# Um Sistema de Tipos para uma Linguagem de Transformação

**Alexandra Barros**[1], **Paulo Borba**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 – 50.732-970 – Recife, PE

{abab, phmb}@cin.ufpe.br

***Abstract.*** *We present a type system for JaTS, a language for specifying transformations to Java programs. In the current version of the language, the notion of type already exists, but is implicit and informal. Here we formally specify the type system of JaTS so that programs produced by transformations written in JaTS do not contain syntatic and some semantic errors.*

***Resumo.*** *Este artigo apresenta 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 de maneira implícita e informal. É necessário especificar o sistema de tipos de JaTS com precisão para garantir que programas produzidos por transformações não contém erros sintáticos e alguns erros semânticos.*

## 1. Introdução

É bem sabido que o uso de tipos em linguagens é útil sobre vários aspectos [Cardelli and Wegner, 1985]. Na programação, os tipos podem esclarecer várias questões assim como impor certas limitações que irão ajudar a reforçar a corretude semântica dos programas. O principal propósito de um sistema de tipos é prevenir a ocorrência de erros durante a execução de um programa [Cardelli, 2004]. Neste trabalho, estamos interessados na confiabilidade que um sistema de tipos bem projetado fornece a uma linguagem para especificar transformações de programas. Com esse sistema de tipos queremos garantir a corretude sintática e parte da corretude semântica dos programas gerados. Apesar de não ser apresentada uma prova formal sobre a corretude da transformação, a argumentação informal deixa claro que os programas gerados não apresentam erros de sintaxe e estão livres de alguns erros de tipos.

Transformação de programas é uma técnica poderosa no suporte a atividades de engenharia de software como *refactoring* [Fowler, 2000, Opdyke, 1992], desenvolvimento formal de software [Borba and Sampaio, 2000, Morgan, 1990], geração de código [Felix and Hausler, 1999] e tradução de linguagens [Felix and Hausler, 1999]. Embora transformações de programas auxiliem na resolução de diversos problemas importantes, 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 do desenvolvedor e diminui a chance de que erros sejam introduzidos.

Para implementar transformações de programas, é necessário representá-las em uma metalinguagem. Uma metalinguagem é uma linguagem de programação na qual são construídos programas para representar programas objeto. LET [Felix and Hausler, 1999], TXL [Cordy and Carmichael, 1993], Draco-PUC [Bergmam et al., 1996] e JPearl [Maia and Oliveira, 2002] são exemplos de

metalinguagens. Linguagem objetos são linguagens utilizadas para construir programas objeto onde serão aplicadas as transformações. Alguns exemplos são Java e C#.

As metalinguagens oferecem recursos poderosos e de fácil utilização para a construção de transformações. Porém, a metaprogramação (ou seja, a programação em uma metalinguagem), especialmente em contextos não-tipados, é notoriamente sujeita a erros [Chen and Xi, 2003b]. Alguns trabalhos recentes contribuíram para a formalização de abordagens para a metaprogramação tipada e descreveram como implementar transformações de programas que podem representar o sistema de tipos da linguagem objeto [Chen and Xi, 2003a, Chen and Xi, 2003b]. Porém, a maioria das metalinguagens existentes ainda não possui um sistema de tipos.

Várias ferramentas para transformação de programas já foram implementadas. Castor e Borba [Castor and Borba, 2001] definiram JaTS - uma metalinguagem para especificação de transformações de programas Java, que foi posteriormente implementada por um sistema de mesmo nome [Castor et al., 2001]. 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 da linguagem objeto. Isto diminui o *gap* semântico entre a linguagem objeto e a metalinguagem, facilitando o aprendizado desta última. Além disso, como JaTS foca especificamente na linguagem Java, é possível especificar transformações que levam a semântica da linguagem objeto em consideração [Castor and Borba, 2001], ao invés de transformações puramente sintáticas [Felix and Hausler, 1999, Cordy and Carmichael, 1993]

Apesar das vantagens apresentadas pela linguagem JaTS, ela ainda não possui um sistema de tipos bem definido. Conseqüentemente, um programador pode criar uma transformação que é ambígua ou que gera código Java incorreto. Para contornar esses problemas, é necessário definir um sistema de tipos para JaTS, de modo que erros de tipos em uma transformação (metaprograma) possam ser detectados em tempo de compilação.

Este trabalho define um sistema de tipos para JaTS. Na implementação atual da linguagem, a noção de tipos já existe, mas apenas de maneira implícita e informal. É necessário especificar o sistema de tipos de JaTS com precisão, a fim de impedir a criação de transformações que gerem código Java incorreto. Neste trabalho, nos concentramos em um subconjunto significativo da linguagem JaTS e definimos um sistema de tipos modular que permite que esse subconjunto seja estendido posteriormente para abordar os demais aspectos da linguagem. Para modelar formalmente esse sistema, são usadas regras de tipos [Cardelli, 2004] e um cálculo minimal para Java [Igarashi et al., 1999].

Este trabalho está organizado da seguinte maneira. Primeiro, apresentamos uma descrição informal da sintaxe e da semântica de JaTS através de exemplos simples, mas expressivos. A seguir, descrevemos os problemas presentes na linguagem que permitem a criação de transformações ambíguas e que geram código Java incorreto. Na Seção 4. apresentamos o sistema de tipos da linguagem JaTS através de regras de tipos formais e descrições textuais informais. Na Seção 5. descrevemos os trabalhos relacionados. Por fim, na Seção 6. apresentamos algumas considerações finais e sugestões para trabalhos futuros.

## 2. A linguagem JaTS

Transformações JaTS são escritas em uma linguagem que estende Java com construções de JaTS. O objetivo dessas construções é permitir a manipulação dos programas sendo transformados através de casamento de padrões. Em uma transformação, cada construção

de JaTS corresponde a uma estrutura sintática de Java e casa com essa estrutura. Nesta seção descrevemos a sintaxe e a semântica de algumas das construções de JaTS e como é realizado o casamento de padrões.

Uma transformação em JaTS é composta por uma pré-condição, que deve ser satisfeita para que a transformação seja aplicada, um *template* fonte e um *template* destino. O *template* fonte da transformação é casado com o tipo Java sendo transformado. Conseqüentemente, esses dois *templates* devem possuir estruturas sintáticas similares. O *template* destino define o tipo que será produzido pela transformação e como o código será gerado.

A aplicação de uma transformação é realizada em 3 fases: *parsing*, transformação e *pretty-printing*. A primeira e a última fase são bem conhecidas na comunidade de linguagens de programação. A fase de transformação pode ser sub-dividida em três etapas: casamento, substituição e processamento. A primeira etapa casa a árvore sintática do *template* fonte com a árvore sintática do tipo Java que está sendo transformado. Um nó do *template* fonte casa com um nó do tipo Java em duas ocasiões: (i) caso eles sejam idênticos; ou (ii) caso o nó do *template* fonte seja uma variável JaTS. Nesta etapa, é produzido um mapeamento de variáveis JaTS para os valores com os quais elas foram casadas. Na segunda etapa, substituição, as ocorrências de variáveis JaTS no *template* destino da transformação são substituídas pelos valores aos quais essas variáveis foram mapeadas na fase de casamento. A última etapa, processamento, consiste em executar algumas estruturas de JaTS no *template* destino da transformação.

A Figura 1 apresenta uma transformação em JaTS que usa algumas das construções da linguagem: variáveis JaTS, declarações opcionais e declarações executáveis. A linguagem JaTS possui outras construções: declarações iterativas, pré-condições e declarações condicionais. Uma descrição de todas as construções existentes na linguagem JaTS está disponível em outro trabalho [Castor and Borba, 2001].

**Template Fonte**

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

**Template Destino**

```
class #C{

  FieldDeclaration:#<#result = #fd ::  #result.removeAcessModifiers();
                     #result.addModifier(''private'')>#

  public #<#fd.getType()># #<#fd.getName().addPrefix(''get'')># (){
    return this.#<#fd.getName()>#;
  }
}
```

**Figura 1: Transformação em JaTS.**

As construções mais simples da linguagem consistem apenas de um identificador Java precedido do caractere '`#`'. Essas construções são chamadas de **variáveis JaTS** e são usadas como *placeholders* nas transformações. Por exemplo, na Figura 1, a variável `#C` é usada como *placeholder* para o nome da classe.

Uma variável JaTS pode ser associada a um tipo que corresponde a uma das construções sintáticas de Java. Em muitos casos, 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. Porém, há situações em que é necessário declarar o tipo de algumas variáveis, a fim de especificar corretamente a semântica pretendida para a transformação. É o que ocorre com a variável JaTS `#fd`, declarada como sendo do tipo `FieldDeclaration`. Sendo assim, ela deverá casar apenas com uma declaração de atributo.

Em algumas transformações, uma estrutura no *template* fonte precisa ser casada com um elemento que pode ou não existir no tipo fonte Java. As **declarações opcionais** permitem que casamentos desse tipo sejam especificados. Para indicar que uma estrutura no *template* fonte da transformação é opcional basta declará-la entre os símbolos "`#[`" e "`]#`". Neste caso, 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. Por exemplo, na Figura 1, o *template* fonte da transformação casa com qualquer classe que estenda ou não `Object` explicitamente. Ou seja, a declaração "`extends Object`" é opcional. Caso a mesma estrutura também aparecesse no template destino, a declaração "`extend Object`" só apareceria no programa gerado se esta também estivesse no programa fonte. Essa característica da linguagem JaTS possibilita a construção de transformações mais genéricas, uma vez que detalhes irrelevantes, no contexto da transformação, podem ser abstraídos. No exemplo da Figura 1, o objetivo é retirar a declaração "`extends Object`" caso ela exista.

As declarações executáveis aparecem apenas no *template* destino de uma transformação, entre os símbolos "`#<`" e "`>#`", e podem extrair e modificar informações das declarações originais, presentes no *template* fonte. Declarações executáveis podem executar métodos em objetos que representam nós da árvore sintática do tipo fonte Java. Essas declarações contêm código Java que deve ser executado com o objetivo de gerar outras declarações ou, de maneira geral, construções Java válidas. Declarações executáveis são processadas na etapa de processamento de uma transformação.

Existem dois tipos de declarações executáveis. O primeiro é chamado **declaração de extração de informação** (DEI). Como exemplo de DEI temos "`#<#fd.getName()>#`". Como `#fd` está associado à declaração de um atributo, a DEI produz como resultado o nome do atributo. O segundo tipo de declaração executável modifica as declarações originais e é chamado de **declarações de modificação de informação** (DMI). Uma DMI faz uso de uma variável temporária para armazenar um clone do objeto sobre o qual deseja-se invocar um ou mais métodos. A declaração "`#result = #fd`" antes do operador "`::`" indica que a variável `#result` será mapeada a uma cópia do valor que foi mapeado a `#fd`. Os métodos no corpo dessa DMI modificam o valor mapeado a `#result`. Depois de sua execução, o valor modificado é retornado como o resultado da DMI.

O *template* fonte da transformação da Figura 1 casa com qualquer classe que contenha apenas um atributo e que estenda ou não a classe `Object` explicitamente. A aplicação dessa transformação ao Programa Objeto apresentado na Figura 2 produz como resultado o Programa Gerado que aparece na mesma figura. Tais programas não são equivalentes e não precisam ser, já que o objetivo de qualquer transformação em JaTS não é realizar um *refactoring*, tipo específico de transformação que preserva o comportamento do programa.

O sistema de tipos descrito neste trabalho aborda um subconjunto significativo da linguagem JaTS: variáveis JaTS, representadas pelo nome `JVar`, declarações executáveis de extração de informação, `ExecutableDeclaration`, e declarações opcionais, que foram restringidas a declarações de superclasse representadas por `ParentDeclaration`. Na sintaxe, as construções `SourceTemplate` e `DestinationTemplate` representam os *templates* fonte e destino respectivamente. Ambos constituem apenas uma declaração de

**Programa Objeto**

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

**Programa Gerado**

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

**Figura 2: Aplicação da transformação.**

classe, ou `ClassDeclaration`. A construção `Identifier` pode ser uma variável JaTS, um identificador Java ou uma DEI; `Name` representa uma lista de identificadores separados por ponto, como um nome qualificado; `Type` pode ser um `Name` ou um tipo primitivo como `int` ou `boolean`. A Figura 3 define a construções sintáticas restantes.

Optamos por esse subconjunto, pois ele constitui uma parte bastante relevante da linguagem e pode ser facilmente estendido para englobar as construções restantes. Por exemplo, como `MethodCall` se assemelha a estruturas de controle como `if` ou `while`, optamos por deixar essas últimas de fora para não tornar o trabalho repetitivo. Também nos baseamos no trabalho de Igarashi [Igarashi et al., 1999] para definir esse subconjunto.

```
ClassDeclaration      ::=  'class' TypeIdentifier
                           (ε | ParentDeclaration)
                           '{' (ClassBodyDeclaration)* '}'
ClassBodyDeclaration  ::=  (FieldDeclaration | MethodDeclaration)*
FieldDeclaration      ::=  Type Identifier';'
MethodDeclaration     ::=  Type Identifier '('FormalParameter')'
                           '{'Block'}'
FormalParameter       ::=  Type Identifier
Block                 ::=  'return' Expression';'
                           | FieldAccess '=' Expression';' Block
Expression            ::=  MethodCall | FieldAccess | Identifier
MethodCall            ::=  (Expression'.'Identifier | Identifier)
                           '('(Expression | ε)')'
FieldAccess           ::=  Identifier'.'Identifier
```

**Figura 3: Subconjunto abordado de declarações e expressões.**

## 3. Deficiências da Linguagem JaTS

O primeiro problema encontrado diz respeito à ambigüidade presente nas declarações de variáveis JaTS. Como foi visto na seção anterior, para algumas variáveis JaTS é opcional a declaração explícita de seu tipo, pois ele pode ser inferido a partir do local onde a variável aparece. Dado que uma mesma variável JaTS pode ser usada mais de uma vez, contanto que o mesmo valor seja associados a ela em todos os casamentos de que participar, a transformação da Figura 4 é considerada válida. Neste exemplo, não é possível determinar se a variável JaTS `#C` é do tipo `Identifier`, pois casa com o nome de uma classe, ou do tipo `Type`, pois também casa com o tipo de um atributo. Essa ambigüidade dificulta a construção de declarações executáveis que utilizem essa variável JaTS, pois os métodos que podem ser chamados a partir dela dependem do seu tipo.

**Template Fonte**

```
class #C extends Name:#D{
   public #C #a;
}
```

**Template Destino**

```
class #D{}
```

**Figura 4: Ambigüidade na definição do tipo da variável `#C` e possibilidade de geração de código incorreto devido a má utilização da variável `#D`.**

Outro problema encontrado na linguagem é referente às variáveis JaTS dos tipos `Identifier`, que representa identificadores, `Name`, que representa nomes qualificados (sequências de identificadores separados por "."), e `Type`, que representa tipos em geral. Esses tipos de varíavel são totalmente compatíveis, ou seja, onde podemos declarar variáveis do tipo `Type` também podemos declarar variáveis dos tipo `Name` ou `Identifier`, e o mesmo ocorre para outras combinações. Essa característica confunde usuários da linguagem e torna as transformações fortemente dependentes do programa objeto dado como entrada. Na figura 4, nenhum erro ocorreria caso o valor mapeado à variável `#D` fosse um identificador, como `ContaAbstrata`. Porém, caso esse valor fosse um nome qualificado, como `banco.conta.ContaAbstrata`, seria gerada uma classe com o nome `banco.conta.ContaAbstrata`, o que não é válido de acordo com a sintaxe de Java [Gosling et al., 1996].

No *template* destino de uma transformação JaTS, é possível adicionar livremente código Java. Como a versão atual da implementação de JaTS não possui qualquer suporte para verificação de tipos Java dentro da transformação, é possível escrever código Java incorreto no *template* destino e esse código será gerado normalmente. Esse problema ocorre independentemente do programa fonte dado como entrada e para solucioná-lo, é preciso fazer também a verificação de tipos Java, além da verificação de tipos JaTS, no *template* destino da transformação antes dela ser aplicada a algum programa objeto. O método `getString()` da Figura 5 exemplifica essa situação.

Os erros descritos até o momento podem ser evitados aplicando a verificação de tipos à transformação antes desta ser aplicada ao programa fonte. No entanto, existe um conjunto de erros bastante interessantes e difíceis de tratar que dependem do programa fonte para ocorrer. Neste caso, faz-se necessária uma verificação de tipos em tempo de transformação, ou seja, quando a transformação estiver sendo aplicada ao programa fonte. Por exemplo, para que não seja gerado código Java incorreto para o método `getDobro()` do *template* destino da Figura 5, a variável JaTS `#t` deve casar com um tipo numérico.

**Template Fonte**

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

**Template Destino**

```
class #C{
   public #t #a;
   public int getDobro(){
      return #a*2;
   }
   public String getString(){
      return 0;
   }
}
```

**Figura 5: Transformação que gera código Java incorreto.**

## 4. Um Sistema de Tipos para JaTS

Na implementação atual de JaTS a noção de tipos já existe, mas foi definida de maneira implícita e informal. Esta seção apresenta uma especificação formal do sistema de tipos de JaTS. A implementação dessa especificação impede a ocorrência dos erros descritos na Seção 3., incluindo os erros que dependem do programa fonte para ocorrer. Optou-se por invalidar transformações que contenham esse tipo de erro, pois as mesmas só podem ser aplicadas a um número restrito de programas fonte. É necessário encorajar a criação de transformações genéricas que possam ser aplicadas a um grande número de programas fonte.A formalização do sistema de tipos irá guiar a implementação deste, tornando-a mais coerente e concisa, pois as regras podem ser facilmente mapeadas em código.

A próxima seção descreve algumas modificações ocorridas na sintaxe da linguagem JaTS para adaptá-la às mudanças causadas pela introdução do sistema de tipos. Por questões de simplicidade, consideramos apenas um subconjunto significativo da linguagem JaTS e não todas as suas construções. Após descrita a nova sintaxe, prosseguimos para a definição das regras nas seções seguintes. O algoritmo para verificação da transformação é descrito informalmente a medida que as regras são apresentadas. As regras de tipo descrevem relações *possui-tipo*, da forma $A \triangleright T$ entre termos $A$ e tipos $T$, e relações *subtipo-de*, da forma $A <: B$ entre tipos. O conjunto de todas as regras de tipos e de subtipos, formam o sistema de tipos de JaTS.

### 4.1. Sintaxe Modificada

A primeira modificação realizada na sintaxe de JaTS soluciona o problema da ambigüidade presente nas declarações de variáveis. O *template* fonte, que antes era formado de uma declaração de classe ou interface, agora é definido como mostra a construção `SourceTemplate` da Figura 6. Na nova sintaxe, variáveis JaTs são declaradas no topo do *template* fonte e têm como escopo a transformação inteira.

```
SourceTemplate      ::=  (ε | JVarDeclarationSet) ClassDeclaration)
JVarDeclarationSet  ::=  'jvar' (JVarDeclaration;)* 'in'
JVarDeclaration     ::=  JVariable ':'  TypeIdentifier';'
```

**Figura 6: Nova sintaxe de um *template* fonte.**

Na Seção 3. foi descrito um sério problema com as variáveis JaTS dos tipos `Type`, `Name` e `Identifier`. Para restringir a permutação dessas variáveis, criamos os tipos `TypeName`, que corresponde ao nome de uma classe, e `TypeFQName`, que corresponde ao nome de um tipo na forma de nome qualificado. Junto com as regras de tipos apresentadas na Seção 4.2., essa sintaxe modificada garante que variáveis destes tipos sempre são usadas da maneira correta. A Figura 7 apresenta novas construções e a modificação feita em `Type`.

```
Type            ::=  PrimitiveType | TypeIdentifier('[]')*
                     | ExecutableDeclaration
TypeIdentifier  ::=  Identifier
TypeFQName      ::=  TypeIdentifier| Name'.'TypeIdentifier
```

**Figura 7: Novas construções para os identificadores da linguagem.**

A Figura 8 apresenta uma transformação que obedece a nova sintaxe de JaTS. O *template* fonte deve casar com um programa fonte que estenda ou não a classe `Object` e

tenha um atributo. O *template* destino remove a cláusula `extends`, caso ela exista, copia o atributo presente no *template* fonte e insere um método `get()` para o atributo. No restante da Seção 4., essa transformação é usada como exemplo para a aplicação das regras de tipo propostas.

**Template Fonte**

```
jvar
   TypeIdentifier:#C;
   FieldDeclaration:#fd;
in
class #C #[extends Object]# { #fd; }
```

**Template Destino**

```
class #C{
   #fd;
   int code;
   public #<#fd.getType()># #<#fd.getName()>#(){
      return this.#fd;
   }
}
```

**Figura 8: Uma transformação JaTS que adere à sintaxe modificada.**

## 4.2. Regras de Subtipo

Para solucionar o problema da permutação de variáveis JaTS dos tipos `Type`, `Name` e `Identifier`, definimos as regras de subtipos apresentadas na Figura 9. O tipos `TypeName` e `TypeFQName` e as regas de subtipos foram necessários porque estas variáveis JaTS, dependendo de sua localização em um *template*, assumem características sintáticas diferentes. Antes, os tipos não representavam bem as características e, além disso, a relação entre eles não estava definida.

```
Identifier <: Name                                  (NAMEID)
TypeIdentifier <: TypeFQName <: Type   (TYPEIDNAME)
```

**Figura 9: Regras de subtipo.**

A regra NAMEID diz que o tipo `Identifier` é subtipo de `Name`, ou seja, uma variável JaTS do tipo `Identifier` pode aparecer em qualquer situação na qual uma variável do tipo `Name` é esperada. O mesmo se aplica à regra TYPEIDNAME. Com as modificações descritas na Seção 4.1., o nome de uma classe passa a casar com uma variável do tipo `TypeIdentifier` e o nome de uma superclasse (especificada em uma cláusula `extends`) com uma variável do tipo `TypeFQName`. Consequentemente, erros como o da transformação da Figura 4 tornam-se detectáveis em tempo de compilação, já que, de acordo com a regra TYPEIDNAME, não é permitido colocar uma variável do tipo `TypeFQName` em um lugar que espera uma variável `TypeIdentifier`.

## 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

```
boolean                    ↦  (_, Type)
char                       ↦  (_, Type)
int                        ↦  (_, Type)
#C                         ↦  (_, TypeIdentifier)
#fd                        ↦  (_, FieldDeclaration)
(#C, code)                 ↦  (int, Identifier)
(#C, #<#fd.getName()>#)    ↦  (#<#fd.getType()>#, Identifier)
```

**Figura 10: Estados inicial de $\Gamma$.**

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 $\Gamma$ 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 $\Gamma$).

Para o sistema de tipos da linguagem JaTS, foi definido um contexto $\Gamma$ um pouco diferente do descrito acima. Como JaTS é uma meta-linguagem, foi realizada uma adaptação no contexto: ao invés de conter mapeamentos de variáveis para tipos, o contexto $\Gamma$ contém dois tipos de mapeamentos: de classes para identificadores (nomes de tipos primitivos, métodos, atributos e variáveis JaTS) que, por sua vez, leva a um par de tipos, e mapeamentos de identificadores para um par de tipos, quando estes não possuem escopo de classe. Para checar por completo uma transformação, é necessário saber o tipo em Java e também o tipo em JaTS de cada identificador. Esses pares são representados por *(J, T)*, onde *J* é o tipo Java do identificador e *T* é o tipo JaTS. Alguns identificadores, como o nome de uma classe, não possuem um tipo Java. Para esses casos, o símbolo '_' é usado no lugar do *J*. Devido a essa adaptação, a relação *possui-tipo* tem agora a forma $A \triangleright$ *(J, T)*.

Antes de iniciar a verificação de tipos da transformação, é necessário construir o estado inicial de $\Gamma$. Para isso, inserimos em $\Gamma$ os identificadores de todos os tipos primitivos e os associamos ao par `(_, Type)`. Em seguida, cada variável JaTS declarada no início da transformação é associada ao par `(_, T)`, onde `T` é seu tipo. Nomes de métodos, atributos e nomes de parâmetros, se não forem variáveis JaTS, são inseridos em $\Gamma$ associados ao par `(_, Identifier)`. Com o objetivo de simplificar as regras de tipos, tais nomes devem ser diferentes. Nomes de classe, superclasse, tipo de método, atributo, parâmetro ou tipo de retorno do método não são inseridos em $\Gamma$, pois eles podem assumir a forma de qualquer identificador válido em Java.

Alguns identificadores do contexto $\Gamma$ possuem um escopo de classe e são representados por um par `(C, id)`, onde `id` é o identificador e `C` a classe onde este se encontra. As variáveis JaTS declaradas no início da transformação e os tipos primitivos possuem contexto global e não são representadas por este par. A Figura 10 mostra o estado inicial de $\Gamma$ para a transformação da Figura 8.

### 4.4. Regras de Tipo

Uma transformação JaTS possui, em um mesmo contexto, variáveis JaTS ou construções Java. Por exemplo, no lugar de um atributo pode haver uma variável JaTS do tipo `FieldDeclaration` ou uma declaração de atributo que esteja de acordo com a linguagem Java. Nas regras de tipo, para remeter a ambas as representações (variáveis JaTS e construções da linguagem Java) usa-se uma **meta-meta-variável**. Elas são escritas em itálico para que seja possível diferenciá-las.

A coleção de todos os identificadores declarados em $\Gamma$, inclusive os que não pos-

suem um contexto global, é indicada nas regras por `dom(Γ)`, o domínio de $\Gamma$. Para obter o tipo de um identificador no contexto $\Gamma$ escreve-se $\Gamma(id)$. Caso o identificador possua tipo `(J, T)` então, $\Gamma(id)$ = `(J, T)`. Se o identificador estiver dentro do contexto de uma classe seu tipo é obtido da seguinte maneira: $\Gamma(\mathtt{C})(id)$, onde `C` é o nome da classe em que está o identificar *id*. A relação $\Gamma, C \vdash A \triangleright T$ (lê-se: *A* possui tipo *T* no contexto $\Gamma$ e na casse C) é utilizada quando a verificação está ocorrendo no corpo da classe. As regras ID e CID verificam se um identificador *id* possui tipo `(J, T)` quando este está em um contexto global e quando está no contexto de uma classe, respectivamente. Para tal, cada regra checa se o retorno da função é `(J, T)`.

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

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

Para tornar mais claro o funcionamento das regras de tipos, elas serão aplicadas à transformação da Figura 8 à medida em que forem descritas. A validação de uma regra pode depender da validação de outras. Sendo assim, a verificação de tipos de cada programa pode ser vista como uma árvore de dependências onde cada nó é uma regra de tipo e seus filhos são as regras de que depende. O algoritmo percorre essa árvore para validar a transformação. Primeiramente, é verificada a corretude do *template* fonte da transformação, utilizando a regra STEMPLATE. Para o *template* fonte ser válido, ou seja, possuir o tipo JaTS `SourceTemplate`, ele deve ser composto de duas construções: *jvSet*, um conjunto de declarações de variáveis JaTS, e *cd*, uma declaração de classe. Essas construções, representadas por meta-meta-variáveis, devem possuir os tipos JaTS `JVarDeclarationSet` e `ClassDeclaration`, respectivamente.

$$\frac{\Gamma \vdash \mathit{jvSet} \triangleright (\_, \mathtt{JVarDeclarationSet}) \qquad \mathit{cd} \triangleright (\_, \mathtt{ClassDeclaration})}{\Gamma \vdash \mathit{jvSet}\ \ \mathit{cd} \triangleright (\_, \mathtt{SourceTemplate})} \quad \text{(STEMPLATE)}$$

Para verificar o tipo de *jvSet*, são utilizadas as regras JVARDECSET e JVAR. De acordo com a primeira, para que a construção *jvSet* seja do tipo `JVarDeclarationSet`, ela deve iniciar com a palavra-chave `jvar`, seguida de uma seqüência de declarações de mesmo tipo JaTS, representada por $\overline{var}$, e terminar com a palavra chave `in`. Cada elemento dessa seqüência deve possuir o tipo JaTS `JVarDeclaration`. Essa verificação é feita utilizando a regra JVAR.

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

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

Depois de analisadas as declarações de variáveis JaTS, verifica-se o tipo de *cd*, que representa uma declaração de classe. Como a classe do *template* fonte da transformação apresenta uma cláusula `extends`, utiliza-se a regra CLASSPARENTDEC para verificar a corretude desta declaração de classe. Existem três meta-meta-variáveis que precisam ter seus tipos JaTS checados. *T*, caso pertença a $\Gamma$, deve possuir tipo JaTS `TypeName`, o que é checado através da regra ID. Caso *T* não pertença a $\Gamma$, esta será ignorada. A segunda meta-meta-variável, *parentDec*, deve possuir tipo JaTS `ParentDeclaration` e `classBody` tipo JaTS `ClassBodyDeclaration`.

$$\frac{\Gamma, T \vdash \mathit{classBody} \triangleright (\_, \texttt{ClassBodyDeclaration}) \qquad \Gamma \vdash T \in \texttt{dom}(\Gamma) \Rightarrow T \triangleright (\_, \texttt{TypeName}) \qquad \mathit{parentDec} \triangleright (\_, \texttt{ParentDeclaration})}{\Gamma \vdash \texttt{class}\ T\ \mathit{parentDec}\{\mathit{classBody}\} \triangleright (\_, \texttt{ClassDeclaration})} \text{(CLASSPARENTDEC)}$$

Como a cláusula `extends` da transformação em questão foi declarada opcional, utiliza-se a regra OPTDEC para verificar o tipo de *parentDec*. De acordo com essa regra, o tipo de uma declaração opcional é igual ao tipo da declaração que ela engloba. Sendo assim, a regra OPTDEC invoca a regra PARENTDEC passando a esta a declaração "`extends Object`", presente na transformação. De acordo com a regra PARENTDEC, uma construção possui tipo JaTS `ParentDeclaration` se obedece ao padrão `extends` *parentName*, onde *parentName* é o nome da superclasse. Caso o *parentName* pertença a $\Gamma$, o que ocorre quando esta é uma variável JaTS, *parentName* deve possuir tipo JaTS `TypeFQName`. Até o momento, transformação da Figura 8 não apresenta erros de tipo.

$$\frac{\Gamma \vdash e \triangleright (\_, \texttt{T})}{\Gamma \vdash \#[e]\# \triangleright (\_, \texttt{T})} \text{(OPTDEC)}$$

$$\frac{\Gamma \vdash T \in \texttt{dom}(\Gamma) \Rightarrow T \triangleright (\_, \texttt{TypeFQName})}{\Gamma \vdash \texttt{extends}\ T \triangleright (\_, \texttt{ParentDeclaration})} \text{(PARENTDEC)}$$

As três regras seguintes (DECS, DECSFD e DECSMD) verificam o tipo de *classBody*. Para esta meta-meta-variável possuir tipo JaTS `ClasBodyDeclaration` ela deve ser composta apenas de construções do tipo `FieldDeclaration` ou `MethodDeclaration`.

$$\frac{\Gamma, \texttt{C} \vdash \mathit{fd} \triangleright (\_, \texttt{FieldDeclaration})}{\Gamma, \texttt{C} \vdash \mathit{fd} \triangleright (\_, \texttt{ClassBodyDeclaration})} \text{(DECSFD)}$$

$$\frac{\Gamma, \texttt{C} \vdash \mathit{fd} \triangleright (\_, \texttt{MethodDeclaration})}{\Gamma, \texttt{C} \vdash \mathit{fd} \triangleright (\_, \texttt{ClassBodyDeclaration})} \text{(DECSMD)}$$

$$\frac{\Gamma, \texttt{C} \vdash d' \triangleright (\_, \texttt{ClassBodyDeclaration}) \qquad d \triangleright (\_, \texttt{ClassBodyDeclaration})}{\Gamma, \texttt{C} \vdash d'\ d \triangleright (\_, \texttt{ClassBodyDeclaration})} \text{(DECS)}$$

O *template* fonte da transformação possui apenas uma declaração de atributo no corpo da classe e esta é uma variável JaTS. Sendo assim, utiliza-se a regra ID para verificar se o tipo JaTS dessa variável é `FieldDeclaration`. Após essa verificação, é terminada a análise do *template* fonte e inicia-se a análise do *template* destino. A regra DTEMPLATE diz que para o *template* destino de uma transformação ser bem tipado ele deve possui uma declaração de classe válida. A verificação de tipos ocorre de forma semelhante nos dois *templates* da transformação.

$$\frac{\Gamma \vdash \mathit{cd} \triangleright (\_, \texttt{ClassDeclaration})}{\Gamma \vdash \mathit{cd} \triangleright (\_, \texttt{DestinationTemplate})} \text{(DTEMPLATE)}$$

O *template* destino possui duas declarações de atributo. A primeira é a variável JaTS `#fd` cuja validação é feita através da regra ID. Para validar a segunda declaração, "`int code;`", utilizada-se a regra FIELDDEC. De acordo com FIELDDEC, uma construção possui tipo JaTS `FieldDeclaration` se casar com o padrão *t id*, onde *t*, caso pertença a $\Gamma$, deve possuir tipo JaTS `Type`, e `id` deve obrigatoriamente estar em $\Gamma$ e possuir tipo JaTS `Identifier`. Sendo assim, a segunda declaração de atributo do *template* destino é válida.

$$\frac{\Gamma, \mathtt{C} \vdash (t \in \mathtt{dom}(\Gamma) \Rightarrow t \triangleright (\_, \mathtt{Type})) \qquad id \triangleright (\_, \mathtt{Identifier})}{\Gamma, \mathtt{C} \vdash tid;\ \triangleright(\_, \mathtt{FieldDeclaration})} \text{ (FIELDDEC)}$$

Como a declaração de método do *template* destino não possui parâmetros formais, utiliza-se que regra METHODDEC para validá-la. A declaração deve casar com o padrão "*ret met* `()` {*block*}", onde: *ret*, se pertencer a $\Gamma$, deve possuir tipo JaTS `Type`, *met* deve pertencer a $\Gamma$ e possuir tipo JaTS `Identifier` e *block*, da mesma forma que *met*, tipo JaTS `JBlock`. Existem duas peculiaridades nessa regra. A primeira delas é o contexto onde o tipo da meta-meta-variável *block* é checado. Para validar o corpo de um método é necessário adicionar ao contexto inicial o identificador `this` mapeando em `(C, Identifier)`. Essa adição é indicada pelo símbolo "$\oplus$". A segunda peculiaridade diz respeito à verificação de tipos Java do método. Ou seja, o tipo Java de *block* deve ser igual ao valor de *ret*.

$$\frac{\begin{array}{c}\Gamma, \mathtt{C} \oplus (\mathtt{this} \mapsto (\mathtt{C}, \mathtt{Indentifer})) \vdash block \triangleright (ret, \mathtt{Block}) \\ \Gamma, \mathtt{C} \vdash (ret \in \mathtt{dom}(\Gamma) \Rightarrow ret \triangleright (\_, \mathtt{Type})) \qquad met \triangleright (\_, \mathtt{Identifier})\end{array}}{\Gamma, \mathtt{C} \vdash ret\ met()\{block\} \triangleright (\_, \mathtt{MethodDeclaration})} \text{ (METHODDEC)}$$

O corpo do método presente no *template* destino possui apenas a expressão "`return this.#fd`". Sendo assim, para validar *block*, utiliza-se a regra BLOCKRET. Essa regra irá invocar EXP para validar o tipo da meta-meta-variável *exp*, que representa a expressão da cláusula `return`. De acordo com EXP uma expressão possui tipo JaTS `Expression` quando é uma declaração de método, atributo ou um identificador. O tipo Java dessa expressão será igual ao tipo Java da declaração.

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright (\mathtt{J}, \mathtt{Expression})}{\Gamma, \mathtt{C} \vdash \mathtt{return}\ exp; \triangleright (\mathtt{J}, \mathtt{Block})} \text{ (BLOCKRET)}$$

$$\frac{\Gamma, \mathtt{C} \vdash exp \triangleright (\mathtt{J}, \mathtt{MethodCall}) \vee exp \triangleright (\mathtt{J}, \mathtt{FieldAccess}) \vee exp \triangleright (\mathtt{J}, \mathtt{Identifier})}{\Gamma, \mathtt{C} \vdash exp \triangleright (\mathtt{J}, \mathtt{Expression})} \text{ (EXP)}$$

A expressão da cláusula `return` da transformação casa com o padrão *v.id*. Então, a regra FACCESS é utilizada para validá-la. De acordo com FACCESS, uma declaração possui tipo JaTS `FieldAccess` se as meta-meta-variáveis *v* e *id* possuírem tipo JaTS `Identifier`. O tipo Java da declaração será igual ao tipo Java de *id*. No caso da transformação da Figura 8, o valor de *id* é a variável JaTS `#fd` que, de acordo com a regra ID, possui tipo JaTS `FieldDeclaration`. Neste caso, a regra FAccess irá invalidar a transformação. Para corrigir a transformação da Figura 8, deve-se colocar no lugar de `#fd` a declaração executável "`#<#fd.getName()>#`". Desta forma a expressão da cláusula `return` é uma declaração executável cujo objeto é a variável JaTS `#fd`. O tipo JaTS da expressão será definido por "`#<#fd.getType()>#`", sendo igual ao valor da meta-meta-variável *ret*. Essa modificação tornaria válida a transformação. Mesmo que o tipo Java não seja conhecido, garante-se que para qualquer programa fonte dado como entrada o valor de *ret* e o tipo Java de *block* serão iguais.

$$\frac{\Gamma \vdash v \triangleright (\_, \mathtt{Identifier}) \qquad id \triangleright (\mathtt{J}, \mathtt{Identifier})}{\Gamma \vdash v.id \triangleright (\mathtt{J}, \mathtt{FieldAccess})} \text{ (FACCESS)}$$

Antes da formalização desse sistema de tipos, a transformação da Figura 8 seria válida e geraria código incorreto. Aplicando previamente as regras descritas, pode-se identificar o erro na transformação e invalidá-la. Da mesma forma, um programador JaTS pode utilizar as regras para analisar qualquer transformação antes de aplicá-la a um programa fonte. Seguindo esse procedimento, ele terá a garantia que os erros descritos na Seção 3. não irão ocorrer.

Erros sintáticos no programa gerado são causados quando os tipos JaTS são mal utilizados. Ou seja, quando uma construção JaTS é colocada, no *template destino*, em um lugar inadequado. Por exemplo, quando uma variável JaTS declarada como `FieldDeclaration` é colocada no lado esquerdo de uma atribuição, ou quando uma expressão executável que produz uma construção do tipo `Name` ocupa o lugar do nome da classe. O sistema de tipos descrito detecta as más utilizações, pois especifica as posições que cada tipo JaTS pode ocupar. Sendo assim, o programa gerado também estará livre de erros de sintaxe.

Não foi possível abordar todos os erros semânticos. Por exemplo, o template destino pode conter uma chamada de métodos cujos parâmetros estão incorretos e essa chamada estará presente no programa gerado. Optamos por simplificar as regras e tratar apenas o tipo de retorno. A regra METHODDEC captura a incompatibilidade entre o tipo de retorno do bloco do método e o tipo de retorno declarado na assinatura. Para tratar outros erros semânticos basta modificar um pouco as regras utilizando o tipo Java das construções.

## 5. Trabalhos Relacionados

Existem várias linguagens e ferramentas para especificar transformações de programas. Entre elas LET, TXL, Draco-PUC e JPearl. As três primeiras linguagens levam em conta somente a sintaxe do programa. As transformações são aplicadas através de casamento de padrões e substituição. O programa fonte dessas linguagens pode estar escrito em qualquer linguagem objeto.Como LET, TXL e Draco-PUC não possuem uma representação semântica da linguagem objeto, não faz sentido possuírem um sistema de tipos. Por este motivo, não é possível garantir que transformações escritas nessas linguagens gerem um programa corretos sintática e semanticamente.

O trabalho proposto por Chen e Xi [Chen and Xi, 2003a] apresenta uma abordagem para implementar transformações de programas capazes de representar a semântica da linguagem objeto. Este trabalho foca no poder de expressividade que uma representação semântica para a linguagem objeto proporciona, ao invés de tentar garantir a corretude do programa gerado pela transformação. JPearl e JaTS são linguagem de transformação capazes de representar a semântica de Java. Contudo, ao contrário de JaTS, JPearl não possui um sistema de tipos formalizado e implementado, o que a torna vulnerável a problemas semelhantes aos descritos na Seção 3..

## 6. Conclusões e Trabalhos Futuros

A linguagem JaTS possuía apenas uma noção de tipos definida de maneira implícita e informal. Por este motivo, apresentava diversas deficiências, entre elas, a possibilidade de implementar e aplicar transformações que geravam código Java incorreto. A definição formal de um sistema de tipos para um subconjunto significativo de JaTS e sua implementação são capazes de garantir a corretude sintática e parte da corretude semântica do programa gerado por uma transformação. O sistema de tipos descrito neste trabalho já foi implementado e estamos trabalhando na extensão do mesmo para tratar um número maior de erros semânticos. No momento, o sistema é bastante útil para garantir a corretude sintática dos programas gerados.

A permutação de variáveis JaTS dos tipos `Type`, `Name` e `Identifier`, como visto na Seção 3. é bastante comum nas transformações, o que aumenta a ocorrência de erros no programa gerado. São ainda mais freqüentes os erros relacionados as tipos Java das construções. Uma simples atribuição, como "`this.#<#fd.getName()># = cod;`", onde

`#fd` é uma declaração de atributo e `cod` é uma variável do tipo `int` passaria despercebida antes da formalização do sistema de tipos. Dependendo do programa fonte dado como entrada, essa atribuição poderia gerar código Java incorreto. Por exemplo, caso a variável `#fd` fosse casada com uma declaração de atributo do tipo `String`. O sistema de tipos filtra grande parte das transformações possíveis de serem implementadas em JaTS e identifica como válidas apenas aquelas que não produzem código Java incorreto, independentemente do programa fonte.

No entanto, algumas transformações que dependem do programa fonte são bastante utilizadas: geração de código de testes para nulidade de parâmetros ou incremento do valor de uma variável JaTS. Nesses casos, o tipo da variável que casará com a variável JaTS não é conhecido. Por este motivo, está sendo analisada a possibilidade de, ao invés de invalidar as transformações JaTS que dependam do programa fonte, gerar um *warning* indicando o erro em potencial.

Está sendo estudada uma forma de generalizar a criação de sistemas de tipo para linguagens de transformação. Acreditamos ser possível dada nossa experiência com o presente trabalho e com outro trabalho de nosso grupo que aborda a geração de sistemas de transformação de programas [Sousa and Borba, 2005].

## 7. Agradecimentos

Gostaríamos de agradecer a Fernando Castor pelas suas sugestões e críticas. Também estamos gratos aos avaliadores anônimos que ajudaram a melhorar este artigo.

## Referências

Bergmam, U., Prado, A., and Leite, J. (1996). Desenvolvimento de sistemas orientados a objetos utilizando o sistema transformacional draco-puc. In *Anáis do X Simpósio Brasileiro de Engenharia de Software*, pages 173–188, São Carlos.

Borba, P. and Sampaio, A. (2000). Basic laws of rool: an object-oriented language. In *Anais do III Workshop de Métodos Formais*, pages 33—44, João Pessoa - PB, Brasil.

Cardelli, L. (2004). Type systems. In Tucker, A. B., editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, Boca Raton, FL, 2nd edition.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *Computing Survey*, 17(4):471–522.

Castor, F. and Borba, P. (2001). A language for specifying java transformations. In *Anais do V Simpósio Brasileiro de Linguagens de Programação*, pages 236–251, Curitiba - PR, Brasil.

Castor, F., Oliveira, K., Souza, A., Santos, G., and Borba, P. (2001). Jats: A java transformation system. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 374–379, Rio de Janeiro - RJ, Brasil.

Chen, C. and Xi, H. (2003a). 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.

Chen, C. and Xi, H. (2003b). Meta-programming through typeful code representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 275–286, Uppsala, Suécia.

Cordy, J. and Carmichael, I. (1993). The txl programming language syntax and informal semantics. Technical report, Queen's University at Kingston, Department of Computing and Information Science.

Felix, M. and Hausler, E. (1999). Let: Uma linguagem para especificar transformações. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, pages 109–121, Porto Alegre - RS, Brasil.

Fowler, M. (2000). *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison Wesley.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Addison-Wesley.

Igarashi, A., Pierce, B., and Wadler, P. (1999). 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.

Maia, M. and Oliveira, A. (2002). Jpearl: Uma linguagem para descrição de reestruturações em programas java. In *Anáis do VI Simpósio Brasileiro de Linguagens de Programação*, pages 166–179, Rio de Janeiro.

Morgan, C. (1990). *Programming from Specifications*. International Series in Computer Science. Prentice-Hall International.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science.

Sousa, A. and Borba, P. (2005). Geração de sistemas de transformação. In *Anais do IX Simpósio Brasileiro de Linguagens de Programação*, Recife - PE, Brasil.