

Translating Java Code to Graph Transformation Systems^{*}

Andrea Corradini¹, Fernando Luís Dotti², Luciana Foss³, and Leila Ribeiro³

¹ Dipartimento di Informatica, Università di Pisa,
Pisa, Italy — andrea@di.unipi.it

² Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul,
Porto Alegre, Brazil — fldotti@inf.pucrs.br

³ Instituto de Informática, Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil — {lfoss,leila}@inf.ufrgs.br

Abstract. We propose a faithful encoding of Java programs (written in a suitable fragment of the language) to Graph Transformation Systems. Every program is translated to a set of rules including some *basic rules*, common to all programs and providing the operational semantics of Java (data and control) operators, and the *program specific rules*, namely one rule for each method or constructor declared in the program.

Besides sketching some potential applications of the proposed translation, we discuss some design choices that ensure its correctness, and we report on how do we intend to extend it in order to handle several other features of the Java language.

1 Introduction

Graph Transformation Systems (GTSS) were introduced about four decades ago as a generalization of Chomski Grammars to non-linear structures. Since then, the theory evolved quickly [24], and a growing community has applied GTSS to diverse application fields [7]. Along the years, GTSS have been used to provide an operational semantics to programming languages [22], to systems made of agents (or actors) interacting via message passing [15, 16, 6], to object-oriented specification formalisms [26], to process calculi [10], and to several other languages and formalisms.

By encoding a computational formalism into GTSS, usually one enjoys several benefits. Firstly, the representation of states as graphs is often quite abstract and intuitive, as it allows to ignore irrelevant details: in fact, graphs are considered usually “up to isomorphism”, and this corresponds to considering states up to the renaming of bound variables or names. Secondly, the rich theory of GTSS provides interesting results that may be applied to the original formalism via the encoding. As an example, the concurrent behavior of GTSS has been thoroughly

^{*} Work partially supported by projects IQ-MOBILE (CNPq and CNR), PLATUS (CNPq), AGILE (EU FET – Global Computing) and SEGRAVIS (EU Reserach Training Network).

studied and a consolidated theory of concurrency is now available, including event structure, process and unfolding semantics [23, 3]; the encoding of process calculi as GTSs equips such calculi with a concurrent operational semantics [10], to which the mentioned results can be applied.

Along these lines, in this paper we propose a translation of Java programs, written in a suitable fragment of the language, into Graph Transformation Systems. The fragment of Java for which we present such an encoding is small, but still significant. It includes operators on primitive data types, class declarations, assignments, conditional statements, (static or instance) method invocations with parameter passing and value returning, and construction of new objects. A *Java declaration*, i.e., a closed set of class definitions, is translated into a *graph of types* and a set of *typed rules* that specify the (potential) behavior of the program. The states of execution of a Java program are represented by graphs which encode aspects related to both the data structures and the control of the program. As expected, information which are not relevant at run-time, like names of local variables or of formal parameters of methods, do not appear explicitly in the graphical representation of states. The rules resulting from the translation of a Java declaration are divided in two kinds: the *basic rules*, that are common to all Java programs and specify the operational meaning of the various (functional and control) operators of the language; and the *program specific rules*, one for each method or constructor, which replace a method/constructor call with the corresponding body. After presenting the translation, we discuss some design choices we made, and also how do we intend to extend the translation in order to handle other features of Java, including arrays, inheritance and multi-threading.

There are several potential applications of the kind of translation we propose: we sketch here two of them. Recently, various analysis techniques for GTSs have been proposed, some of which are based on the unfolding semantics (see, e.g., [2] for the finite-state case, and [4] for the general case). We intend to investigate how far such techniques can be applied to the systems obtained from the translation of Java programs, and thus, indirectly, to the Java programs themselves: the leading intuition is that relevant properties of Java programs can be formulated, in a GTSs setting, as structural properties of the graphs reachable in the system. Having this application in mind, we restricted the format of the rules obtained from the translation in order to match the constraints imposed by [2, 4], the most relevant of which is that rules must be injective.

Another application we have in mind is to provide a pure GTS-based semantics to *Java-attributed* GTSs, the graph transformation model adopted in tools like AGG [1]. In this model, the items of a graph can be associated with arbitrary Java expressions (the *attributes*), and in a rewriting step such expressions are evaluated to compute the attributes of the newly created items. By exploiting the proposed translation of Java to GTS, we may translate a Java-attributed GTS to a plain typed GTS by first encoding the attributes as graphs, and then simulating an attributed graph rewriting step as a sequence of typed graph rewriting steps, where the evaluation of the attributes is performed graphically.

On the one hand, the translation just sketched would allow us to explore the applicability of the above mentioned analysis techniques to Java-attributed GTSS. On the other hand, we are confident that it will provide a formal ground on which to address a well-known limitation of the expressive power of attributed GTSS, namely the fact that graphical items may refer to attributes, but attributes cannot refer back to graphical items. For example, according to the standard approaches [17], it is not possible to define a graph where a vertex has as attribute a list of vertices of the graph itself.

The paper is organized as follows. Section 2 introduces the basics of typed (hyper)graph transformation systems according to the Single-Pushout Approach. Section 3 presents the fragment of Java that we shall consider, and Section 4 shows how to translate programs written in this fragment into typed GTSS. Section 5 discusses some design choices underlying the proposed translation and how do we intend to handle other features of Java, and Section 6 concludes sketching some subjects of future work.

2 Typed Graph Transformation Systems

In this section we recall the definition of graph transformation systems (GTSS) according to the Single-Pushout (SPO) approach [18, 8]. However, it is worth stressing that we could have used equivalently other approaches, like for example the Double-Pushout approach: in fact, for the kind of graphs and rules generated by the translation of Java to GTSS, the two approaches are equivalent.

We shall define GTSS over (typed) *hypergraphs*, i.e., graphs where edges can be connected to any (finite) number of vertices. Graphically, an edge is depicted as a box (whose shape may vary), and the connections to the vertices are drawn as thin lines, called *tentacles*. Usually the tentacles of an edge are labeled by natural numbers: here we propose a slightly more general definition, allowing us to label tentacles with *labels* taken from a fixed set (see Figure 2).

The definition of the SPO approach is based on a category of graphs and *partial* morphisms.

Definition 1 (weak commutativity). *Given two partial functions $f, f' : A \rightarrow B$, we say that f is **less defined** than f' (and we write $f \leq f'$) if $\text{dom}(f) \subseteq \text{dom}(f')$ and $f(x) = f'(x)$ for all $x \in \text{dom}(f)$. Given two partial functions $f : A \rightarrow B$ and $f' : A' \rightarrow B'$, and two total functions $a : A \rightarrow A'$ and $b : B \rightarrow B'$, we say that the resulting diagram **commutes weakly** if $b \circ f \leq f' \circ a$.*

Now we introduce hypergraphs and partial morphisms. Each hyperedge is associated to a *finite, labeled set of vertices* (formally, to a partial function from a fixed set of labels to vertices). This definition makes use of a *labeling functor*, defined as follows: Let L be a set of labels. The labeling functor $\mathcal{I}_L : \text{Set}^P \rightarrow \text{Set}^P$ maps each set $V \in |\text{Set}^P|$ to the set of *L-labeled sets over V* (i.e., to the set of partial functions $L \rightarrow V$), and each partial function $f : V \rightarrow V' \in \text{Mor}_{\text{Set}^P}(V, V')$ to the function $\mathcal{I}_L(f)$, defined for all $l : L \rightarrow V \in \mathcal{I}_L(V)$ as $\mathcal{I}_L(f)(l) = f \circ l$.

Definition 2 ((hyper)graph, (hyper)graph morphism). A **(hyper) graph** $G = (V_G, E_G, c^G)$ over a set of labels L consists of a set of vertices V_G , a set of (hyper)edges E_G , and a total connection function $c^G : E_G \rightarrow \mathcal{I}_L(V_G)$, assigning a finite L -labeled set of vertices to each edge.⁴

A **(partial) graph morphism** $g : G \rightarrow H$ is a pair of partial functions $g_V : V_G \rightarrow V_H$ and $g_E : E_G \rightarrow E_H$ which are weakly homomorphic, i.e., $\mathcal{I}_L(g_V) \circ c^G \geq c^H \circ g_E$ (if an edge is mapped, the corresponding vertices, if mapped, must have the same labels). A morphism is called **total** if both components are total. The category of hypergraphs and partial hypergraph morphisms is denoted by **HGraphP** (identities and composition are defined componentwise).

$$\begin{array}{ccc} E_G & \xrightarrow{g_E} & E_H \\ c^G \downarrow & \geq & \downarrow c^H \\ \mathcal{I}_L(V_G) & \xrightarrow{\mathcal{I}_L(g_V)} & \mathcal{I}_L(V_H) \end{array}$$

To distinguish different kinds of vertices and edges, we will use the notion of *typed hypergraphs*, analogous to typed graphs [5, 16]: every hypergraph is equipped with a morphism *type* to a fixed graph of types.⁵

Definition 3 (typed hypergraphs). Let TG be a fixed graph called the **graph of types**. A **typed hypergraph over TG** is a pair $HG^{TG} = (HG, \text{type}^{HG})$ where HG is a hypergraph called **instance graph** and $\text{type}^{HG} : HG \rightarrow TG$ is a total hypergraph morphism, called the **typing morphism**.

A morphism between typed hypergraphs HG_1^{TG} and HG_2^{TG} is a partial graph morphism $f : HG_1 \rightarrow HG_2$ such that $\text{type}^{HG_1} \geq \text{type}^{HG_2} \circ f$. The category of hypergraphs typed over a graph of types TG , denoted by **THGraphP**(TG), has hypergraphs over TG as objects and morphisms between typed hypergraphs as arrows (identities and composition are the identities and composition of partial graph morphisms).

Definition 4 (graph transformation systems). Given a graph of types TG , a **rule** $r : L \rightarrow R$ is a partial injective graph morphism in **THGraphP**(TG) such that 1) L and R are finite, 2) L has no isolated vertices, 3) r is total on the vertices of L (i.e., vertices are preserved), 4) r is not defined on at least one edge in L (i.e., the rule is consuming). A **graph transformation system** is a pair $\mathcal{G} = (TG, \text{Rules})$ where TG is the graph of types and Rules is a set of rules over TG .

Conditions 3) and 4), as well as the injectivity of rules, are required by the unfolding constructions or by the analysis techniques based on them (see [4, 2]) that we intend to apply to our systems. Condition 2) guarantees that vertices that become isolated have no influence on further rewriting. Thus one can safely assume that isolated vertices are removed by some kind of garbage collection, mitigating condition 3).

⁴ Intuitively, for each edge $e \in E_G$, the partial function $c^G(e) : L \rightarrow V_G$ is defined on a label $a \in L$ iff e has a tentacle labeled a pointing to vertex $c^G(e)(a)$.

⁵ Note that, due to the use of partial morphisms, this is not just a comma category construction: the morphism *type* is total whereas morphisms among graphs are partial, and we need weak commutativity instead of commutativity.

Definition 5 (derivation step, derivation). Let $\mathcal{G} = (TG, Rules)$ be a GTS, $r : L \rightarrow R \in Rules$ be a rule, and G_1 be a graph typed over TG . A **match** for r in G_1 is a total morphism $m : L \rightarrow G_1$ in $\mathbf{THGraphP}(TG)$. A **derivation step** $G_1 \xrightarrow{r,m} G_2$ using rule r and match m is a pushout in the category $\mathbf{THGraphP}(TG)$.

$$\begin{array}{ccc}
 L & \xrightarrow{r} & R \\
 m \downarrow & (PO) & \downarrow m' \\
 G_1 & \xrightarrow{r'} & G_2
 \end{array}
 \quad
 \begin{array}{l}
 \text{A derivation sequence of } \mathcal{G} \text{ is a sequence of derivation steps} \\
 G_i \xrightarrow{r_i, m_i} G_{i+1}, i \in \{0, \dots, n\}, n \in \mathbb{N}, \text{ where } r_i \in Rules \text{ for all} \\
 i \in \{0, \dots, n\}.
 \end{array}$$

3 Java

Java [11] is a general-purpose object-oriented programming language designed to be highly portable and to ease the deployment of programs in distributed settings. Therefore the choice of a run-time environment based on interpretation (the Java Virtual Machine) and allowing for dynamic class loading from remote code bases. Garbage collection is also supported by the run-time environment. Java supports multi-threading as well as a synchronization mechanism close to Monitors. Altogether, the characteristics of Java led many developers to adhere to the language and currently a rich set of Java libraries can be found for the most different application areas.

In the following we present the BNF syntax of the fragment of Java considered in this paper and next a corresponding example of Java code. The fragment includes operators on the primitive data types `char`, `int` and `boolean`, class declarations, assignments, conditional statements, (static or instance) method invocations with parameter passing and value returning, and construction of new objects.

```

Declaration ::= Modifier class Id { ClassBody } | ;
Modifier ::= static | public | private |  $\varepsilon$ 
ClassBody ::= Modifier ClassMember ClassBody | ; |  $\varepsilon$ 
ClassMember ::= MethodDeclaration | ConstructorDeclaration | VarDeclaration
MethodDeclaration ::= MType Id ( ParameterList ) { Block }
MType ::= void | Type
ConstructorDeclaration ::= Id ( ParameterList ) { Block }
VarDeclaration ::= Type VarDeclarationList ;
VarDeclarationList ::= Id VarInitializer | VarDeclarationList , Id VarInitializer
VarInitializer ::= = Expression |  $\varepsilon$ 
ParameterList ::= Type Id ParameterRest |  $\varepsilon$ 
ParameterRest ::= , Type Id ParameterRest |  $\varepsilon$ 
Block ::= BlockStatement Block |  $\varepsilon$ 
BlockStatement ::= VarDeclaration | Statement
Statement ::= StatementExp | { Block } | if ( Expression ) Statement |
    if ( Expression ) Statement else Statement | return Expression ; | ;

```

```

StatementExp ::= Assignment | MethodInvocation | NewExpression |
    this ( ArgumentList )
Assignment ::= QualifiedId = Expression | FieldAccess = Expression
FieldAccess ::= Primary . Id
Primary ::= FieldAccess | MethodInvocation | newExpression | ( Expression )
    | this | Literal | null
Expression ::= Assignment | Expression InfixOp Expression | PrefixOp Expression
    | QualifiedId | Primary
MethodInvocation ::= Id ( ArgumentList ) | Primary . Id ( ArgumentList )
NewExpression ::= new Id ( ArgumentList )
InfixOp ::= || | && | == | != | < | > | + | - | * | /
PrefixOp ::= ! | + | -
ArgumentList ::= Expression ArgumentRest |  $\varepsilon$ 
ArgumentRest ::= , Expression ArgumentRest |  $\varepsilon$ 
Literal ::= Integer | Character | true | false
Type ::= QualifiedId | BasicType
BasicType ::= char | int | boolean
QualifiedId ::= Id Qualification
Qualification ::= . Id Qualification |  $\varepsilon$ 

```

We call a **Java declaration** a set *JSpec* of Java class declarations written in the above fragment of the language, closed under definitions (i.e., every class referred to by a class in *JSpec* is in *JSpec* as well), correct with respect to syntax and static semantics (i.e., the definitions compile safely), and satisfying some additional requirements described next.

Without loss of generality, we assume that all (static or instance) variables of the classes in *JSpec* are declared as **private**, (read/write access to a variable **x** can be provided by the standard accessor methods **getX/setX**); that no variable is accessed using the dot-notation “**class.var**” or “**obj.var**”,⁶ unless **obj** is **this**; and that all local variables of methods are initialized (possibly with the standard default value) at declaration time.

Figure 1 shows a sample Java declaration satisfying the above constraints.

4 Translating Java to GTSs

We present here the translation of Java declarations, as defined in the previous section, into graph transformation rules. A Java declaration *JSpec* is translated into a graph of types and a set of typed rules that specify the (potential) behavior of the program. The graphs which are rewritten represent aspects related to both the data structures and the control of the program under execution.

In the graph modeling the current state of a computation every available data element is represented explicitly, including objects, variables, constant values of basic data types, and also the classes of *JSpec*, which are needed to handle **static** variables and methods. Each such data element is represented graphically by a vertex (its *identity*, typed with the corresponding Java data type) and by an

⁶ This last assumption is necessary even if all variables are **private**.

```

public class LinkedList List {
    private ListNode first;
    private ListNode last;

    public ListNode getFirst() {
        return first;}

    public ListNode getLast() {
        return last;}

    public void setFirst(ListNode f) {
        first=f;}

    public void setLast(ListNode l) {
        last=l;}

    public LinkedList() {
        first = last = null; }

    public boolean isEmpty() {
        return first == null; }

    public void clear() {
        first = last = null; }

    public boolean add( Person x ) {
        if ( isEmpty() ) first = last = new ListNode( x );
        else { last.setNext(new ListNode( x ));
              last = last.getNext(); }
        return true; }

    public boolean remove() {
        if ( isEmpty() ) return false;
        else { first = first.getNext();
              return true; } }
}

public class ListNode {
    private Person element;
    private ListNode next;

    public Person getElement() {
        return element;}

    public ListNode getNext() {
        return next;}

    public void setElement(Person e) {
        element = e;}

    public void setNext(ListNode n) {
        next = n;}

    public ListNode(Person element) {
        this( element, null ); }

    public ListNode(Person element, ListNode next ) {
        this.element = element;
        this.next = next; }
}

public class Person {
    private int identifier;

    public int getIdentifier() {
        return identifier;}

    public void setIdentifier(int n) {
        identifier = n;}

    public Person(int identifier) {
        this.identifier = identifier; }
}

```

Fig. 1. Example of Java Declaration.

edge carrying the relevant information, connected to the identity with a tentacle labeled by `val` (for constants) or `id` (for all other cases). Such edges may have additional tentacles: for example, the edge representing a class (depicted as a double box) has one tentacle for each `static` variable of the class; an edge representing a class instance has one tentacle for each `instance` variable of the class; an edge representing a variable has a `val` tentacle pointing to the (identity node of the) current value of the variable.⁷

The control operators are also represented as hyperedges, and they are connected through suitable tentacles to all the data they act upon. Their operational behavior is specified by rules. The rules are divided in two kinds:

- **Basic rules:** These rules are common to all Java programs, and specify the operational meaning of the various (functional and control) operators of the language, including operators on primitive data types, like arithmetic and relational ones; assignment and `new` operators; `if` and `if-else` control structures; `return` statement; `get` and `set` operators for accessing local variables; and rules for handling value indirections (`val` edges).
- **Program specific rules:** These rules encode the methods and constructors of the classes in *JSpec*. For each method and constructor declaration

⁷ Technically, when restricted to the data part, the graphs we are considering are *term graphs* [20].

in *JSpec* there is one rule, where the left-hand side encodes the method or constructor call and the right-hand side encodes the whole body of the method.

Conceptually, the execution of a method call is modeled by a derivation sequence where the first rule (a program specific one) replaces the method invocation edge by a graph that encodes the body of the method, and next the control edges in the body are evaluated sequentially using the corresponding rules. In order to guarantee that statements are evaluated in the same order as in the Java Virtual Machine, we supply every control edge with two additional tentacles (labeled **in** and **out**) and we use a unique **GO** hyperedge system-wide, acting as a token, to enforce sequential execution. In fact, every rule consumes the **GO** token connected to the **in**-vertex of the control edge (thus the rule cannot be applied if the token is not present), and generates the token again to the **out**-vertex.

For a given Java declaration *JSpec*, the following definition describes the graph of types $TG(JSpec)$ obtained from the translation.

Definition 6 (graph of types of a Java declaration). *Let $JSpec$ be a Java declaration. The graph of types $TG(JSpec) = (V, E, c)$ associated to $JSpec$ is the hypergraph defined as follows:*

- $V = BDT \cup CT \cup \{\circ\}$, where $BDT = \{t \mid t \text{ is a basic data type used in } JSpec\}$, $CT = \{c \mid c \text{ is a class name in } JSpec\}$. Thus there is one vertex for each data type in *JSpec*, and an additional vertex “ \circ ” which is the type of all control vertices.
- $E = \{ \boxed{GO}, \boxed{if}, \boxed{+b}, \boxed{-b}, \boxed{*}, \boxed{/}, \boxed{<}, \boxed{>}, \boxed{\parallel}, \boxed{\&\&}, \boxed{+u}, \boxed{-u}, \boxed{!} \} \cup CH \cup BH \cup CTE$, where
 - *subscripts b and u stand for “binary” and “unary”;*
 - $CH = \{ \boxed{C}, \boxed{C}, \boxed{null^C}, \boxed{new^C}, \boxed{\langle init \rangle_p^C}, \boxed{this^C} \mid C \text{ is a class in } JSpec \} \cup \{ \boxed{m_p^C} \mid m \text{ is a (static or instance) method in } C \text{ with parameter type list } p \} \cup \{ \boxed{\langle init \rangle_p^C} \mid \text{there is a constructor with parameter type list } p \text{ in } C, \text{ or } C \text{ has no constructors and } p \text{ is the empty list} \}$;
 - $BH = \{ \boxed{var^t}, \boxed{get^t}, \boxed{set^t}, \boxed{return^t}, \boxed{val^t}, \boxed{==^t}, \boxed{!=^t} \mid t \text{ is a basic data type or a class in } JSpec \}$;
 - $CTE = \{ \boxed{cte_t} \mid cte \text{ is a constant of a basic data type } t \text{ in } JSpec \}$.
- The set of tentacle labels L is defined as $L = \{g, \text{exp}, \text{in}, \text{out}, \text{out}_T, \text{out}_F, \text{left}, \text{right}, \text{result}, \text{val}, \text{id}, \text{target}, \text{value}\} \cup NAT \cup PAR$, where $NAT = \{\text{at} \mid \text{at} \text{ is a (static or instance) variable name of a class } C \in CH\}$ and $PAR = \{i \mid i \in \mathbb{N} \wedge 0 < i \leq n, \text{ where } n \text{ is the maximum number of formal parameters of any method or constructor in } JSpec\}$.
- The connection function $c : E \rightarrow (L \rightarrow V)$ is defined as follows, where for each edge $e \in E$ the graph of the partial function $c(e) : L \rightarrow V$ is shown:

- $c(\boxed{GO}) = \{(g, \circ)\};$
- $c(\boxed{if}) = \{(\text{exp}, \text{boolean}), (\text{in}, \circ), (\text{out}_T, \circ), (\text{out}_F, \circ)\};$
- $\forall op \in \{\boxed{+b}, \boxed{-b}, \boxed{*}, \boxed{/}\}. c(op) = \{(\text{left}, \text{int}), (\text{right}, \text{int}), (\text{result}, \text{int})\};$
- $\forall op \in \{\boxed{>}, \boxed{<}\}. c(op) = \{(\text{left}, \text{int}), (\text{right}, \text{int}), (\text{result}, \text{boolean})\};$
- $\forall op \in \{\boxed{|||}, \boxed{\&\&}\}. c(op) = \{(\text{left}, \text{boolean}), (\text{right}, \text{boolean}), (\text{result}, \text{boolean})\};$
- $c(\boxed{!}) = \{(\text{right}, \text{boolean}), (\text{result}, \text{boolean})\};$
- $c(\boxed{C}) = \{(\text{at}_1, t_1), \dots, (\text{at}_n, t_n), (\text{id}, C)\}$, where n is the total number of static variables in C , and, for each $0 < i \leq n$, at_i is a static variable name in C and $t_i \in BDT \cup CT$ is its type;
- $c(\boxed{C}) = \{(\text{at}_1, t_1), \dots, (\text{at}_n, t_n), (\text{id}, C)\}$, where n is the total number of instance variables in C , and, for each $0 < i \leq n$, at_i is a instance variable name in C and $t_i \in BDT \cup CT$ is its type;
- $c(\boxed{\text{null}^C}) = \{(\text{val}, C)\};$
- $c(\boxed{m_p^C}) = \{(1, t_1), \dots, (n, t_n), (\text{target}, C), (\text{in}, \circ), (\text{out}, \circ)\} \cup RET_m$, where $[t_1, \dots, t_n]$ is exactly the parameter type list p of method m , and if the type t of method m is different from `void`, then $RET_m = \{(\text{return}, t)\}$ else $RET_m = \emptyset$;
- $c(\boxed{\text{new}^C}) = \{(\text{target}, C), (\text{return}, C), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\langle \text{init} \rangle_p^C}) = \{(1, t_1), \dots, (n, t_n), (\text{target}, C), (\text{in}, \circ), (\text{out}, \circ)\}$, where $[t_1, \dots, t_n]$ is the parameter type list p of the constructor;
- $c(\boxed{\text{this}^C}) = \{(\text{target}, C), (\text{result}, C), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\text{var}^t}) = \{(\text{id}, t), (\text{val}, t)\};$
- $c(\boxed{\text{get}^t}) = \{(\text{target}, t), (\text{result}, t), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\text{set}^t}) = \{(\text{target}, t), (\text{value}, t), (\text{result}, t), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\text{return}^t}) = \{(\text{value}, t), (\text{result}, t), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\text{val}^t}) = \{(\text{return}, t), (\text{value}, t)\};$
- $c(\boxed{==^t}) = \{(\text{right}, t), (\text{left}, t), (\text{result}, \text{boolean}), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{!=^t}) = \{(\text{right}, t), (\text{left}, t), (\text{result}, \text{boolean}), (\text{in}, \circ), (\text{out}, \circ)\};$
- $c(\boxed{\text{cte}_t}) = \{(\text{id}, t)\}.$

For each class $C \in JSpec$, the edge labeled \boxed{C} represents the class itself, providing access to its static variables: it is assumed to be unique in any legal graph representing an execution state of a Java program, and it will be the target of all *static* method invocation, as well as of the *new* operator. The edges labeled \boxed{C} represent instead instances of C , each connected to its private copy of the instance variables of C .

Concerning the rules obtained by the translation, many of the **basic rules** for the Java operators in the sub-language we consider are shown in Figure 2.

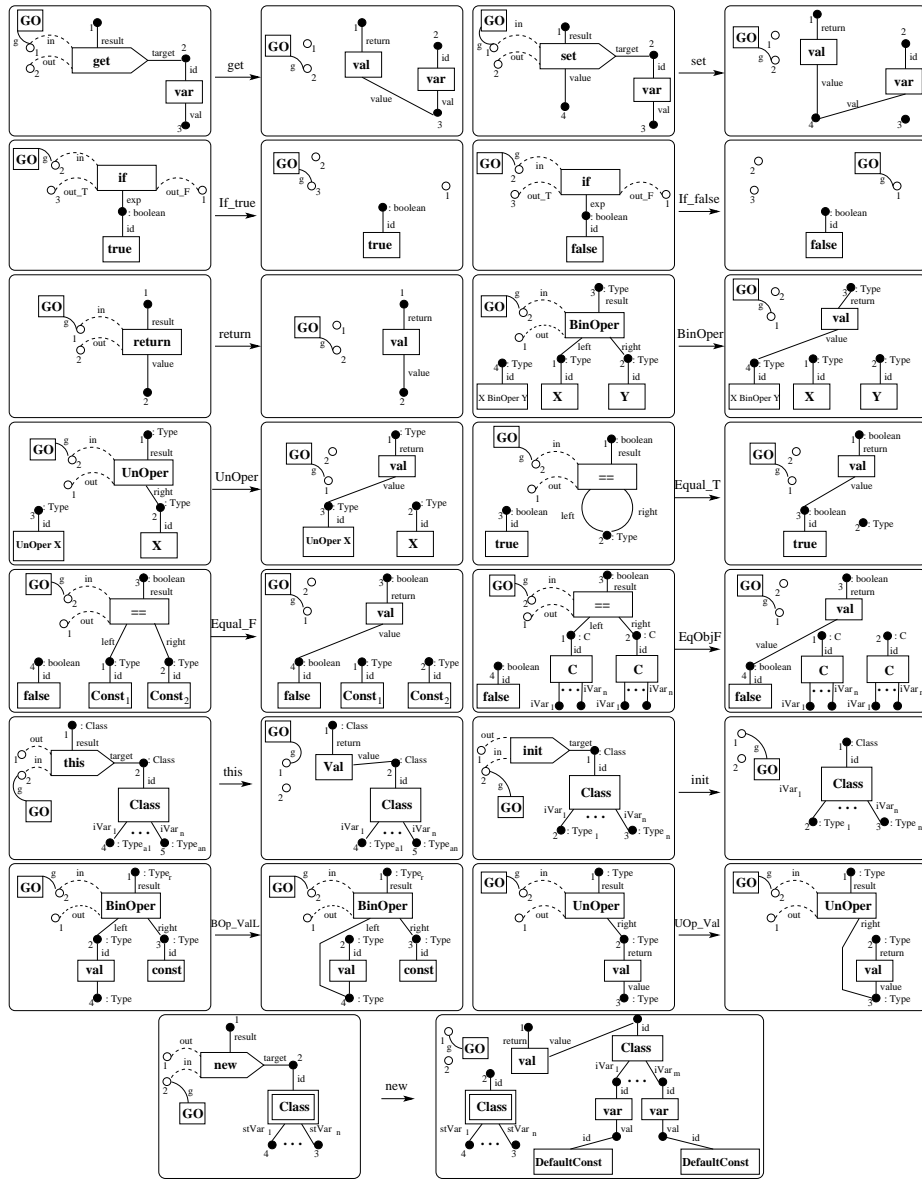


Fig. 2. Basic Rules

Several of such rules actually are rule schemata, and represent the (finite, even if large) collection of all their instances. Some of the rules, including those handling the equality operator and the `val` edges, are discussed in Section 5.

As far as the **program specific rule** $r : L \rightarrow R$ encoding a method (or constructor) of the Java declaration is concerned, it is obtained as follows:

- the left-hand side L is the translation of the method invocation (see Figure 3(d) for a static method invocation);
- the right-hand side R is obtained by translating the body of the method; this translation is obtained by replacing every Java statement in the method body by a corresponding graph – as shown informally in Figure 3 – and connecting such (sub)graphs sequentially using the control tentacles;
- the partial morphism r preserves all the items of its source, but for the edge representing the method being called.

Figure 4 shows some of the rules resulting from the translation of the Java declaration of Figure 1. Currently, such translation is defined in a precise but informal way. We intend to formalize it using, for example, *pair grammars* [21].

5 Some considerations about the proposed translation

In this section we first discuss the design choices underlying some of the rules presented in the previous section. Next we describe the way we intend to extend our translation to a larger fragment of Java.

Handling of val edges. To model the fact that an expression returns a value, we included a special *indirection* edge, labeled with `val`, linking the vertex where the result should be placed to the identity of the result value (see for example, in Figure 2, the rules `get`, which makes the value of the target variable accessible at vertex 1 of the left-hand side, and `set`, which implements an assignment by changing the value of the target variable and returning the assigned value). In order to get rid of the `val` edges, which is necessary because otherwise the execution cannot proceed, we need to add for each operator edge and for each tentacle through which the operator may access a data value, an auxiliary rule that allows to bypass any `val` edge connected to that tentacle. Rule schemata `BOp_ValL` and `UOp_Val` of Figure 2 show the shape of such auxiliary rules.

It is worth noting that the simpler solution of adding a single non-injective rule with a `val` edge in the left-hand side and a single node in the right-hand side (thus the rule would consume the edge, “merging” the two nodes connected to it), would not be legal according to Definition 4.

Handling of the equality operator “==”. Rule `Equal_T` of Figure 2 shows that “==” is interpreted as *reference identity*, and this both for reference types (classes) and for basic data types. The correctness of this rule is ensured by assuming (as in [14]) that all constants of basic data types are present in the start graph, and that they are preserved by all rules. Such an assumption allows to deal with values of all data types uniformly at the graphical level (that is, using non-attributed

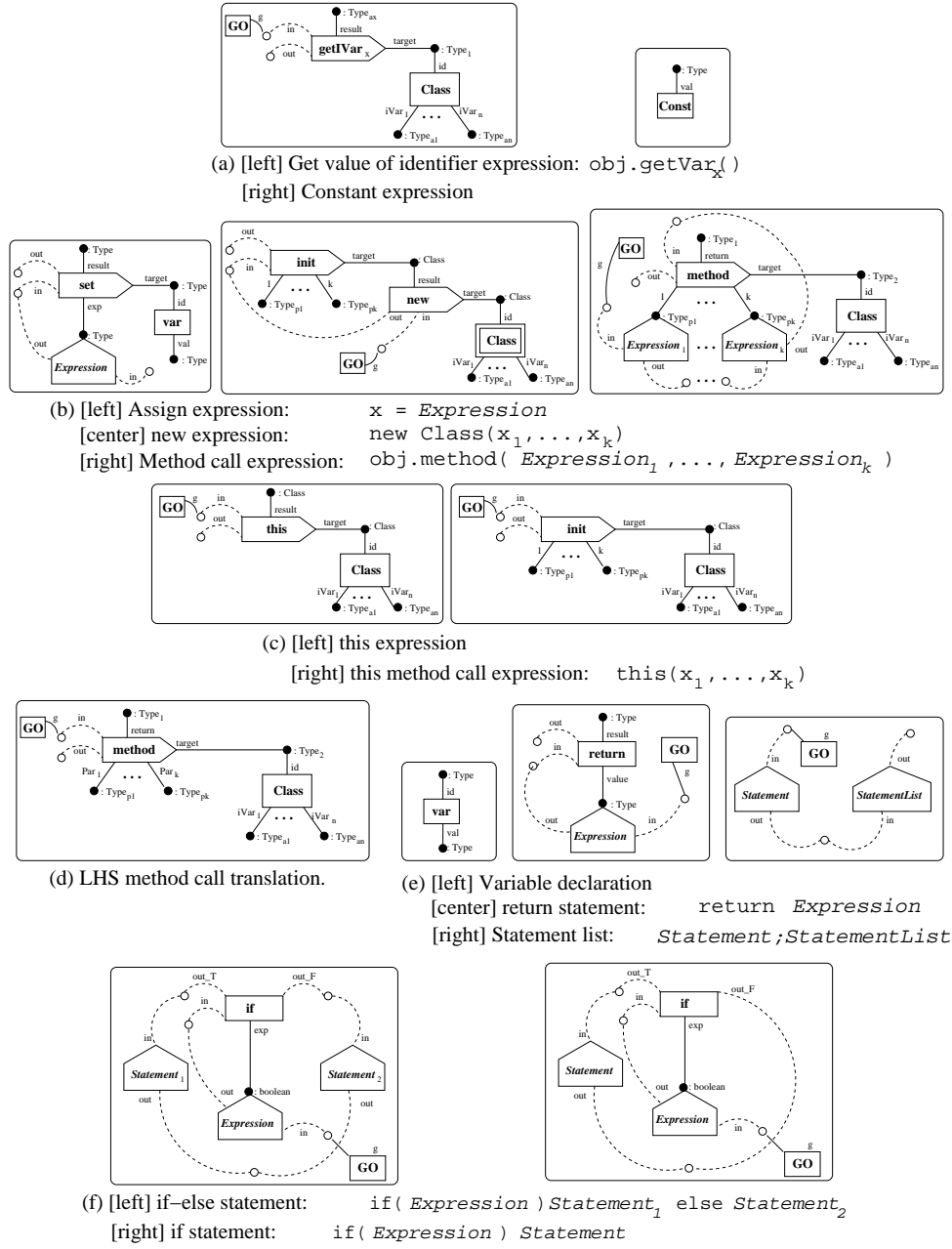


Fig. 3. Translation schema for Java statements.

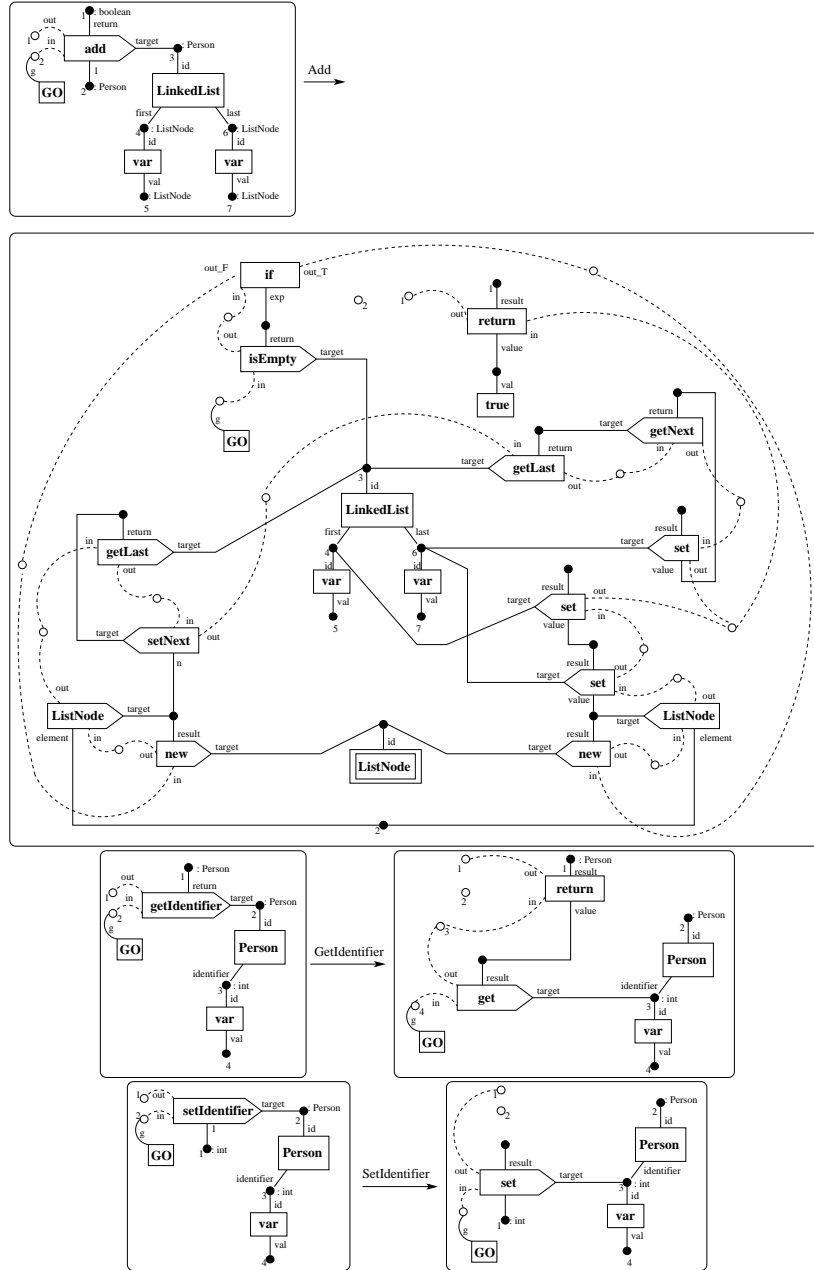


Fig. 4. The rules obtained from the translation of some instance methods of the Java declaration of Figure 1, namely `add(Person x)` of class `LinkedList`, and `getIdentifier()` and `setIdentifier(int n)` of class `Person`.

graphs), at the price of making the size of the graphs practically unmanageable. Notice also that `Equal_F` is a schema that represents one such rule for each pair of distinct constants `Const1` and `Const2` of basic data types.

As a topic of further research, we intend to explore a different encoding making use of attributed graphs, where only values of basic data types would be allowed as attributes, and where equality on basic data types (as well as the other predefined operators) would be evaluated in a suitable algebra. This would reduce drastically the size of graphs as well as the number of rules, still having at our disposal a rich theory, thanks to the recent results presented in [9] which extend to *attributed* GTSs many classical results of *typed* GTSs.

Rule `EqualObjF`, states that “`==`” returns `false` if applied to two distinct instances of a given class. But such a rule could be applied, using a non-injective match, also to a match of rule `Equal_T`, leading to an incorrect result. There are several ways to fix this problem and to ensure correctness: (1) we could require that all matches are *injective* (actually, *edge-injectivity* would suffice); (2) we could establish a priority between rules `Equal_T` and `EqualObjF`; or, (3) we could use GTSs with negative application conditions. Both solutions (1) and (3) have a solid theoretical background (see [13, 12]), but we still have to study in depth how the various solutions affect the applicability of the analysis techniques we are interested in.

Iterative statements. The fragment of Java we considered does not include any kind of loops (`while`, `for` or `do-while` statements), which nevertheless can be simulated by recursive method invocations: these are in general computationally more expensive, but efficiency considerations go beyond the scope of this paper. It is quite evident that iterative statements cannot be handled like the other operators with a fixed set of basic rules. In fact, with the proposed rules, the edge representing an operator is deleted when the operator is evaluated, and thus it cannot be evaluated more than once. Instead, each iterative statement could be represented by a distinct edge, with a corresponding rule replacing that edge with the body of the statement: a solution basically equivalent to a recursive method invocation.

We started investigating how to extend the proposed translation to several other features of Java. **Arrays** could be modeled by adding `ARRAY` edges with the desired arity(ies), and corresponding rules for getting and setting the array element values. However, the most naive solution is not satisfactory, as it requires one copy of such rules for each legal index. **Multi-threading** could be modelled within our setting by adding multiple `GO` edges to create different execution threads (synchronization mechanisms would also map to corresponding manipulations of `GO` edges – for instance, the notion of “join” could be mapped to the use of counters and elimination of some `GO` edges until a next command should be executed). Concerning **inheritance**, we are investigating the use of `var` edges to model sub-class relations (a related approach has been proposed in [19]). The idea consists of adding to the graph of types one `var` edge for each pair of classes related by inheritance: this would allow to model the fact that

a variable of type `Class1` can store objects of class `Class2` if `Class2` extends `Class1`.

6 Conclusion and Future Work

In this paper we presented a translation of a fragment of the Java language to typed Graph Transformation Systems. The idea was to encode values of data types as well as Java control structures graphically, and to use rules to simulate the execution of Java programs.

The proposed translation is to be considered as a first contribution of a long-term research activity, and there are several topics for further work.

Besides addressing the translation of other features of Java, as sketched in the previous section, we intend to investigate the encoding of a relevant part of the Java Virtual Machine, for example following the approach taken in [25], and to explore how far a graphical specification method based on GTSs could compete with the approach based on Abstract State Machines. Modelling the JVM would be the natural way to address certain aspects of the language that we deliberately ignored here, including, for example, dynamic class loading and garbage collection.

In particular, the handling of garbage has a direct impact on the applicability of the analysis techniques developed for GTSs. Notice that besides the usual *data garbage* including the objects that are not reachable anymore, since we also represent graphically the control structures, our graphs may include *control garbage*, for example the control structures belonging to branches of computations which are not executed. A direction we are investigating is to get rid of garbage in the analysis phase at an abstract level. More precisely, let us mention that the analysis technique proposed in [2] applies to GTSs that are finite-state *up to isolated nodes*. In the same vein, we would like to deal with graphs encoding Java computational states *up to garbage*.

Acknowledgement We would like to thank the anonymous referees for their detailed comments and constructive suggestions.

References

1. The AGG website. <http://tfs.cs.tu-berlin.de/agg>.
2. P. Baldan, A. Corradini, and B. König. Verifying Finite-State Graph Grammars: an Unfolding-Based Approach. In *Proc. of CONCUR'04*. Springer, 2004. To appear.
3. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In *Proc. of FoSSaCS'99*, volume 1578 of *LNCS*, pages 73–89. Springer, 1999.
4. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT'02*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
5. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.

6. F. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *Formal Methods for Open Object-based Systems IV*, pages 45–64. Kluwer Academic Publishers, 2000.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages, and Tools*. World Scientific, 1999.
8. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and comparison with Double Pushout Approach. In Rozenberg [24].
9. H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In *Proc. of ICGT'04*, LNCS. Springer, 2004. This volume.
10. F. Gadducci and U. Montanari. A Concurrent Graph Semantics for Mobile Ambients. In *Proc. of MFPS'01*, volume 45 of *ENTCS*. Elsevier Science, 2001.
11. J. Gosling, B. Joy, Steele G., and G. Bracha. *The Java Language Specification. 2nd Edition*. Sun Microsystems, Inc., 2000. <http://java.sun.com/docs/books/jls/>.
12. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.
13. A. Habel, J. Müller, and D. Plump. Double-Pushout Graph Transformation Revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
14. R. Heckel, J.M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Proc. of ICGT'02*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
15. D. Janssens and G. Rozenberg. Actor grammars. *Mathematical Systems Theory*, 22:75–107, 1989.
16. Martin Korff. True Concurrency Semantics for Single Pushout Graph Transformations with Applications to Actor Systems. In *Proc. of IS-CORE'94*, pages 33–50. World Scientific, 1995.
17. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, 1993.
18. Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
19. A.P. Lüdtke Ferreira and L. Ribeiro. Towards Object-Oriented Graphs and Grammars. In *Procs. FMOODS'03*, volume 2884 of *LNCS*, pages 16–31. Springer, 2003.
20. Detlef Plump. Term graph rewriting. In Ehrig et al. [7], chapter 1, pages 3–62.
21. Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 6:560–59, 1971.
22. Terrence W. Pratt. Definition of Programming Language Semantics using Grammars for Hierarchical Graphs. In *Proc. 1st Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 389–400. Springer, 1979.
23. Leila Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
24. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.
25. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
26. A. Wagner and M. Gogolla. Defining Operational Behaviour of Object Specifications by Attributed Graph Transformations. *Fundamenta Informaticae*, 26:407–431, 1996.