triangleright \mathtt{JVarDeclaration}}{\Gamma \vdash \mathtt{jvar}\ \overline{var}\ \mathtt{in} \triangleright \mathtt{JVarDeclarationSet}} \quad \text{(JVARDECSET)}$$

$$\frac{\Gamma \vdash v \triangleright \mathtt{T}}{\Gamma \vdash \mathtt{T} : v; \triangleright \mathtt{JVarDeclaration}} \quad \text{(JVAR)}$$

$$\frac{\begin{array}{c}\Gamma, T \vdash classBody \triangleright \mathtt{ClassBodyDeclarations} \\ \Gamma \vdash T \in \mathtt{dom}(\Gamma) \Rightarrow T \triangleright \mathtt{TypeIdentifier}\end{array}}{\Gamma \vdash \mathtt{class}\ T\{classBody\} \triangleright \mathtt{ClassDeclaration}} \quad \text{(CLASSDEC)}$$

$$\frac{\begin{array}{c}\Gamma, T \vdash classBody \triangleright \mathtt{ClassBodyDeclarations} \\ \Gamma \vdash (T \in \mathtt{dom}(\Gamma) \Rightarrow T \triangleright \mathtt{TypeIdentifier}) \qquad parentDec \triangleright \mathtt{ParentDeclaration}\end{array}}{\Gamma \vdash \mathtt{class}\ T\ parentDec\{classBody\} \triangleright \mathtt{ClassDeclaration}} \quad \text{(CLASSPARENTDEC)}$$

$$\frac{\Gamma \vdash T \in \mathtt{dom}(\Gamma) \Rightarrow T \triangleright \mathtt{TypeName}}{\Gamma \vdash \mathtt{extends}\ T \triangleright \mathtt{ParentDeclaration}} \quad \text{(PARENTDEC)}$$

$$\frac{\Gamma \vdash e \triangleright \mathtt{T}}{\Gamma \vdash \#[e]\# \triangleright \mathtt{T}} \quad (\textsc{OptDec})$$

$$\frac{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{FieldDeclaration}}{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{ClassBodyDeclarations}} \quad (\textsc{DecsFD})$$

$$\frac{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{MethodDeclaration}}{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{ClassBodyDeclarations}} \quad (\textsc{DecsMD})$$

$$\frac{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{ForAllDeclaration}}{\Gamma, \mathtt{C} \vdash fd \triangleright \mathtt{ClassBodyDeclarations}} \quad (\textsc{DecsFA})$$

$$\frac{\Gamma, \mathtt{C} \vdash d' \triangleright \mathtt{ClassBodyDeclarations} \qquad d \triangleright \mathtt{ClassBodyDeclarations}}{\Gamma, \mathtt{C} \vdash d'\ d \triangleright \mathtt{ClassBodyDeclarations}} \quad (\textsc{Decs})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \vdash t \in \mathtt{dom}(\Gamma) \Rightarrow t \triangleright \mathtt{Type} \wedge t \neq \mathtt{void} \\ \Gamma, \mathtt{C} \vdash id \triangleright \mathtt{Identifier}\end{array}}{\Gamma, \mathtt{C} \vdash t\ id; \triangleright \mathtt{FieldDeclaration}} \quad (\textsc{FieldDec})$$

$$\frac{\Gamma \vdash t \triangleright \mathtt{T}}{\Gamma \vdash \mathtt{t} \triangleright \mathtt{T}^*} \quad (\textsc{SetsUnit})$$

$$\frac{\Gamma \vdash t' \triangleright \mathtt{T} \qquad t \triangleright \mathtt{T}^*}{\Gamma \vdash \mathtt{t'\ t} \triangleright \mathtt{T}^*} \quad (\textsc{Sets})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \oplus (\mathtt{this} \mapsto \mathtt{Indentifer}, id \mapsto \mathtt{Identifier}) \vdash block \triangleright \mathtt{Block} \\ \Gamma, \mathtt{C} \vdash (ret \in \mathtt{dom}(\Gamma) \Rightarrow ret \triangleright \mathtt{Type}) \qquad met \triangleright \mathtt{Identifier} \qquad t\ id \triangleright \mathtt{FormalParameter}\end{array}}{\Gamma, \mathtt{C} \vdash ret\ met(t\ id)\{block\} \triangleright \mathtt{MethodDeclaration}} \quad (\textsc{MethodDec})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression} \\ \Gamma, \mathtt{C} \vdash (id \triangleright \mathtt{Identifier} \wedge id \neq \mathtt{this}) \vee id \triangleright \mathtt{FieldAccess}\end{array}}{\Gamma, \mathtt{C} \vdash id = exp; \triangleright \mathtt{Block}} \quad (\textsc{BlockExp})$$

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression}}{\Gamma, \mathtt{C} \vdash \mathtt{return}\ exp; \triangleright \mathtt{Block}} \quad (\textsc{BlockRet})$$

$$\frac{\Gamma, \mathtt{C} \vdash b' \triangleright \mathtt{Block} \qquad b \triangleright \mathtt{Block}}{\Gamma, \mathtt{C} \vdash b'\ b \triangleright \mathtt{Block}} \quad (\textsc{Block})$$

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{MethodCall} \vee exp \triangleright \mathtt{FieldAccess}}{\Gamma, \mathtt{C} \vdash exp \triangleright \mathtt{Expression}} \quad (\textsc{Exp})$$

$$\frac{\Gamma \vdash id \triangleright \mathtt{Identifier}}{\Gamma \vdash id() \triangleright \mathtt{MethodCall}} \quad (\textsc{MetCall})$$

$$\frac{\Gamma \vdash exp \triangleright \mathtt{Expression} \qquad id \triangleright \mathtt{Identifier}}{\Gamma \vdash exp.id() \triangleright \mathtt{MethodCall}} \quad (\textsc{MetCallExp})$$

$$\frac{\Gamma \vdash v, i \triangleright \mathtt{Identifier} \wedge i \neq \mathtt{this}}{\Gamma \vdash v.i \triangleright \mathtt{FieldAccess}} \quad (\textsc{FAccess})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \vdash t \in \mathtt{dom}(\Gamma) \Rightarrow t \triangleright \mathtt{Type} \wedge t \neq \mathtt{void} \\ \Gamma, \mathtt{C} \vdash id \in \mathtt{dom}(\Gamma) \Rightarrow id \triangleright \mathtt{Identifier}\end{array}}{\Gamma, \mathtt{C} \vdash t\ id \triangleright \mathtt{FormalParameter}} \quad (\textsc{FormParam})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \vdash s \triangleright \mathtt{T}^* \\ \Gamma, \mathtt{C} \oplus (u \mapsto \mathtt{T}) \vdash decs \triangleright \mathtt{ForAllBlock}\end{array}}{\Gamma \vdash \mathtt{forall}\ u\ \mathtt{in}\ s\{decs\} \triangleright \mathtt{ForAllDeclaration}} \quad (\textsc{ForAll})$$

$$\frac{\Gamma \vdash fd \triangleright \mathtt{FieldDeclaration}}{\Gamma \vdash fd \triangleright \mathtt{ForAllBlock}} \quad (\textsc{ForAllBlockFD})$$

$$\frac{\Gamma \vdash fd \triangleright \mathtt{MethodDeclaration}}{\Gamma \vdash fd \triangleright \mathtt{ForAllBlock}} \quad (\textsc{ForAllBlockMD})$$

$$\frac{\Gamma \vdash d' \triangleright \texttt{ForAllBlock} \qquad d \triangleright \texttt{ForAllBlock}}{\Gamma \vdash d', d \triangleright \texttt{ForAllBlock}} \quad \text{(ForAllDecs)}$$

$$\frac{\mathrm{f}(\Gamma) \models exp \triangleright T}{\Gamma \vdash \# < exp > \# \triangleright \texttt{T}} \quad \text{(ExecExp)}$$

REFERÊNCIAS BIBLIOGRÁFICAS

[AUS86] Alfred Aho, Jeffrey Ullman, and Ravi Sethi. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[bce] Byte code engineering library. Endereço: http://jakarta.apache.org/bcel/.

[BS00] Paulo Borba and Augusto Sampaio. Basic laws of rool: an object-oriented language. In *Anais do III Workshop de Métodos Formais*, pages 33—44, João Pessoa - PB, Brasil, 2000.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[Car04] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, Boca Raton, FL, 2nd edition, 2004.

[CB01] Fernando Castor and Paulo Borba. A language for specifying java transformations. In *Anais do V Simpósio Brasileiro de Linguagens de Programação*, pages 236–251, Curitiba - PR, Brasil, 2001.

[COS+01] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. Jats: A java transformation system. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 374–379, Rio de Janeiro - RJ, Brasil, 2001.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Survey*, 17(4):471–522, December 1985.

[CX03a] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Partial evaluation and semantics-based program manipulation*, pages 20–28, San Diego - California, USA, 2003.

[CX03b] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 275–286, Uppsala, Suécia, 2003.

[FH99] Marcelo Felix and Edward Hausler. Let: Uma linguagem para especificar transformações. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, pages 109–121, Porto Alegre - RS, Brasil, 1999.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison Wesley, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Oriented Object Software*. Addison Wesley, 1995.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 161–173, November 2002.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *Proceedings of the ACM 1999 Symposium on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146, Denver - Colorado, USA, 1999.

[jbo] Jboss application server. Endereço: http://www.jboss.org/products/jbossas.

[K+97] Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242. Springer Verlag LNCS 1241, 1997.

[Lad03] Ramnivas Laddad. *AspectJ in Action*. Manning, Greenwich, CT, USA, 2003.

[Mor90] Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice-Hall International, 1990.

[Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992.

[RJ99] Donald Bradley Roberts and Ralph Johnson. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[SB02] Sérgio Soares and Paulo Borba. Aspectj – programação orientada a aspectos. In *Anáis do VI Simpósio Brasileiro de Linguagens de Programação, SBLP 2002*, pages 36–55. PUC-Rio, June 2002.