UNU/IIST

International Institute for
Software Technology

# Graph-Based Type System, Operational Semantics and Implementation of an Object-Oriented Programming Language

Wei Ke, Zhiming Liu, Shuling Wang and Liang Zhao

May 2009

UNU-IIST Report No. 410

$\mathcal{R}$

# UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research [R], Technical [T], Compendia [C] or Administrative [A]. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

The United Nations
University

**UNU/IIST**

**International Institute for
Software Technology**

P.O. Box 3058
Macao

# Graph-Based Type System, Operational Semantics and Implementation of an Object-Oriented Programming Language

## Wei Ke, Zhiming Liu, Shuling Wang and Liang Zhao

### Abstract

We present a graph-based model of a type system and an operational semantics for an object-oriented programming language. In this model, we define class graphs, object graphs and state graphs that naturally capture the essential OO features. The type system checks whether an expression, a command or a program is type-correct, based on its class graph and type context, which is a graph too. The operational semantics is a small-step one defined in the style of classical structural operational semantics, in which an execution step of a command is defined as a transition from one state graph to another obtained by simple operations on graphs. Both the type system and the semantics are fully implemented. On the one hand, working out the implementation has helped us to understand the semantics to make sure our definition is correct. On the other hand, the implementation will be used as the basis for our future development of simulation and validation techniques for OO programs, with the visualization of stage graph transitions during the execution. The motivation of this work is the development of techniques for reasoning about properties of OO programs stated in a logic of graphs. We show the promises

and feasibility of this by formulating a range of assertions about objects.

**Keywords**: OO programs, type system, operational semantics, class graphs, state graphs

**Wei Ke** is a PhD student registered in Beihang University, Beijing, and a researcher in Macao Polytechnic Institute, Macao. He is working on projects in object-oriented programming language semantics. Email: wke@ipm.edu.mo

**Zhiming Liu** is a Senior Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasizing sound methods and tools for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for object-oriented development. Email: Z.Liu@iist.unu.edu

**Shuling Wang** is a postdoctoral research fellow of UNU-IIST, working with Dr. Zhiming Liu and Dr. Xu Wang. Her research interest is in formal methods of object-oriented programs, component systems, and concurrent systems, including syntax, semantics, and verification. Email: wsl@iist.unu.edu

**Liang Zhao** is a PhD student in Computer Science of the joint PhD program of UNU-IIST and University of Pisa. His research interests include semantics and type systems of programming languages, graph transformation, and formal methods for object-oriented development. Email: liang@iist.unu.edu

**Copyright © 2009** by UNU-IIST

# Contents

**1 Introduction** **1**

**2 An Object-oriented Language** **2**

**3 Class Graphs, Object Graphs and State Graphs** **3**
3.1 Class graphs . . . 4
3.2 Object graphs . . . 5
3.3 State graphs . . . 6
3.4 Correctly typed object graphs and state graphs . . . 8
3.5 Graph operations . . . 9

**4 Type System** **11**
4.1 Type checking of expressions . . . 13
4.2 Type checking of commands . . . 13
4.3 Type checking of class graphs and programs . . . 15

**5 Operational Semantics** **15**
5.1 Evaluation of expressions . . . 15
5.2 Type safety of expressions . . . 16
5.3 Semantic rules . . . 18
5.4 Type safety of commands . . . 21
5.5 Semantics and type safety of programs . . . 24

**6 Implementation** **24**
6.1 Overview of design and issues . . . 24
6.2 Source files . . . 27
6.3 Increment graph pattern . . . 30
6.4 Class structure . . . 31
6.5 Language and parsing . . . 33
6.6 Type checking and graph transforming . . . 34

**7 Application** **34**

**8 Conclusions** **36**

**A Syntax Diagram of the Implemented rCOS** **37**

# 1 Introduction

The behavior of an OO program is complex and reasoning about it is hard. The main reason is that its execution states have complex structures and properties of related objects. These structures are determined by the class structure of the program. Complexity is in general the cause of breakdowns of a system and OO programs are typically prone to errors of a null pointer (or reference), an inaccessible object and aliases [12].

A formal semantic model in general makes (or should make) two major contributions. The first is to provide *conceptual clarification* for better understanding so as to master the complexity better, and the second is to support the development of techniques and tools for reasoning about programs. The work we present in this paper primarily aims at the former, but with a promise of establishing a basis for advancing the state of the art of the techniques and tool support for verification and analysis of OO programs.

We define an operational semantics for an OO programming language that is originally defined with a denotational semantics and a refinement calculus [11, 23] for the rCOS method of component-based model driven design [17, 20]. For this, we define objects of a class and execution states of a program as directed labeled graphs. A node represents an object or a simple datum. However, in the former case, a node is not labeled by an explicit reference value, but by the name of its runtime type that is a name of a class of the program. An edge is labeled by the name of a field of the source object referring to the target object.

An invocation to a method of an object does not only access to and operate on the fields of "*this*" object (*self* instead of "*this*" is used in this paper), but also the temporarily declared variables and the scope of the execution changes when another method is called inside this method. For this, we define a state graph in which edge labels of the temporary variables are arranged on the top of the graph in a *stack*, according to their scopes, by specially \$-labeled edges.

Changes on states can be simply defined in terms of operations on these graphs, such as swinging a path (a reference variable) to another node (object) and deleting a node (an object). A structural operational semantics can be easily defined in the classical manner: an execution step of a command from a state is a transition to another state graph via the defined operations on the initial graph. A change of the scope of execution is done by pushing in or popping out a scope node of the state graph, and garbage collection can be performed by deleting all the nodes that are not reachable from the root.

Properties of objects and states, such as conflict-freedom among aliases, accessibility of one object by another and absence of null references, can be described as predicates [7] of their corresponding graphs. This allows the graphs to be used to interpret a graph based logic, such as Logics of Aliasing [4], and the operational semantics as the basis to develop a graph-based Hoare-logic for static analysis of OO programs.

While we are lifting objects and states to graphs and treating them as instance values of variables

in the manner as we model programs with only pure data, we also lift the class definitions of a program and the declarations as a whole as type graphs, called *class graphs* [23]. This allows us to define a simple type system of the language. Furthermore, with structural refinement relations defined for class graphs in our earlier work [23], the operational semantics will support a rewriting system for proving equivalence up-to structural refinement mapping among programs.

In this paper, we also show both the type system and the operational semantics are easy to implement. The implementation is written in Java directly according to the semantic rules, and a program produces the visualized graphs step by step during its execution. Therefore, the language can be directly used for simulation and validation.

We introduce in the next section the syntax of our OO language. We define in Section 3 class graphs, object graphs and state graphs, followed by their operations. Our type system and operational semantics are provided in Section 4 and Section 5, respectively. Their implementation is introduced in Section 6. We show in Section 7 examples of properties of programs that can be stated and analyzed in this model without formality. Conclusions are drawn in Section 8 with a discussion on related work and future work.

## 2 An Object-oriented Language

We assume four disjoint sets: $\mathcal{C}$ of class names, $\mathcal{D}$ of names of primitive data types such as *Int* and *Bool*, $\mathcal{A}$ of names of attributes and variables, $\mathcal{M}$ of names of methods. Let $\mathcal{T}$ be the union of $\mathcal{C}$ and $\mathcal{D}$. The OO programming language we consider is that of rCOS [11] and its syntax is given in Fig. 1. It supports most of the essential object-oriented features, including inheritance, type casting, dynamic binding and recursive objects, and it is now being extended to a component-based architectural description language with facilities of interfaces and compositions [20].

In Fig. 1, the terminals $T$ and $S$ are type names in $\mathcal{T}$, $a$ an attribute (or field) name, $m$ a method name, $d$ a constant datum of a primitive type, $f$ a built-in operation of a primitive data type, and $x$ and $y$ variables. We use $\overline{u}$ to denote a sequence of elements $u_1 \cdot u_2 \cdots u_k$ and $\varepsilon$ for the empty sequence. The concatenation of two sequences is denoted by $\overline{u} \cdot \overline{v}$. We do not distinguish between an element and a singleton sequence.

***Program, class and method***. The language is similar to Java. A program *prog* is a sequence of class declarations *cdecls* followed by a main method *Main*. A class $C$ can be declared as public or private (the default is public), and may have a *direct superclass* $D$, declared as class $C$ extends $D$. An attribute (or field) declaration *adef* consists of a visibility annotation (private, protected, or public), its type, name and initial value, which is a literal. We do not consider attribute overriding. A method declaration consists of the method name $m$, its value parameters ($\overline{S\ x}$), result parameters ($\overline{T\ y}$), and the body $c$. Because rCOS is also used as a specification language, it allows a method to return a number of outputs. We would like to follow the classical manner in defining the semantics and do not want expressions to have side effects. This is why we allow a method to have result parameters instead of returning a value directly. For simplicity, we assume

$$
\begin{array}{lcl}
prog & ::= & cdecls \bullet Main \\
cdecls & ::= & cdecl \mid cdecl; cdecls \\
cdecl & ::= & [\mathsf{private}]\ \mathsf{class}\ C\ \ [\mathsf{extends}\ D]\, \{\overline{adef}\,; \overline{mdef}\ \} \\
adef & ::= & visib\ T\ a = l \\
visib & ::= & \mathsf{private} \mid \mathsf{protected} \mid \mathsf{public} \\
mdef & ::= & m(\overline{S\ x}; \overline{T\ y})\{c\} \\
c & ::= & \mathsf{skip} \mid C.new(le) \mid le := e \mid \mathsf{var}\ T\ x\, [= e]\ \mid \mathsf{end}\ x \\
 & \mid & e.m(\overline{e}; \overline{le}) \mid c; c \mid c \lhd b \rhd c \mid b * c \\
b & ::= & true \mid false \mid e = e \mid \neg b \mid b \wedge b \mid b \vee b \\
e & ::= & le \mid self \mid (C)e \mid l \mid f(\overline{e}) \\
le & ::= & x \mid e.a \\
l & ::= & d \mid null \\
Main & ::= & (\overline{ext}; c) \\
ext & ::= & T\ x = l
\end{array}
$$

Figure 1: The rCOS syntax

all methods are public, and can be inherited by a subclass. As a key feature of OO, a method is allowed to be overridden in a subclass, but its signature, i.e. types of parameters, should be preserved. The main method also declares the *external variables* $\overline{ext}$ of the program, which are accessible in the main method. Each external variable declaration consists of its type, name and initial value. We could follow Java to declare a class with $\overline{ext}$ as its attributes and the method $main()$, but it would cause some hiccups in our discussion.

***Command***. $e.m(\overline{e}; \overline{le})$ denotes the method call within the object that $e$ refers to, where $\overline{e}$ and $\overline{le}$ are respectively the actual value parameters and result parameters. $C.new(le)$ creates an object of class $C$ whose attributes are initialized with the initial values as declared in $C$, and then attach it to $le$. Command $\mathsf{var}\ T\ x = e$ declares a local variable $x$ of type $T$ with initial value $e$, while $\mathsf{end}\ x$ ends the scope of the local variables $x$.

***Expression***. Expressions include *assignable expressions* $le$, or we say left-side expressions sometimes, the special variable $self$ that represents the currently active object, expressions with type casting $(C)e$, literals $l$, and expressions $f(\overline{e})$ constructed with built-in operations $f$ of primitive types. Note that $C.new(le)$ is a command rather than an expression, thus expressions in rCOS have no side effects.

# 3 Class Graphs, Object Graphs and State Graphs

We define class graphs, object graphs, and state graphs in this section and discuss their relations. We also define graph operations that we need in the upcoming type system and operational semantics.

### 3.1 Class graphs

The class structure of an object-oriented program can be represented as a *directed and labeled graph* [23]. A node represents a class of objects or a type of data and it is labeled by a type name in $\mathcal{T}$. All nodes are labeled by different names (this is different from object graphs that are to be defined later). An edge is labeled by the name of an attribute, or a designated symbol $\rhd$ to represent the inheritance of one class from another.

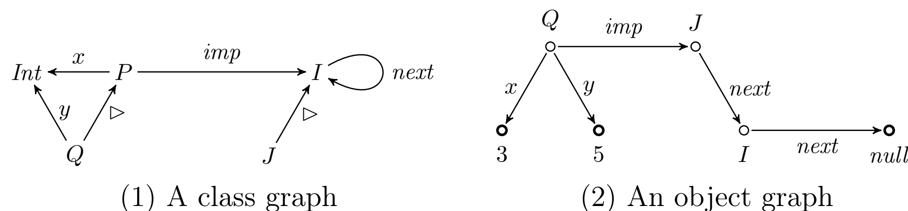
**Definition 1 (Class Graph)** *A class graph is a directed and labeled graph* $\Gamma = \langle N, E, M\rangle$, *where*

- $N \subseteq \mathcal{T}$, *denoted by* $\Gamma.node$, *is the set of nodes. Each node represents a class or a primitive data type,*
- $E \subseteq N \times (\mathcal{A} \cup \{\rhd\}) \times N$ *is the set of edges, denoted by* $\Gamma.edge$. *An edge* $(C, a, D) \in E$ *means class* $C$ *has an attribute* $a$ *of type* $D$, *and an edge* $(C, \rhd, D) \in E$ *says* $C$ *is a direct subclass of* $D$,
- $M$ *is a function that maps each class node to a set of method definitions, denoted by* $\Gamma.method$, *and* $m(S\ x; T\ y)\{c\} \in M(C)$ *means the method* $m$ *is defined in class* $C$.

We use $C \rhd_\Gamma D$ to denote that $C$ is a direct subclass of $D$ in $\Gamma$, and $\preccurlyeq_\Gamma$ the subtype relation defined by $\Gamma$, which is the extension of the reflexive and transitive closure of $\rhd_\Gamma$ on $\mathcal{T}$. We always omit the subscript $\Gamma$ when there is no confusion.

Not every class graph defined above can represent the class declarations of a syntactical program. We thus define the *well-formed* graphs. A class graph $\Gamma = \langle N, E, M\rangle$ is well-formed if the following conditions hold.

- Data types can only be used to label leaves: $(C, a, D) \in E$, implies $C \in \mathcal{C}$.
- Labels of outgoing edges from a node are different, and thus attributes of a class are distinct and there is no multiple inheritance.
- Conditions for the inheritance:
  - the inheritance relation is only defined among classes,
  - there is no cycle formed by $\rhd$ edges,
  - no attribute overriding is allowed: if $C_1 \preccurlyeq C$, $C_1 \neq C$ and $(C, a, D) \in E$, $(C_1, a, T) \notin E$ for any type $T$.
- Conditions for methods:
  - names of methods defined in each class are distinct,




(1) A class graph (2) An object graph

Figure 2: Class and object graphs

- class types of parameters of a method and class types used in the method body must be nodes of the graph: for each class name $C$ that occurs in $m(S\ x; T\ y)\{c\}$, $C \in N$,
- overriding of a method preserves the method signature: if $m(S\ x; T\ y)\{c\} \in M(C)$, $C' \preccurlyeq C$ and $m(S'\ x'; T'\ y')\{c'\} \in M(C')$, $S' = S$ and $T' = T$.

In the rest of the paper, a class graph always means a well-formed class graph unless it is stated otherwise. An example of class graph is shown in Fig. 2(1).

To represent more static features of the program, we extend the class graph with a set of functions. For a class $C$ in a class graph $\Gamma = \langle N, E, M\rangle$, $attr(C)$ denotes the labels of the outgoing edges from $C$, i.e. the attributes directly defined in $C$, and $Attr(C)$ the set of attributes of $C$ as well as those of all its superclasses, i.e. the *actual attributes* of $C$ including those through inheritance. They are defined by

$$\begin{array}{rcl} attr(C) & \widehat{=} & \{a \in \mathcal{A} \mid \exists T \in N \bullet (C, a, T) \in E\} \\ Attr(C) & \widehat{=} & \{a \in \mathcal{A} \mid \exists D \bullet C \preccurlyeq D \wedge a \in attr(D)\}. \end{array}$$

For an attribute $a \in Attr(C)$, we use $dtype(C, a)$ to denote its *declared type* $T$, i.e. $(D, a, T) \in E$ for some $D \succcurlyeq C$, $init(C, a)$ its initial value and $visib(C, a)$ its visibility, which is either private, protected or public. We abuse the notation $visib()$ to denote also the visibility of classes, so that $visib(C) \in \{\mathsf{private}, \mathsf{public}\}$ for each class $C$. Besides, we introduce two partial functions $mtype(C, m)$ and $mbody(C, m)$ for looking up the parameter type (signature) and the body of a method $m$ from a class $C$, respectively. They will be used in defining the type checking and the semantic rules of method invocations.

$$mtype(C, m) \widehat{=} \begin{cases} (S; T) & \text{if } m(S\,x; T\,y)\{c\} \in method(C) \\ mtype(D, m) & \text{otherwise, if } C \rhd D \end{cases}$$

$$mbody(C, m) \widehat{=} \begin{cases} (x; y; c) & \text{if } m(S\,x; T\,y)\{c\} \in method(C) \\ mbody(D, m) & \text{otherwise, if } C \rhd D \end{cases}$$

## 3.2 Object graphs

An object graph describes a family of objects and their relations. A node represents either an object, called an *object node* and labeled by its class, or a constant value, called a *value node* and

labeled by the constant. An edge represents an attribute of the source object, and its target is the node representing the object or value that the attribute refers to.

Let $\mathcal{N}$ be an infinite set of node names and $\mathcal{L}$ the set of constant values (literals) including the *null* object and values of primitive types. We introduce the special symbol $\perp \notin \mathcal{L}$ to represent an "undefined value" and define $\mathcal{L}_\perp \mathrel{\widehat{=}} \mathcal{L} \cup \{\perp\}$.

**Definition 2 (Object Graph)** *An object graph is a directed and labeled graph $G = \langle N, E, T, F \rangle$, where*

- *$N \subseteq \mathcal{N}$ is the set of nodes, denoted by $G.node$,*
- *$E \subseteq N \times \mathcal{A} \times N$ is the set of edges, denoted by $G.edge$,*
- *$T : N \rightharpoonup \mathcal{C}$ is a partial mapping from nodes to types, denoted by $G.type$,*
- *$F : N \rightharpoonup \mathcal{L}_\perp$ is a partial mapping from nodes to values, denoted by $G.value$,*

*such that*

1. *a node is either an object node or a value node: $\mathbf{dom}(T) \cap \mathbf{dom}(F) = \emptyset$ and $\mathbf{dom}(T) \cup \mathbf{dom}(F) = N$,*
2. *labels of the outgoing edges from a node are different, and*
3. *all value nodes are leaves, having no outgoing edges.*

An example of object graph is shown in Fig. 2(2).

***Edge and Path***. Let $G$ be an object graph. We write $n_1 \xrightarrow{a} n_2$ for an edge $(n_1, a, n_2) \in G.edge$. For a set $ns \subseteq G.node$ of nodes (or a single node), $in(ns)$ and $out(ns)$ respectively denote the sets of incoming edges to and outgoing from them. For a non-empty path $p$, i.e. a sequence of consecutive edges, we define $source(p)$ and $target(p)$ to be the source and target node of $p$, $first(p)$ and $last(p)$ the first and last edge, respectively. Note that these notations for edges and paths are applicable to all directed and labeled graphs, not only object graphs.

### 3.3 State graphs

A state at a moment of time in the execution of an OO program consists of the existing objects, the attribute links between them and values of the data attributes at that time. These at that time form an object graph.

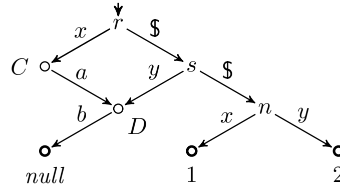



Figure 3: A state graph

Roughly speaking, each step of the execution of the program in a state is to change the state by creating a new object, deleting an old object, forming a new link, removing an existing link, changing a link, or modifying a data attribute. Obviously, all these changes of the state can be considered as simple operations on the initial object graph.

However, we are interested in a small step semantics, and we need to define semantics of changes of local variables and nested method invocations. For this, we define the notion of *state graphs* that introduces stacks into object graphs.

**Definition 3 (State Graph)** *A state graph is a rooted, directed and labeled graph* $G = \langle N, E, T, F, r\rangle$*, where*

- $N$*,* $T$ *and* $F$ *are defined as in Definition 2 of object graphs,*
- $E \subseteq N \times (\mathcal{A} \cup \{self, \$\}) \times N$ *is the set of edges, denoted by* $G.edge$*,*
- $r \in N$ *is the root of the graph, i.e. without incoming edges, denoted by* $G.root$*,*
- *starting from* $r$*, the* $-edges, *if there are any, form a path such that except* $r$ *each node on the path has only one incoming edge.*

We call the $-path of $G$ the *stack* of the state graph and call the nodes on this path the *scope nodes*. When entering a new scope, a new node representing this scope is pushed onto the top of the stack, and when exiting a scope, the top node is popped out (together with the outgoing edges from it). A state graph in shown in Fig. 3. When the execution enters $\mathsf{var}\ y; \mathsf{var}\ x; \cdots; \mathsf{end}\ x; \mathsf{end}\ y$, it pushes a new scope node $s$ onto the top of node $n$ with variable $y$ being attached to it; then the execution proceeds to $\mathsf{var}\ x; \cdots; \mathsf{end}\ x; \mathsf{var}\ y$, a new scope is entered and thus a new scope node $r$ is pushed on the top of node $s$ with the newly declared variable $x$ being attached to it. When $\mathsf{end}\ x$ is executed, $r$ together with $x$ is popped out, then in the same way node $s$ will be popped out together with $y$.

A state graph represents a proper execution state of a program only if it satisfies the conditions (2)&(3) of object graphs and the following two well-formedness conditions:

1. a node is either a scope node, an object node or a value node, and

2. the source of each edge labeled by *self* is a scope node and its target is an object node.

In the rest of the paper, we always assume a state graph well-formed. Besides, a state graph is called *stable* if it does not contain $-edges, i.e. the stack is empty.

***Trace and graph isomorphism***. A node $n$ is *accessible* in $G$, denoted by $\mathsf{access}(G, n)$, if it is reachable via a path staring from the root node, and $G$ is *connected* if all nodes are accessible. The sequence of edge labels $a_1.a_2 \ldots a_k$ uniquely determines the target node of a path from the root node $G.root \xrightarrow{a_1} \cdots \xrightarrow{a_k} n_k$, and it therefore uniquely represents an object or a value, depending on the type of the target node. We call such a sequence of edge labels a *trace* and ignore the difference between a path starting from the root and its trace.

In an abstract model, we do not distinguish graphs with only different choice of names of their nodes from $\mathcal{N}$, and this can be formalized by the notion of graph isomorphism. Two state graphs $G$ and $G'$ are *isomorphic* if there is a bijective function $g$ from $G.node$ to $G'.node$, such that

1. $g(G.root) = G'.root$,
2. $n_1 \xrightarrow{a} n_2 \in G.edge$ iff $g(n_1) \xrightarrow{a} g(n_2) \in G'.edge$, and
3. $G.type(n) = G'.type(g(n))$ and $G.value(n) = G'.value(g(n))$

Any two isomorphic state graphs have the same set of traces. Furthermore, we can identify value nodes with their traces, and combine different leaves with equal values into one, that is to make the mapping $G.value$ injective. From now on, we do not distinguish a value node from its value. And we assume a value node is in the state when needed, as otherwise it can always be added.

## 3.4 Correctly typed object graphs and state graphs

We have two kinds of types, namely class types and primitive data types. However, they are not enough if we want to reason about the type of the literal *null*, which is likely to be a value in an object or state graph. For this purpose, we introduce a special type *Null*, and assume that it is a subtype of every class type. Note that such an assumption does not lead to multiple inheritance, since *Null* does not inherit attributes or methods from any class. We use $\mathbf{T}(l)$ to denote the type of a literal $l$. For example, $\mathbf{T}(5) = Int$ and $\mathbf{T}(null) = Null$.

The allowable objects and states of a program, both being represented by graphs, are determined by the class declarations of the program, which is represented by a class graph too. For a class graph $\Gamma$, an object graph or a state graph $G$ is *correctly typed* with respect to $\Gamma$, or $\Gamma$-typed, if the following conditions hold.

1. The type of each object is defined in the class graph: for each object node $n$ of $G$, $G.type(n) \in \Gamma.node$.

2. Each attribute is correctly typed according to the class graph: for each object node $n$ and edge $n \xrightarrow{a} n'$ of $G$, there exist nodes $C, T \in \Gamma.node$ such that $n \preccurlyeq C$, $C \xrightarrow{a} T \in \Gamma.edge$ and $n' \preccurlyeq T$.

Here and in the following discussion, we abuse the notation $n \preccurlyeq T$ to denote that the type of an object node or a value node $n$ is a subtype of $T$, specifically:

- if $n$ is an object node, $G.type(n) \preccurlyeq T$;
- if $n$ is a value node, $\mathbf{T}(G.value(n)) \preccurlyeq T$ unless $G.value(n) = \bot$.

A state graph $G$ is a *valid* state of a program *prog* if the following conditions hold.

1. $G$ is correctly typed with respect to the class graph of *prog*.
2. The last node $n_0$ of the stack of $G$ records the external variables of *prog*: for each edge $n_0 \xrightarrow{x} n$ of $G$, $x$ is an external variable of *prog* declared with some type $T$ and $n \preccurlyeq T$.

In the rest of the paper, we are only interested in correctly typed object graphs and valid state graphs, while we do not explicitly mention the program or its class graph when there is no confusion.

### 3.5 Graph operations

We define a few basic operations on state graphs, which we are to use in the semantic definitions.

***Swing an edge***. The most often operation for changing a state $G$ is done by an assignment $le.a := e$, where $le$ can be the empty path $\varepsilon$. It causes the swing of the $a$-edge to point to the object or value of $e$. For an edge $d = n_1 \xrightarrow{a} n_2$ and a node $n$ of $G$,

$$swing(G, d, n) \mathrel{\widehat{=}} G'$$

such that $G'$ is the same as $G$ except that

$$G'.edge = (G.edge \setminus \{d\}) \cup \{n_1 \xrightarrow{a} n\}.$$

For a trace $p$, we use $swing(G, p, n)$ for $swing(G, last(p), n)$, i.e. swinging a path means swinging its last edge. See Fig. 4. In the following operations, when defining a new graph, we just list the part different from the old graph.

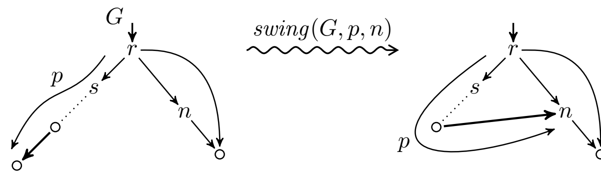



Figure 4: Path swing

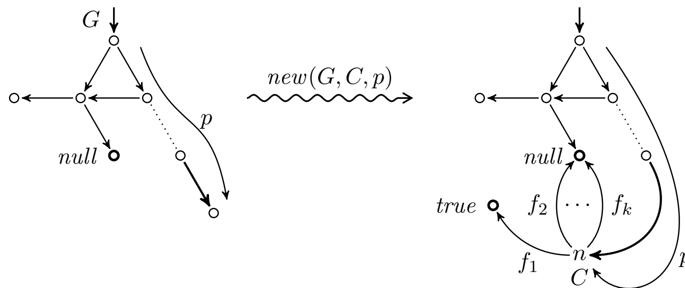



Figure 5: Object creation

***Create an object.*** Adding an object node is slightly tricky and we need to consider the type of the node and its attributes. Creating a new object of class $C$ and attaching it to trace $p$ in $G$ is defined by

$$new(G, C, p) \mathrel{\widehat{=}} swing(G', p, n)$$

such that $n \notin G.node$, and

$$\begin{aligned} G'.node &= G.node \cup \{n\} \\ G'.edge &= G.edge \cup \{n \xrightarrow{a} init(C, a) \mid a \in Attr(C)\} \\ G'.type &= G.type \cup \{n \mapsto C\}. \end{aligned}$$

An example of object creation is shown in Fig. 5. In this example, an object $n$ of class $C$ is created, with attributes $f_1, \cdots, f_k$ set to their corresponding initial values. Then the path $p$ is swung to the new object.

***Garbage collection.*** After applying a graph operation, some nodes can become inaccessible. Let $garbage(G)$ be the set of all the inaccessible nodes in $G$. We define the garbage collection below.

$$G^{\bullet} \mathrel{\widehat{=}} delete(G, garbage(G))$$

where $delete(G, ns)$ removes from $G$ the nodes $ns$ and their incoming and outgoing edges, including removing $ns$ from the domain of $G.type$ and $G.value$.

***Stack operations.*** For a sequence of variables $\overline{x} = x_1 \cdots x_k$ and nodes $\overline{n} = n_1 \cdots n_k$, $push(G, \overline{x}, \overline{n})$ adds a new scope with outgoing edges labeled by $\overline{x}$ and pointing to the nodes $\overline{n}$, accordingly:

$$push(G, \overline{x}, \overline{n}) \mathrel{\widehat{=}} G'$$

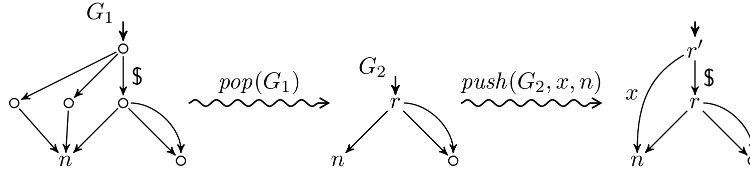



Figure 6: Stack push and pop

such that $r' \notin G.node$, and

$$\begin{aligned} G'.node &= G.node \cup \{r'\} \\ G'.edge &= G.edge \cup \{r' \xrightarrow{x_1} n_1, \cdots, r' \xrightarrow{x_k} n_k, r' \xrightarrow{\$} r\} \\ G'.root &= r'. \end{aligned}$$

As shown in Fig. 6, ending a scope pops the root, if it exists, out of the stack by simply removing it from the graph, but the next node on the stack becomes the root.

$$pop(G) \mathrel{\widehat{=}} G'^{\bullet} \qquad \text{if } r \xrightarrow{\$} r_{next} \in G.edge$$

such that

$$\begin{aligned} G'.edge &= G.edge \setminus \{r \xrightarrow{\$} r_{next}\} \\ G'.root &= r_{next}. \end{aligned}$$

# 4 Type System

The type system checks whether a program is type-correct (or well-typed) before executing the program. It is constructed in a bottom-up way: from the checking of expressions and commands to that of methods, class graphs and, finally, programs.

Generally, the type-correctness of an expression or a command depends on a specifical class graph. For example, the command $C.new(o); o.a := 1$ is well-typed only under a class graph with a class $C$ which has an integer attribute $a$. In fact, a class graph itself is still not enough to decide whether an expression (or a command) is type-correct. For a variable $x$, it is always well-typed inside a local scope var $T\ x; \cdots;$ end $x$, but is likely to be ill-typed (undefined) outside. This tells us that the type-correctness of an expression also depends on its type environment, which can be visualized by a graph, called *type context*.

**Definition 4 (Type Context)** *A type context is a rooted, directed and labeled graph* $\Delta = \langle N, E, r\rangle$*, where*

- $N \subseteq \mathcal{N} \cup \mathcal{T}$ *is the set of nodes, denoted by* $\Delta.node$*, including* scope nodes $N \cap \mathcal{N}$ *and* type nodes $N \cap \mathcal{T}$*,*
- $E \subseteq N \times (\mathcal{A} \cup \{self, \$\}) \times N$ *is the set of edges, denoted by* $G.edge$*, and*
- $r \in N \cap \mathcal{N}$ *is the root of the graph, i.e. without incoming edges, denoted by* $G.root$*,*

*such that*

1. *labels of the outgoing edges from a node are different,*
2. *an \$-edge is associated with two scope nodes; while an edge labeled by a variable name (or self) starts from a scope node and ends at a type node, and*
3. *starting from the root r, the \$-edges, if there are any, form a path such that except r each node on the path has only one incoming edge.*

A type context represents a snapshot of the type environment at a time of type checking, recording the types of variables (including *self*) declared in all the scopes at that time. Similar to state graphs, a type context has a stack structure. When the type checking enters a new scope, a node with outgoing edges recording the variables in the new scope is pushed onto the top of the stack; when the checking exits a scope, the top node of the stack, together with its outgoing edges, is popped out, so that the type context recovers to the one exactly before entering the scope.

The notion of trace and graph isomorphism is also suitable for type contexts. Therefore, we do not distinguish type contexts differing only in the choice of names of their scope nodes from $\mathcal{N}$. Moreover, we assume a type node is in a type context when needed, as otherwise it can always be added.

Let $\Delta$ be a type context, $n$ be one of its scope nodes and $w$ be a variable name or *self*, we define a partial function $search(\Delta, n, w)$ that searches for the paths of $w$ in $\Delta$ from $n$ node-by-node down the stack.

$$search(\Delta, n, w) \mathrel{\widehat{=}} \begin{cases} n \xrightarrow{w} n' & \text{if } \exists n' \bullet n \xrightarrow{w} n' \in \Delta.edge \\ n \xrightarrow{\$} n_1 \cdot search(\Delta, n_1, w) & \text{otherwise, if } \exists n_1 \bullet n \xrightarrow{\$} n_1 \in \Delta.edge \end{cases}$$

Note that the recursion always terminates as there is no loop formed by \$-edges. Based on this function, we define $\Delta(w)$ as the type of $w$ in $\Delta$:

$$\Delta(w) \mathrel{\widehat{=}} target(search(\Delta, \Delta.root, w)).$$

So, $\Delta(w)$ exists if and only if $w$ *occurs in* $\Delta$, i.e. there exists an edge in $\Delta$ labeled by $w$. Moreover, we use $var(\Delta, n)$ to denote the set of variables (together with *self*) recorded in the scope node $n$ of $\Delta$, and $size(\Delta)$ the size of $\Delta$, i.e. the length of its \$-path.

$$var(\Delta, n) \mathrel{\widehat{=}} \{w \in \mathcal{A} \cup \{self\} \mid \exists n' \bullet n \xrightarrow{w} n' \in \Delta.edge\}$$
$$size(\Delta) \mathrel{\widehat{=}} \#\{n \xrightarrow{\$} n' \in \Delta.edge\},$$

$$\text{(T-Var)}\ \frac{\Delta(x) = T}{\Delta \vdash x : T} \qquad \text{(T-Self)}\ \frac{\Delta(\mathit{self}) = C}{\Delta \vdash \mathit{self} : C} \qquad \text{(T-Attr)}\ \frac{\Delta \vdash e : C \quad \mathit{dtype}(C, a) = T \quad C \text{ sees } a}{\Delta \vdash e.a : T}$$

$$\text{(T-UCast)}\ \frac{\Delta \vdash e : C' \quad C' \preccurlyeq C}{\Delta \vdash (C)e : C} \qquad \text{(T-DCast)}\ \frac{\Delta \vdash e : C' \quad C \preccurlyeq C' \quad C \neq C'}{\Delta \vdash (C)e : C}$$

$$\text{(T-Lit)}\ \frac{\mathbf{T}(l) = B}{\Delta \vdash l : B} \qquad \text{(T-Op)}\ \frac{f : B \to B' \quad \Delta \vdash e : B}{\Delta \vdash f(e) : B'}$$

Figure 7: Type checking of expressions

where $\#s$ is the number of elements of a set $s$. We write $\mathit{var}(\Delta)$ as a shorthand for $\mathit{var}(\Delta, \Delta.\mathit{root})$, i.e. the set of variables in the current scope.

Since type contexts and state graphs are similar in the stack structure, most stack-related operations defined for one kind of graphs are also applicable to the other kind. For example, we can directly use $\mathit{search}(G, n, w)$ for a state graph $G$ and $\mathit{pop}(\Delta)$ for a type context $\Delta$ without redefining them.

## 4.1 Type checking of expressions

For a type context $\Delta$, a class graph $\Gamma$, an expression $e$ and a type $T$, we use $\Delta \vdash_\Gamma e : T$ to denote that $e$ is well-typed and of type $T$ under $\Delta$ and $\Gamma$. In this notation, the subscript $\Gamma$ can be omitted when there is no confusion. The typing rules for expressions are given in Fig. 7. In these rules, $C$ sees $a$ means

1. $\mathit{visib}(C, a) = \mathsf{public}$, or
2. $\mathit{visib}(C, a) = \mathsf{protected}$ and $\Delta(\mathit{self}) \preccurlyeq C$, or
3. $\mathit{visib}(C, a) = \mathsf{private}$ and $\Delta(\mathit{self}) = C$.

## 4.2 Type checking of commands

Intuitively, a type-correct command should satisfy certain conditions. For example, the types of the two expressions in the left side and right side of an assignment should match with each other, and the types of actual parameters in a method invocation should be consistent with the signature of the method. Such kind of requirements is captured by the notion of *local well-typedness*. For type contexts $\Delta$, $\Delta'$, a class graph $\Gamma$ and a command $c$, we use $\Delta \to \Delta' \vdash_\Gamma c$ to denote that $c$ is

$$(\text{T-SKIP})\ \Delta \vdash \mathsf{skip} \qquad (\text{T-NEW})\ \frac{\Delta \vdash le : C' \quad C \preccurlyeq C'}{\Delta \vdash C.new(le)}$$

$$(\text{T-ASSIGN})\ \frac{\Delta \vdash le : T' \quad \Delta \vdash e : T \quad T \preccurlyeq T'}{\Delta \vdash le := e} \qquad (\text{T-DECL})\ \frac{\left[\Delta \vdash e : T' \quad T' \preccurlyeq T\right]}{\Delta \rightarrow push(\Delta, x, T) : size(\Delta) \vdash \mathsf{var}\ T\ x\,[= e]}$$

$$(\text{T-END})\ \frac{\mathsf{var}(\Delta) = \{x\} \quad size(\Delta) \geq 1}{\Delta \rightarrow pop(\Delta) : size(\Delta) - 1 \vdash \mathsf{end}\ x}$$

$$(\text{T-INVK})\ \frac{\Delta \vdash e : C \quad \Delta \vdash ve : S' \quad \Delta \vdash re : T' \quad mtype(C, m) = (S; T) \quad S' \preccurlyeq S \quad T \preccurlyeq T'}{\Delta \vdash e.m(ve; re)}$$

$$(\text{T-SEQ})\ \frac{\Delta \rightarrow \Delta'' : k_1 \vdash c_1 \quad \Delta'' \rightarrow \Delta' : k_2 \vdash c_2}{\Delta \rightarrow \Delta' : \min(k_1, k_2) \vdash c_1; c_2} \qquad (\text{T-IF})\ \frac{\Delta \vdash b : Bool \quad \Delta \vdash c_1 \quad \Delta \vdash c_2}{\Delta \vdash c_1 \triangleleft b \triangleright c_2}$$

$$(\text{T-WHILE})\ \frac{\Delta \vdash b : Bool \quad \Delta \vdash c}{\Delta \vdash b * c}$$

Figure 8: Type checking of commands

locally well-typed under $\Delta$ and $\Gamma$, and that the type context turns to $\Delta'$ after the checking of $c$. Furthermore, a locally well-typed command $c$ (under $\Delta$ and $\Gamma$) is well-typed, denoted as $\Delta \vdash_\Gamma c$, if its variable declarations and endings are always matched. In these notations, the subscript $\Gamma$ can always be omitted when there is no confusion.

We would expect that $\Delta \vdash c$ is equivalent to $\Delta \rightarrow \Delta \vdash c$, but it is not true. Suppose $c_1 = \mathsf{var}\ x; \mathsf{end}\ x$ and $c_2 = \mathsf{end}\ x; \mathsf{var}\ x$. It is easy to find a type context $\Delta_0$ such that both $c_1$ and $c_2$ are locally well-typed under $\Delta_0$. However, in the sense of well-typedness, $c_2$ is quite different from $c_1$: it can not be well-typed under any type context since its variable declaration and ending are not matched. In order to distinguish between commands such as $c_1$ and $c_2$, we define $\Delta \vdash c$ by $\Delta \rightarrow \Delta : size(\Delta) \vdash c$. The extended notation $\Delta \rightarrow \Delta' : k \vdash c$ means $\Delta \rightarrow \Delta' \vdash c$ and the natural number $k$ is the smallest size of type contexts during the checking of $c$. Now, the difference of $c_1$ and $c_2$ is clear: $\Delta_0 \rightarrow \Delta_0 : size(\Delta_0) \vdash c_1$ but $\Delta_0 \rightarrow \Delta_0 : size(\Delta_0) - 1 \vdash c_2$. We give the type checking rules for commands in Fig. 8.

### 4.3 Type checking of class graphs and programs

Based on the type system of commands established so far, we are able to check the type-correctness of class graphs and programs. A class graph is well-typed if each method defined in the graph is well-typed, and a method definition is well-typed if its body command is well-typed according to its formal parameters.

**Definition 5 (Well-typed Method)** *A method* $m(S\ x; T\ y)\{c\}$ *defined in class* $C$ *of class graph* $\Gamma$ *is well-typed, denoted by* $\vdash_\Gamma C :: m$*, if* $\Delta \vdash_\Gamma c$*, where* $\Delta = \langle\{r, C, S, T\}, \{r \xrightarrow{self} C, r \xrightarrow{x} S, r \xrightarrow{y} T\}, r\rangle$.

**Definition 6 (Well-typed Class Graph)** *A class graph* $\Gamma$ *is well-typed, if for each attribute* $a$ *defined in some class* $C$, $\mathbf{T}(init(C, a)) \preccurlyeq dtype(C, a)$*, and for each method* $m$ *defined in some class* $C$, $\vdash_\Gamma C :: m$.

Let $prog = cdecls \bullet Main$ be an OO program and $\Gamma$ be the graph representation of its class declaration section *cdecls*. In this case, *prog* is denoted by $\Gamma \bullet Main$. It is well-typed if $\Gamma$ is well-typed and *Main* is well-typed under $\Gamma$.

**Definition 7 (Well-typed Program)** *A program* $prog = \Gamma \bullet Main$ *is well-typed, where* $Main = (T_1\ x_1 = l_1, \cdots, T_k\ x_k = l_k; c)$*, if*

- $\Gamma$ *is well-typed,*
- *for each class name* $C$ *occurring in Main,* $visib(C) = \mathsf{public}$,
- *for* $1 \leq i \leq k$, $\mathbf{T}(l_i) \preccurlyeq T_i$*, and*
- $\Delta_{init} \vdash_\Gamma c$*, where* $\Delta_{init} = \langle\{r, T_1, \cdots, T_k\}, \{r \xrightarrow{x_i} T_i \mid 1 \leq i \leq k\}, r\rangle$.

## 5 Operational Semantics

Using the state graphs, we now simply follow the classical routine to define the evaluation of an expression and then the state transition rules.

### 5.1 Evaluation of expressions

Given a state graph $G$, the evaluation of an expression $e$ returns an object node or a constant value. We use $eval(G, e)$ to denote the value of $e$, and $rtype(G, e)$ to denote the *runtime type* or

*current type* of $e$ in state $G$.

In order to evaluate an expression, we should first calculate its trace. Given a state graph $G$, we use $trace(G, e)$ to denote the *trace* of an expression $e$. It is calculated inductively, using the function $search()$ defined in Section 4.

$$\begin{aligned} trace(G, w) &\mathrel{\widehat{=}} search(G, G.root, w) \\ trace(G, e.a) &\mathrel{\widehat{=}} trace(G, e).a \\ trace(G, (C)e) &\mathrel{\widehat{=}} trace(G, e) \end{aligned}$$

where $w$ is a variable or *self*. For the example $G_0$ in Fig. 2(2), $trace(G_0, x) = x$ and $trace(G_0, y) = \$.y$. From now on, when there is only one state graph, we omit the argument $G$ in the graph operations that we have defined.

The value and the runtime type of an expression $e$ in $G$ are determined inductively as follows.

1. If $e$ is a constant value $l$, then $eval(e) = l$ and $rtype(e) = \mathbf{T}(l)$,
2. If $e$ is a variable $x$ or *self*, $e$ can be evaluated in $G$ only when $trace(e)$ exists in $G$. Let $n = target(trace(e))$. If $n$ is an object node, $eval(e) = n$ and $rtype(e) = G.type(n)$, otherwise $eval(e) = G.value(n)$ and $rtype(e) = \mathbf{T}(eval(e))$.
3. If $e$ is $e'.a$, $e$ can be evaluated in $G$ only when $trace(e)$ exists in $G$. Let $n = target(trace(e))$. If $n$ is an object node, $eval(e) = n$ and $rtype(e) = G.type(n)$, otherwise $eval(e) = G.value(n)$ and $rtype(e) = \mathbf{T}(eval(e))$.
4. If $e$ is of the form $(C)e'$, then $eval(e) = eval(e')$ and $rtype(e) = rtype(e')$, provided that $rtype(e') \preccurlyeq C$.
5. For $e = f(e')$, $eval(e) = f(eval(e'))$ and $rtype(e) = \mathbf{T}(eval(e))$.

Note that for an expression $e = e'.a$ such that $eval(e)$ exists, $eval(e')$ must be an object node with an attribute $a$. We call $eval(e')$ the *parent object* of $e'.a$.

## 5.2 Type safety of expressions

We will show that well-typed expressions have good properties in evaluation. For this purpose, we need to study the consistency relation between type contexts and state graphs.

A type context $\Delta$ and a state graph $G$ are consistent if they have the similar structure. It is defined inductively by the following conditions, where $w$ is either a variable $x$ or *self*.

- For $w \in var(\Delta)$, $w \in var(G)$ and $G(w) \preccurlyeq \Delta(w)$.

- For $w \in var(G)$, $w \in var(\Delta)$ if $w$ occurs in $\Delta$.
- $pop(\Delta)$ and $pop(G)$ are consistent, unless $size(\Delta) = size(G) = 0$.

By induction on the size of $\Delta$, it is easy to verify that $\Delta$ and $G$ are consistent implies $size(\Delta) = size(G)$ and $G(w) \preccurlyeq \Delta(w)$ for any $w$ occurring in $\Delta$.

We expect to prove that a well-typed expression can be evaluated to a value, but there are the following cases of exceptions.

- **Exception 1 (null reference)**: the evaluation of an expression $e.a$ or the execution of a command $e.m(ve, re)$ fails, if $e$ is evaluated to *null*.
- **Exception 2 (uninitialized variable)**: the evaluation of a variable fails, if it is not initialized (defined) after declaration.
- **Exception 3 (illegal downcast)**: the evaluation of an expression $(C)e$ fails, if the runtime type of $e$ is not a subtype of $C$.

These three cases of exceptions can not be checked and avoided statically. However, the type system can help us to preclude all the other failures, i.e. the evaluation of a well-typed expression will not "go wrong".

**Theorem 1 (Type safety of expressions)** *Let $\Delta$ be a type context, $e$ be an expression, $T$ be a type and $G$ be a state graph consistent with $\Delta$. If $\Delta \vdash e : T$, then $eval(G, e)$ exists and $rtype(G, e) \preccurlyeq T$, unless one of the exception cases happens.*

*Proof.* By induction on the structure of $e$. Suppose none of the exception cases happens.

- *Case $x$* (And similarly *Case self*).

  Since $\Delta \vdash x : T$ can only be resulted from (T-VAR), we have $\Delta(x) = T$. So, $target(trace(x)) = G(x) \preccurlyeq \Delta(x) = T$, note that $\Delta$ is consistent with $G$. Let $n = target(trace(x))$. If $n$ is an object node, $eval(x) = n$ and $rtype(x) = G.type(n) \preccurlyeq T$. If $n$ is a value node, $eval(x) = G.value(n) \neq \perp$ and we still have $rtype(x) = \mathbf{T}(eval(x)) \preccurlyeq T$, provided that Exception 2 does not happen.

- *Case $e'.a$.*

  Since $\Delta \vdash e'.a : T$ can only be resulted from (T-ATTR), we have $\Delta \vdash e' : C$ and $dtype(C, a) = T$ for some class $C$. From the deduction hypothesis, $eval(e')$ exists and $rtype(e') \preccurlyeq C$. Since Exception 1 does not happen, $eval(e') \neq null$. So, $eval(e') = target(trace(e')) = n'$ and $rtype(e') = G.type(n') = C' \preccurlyeq C$ for some node $n'$ and class $C'$. From $dtype(C, a) = T$ and $C' \preccurlyeq C$, we know $dtype(C', a) = T$, thus there exists

$n' \xrightarrow{a} n \in G.edge$ for some node $n$ such that $n \preccurlyeq T$. So, $target(trace(e'.a)) = n$. If $n$ is an object node, $eval(e'.a) = n$ and $rtype(e'.a) = G.type(n) \preccurlyeq T$. If $n$ is a value node, $eval(e'.a) = G.value(n)$ and we still have $rtype(e'.a) = \mathbf{T}(eval(x)) \preccurlyeq T$, provided that Exception 2 does not happen.

- *Case* $(C)e'$.

  Since $\Delta \vdash (C)e' : T$ can only be resulted from (T-UCAST) or (T-DCAST), we have $T = C$, $\Delta \vdash e' : C'$ from some $C'$ satisfying $C' \preccurlyeq C$ or $C \preccurlyeq C'$. From the deduction hypothesis, $eval(e')$ exists and $rtype(e') \preccurlyeq C'$. In the case $C' \preccurlyeq C$, $rtype(e') \preccurlyeq C$; and in the case $C \preccurlyeq C'$, $rtype(e') \preccurlyeq C$ still holds since Exception 3 does not happen. So we always have $eval((C)e') = eval(e')$ and $rtype((C)e') = rtype(e') \preccurlyeq C = T$.

- *Case* $l$.

  Since $\Delta \vdash l : T$ can only be resulted from (T-LIT), we know that $T$ is a primitive data type such that $\mathbf{T}(l) = T$. As a result, $eval(l) = l$ and $rtype(l) = \mathbf{T}(l) = T \preccurlyeq T$.

- *Case* $f(e')$.

  Since $\Delta \vdash f(e') : T$ can only be resulted from (T-OP), we know that $T$ is a primitive data type, $f : B \rightarrow T$ and $\Delta \vdash e' : B$ for some $B$. From the deduction hypothesis, $eval(e')$ exist and $rtype(e') = \mathbf{T}(eval(e')) \preccurlyeq B$, which implies $\mathbf{T}(eval(e')) = B$ since both of them are primitive data types. As a result, $eval(f(e')) = f(eval(e'))$ and $rtype(f(e')) = \mathbf{T}(f(eval(e'))) = T \preccurlyeq T$.

### 5.3 Semantic rules

We define a small step semantics for our language by giving the transition relation between *configurations*. There are two kinds of configurations:

- a *non-terminated configuration* is a pair $\langle c, G\rangle$, where $c$ is a command, and $G$ is a state, and
- a *terminated configuration* is a state $G$, representing the completion of the execution of a command.

Fig. 9 gives the semantic rules that are relevant to the object-oriented features and the graph notation of states. The rules of sequential composition, conditional choice and iteration are defined in the standard way in which an operational semantics for an imperative language is defined.

The assignment $le := e$ swings the trace of $le$ to the value of $e$, and $C.new(le)$ creates a new initial instance of $C$ and swings the trace of $le$ to point to the new instance. A local variable declaration `var` $T\ x = e$ adds the variable $x$ to a new scope by pushing it onto the stack of the state and initiates it to the value of $e$; while `end` $x$ pops the root out of the state.

$$(\text{SKIP})\ \langle \mathsf{skip}, G\rangle \to G \qquad (\text{ASSIGN})\ \langle le := e, G\rangle \to \mathit{swing}(G, \mathit{trace}(le), \mathit{eval}(e))$$

$$(\text{NEW})\ \langle C.\mathit{new}(le), G\rangle \to \mathit{new}(G, C, \mathit{trace}(le)) \qquad (\text{DECL})\ \langle \mathsf{var}\ T\ x = e, G\rangle \to \mathit{push}(G, x, \mathit{eval}(e))$$

$$(\text{DECL-U})\ \langle \mathsf{var}\ T\ x, G\rangle \to \mathit{push}(G, x, \perp) \qquad (\text{END})\ \langle \mathsf{end}\ x, G\rangle \to \mathit{pop}(G)$$

$$(\text{ENTER})\ \begin{array}{l}\langle \mathit{enter}(C, S, T, x, y, e, ve, re), G\rangle \\ \quad \to \mathit{push}(G, \mathit{self} \cdot x \cdot y \cdot y^*, \mathit{eval}(e) \cdot \mathit{eval}(ve) \cdot \perp \cdot \mathit{po}(G, re))\end{array}$$

$$(\text{LEAVE})\ \langle \mathit{leave}(y, re), G\rangle \to \mathit{pop}(\mathit{swing}(G, \mathit{spo}(G, y^*, re), \mathit{eval}(y)))$$

$$(\text{INVK})\ \frac{\mathit{rtype}(e) = C \quad \mathit{mtype}(C, m) = (S; T) \quad \mathit{mbody}(C, m) = (x; y; c)}{\langle e.m(ve; re), G\rangle \to \langle \mathit{enter}(C, S, T, x, y, e, ve, re); c; \mathit{leave}(y, re), G\rangle}$$

$$(\text{SEQ})\ \frac{\langle c_1, G\rangle \to \langle c_1', G'\rangle}{\langle c_1; c_2, G\rangle \to \langle c_1'; c_2, G'\rangle} \qquad (\text{SEQ-PRI})\ \frac{\langle c_1, G\rangle \to G'}{\langle c_1; c_2, G\rangle \to \langle c_2, G'\rangle} \qquad (\text{IF-F})\ \frac{\mathit{eval}(b) = \mathit{false}}{\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_2, G\rangle}$$

$$(\text{IF-T})\ \frac{\mathit{eval}(b) = \mathit{true}}{\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_1, G\rangle} \qquad (\text{WHILE-F})\ \frac{\mathit{eval}(b) = \mathit{false}}{\langle b * c, G\rangle \to G} \qquad (\text{WHILE-T})\ \frac{\mathit{eval}(b) = \mathit{true}}{\langle b * c, G\rangle \to \langle c; b * c, G\rangle}$$

Figure 9: Operational semantics for commands in rCOS

Intuitively, a method invocation $e.m(ve, re)$ first records the value of the actual value parameter $ve$ in the formal value parameter of $m$, and then executes the method body. At the end it returns the value of the formal return parameter of $m$ to the actual return parameter $re$. However, the precise definition is complicated by the following issues.

***Method look-up***. First, dynamic binding of the method to the runtime type of $e$ requires the look-up for the signature $\mathit{mtype}(C, m) = (S; T)$ and the definition $\mathit{mbody}(C, m) = (x; y; c)$ of $m$. This is handled in Rule (INVK).

***Entering method body***. Then, the parent object of actual result parameter $re$ in the initial state should be recorded before it is possibly changed by the body command of the method. This is the "early result parameter binding" semantics. For this, in addition to have $\mathit{self}$ for recording $e$, the formal value parameter $x$ for holding the actual value parameter $ve$ and the formal return parameter $y$ being $\perp$, we need an auxiliary variable $y^*$, which corresponds to the formal return parameter $y$ and does not occur in the program, to record the parent object of $re$ in the initial state. Thus, we introduce an *implementation command* $\mathit{enter}(C, S, T, x, y, e, ve, re)$ whose semantics is defined by Rule (ENTER), that sets a new scope with variables $\mathit{self}$, $x$, $y$ and $y^*$ that are respectively initialized properly according to the above discussion. Function $\mathit{po}(G, re)$

class $N$ $Int\, a; N\, ln$

class $L$
  $N\, h; N\, p;$
  $m(N\, s; Int\, r)$
    if $s \neq null$
      $self.m(s.ln; r);$ ♠
      if $self.p = null \vee s.a \leq self.p.a$
        $self.p := s; r := s.a$
    else $self.p := null$

invocation: $self.m(self.p; self.p.a)$

Figure 10: A state graph at ♠ of the second nested invocation of $m$

returns the parent object of $re$ in $G$ that is going to be recorded by $y^*$.

$$po(G, re) \mathrel{\widehat{=}} \begin{cases} eval(G, e) & \text{if } re = e.a \\ \bot & \text{otherwise} \end{cases}$$

***Leaving method body***. When the execution is leaving the body of $m$, if $re$ is of the form $e.a$, the attribute $a$ of the old parent object of actual parameter $re$ must be swung to the value of the formal result parameter $y$. For this, we recover the trace by the function

$$spo(G, y^*, re) \mathrel{\widehat{=}} \begin{cases} y^*.a & \text{if } re = e.a \\ \$.trace(pop(G), x) & \text{if } re = x \end{cases}$$

The return of the method invocation is carried out by the implementation command *leave* $(y, re)$ whose semantics is in Rule (Leave) defined by the swing and pop operations.

Note that the use of these implementation commands instead of directly to use commands var $S$ $x = ve$; var $T$ $y$ and $re := y$; end $y$; end $x$ is to avoid possible name conflicts between actual parameters $ve, re$ and formal parameters $x, y$ used as local variables in the method body. Instead of implementation commands, literal values of the form Val $v$ are used in [16], that actually model addresses of variables.

**Example 1** *Fig. 10 presents an example to illustrate the problem for method invocation we discussed above. The program defines a node class $N$ and a list class $L$. The method $m$ of $L$ recursively traverses a list pointed to by $s$, and searches for the node with the minimum value, which will be returned by $r$. We number the roots which represent different local scopes for recursive calls of $m$, and discriminate local variables in these scopes by using the numbers as subscripts. When $self.m(self.p; self.p.a)$ is invoked, at the second entering of the recursive call (denoted by ♠), the formal value variable $s_{(3)}$ is initialized with the $s_{(2)}.ln$ of the outer scope; and at its exit, the actual result parameter $r_{(2)}$ of the outer scope will be set with $r_{(3)}$. On the other hand, the parent object of the actual result parameter $self.p.a$ gets changed during the method invocation because $self.p$ is assigned to refer to another object inside $m$. We use the corresponding*

*variable $r^*$ to remember the parent object of $self.p.a$ in the initial state, and at the end, we pass back the value of the formal result parameter $r$ to $r^*.a$.*

## 5.4 Type safety of commands

The type checking rules for the implementation commands *enter* and *leave* are shown in Fig. 11.

$$(\text{T-Enter})\ \frac{\Delta \vdash ve : S' \quad S' \preccurlyeq S \quad \Delta \vdash re : T'}{\Delta \rightarrow push(\Delta, self \cdot x \cdot y, C \cdot S \cdot T) : size(\Delta) \vdash \mathit{enter}(C, S, T, x, y, e, ve, re)}$$

$$(\text{T-Leave})\ \frac{\Delta \vdash y : T \quad pop(\Delta) \vdash re : T' \quad T \preccurlyeq T'}{\Delta \rightarrow pop(\Delta) : size(\Delta) - 1 \vdash \mathit{leave}(y, re)}$$

Figure 11: Type checking of implementation commands

We have already introduced three cases of exceptions in Section 5.2 which can not be avoided by type checking. If none of these exception cases occurs, the execution of a locally well-typed command will not be blocked.

**Theorem 2 (Type safety of commands)** *Let $\Delta$, $\Delta_{end}$ be two type contexts, $c$ be a command and $G$ be a state graph consistent with $\Delta$. If $\Delta \rightarrow \Delta_{end} \vdash c$, then*

- *either there exist a state graph $G'$ consistent with $\Delta_{end}$ such that $\langle c, G\rangle \rightarrow G'$,*
- *or there exist a configuration $\langle c', G'\rangle$ and a type context $\Delta'$ consistent with $G'$ such that $\langle c, G\rangle \rightarrow \langle c', G'\rangle$ and $\Delta' \rightarrow \Delta_{end} \vdash c'$,*

*unless one of the exception cases (see Section 5.2) happens.*

*Proof.* By induction on the structure of $c$. Suppose none of the exception cases happens.

- *Case* skip.

  Since $\Delta \rightarrow \Delta_{end} \vdash \mathsf{skip}$ can only be resulted from (T-Skip), we have $\Delta_{end} = \Delta$. Thus $G$ is consistent with $\Delta_{end}$. According to (Skip), $\langle \mathsf{skip}, G\rangle \rightarrow G$.

- *Case* $C.new(le)$.

  Since $\Delta \rightarrow \Delta_{end} \vdash C.new(le)$ can only be resulted from (T-New), we have $\Delta_{end} = \Delta$, $\Delta \vdash le : C'$ and $C \preccurlyeq C'$ for some $C'$. According to Theorem 1, $eval(le)$ exists, thus $trace(le)$ and $G' = new(G, C, trace(le))$ also exist. Here, $C \preccurlyeq C'$ ensures that $G'$ is also

a valid state graph. According to (NEW), $\langle C.new(le), G\rangle \rightarrow G'$. Note that the graph operation $new()$ does not affect scope nodes, and at most changes one of the their outgoing edges: when $le = x$, it swings such an $x$-labeled edge to an object node $n$ of class $C$. Even in this case, we still have $G'(x) \preccurlyeq \Delta(x)$ since $G'.type(G'(x)) = G'.type(n) = C \preccurlyeq C' = \Delta(x)$. As a result, $G'$ is consistent with $\Delta_{end}(= \Delta)$.

- *Case* $le := e$.

  Since $\Delta \rightarrow \Delta_{end} \vdash le := e$ can only be resulted from (T-ASSIGN), we have $\Delta_{end} = \Delta$, $\Delta \vdash le : T'$, $\Delta \vdash e : T$ and $T \preccurlyeq T'$ for some $T$ and $T'$. According to Theorem 1, $eval(le)$, $eval(e)$ exist and $rtype(e) \preccurlyeq T$. Thus $trace(le)$ and $G' = swing(G, trace(le), eval(e))$ also exist. Here, the validity of $G'$ is ensured by $rtype(e) \preccurlyeq T \preccurlyeq T'$. According to (ASSIGN), $\langle le := e, G\rangle \rightarrow G'$. Note that the graph operation $swing()$ does not affect scope nodes, and at most changes one of the their outgoing edges: when $le = x$, it swings such an $x$-labeled edge to the node $n$ representing the value of $e$. Even in this case, we still have $G'(x) = n \preccurlyeq rtype(e) \preccurlyeq T' = \Delta(x)$. As a result, $G'$ is consistent with $\Delta_{end}(= \Delta)$.

- *Case* $\mathsf{var}\ T\ x\,[= e]$.

  We only consider the case that $x$ is initialized by $e$. The proof is similar (and simpler) for the uninitialized case. Since $\Delta \rightarrow \Delta_{end} \vdash \mathsf{var}\ T\ x = e$ can only be resulted from (T-DECL), we have $\Delta \vdash e : T'$, $T' \preccurlyeq T$ and $\Delta_{end} = push(\Delta, x, T)$ for some $T'$. According to Theorem 1, $eval(e)$ exists and $rtype(e) \preccurlyeq T' \preccurlyeq T$, which means $G' = push(G, x, eval(e))$ also exists. Then, according to (DECL), $\langle \mathsf{var}\ T\ x = e, G\rangle \rightarrow G'$. Note that $G'$ is consistent with $\Delta_{end}$, since $var(G') = \{x\} = var(\Delta_{end})$, $G'(x) \preccurlyeq rtype(e) \preccurlyeq T = \Delta_{end}(x)$.

- *Case* $\mathsf{end}\ x$.

  Since $\Delta \rightarrow \Delta_{end} \vdash \mathsf{end}\ x$ can only be resulted from (T-END), we have $\Delta_{end} = pop(\Delta)$, thus $pop(G)$ is consistent with $\Delta_{end}$. According to (END), $\langle \mathsf{end}\ x, G\rangle \rightarrow pop(G)$.

- *Case* $\mathsf{enter}(C, S, T, x, y, e, ve, re)$.

  Since $\Delta \rightarrow \Delta_{end} \vdash \mathsf{enter}(C, S, T, x, y, e, ve, re)$ can only be resulted from (T-ENTER), we have $\Delta_{end} = push(\Delta, self \cdot x \cdot y, C \cdot S \cdot T)$, $\Delta \vdash ve : S'$, $S' \preccurlyeq S$ and $\Delta \vdash re : T'$ for some $S'$, $T'$. We also have $rtype(e) = C$ (thus $eval(e)$ exists), note that $\mathsf{enter}(C, S, T, x, y, e, ve, re)$ can only be triggered by (INVK). According to Theorem 1, $eval(ve)$, $eval(re)$ exist and $rtype(ve) \preccurlyeq S' \preccurlyeq S$, which implies $po(G, re)$ also exists. As a result, $G' = push(G, self \cdot x \cdot y \cdot y^*, eval(e) \cdot eval(ve) \cdot \perp \cdot po(G, re))$ always exists. According to (ENTER), $\langle \mathsf{enter}(C, S, T, x, y, e, ve, re), G\rangle \rightarrow G'$. From (1) $var(G') = \{self, x, y, y^*\} = var(\Delta_{end}) \cup \{y^*\}$, (2) $y^*$ does not occur in $\Delta_{end}$, (3) $G'(self) \preccurlyeq rtype(e) = C = \Delta_{end}(self)$, (4) $G'(x) \preccurlyeq rtype(ve) \preccurlyeq S = \Delta_{end}(x)$, and (5) $G'(y) \preccurlyeq \Delta_{end}(y)$ as $G'.value(G'(y)) = \perp$, we conclude that $G'$ is consistent with $\Delta_{end}$.

- *Case* $\mathsf{leave}(y, re)$.

  Since $\Delta \rightarrow \Delta_{end} \vdash \mathsf{leave}(y, re)$ can only be resulted from (T-LEAVE), we have $\Delta_{end} = pop(\Delta)$, $\Delta \vdash y : T$, $pop(\Delta) \vdash re : T'$ and $T \preccurlyeq T'$ for some $T$, $T'$. According to Theorem 1, $eval(pop(G), re)$, $eval(y)$ exist and $rtype(y) \preccurlyeq T \preccurlyeq T'$. Thus $spo(G, y^*, re)$, $G'' = swing(G, spo(G, y^*, re), eval(y))$ and $G' = pop(G'')$ also exist. Here, the validity

of $G''$ is ensured by $rtype(y) \preccurlyeq T'$, and $G'$ is valid since $G''$ is valid. According to (LEAVE), $\langle \mathsf{leave}\,(y, re)\,, G\rangle \to G'$. Note that the graph operation $swing()$ does not affect scope nodes, and at most changes one of the their outgoing edges: when $re = x$, it swings such an $x$-labeled edge to the node $n$ representing the value of $y$. Even in this case, we still have $n \preccurlyeq rtype(y) \preccurlyeq T'$. As a result, $G''$ is consistent with $\Delta$, which implies $G'$ is consistent with $\Delta_{end}$.

- *Case* $e.m(ve; re)$.

  Since $\Delta \to \Delta_{end} \vdash e.m(ve, re)$ can only be resulted from (T-INVK), we have $\Delta_{end} = \Delta$, $\Delta \vdash e : C'$, $\Delta \vdash ve : S'$, $\Delta \vdash re : T'$, $mtype(C', m) = (S; T)$, $S' \preccurlyeq S$ and $T \preccurlyeq T'$ for some $C'$, $S$, $T$, $S'$ and $T'$. According to Theorem 1, $eval(e)$ exists and $rtype(e) \preccurlyeq C'$. Since Exception 1 does not happen, we have $eval(e) \neq null$, which means $rtype(e) = C$ for some class $C \preccurlyeq C'$. So, $mtype(C, m) = mtype(C', m) = (S; T)$. Suppose $mbody(C, m) = (x; y; c_0)$ and let $c' = \mathsf{enter}\,(C, S, T, x, y, e, ve, re)\,; c_0; \mathsf{leave}\,(y, re)$. According to (INVK), $\langle e.m(ve, re), G\rangle \to \langle c', G\rangle$. Then, we need to show that $c'$ is locally well-typed. This is done by three steps. (1) From (T-ENTER), we have $\Delta \to \Delta' \vdash \mathsf{enter}\,(C, S, T, x, y, e, ve, re)$, where $\Delta' = push(\Delta, self \cdot x \cdot y, C \cdot S \cdot T)$. (2) Here, $m$ must be a well-typed method defined in some superclass $C''$ of $C$, i.e. $\Delta_1 \vdash c_0$ for $\Delta_1 = \langle \{r, C'', S, T\}, \{r \xrightarrow{self} C'', r \xrightarrow{x} S, r \xrightarrow{y} T\}, r\rangle$. Note that the change of the type of $self$ to a subclass in the type context does not affect the type-correctness of a command, we have $\Delta_2 \vdash c_0$ for $\Delta_2 = \langle \{r, C, S, T\}, \{r \xrightarrow{self} C, r \xrightarrow{x} S, r \xrightarrow{y} T\}, r\rangle$. Since $\Delta'$ is simply an extension of $\Delta_2$, we have $\Delta' \vdash c_0$, which implies $\Delta' \to \Delta' \vdash c_0$. (3) From (T-LEAVE), we have $\Delta' \to \Delta \vdash \mathsf{leave}\,(y, re)$, provided that $\Delta' \vdash y : T$ and $pop(\Delta') = \Delta$. Finally, according to (1), (2) and (3), we conclude that $\Delta \to \Delta \vdash c'$ by using (T-SEQ).

- *Case* $c_1; c_2$.

  Since $\Delta \to \Delta_{end} \vdash c_1; c_2$ can only be resulted from (T-SEQ), we have $\Delta \to \Delta_{mid} \vdash c_1$ and $\Delta_{mid} \to \Delta_{end} \vdash c_2$ for some $\Delta_{mid}$. From the deduction hypothesis, there are two cases: either (1) $\langle c_1, G\rangle \to G'$ for some $G'$ consistent with $\Delta_{mid}$; or (2) $\langle c_1, G\rangle \to \langle c_1', G'\rangle$ and $\Delta' \to \Delta_{mid} \vdash c_1'$ for some $\Delta'$, $c_1'$ and $G'$ consistent with $\Delta'$. In the case (1), we have $\langle c_1; c_2, G\rangle \to \langle c_2, G'\rangle$ according to (SEQ-PRI). In the case (2), we have $\langle c_1; c_2, G\rangle \to \langle c_1'; c_2, G'\rangle$ according to (SEQ), and $\Delta' \to \Delta_{end} \vdash c_1'; c_2$ according to (T-SEQ).

- *Case* $c_1 \lhd b \rhd c_2$.

  Since $\Delta \to \Delta_{end} \vdash c_1 \lhd b \rhd c_2$ can only be resulted from (T-IF), we have $\Delta_{end} = \Delta$, $\Delta \vdash b : Bool$, $\Delta \vdash c_1$ and $\Delta \vdash c_2$. So, $eval(b)$ equals $true$ or $false$, $\Delta \to \Delta_{end} \vdash c_1$ and $\Delta \to \Delta_{end} \vdash c_2$. If $eval(b) = true$, we have $\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_1, G\rangle$ according to (IF-T). Otherwise, we get $\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_2, G\rangle$ according to (IF-F).

- *Case* $b * c_1$.

  Since $\Delta \to \Delta_{end} \vdash b * c_1$ can only be resulted from (T-WHILE), we have $\Delta_{end} = \Delta$, $\Delta \vdash b : Bool$ and $\Delta \vdash c_1$. As a result, $eval(b)$ equals $true$ or $false$ and $\Delta \to \Delta \vdash c_1$. If $eval(b) = false$, we have $\langle b * c_1, G\rangle \to G$ according to (WHILE-F). Otherwise, we get $\langle b * c_1, G\rangle \to \langle c_1; b * c_1, G\rangle$ according to (WHILE-T), and $\Delta \to \Delta_{end} \vdash c_1; b * c_1$ according to (T-SEQ).

### 5.5 Semantics and type safety of programs

The semantics of a program is the execution of the main command under the initial state, which is a stable state whose root records the external variable referring to their initial values. Specifically, the initial configuration of $prog = \Gamma \bullet (T_1\ x_1 = l_1, \cdots, T_k\ x_k = l_k; c)$ is $\langle c, G_{init} \rangle$, where $G_{init} = \langle \{r, n_1, \cdots, n_k\}, \{r \xrightarrow{x_i} n_i \mid 1 \leq i \leq k\}, \emptyset, \{n_i \mapsto l_i \mid 1 \leq i \leq k\}, r \rangle$.

If none of the exception cases occurs, the execution of a well-typed program will not "go wrong". And if it terminates, the final state is also a stable one.

**Theorem 3 (Type safety of programs)** *For a well-typed program prog* $= \Gamma \bullet (T_1\ x_1 = l_1, \cdots, T_k\ x_k = l_k; c)$*,*

- *either there exists a stable state graph* $G_{end}$ *consistent with* $\Delta_{init}$ *(see Definition 7) such that* $\langle c, G_{init} \rangle \to G_{end}$*,*
- *or there exist a configuration* $\langle c', G' \rangle$ *and a type context* $\Delta'$ *consistent with* $G'$ *such that* $\langle c, G_{init} \rangle \to \langle c', G' \rangle$ *and* $\Delta' \to \Delta_{init} \vdash c'$*,*

*unless one of the exception cases (see Section 5.2) happens.*

*Proof.* Since *prog* is well-typed, we have $\Delta_{init} \vdash c$, which implies $\Delta_{init} \to \Delta_{init} \vdash c$. Note that $\Delta_{init}$ and $G_{init}$ are consistent, we can apply Theorem 2 and the proof is completed.

## 6 Implementation

### 6.1 Overview of design and issues

We use Java as our implementation language, and ANTLR as our parser generator, because of its Java origin and powerful grammar specification language. The implementation consists of

1. a parser, that translates source programs into class graphs and command sequences;
2. a type checker, that checks class graphs and commands following the well-typedness rules and definitions;
3. a transformer, that transforms state graphs by applying the main command to an initial graph, in the manner of decomposed small steps, following the semantic rules;

4. an observer, that intercepts, layouts and exports intermediate graphs to descriptions in the PGF/TikZ language to be incorporated into TeX documents.

Fig. 12 shows the overall data flow of how a program is processed, where $\Delta$, $\Gamma$ and $G$ stand for type contexts, class and state graphs, respectively. The Kamada-Kawai algorithm [14] is used in our implementation to auto-layout state graphs, and the results appear to be reasonably pleasing.

***Graph representation.*** Class graphs and state graphs forbid a node having outgoing edges with duplicated labels. This enables us to store the nodes $N \subseteq \mathcal{N}$ and the edges $E \subseteq \mathcal{N} \times \mathcal{A} \times \mathcal{N}$ of a graph $G$ in a mapping $S : \mathcal{N} \to \mathcal{A} \to \mathcal{N}$. With such a representation, it is efficient to retrieve the target from a source and a label, and all the outgoing edges from a source. Leaf nodes are also stored as sources mapped to nothing in the mapping for membership tests. We have

$$N = \mathbf{dom}\, S, \quad \mathit{label}(\mathit{out}(n)) = \mathbf{dom}\, S(n) \quad \text{and} \quad \mathit{target}(n, a) = S(n)(a),$$

where $n$ is a node and $a$ is a label.

Nodes are implemented as a Java interface `Node`, and they are identified by instance identities of the objects, such as type names and constant values, implementing the interface. These objects can thus be treated as nodes directly.

The Java API employs two kinds of equality: reference equality (`==`) and content equality (`equals`). Since the identity is the only thing significant for a node, it suits for reference equality. Names, values and labels have their external meanings, they are constructed in various places, thus must be identified by their contents. However, each name or value instance stored in a graph is also a node, we introduce an additional mapping to look up its identity (if exists in the graph) from its content. This resembles the `intern` method of class `String` in the Java API. Ordered contents can be stored as keys in class `TreeMap`, while unordered object identities have to be used as hash values and stored in class `HashMap`.

***Graph transformation and optimization.*** State graphs are immutable in our implementation, which allows intermediate graphs to be retrieved. Every transformation of a graph returns a new graph. Different graphs may share nodes and labels, while each keeps its own mappings of the elements.

A garbage collection operation (*gc*) is performed after each step of execution, and it is effectively done by a depth first search of the graph (*dfs*). Primitive graph operations are first performed on a temporary graph, that we call the *increment graph*, and their grand effect is added upon

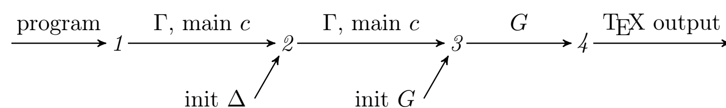



Figure 12: Data flow of program processing

the base graph with only one *dfs*. The relations in the increment graph override those in the base graph. There is no need for a decrement graph, since the only operation that may remove elements is the *gc*.

A new graph is constructed while performing the *dfs* by adding visited entries to the mappings representing the new graph. We never remove entries from the mappings, allowing us to implement our curried mappings using `Maps` of `Maps`, where removals of entries may cause a *domino effect*.

In the theory, we always start searching for a path from the root. Sometimes we need to first pop a graph and then search for a path. Popping a graph generates a new graph and it is expensive. We introduce *pointed paths* in the implementation to allow specifying a search origin point in a path. We can specify the scope node next to the root as the origin in the same graph when popping for searching is needed, eliminating thecreation of a new graph.

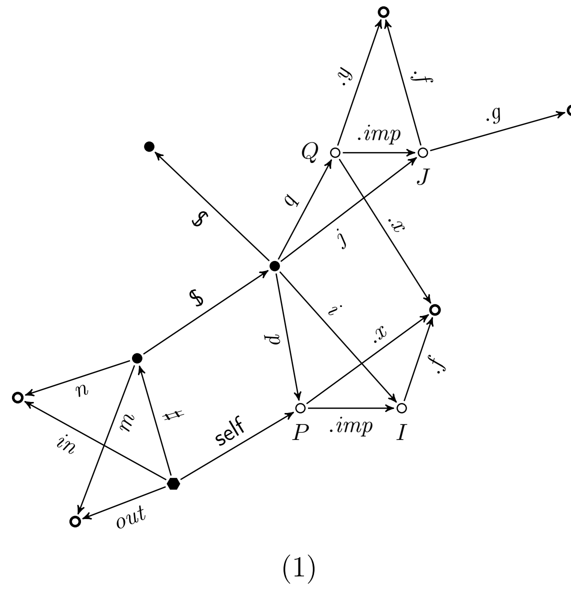
***Frame links***. For the presence of global variables, which are visible in all methods, [1] the type checking stacks up two scopes for method bodies: the global variables and the method local variables, including *self* and the parameters. On the other hand, in the graph transforming, all scope nodes from the root down to the bottom are available for path searching. This produces an inconsistency between the type system and the semantics, where a label accepted as a global variable by the type system may be intercepted by a local variable in some middle scope with the same label during program execution. To deal with this problem, we either reject global variables in all method bodies during the type checking or block the middle scopes and search the global scope separately at runtime. In our implementation, we take the latter approach by introducing another special frame link label '$\sharp$' to link the new root created for a method invocation with the old root, separating the two method frames. The searching of a path stops at the frame link, preventing a method body from seeing the variables outside, then, if the path is not found, the search continues in the global scope, which is located at the bottom of the $\mathtt{\$}/\sharp$-path. This exactly matches the type checking process.

***Examples***. Two examples in Fig. 13 illustrate the graph system. Fig. 13(1) is an instance of the bridge pattern, and Fig. 13(2) illustrates the capturing of parent objects of actual result parameters. They are generated by the auto-layout algorithm.

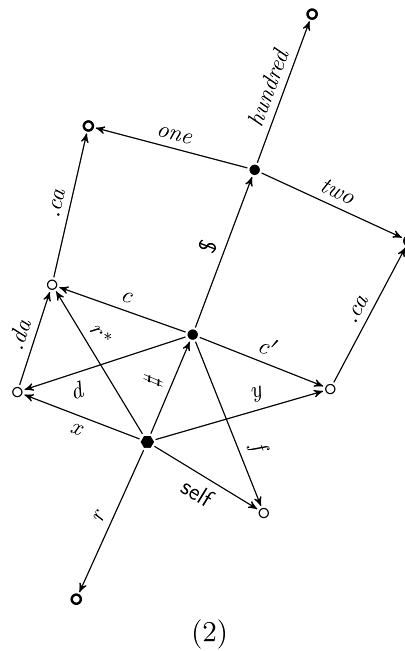
In Fig. 13(1), there is an explicit pointer in each abstraction object pointing to its implementor object. It is hard to see subclass relations in state graphs, since attribute origins are merged along the inheritance path when objects are instantiated. $Q$ is a subclass of $P$, and they are abstractions; $J$ is a subclass of $I$, and they are implementors. We also see the relation between formal and actual parameters, for the graph is obtained within a method call.

In Fig. 13(2), the result parameter $r$ will be bound to an attribute $c.ca$ upon the method return. We record the parent object $c$ using an edge with auxiliary label $r^*$ to avoid losing it, in the case of $c$ being pointed to somewhere else during the method call.

[1] Our theory restricts the visibility of global variables only to the main command for simplicity, while in our implementation, we bring back the usual meaning of global to be practical.




(1)




(2)

Figure 13: Examples of auto-generated state graphs

## 6.2 Source files

Following the common Java project approach, the source files are stored in `src/` directory while compiled objects are stored in `bin/` directory. There are some rCOS programs (`.r` files) as examples in `examples/` and the packed binary package in `jar/`.

The source files are further divided into three packages:

- *compiler*, consisting of the grammar files, the generated parsers, the drivers to initiate and monitor parsing, type checking, graph transforming and output generating, the symbol table management and the user front end;
- *graph*, consisting of the graph management and operations, the expression and command management and operations, including the state transformation and the type checking, the type context management and other core facilities such as the exception and sequence libraries;
- *illustrator*, consisting of the graph layouter and the output renderers, including the PGF/T*ik*Z file generator.

A description of each of the individual files is listed below.

***compiler*** —

`DebugInfo.java` manages the symbol table, linking commands to their corresponding source code. It also records observation points in a program.

`rCOS.g` is the grammar file, parsing source texts to ASTs.

`rCOSWalker.g` is the tree grammar file, paring ASTs to expressions, commands and class graphs.

`Runner.java` is the driver and user front end. It accepts user commands, loads and runs user programs, handles exceptions and reports error messages. It contains the program entry point.

`VarScope.java` Variable scope management, used in parsing to determine when to add *self* to attributes.

***graph*** —

`Cascade.java` is a generic class for curried mappings of type $P \to (S \to V)$, where the outer mapping is a `HashMap` and the inner mapping is a `TreeMap` (a simpler user class `ArrayMap` is ultimately used instead for efficiency, since we don't have many keys, corresponding to attributes and local variables, in a mapping and there are too many object creations in tree operations.)

`CLL.java` is a circular doubly linked list implementation with generic elements. It is used for sequences, stacks and simple containers throughout the program.

`Command.java` contains the classes for commands, including the abstract base and the concrete rCOS commands.

`Context.java` stores the type context, including a class graph $\Gamma$ as the base and a dynamically changing context $\Delta$ as the increment. It is an instantiation of the increment graph pattern. It provides methods to create and update the context during parsing and type checking.

`Edge.java` defines the structure for storing edge tuples.

`Ex.java` contains the customized exception classes, including normal exceptions that are raised when user programs go wrong, and internal exceptions that are raised when the implementation itself goes wrong.

`Expr.java` contains the classes for expressions, including the abstract base and the concrete rCOS expressions, of various forms.

`Graph.java` implements the increment graph pattern, including the abstract graph, the base graph and the composition graph. A graph is implemented as a `Cascade` accompanied with a reverse mapping from contents to references for value nodes.

`GraphExp.java` defines the graph exporter interface, for the other parts to retrieve the structure of a graph.

`Label.java` contains the classes for edge labels, including the abstract base and the concrete labels, such as attribute names, variable names, the scope link and the frame link.

`Lookup.java` defines the interface for a subset of mapping operations, including only the member test for keys and the iterating of entry pairs in a mapping. It also contains the classes for empty lookups and the composition lookups.

`Method.java` defines the structure for storing parameter lists and bodies of methods.

`Name.java` contains the classes for type names (classes and primitive types) and method names, including an abstract base class.

`Node.java` defines the interface for nodes, with only one method to test if a node is also a value.

`Path.java` defines the structure for storing pointed paths, each consisting of a node as the search origin and a sequence of labels.

`Seq.java` is a repository of common higher order iterables (sequences), such as concatenating, filtering, mapping and zipping. Also some related classes for products and delegates, such as pairs, triples and predicates are defined here.

`State.java` stores the state graph, including a store as the base and a transformer as the increment. It is an instantiation of the increment graph pattern. It provides methods to perform state graph operations during running of a program.

`Value.java` contains the classes for constant values, including the abstract base and the concrete values, such as integers, booleans, strings and the null reference.

`Variant.java` is the generic base class of variant classes, such as names, labels and values. A variant has a tag to identify its kind, and two variants of different kinds can be compared by their tags.

***illustrator*** —

`GraphData.java` stores nodes and edges of a graph explicitly for further processing. It also maintains node identities as integers across different graphs.

`GraphDump.java` defines the interface to invoke graph output.

`KKLayout.java` implements the Kamada-Kawaii graph layout algorithm.

`TextAdjListDump.java` writes out graph relations in plain text, mainly for debugging.

`TikzGraphDump.java` writes out graph layouts in PGF/TikZ scripts incorporated in a standalone LaTeX file.

## 6.3 Increment graph pattern

As discussed in the above sections, the graph transformations in type checking and program execution are of correspondence, particularly in the scope stacks. In type checking, a stack can be reverted to its previous states in place, by popping newly pushed nodes, all the nodes and edges in the previous states remain unchanged. In program execution, changes not only occur in the newly pushed scope, but also in other parts of the graph. Even in the former case, popping nodes requires considerable operations if changes are blended into originals, especially deletions, which are costly in our curried mapping design.

To deal with the problems, we implement an increment graph pattern, used in both context (class) graphs and state graphs. An increment graph pattern is a composition of a pair of graphs $(i, b)$, where $i$ is the increment graph, and $b$ is the base graph. The composition itself inherits the `Graph` abstract class, thus is also a graph. The composition preserves the tail, the base graph $b$, by adding all the subsequent changes only in the increment graph $i$. Thus, we are able to extract the previous graphs very easily. The composition graph can be recursively constructed, forming a list of graphs in the form of $(i_n, (\ldots (i_2, (i_1, b)) \ldots))$, where any one tail portion of the list is a consistent graph, and any one increment graph contains only the changes, which must be incorporated with its base graph to be interpreted correctly. There are cases that the increment graph and the base graph contain duplicated graph elements, where the elements in the increment graph override those in the base graph when the composition is under consideration.

Since we are able to eliminate deletions with the increment graph pattern, the composition of an increment graph and a base graph is relatively simple. Here are the fundamental graph operations to consider:

- `addNode` and `addEdge`. We delegates the adding of new nodes and new edges to the increment graph.
- `containsNode` and `containsEdge`. We first look in the increment graph for the existence of the specified node or edge, if not found, we then look in the base graph. An edge is identified by a source node and an outgoing label.
- `getTg`. We return the target node from an edge which is in the form of a source node and an outgoing edge label, if the edge is found in the increment graph, we return its target node, otherwise, we pass the call to the base graph.
- `getValue`. We return the node represent the specified constant value in the graph by first looking in the increment graph, then the base graph, similar to `getTg`.

- `ensureValue`. Just as `getValue`, except that if the constant value is not found in the graph, we make a new node to represent the value in the increment graph.
- `getOutEdgesLookup`. We return a lookup table of the outgoing edges from the specified node. We need to composite the two lookup tables obtained from the increment graph and the base graph, with the one from the increment graph overriding the one from the base graph. There is a class `Lookup.List` to implement this composition, where the iterating of table entries is performed by concatenating the entry sequence of the head table and the filtered entry sequence of the tail table. The filtering is done by a predicate which drops those entries occurred in the head table.

Other graph operations, which handle general graphs uniformly, can be based on the above fundamental operations.

## 6.4 Class structure

Fig. 14 shows the relations of the classes relevant to the language elements, such as expressions and commands. A class name *.Self* with a dot prefix is a shorthand for its full name *Expr.Self* qualified by its ultimate super class in the diagram. The classes are divided as fine as possible to allow the Java type system to enforce their proper usages. We can see that graph elements, `Node`, only occur in auxiliary commands, *enter* and *leave*. The language can be described by these classes independent to the graph model.

Here, we give the description of some of the classes and features that are otherwise not discussed elsewhere:

- `Graph<V extends Node & Comparable<V>>`. This is a generic graph class that accepts nodes, of type `V`, with external identity, if the nodes implements `Comparable`. This allows some nodes, called value nodes, to be identified by content equality rather than reference equality. The graph uses a reverse lookup table to locate the value nodes by their contents. In state graphs, constant values are the value nodes, while in class graphs, type names are the value nodes.
- `Graph.Base`. This is the concrete graph that implements the core graph update and query operations. Updates are performed in place so that they can be carried out efficiently. Combined with the increment graph pattern, it can be used to implement immutable graphs while allowing frequent graph updates.
- `Command`. This is the abstract base of all commands requiring methods to check the local well-typedness and perform the small step execution. The small step execution executes the first atomic sub-command, and returns a configuration consists of the rest of the command and the result state graph. The class also implements the well-typedness checking based on the local well-typedness checking.

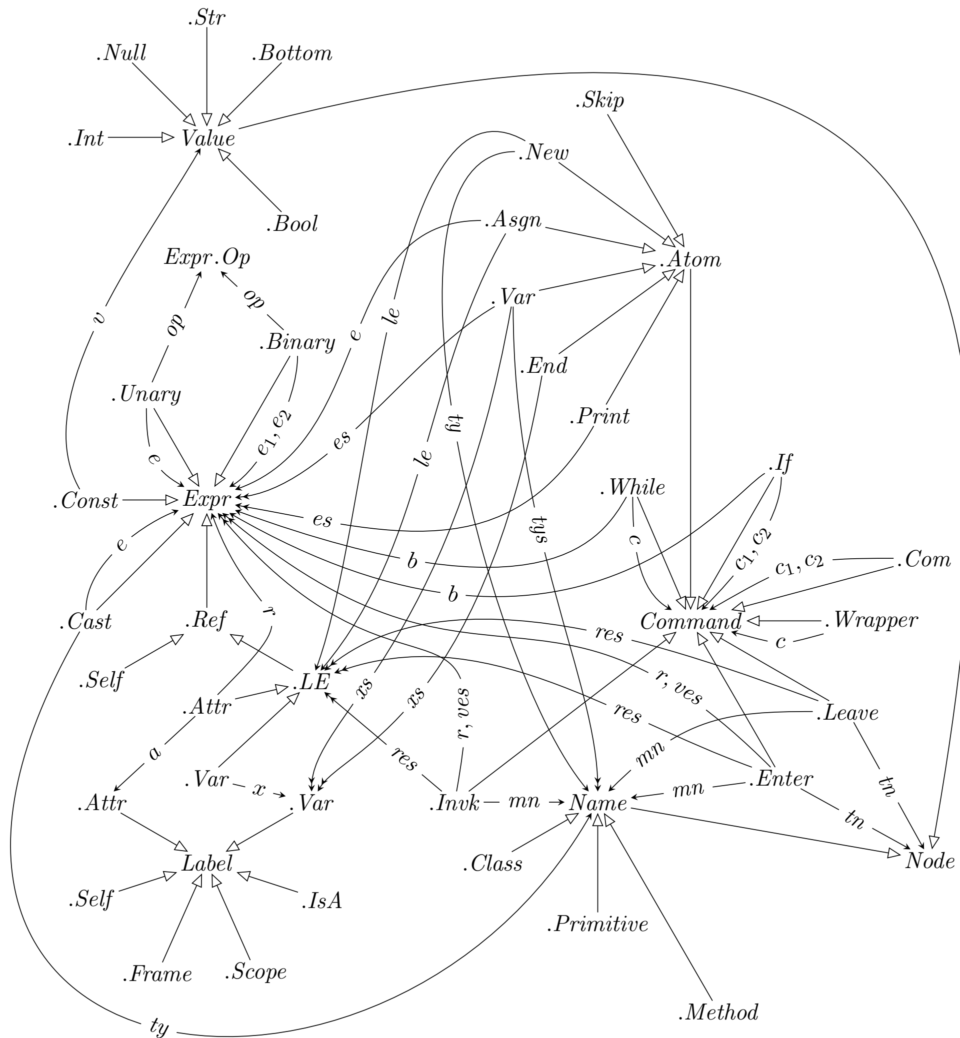



Figure 14: Classes and relations for language elements

- `Command.Wrapper`. Every command is designed to have an optional inspector (or hook), which is called before and after the execution of the command. The output of graph drawings and the user break mechanism are based on these hooks. Following the small step semantics, a compound command is split into atomic commands in the form of a head and a tail, where the head is an atomic command to execute immediately, and the tail is a smaller, possibly compound, command. We don't have the after-point of a compound command, since its tail, once returned from the initial splitting, is merged into the tail of the whole program. The wrapper command wraps a compound command to mark out its literal boundary, when executed, it inserts an empty command to the end of the compound command, to invoke the inspector at the after-point.

- `Command.Var`. This command is extended to allow multiple variable declarations. The variables being declared can all be referenced throughout the declaration, and variables with the same names in outer scopes are hidden. The order of the variables being declared

is insignificant, however, if the initializer of a variable refers to, i.e. depends on, another variable being declared, there must exists a topological order of dependency among the variables. Upon the creation of a `Command.Var` instance, a topological sort is performed to reorder the declaration sequence. If such a sorting fails, an error of circular dependency is reported.

This behavior is slightly different from the type checking rules and the semantics rules, where var $T\,x = x$ is allowed if there is an $x$ in some outer scope. The implementation rejects this as a circular dependency. Accepting such a command makes the language more context sensitive and literally confusing, that we'd better keep it only in the theory.

In a multiple variable declaration, when properly reordered, variable initializers are type checked under the scope where all the being declared variables have been pushed. Upon the execution of such a command, variables are added to the state graph iteratively in order, so that the initializers can be evaluated iteratively without dependency problems.

- `Expr` and `Expr.Ref`. `Expr` is the abstract base of all expressions, requiring methods to check the well-typedness and evaluate the expression. The evaluation results a node, or a constant value, which is also a node. If the expression is a reference, it should be of abstract class `Expr.Ref`, which requires a method to halfway evaluate it to a pointed path, given the starting search point. A reference expression can be fully evaluated based on the pointed path, by finding the target node of the path.

## 6.5 Language and parsing

***Language***. The language defined in the implementation is a direct dialect of that defined in Section 2, with some complementary details. The visibility, public, protected or private, of classes and class members is not implemented in the current implementation for simplicity. The full syntax is shown in Appendix A.

To practically use the language in programming, we predefine some operations and a `print` command. The operations include basic arithmetic and logic operations, random number generation `rnd`, string operation `length` to return the length of a string, string conversion `string` to convert a value to its string representation, and substring operations `take` and `drop` to respectively return the front and rear parts of a string.

***Parsing***. A source program is first parsed into an abstract syntax tree (AST), then the expressions, the commands and the class graph corresponding to the program are created by walking through the AST. The processing is directly specified in the ANTLR grammars.

A simple scope context is maintained during the tree walking, to lookup whether a variable is defined in the current scope. This is to determine the type of a label, being an attribute of *self* or a variable. If a label is defined in the current scope, a variable is created, otherwise, an attribute with *self* as its parent is created. The scope context is more precise in the type checking phase, but we decide not to change the type of a program element beyond parsing,

therefore, this completion of attributes with *self* is moved to the parsing phase.

### 6.6 Type checking and graph transforming

A context graph, implemented in the `Context` class, consists of a base graph $\Gamma$ and a stack of increment graphs $\Delta$. The class provides methods to progressively build up the base graph during parsing. After parsing, the base graph $\Gamma$ no long changes, and all the scope creations and destroys are done in the increment graphs $\Delta$, efficiently by using the increment graph pattern.

By the increment graph pattern, a pushing of a new scope cons an increment graph to the head of the context stack, forming a new composition graph, we then add new variables to the scope by updating the composition graph, that is, the increment graph component of the composition. When a popping is needed, we simply return the base graph component of the composition as the result graph.

The `Context` class also provides methods to carry out the type checking of commands, methods and class graphs. It checks for the inheritance loop in the class graph, builds necessary scopes and delegates the type checking tasks to the corresponding methods of commands and expressions.

A transforming state graph, implemented in the `State.Transformer` class, consists of a base state graph `State.Sto` combined with an increment graph $i$ to form a composition graph. The `Transformer` class provides methods to perform the graph operations, such as *swing* and *add*. All updates to the transforming state graph take place only in $i$, keeping the base state graph immutable. The graph operations are initiated by the commands of a program, starting from the global variable declaration followed by the main command. When a small step of execution completes, a depth first search is performed on the transforming state graph, copying all the reachable elements to a new `Sto`, reflecting the grand effect of the operations accumulated in the transforming.

When the main Java class of the implementation, `Runner`, starts the type checking and the graph transforming process, the class graph $\Gamma$, the associated methods, the main command and the global variable declaration are constant thereafter. The current running command are pushed onto a call stack in the before-point inspector, and popped out in the after-point inspector. This approach maintains the nested invocation structure under the small step execution by the help of the `Command.Wrapper` class. The `Runner` catches exceptions, fetches the topmost command on the call stack, and looks up the symbol table for the command to report runtime errors.

## 7 Application

A main motivation of the semantics is that we wish to help in reasoning about OO programs. For this it is crucial that properties can be clearly, easily and precisely thought about, described and

understood. The advantage of our model in this aspect comes from intuitiveness and theoretical maturity of graphs. We show here some examples.

***Graph assertions***. As shown in [12], many important properties of OO programs can easily be interpreted as assertions of state graphs. Simple but useful assertions include *acyclic nodes*, *acyclic graphs*, *sink* (or *leaf*) *nodes*, and *reachability* (*credibility*) of one node from another.

There are more subtle and interesting properties, such as *dominance* of one node by another. Node $n_1$ *dominates* node $n_2$, denoted by $n_1$ $\mathsf{dominates}$ $n_2$, if any trace to $n_2$ passes through $n_1$. It holds for $G$ if $n_2 \notin (G \setminus \{n_1\})^{\bullet}.node$, where $G \setminus \{n_1\}$ returns the graph after removing the node $n_1$ and all its adjacent edges from $G$. Similarly, an edge $d$ is the *bridge* for node $n$, denoted by $d$ $\mathsf{bridges}$ $n$, if any trace to $n$ goes through $d$. It holds for $G$ if $n \notin (G \setminus \{d\})^{\bullet}.node$, where $G \setminus \{d\}$ returns the graph after removing the edge $d$ from $G$.

***Separation logic of graphs***. For a connected state graph $G$, let $G.store$ be the subgraph, called the *store* of $G$, that contains the nodes on the \$-path of $G$ and their outgoing edges. The subgraph obtained from $G$ by removing the edges of the store (and the nodes that become isolated because of the removal of these edges) is called the *heap* of $G$, denoted by $G.heap$. Note that $G = G.heap \cup G.store$.

The separation logic [21, 19] can be interpreted in our model. A state $G$ is a *separating composition* of two graphs $G_1$ and $G_2$, denoted by $G = G_1 * G_2$, if $G = G_1 \cup G_2$, $G_1.store = G_2.store$ and $G_1.heap.edge \cap G_2.heap.edge = \emptyset$. The separating conjunction $p * q$, asserting that the heap graph can be split into two object graphs for which $p$ and $q$ hold respectively, is defined as

$$[\![p * q]\!]\, G \mathrel{\widehat{=}} \exists G_1, G_2 \bullet G = G_1 * G_2 \wedge [\![p]\!]\, G_1 \wedge [\![q]\!]\, G_2$$

For example, assume that $q$ is an invariant of a class $C$. To be sure that a method (that possibly overrides a method of $C$) of an object of a subclass $D$ of $C$ preserves this invariant, the assertion $\{q * true\} mbody(D, m) \{q * true\}$ is checked. Notice that $q$ only mentions fields of $C$, and the separation is to divide the state of the object into the attributes inherited from $C$ and those newly declared in $D$.

Chen and Sanders [7] propose a pointer logic based on a mixed model of graphs and functions, that extends separation logic with more flexible relational compositions. Our graphs are simpler, but can also define those compositional relations such as the relation $G_1 \mathsf{access}\, G_2$ asserts that there is a node (object) of $G_1$ can access a node of $G_2$.

Hoare and O'Hearn [13] propose a unification of the ideas of separation in Concurrent Separation Logic [5]. We can also write properties by the idea of trace separation because in our model traces and nodes are unified.

***Object aliasing and confinement***. In an OO program, an object is referred by a navigation expression that is evaluated as a trace in our model. Two expressions are *aliasing*, denoted by $e_1 \diamond e_2$, if their traces target at the same object. This is obviously an equivalence relation, and thus aliasing expressions share many properties. For example, they can reach the same objects,

and they reach any of these objects through the same paths. Formally, let $p$ be a sequence of attribute names, $e_1\langle p\rangle e_2$ means that the object referred to by $e_2$ can be reached from the object referred to by $e_1$ via $p$. We have $e \diamond e_1 \wedge e_1\langle p\rangle e_2 \Rightarrow e\langle p\rangle e_2$, and $e_1\langle p\rangle e_2 \wedge e_2 \diamond e \Rightarrow e_1\langle p\rangle e$. Notice that aliasing is also a cause of cycles in the state.

There exist language mechanisms for managing aliasing and encapsulation of heap-allocated objects. Ownership [9, 8] is one of them, and it provides a notion of object-level encapsulation. Each object has an owner, and it can only be accessed through its owner, i.e. it is dominated by its owner:

$$e_1 \ \mathsf{owns}\ e_2 \mathrel{\widehat{=}} \mathit{trace}(e_1)\ \ \mathsf{dominates}\ \ \mathit{trace}(e_2)$$

Another approach is to ensure unique or aliasing free references [18, 3]. A variable or field annotated by the keyword **unique** is the only name to refer the object. We define $\mathsf{uniq}\ e$ to denote that $e$ is the unique trace to its target object.

$$\mathsf{uniq}\ e \mathrel{\widehat{=}} e = \mathit{null} \vee \forall d \in \mathit{trace}(e) \bullet d\ \ \mathsf{bridges}\ \ \mathit{trace}(e)$$

# 8 Conclusions

It is hard to conclude what cannot be done in an existing semantics can be done by the semantics presented here. Different semantic models provide different ways of thinking about the programs they define. Compared to other semantic models in the large body of literature, the simple graph model provides, in our opinion, more clarity on the OO features, including inheritance, type casting, dynamic binding, aliasing, local variable declaration, and early binding of result parameters in method invocations. Furthermore, the discussion on possible applications in Section 7 shows that this model is simple enough and helps in formulating clear assertions about executions of programs. It is also rich enough for defining more sophisticated language mechanisms, such as ownership and confinement, and a powerful logic to describe and prove important properties of programs.

***Related work***. There is a big number of traditional semantic theories of OO programs (e.g. [16, 1, 11, 6]), operational or denotational, that use the basic theory of sets, functions and relations in defining the states of a program. As pointed out in [15], this approach "often needs to include in the syntax definition of runtime concepts", such as locations to indicate a value that may possibly change over time. This and the lack of clarity about the structural properties of the states of OO programs are the main source of the complexity of these traditional theories, and possibly the main reason why there is still a fairly well understood and working logic analog to the Hoare-logic or Dijkstra calculus of predicate transformers for OO programming. The complexity also causes difficulty when we formulate properties of executions and it is also why it is hard to define a fully abstract denotational semantics for OO programming. We believe that a trace model similar to that of Hoare and He [12] can be defined for our language and proven fully abstract w.r.t. to the operational semantics given in this paper.

Another advantage of our approach, and graph-based approach in general [15], is that it allows

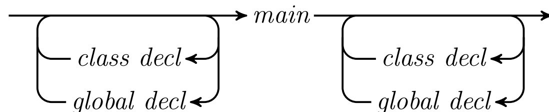
us to use the single mathematical structure – a graph, for the static class structure – the class graph, runtime state – the state graph, and the flow of control – the transition graph, of the program.

There is a large community working on graph-based approaches to software design, known as the area of graph transformations [22]. Much of the work in this area is about models of software architectures [2]. There is however some work on defining programming languages, including execution semantics, by graph transformation systems (e.g. [15, 10]). However, it heavily uses the Rich Abstract Syntax Graph (R-ASG) to gain the power of unification of context information and formal syntactical transformations from programs to graphs. It left the formal semantics of R-ASG undefined. Therefore, while simulation of a program is nicely supported by a tool, it is not clear how assertions of executions can be formulated and reasoned about. Our work is mainly inspired ideas of Hoare and He [12] with the aim to support clear and expressive description and logical verification of properties of executions of programs.

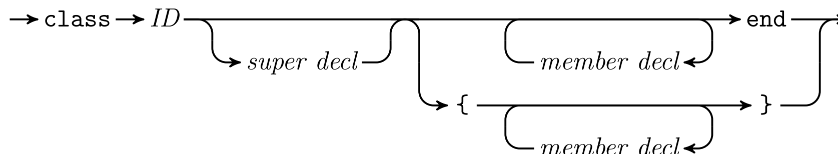
***Future work***. This is the start of a research endeavor. We plan to develop a graph assertional logic for static analysis of OO programs, and then investigate its application in automated techniques of verification and analysis. We will define a fully abstract semantics for rCOS, that is now used for component-based model driven development [17]. Further, we plan to extend the work to define an operational semantics of multi-thread programs for their verification and analysis.

# A Syntax Diagram of the Implemented rCOS

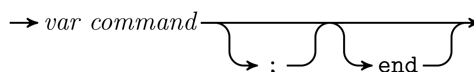
*program* :




*class decl* :

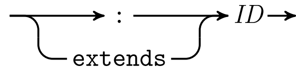



*global decl* :

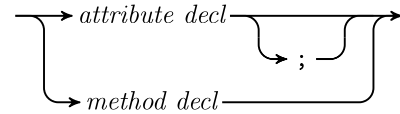


*main* :

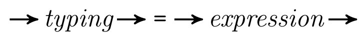
*super decl* :



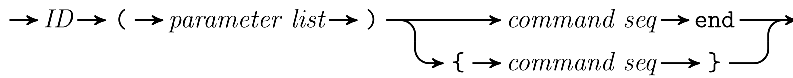

*member decl* :



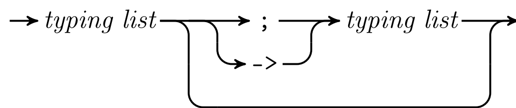

*attribute decl* :



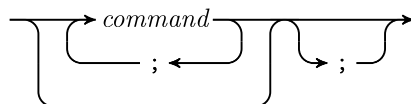

*method decl* :



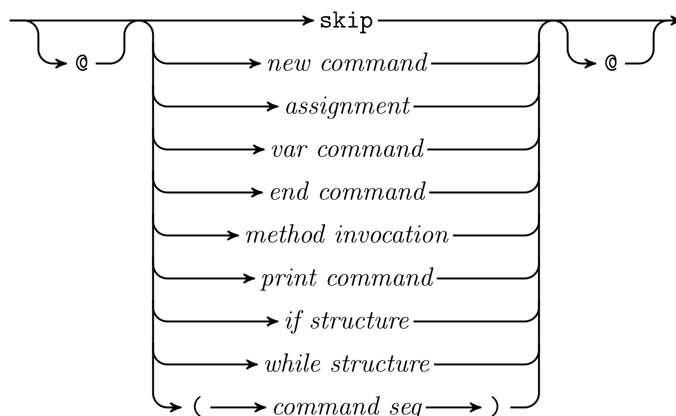

*parameter list* :



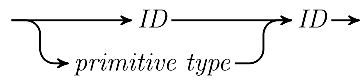

*command seq* :



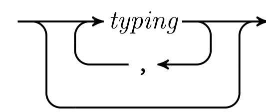

*command* :



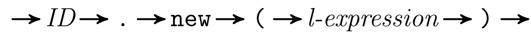
*typing* :



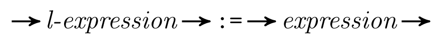

*typing list* :



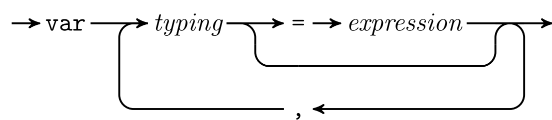

*new command* :



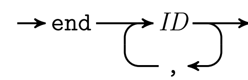

*assignment* :



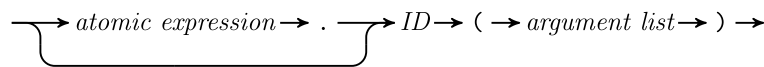

*var command* :



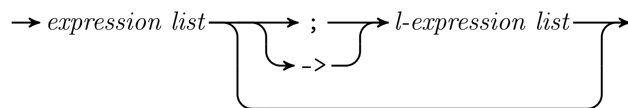

*end command* :



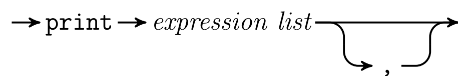

*method invocation* :




*argument list* :

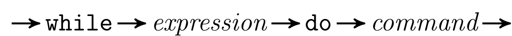



*print command* :

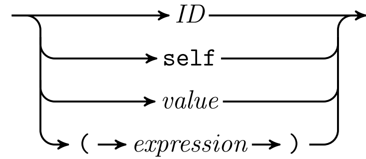



*if structure* :

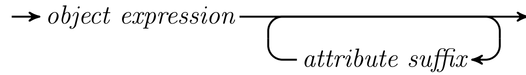
*while structure* :



*object expression* :

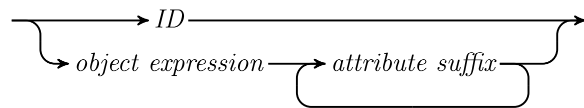



*atomic expression* :

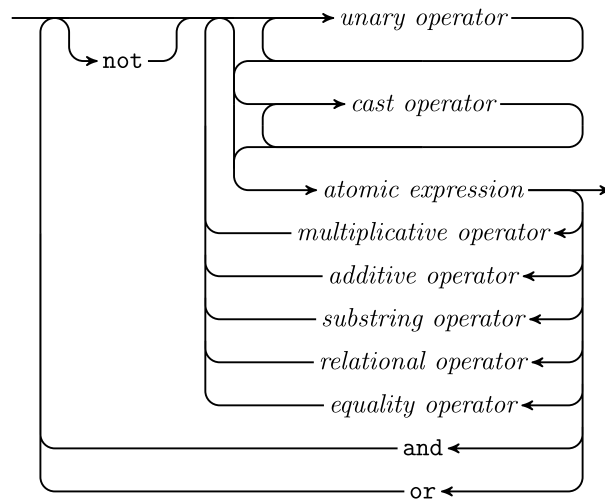



*attribute suffix* :

*l-expression* :

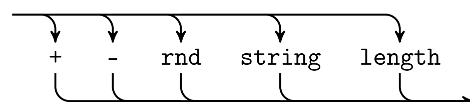



*expression* :

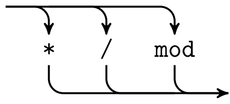



*cast operator* :

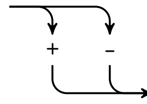
*unary operator* :



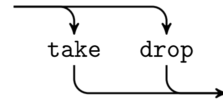
*multiplicative operator* :



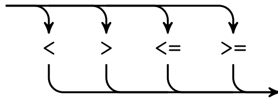

*additive operator* :



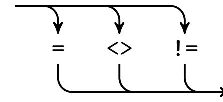

*substring operator* :



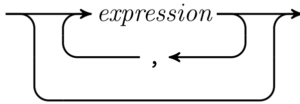

*relational operator* :



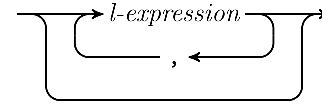

*equality operator* :



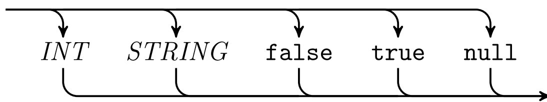

*expression list* :



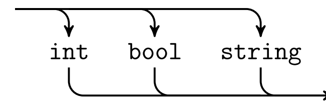

*l-expression list* :



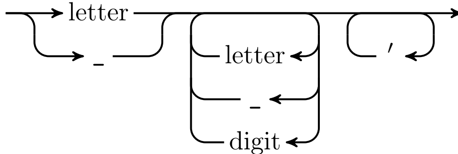

*value* :



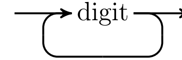

*primitive type* :



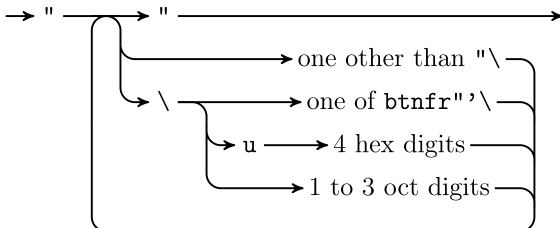

*ID* :




*INT* :




*STRING* :




# References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Fourth Working IEEE/IFIP Conference on Software Architecture*, pages 155–164, Washington, DC, USA, 2004. IEEE Computer Society.

[3] J. Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, 2001.

[4] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *Static Analysis, LNCS 3148*, pages 344–360. Springer, 2004.

[5] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[6] A. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *World Congress on Formal Methods (2), LNCS 1709*, pages 1439–1460. Springer, 1999.

[7] Y. Chen and J. W. Sanders. Compositional reasoning for pointer structures. In *Mathematics of Program Construction, LNCS 4014*, pages 115–139. Springer, 2006.

[8] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming, LNCS 2072*, pages 53–76. Springer, 2001.

[9] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.

[10] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In *Graph Transformations, LNCS 3256*, pages 383–398. Springer, 2004.

[11] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006.

[12] C. A. R. Hoare and J. He. A trace model for pointers and objects. In *13th European Conference on Object-Oriented Programming, LNCS 1628*, pages 1–17. Springer, 1999.

[13] T. Hoare and P. O'Hearn. Separation logic semantics for communicating processes. *Electron. Notes Theor. Comput. Sci.*, 212:3–25, 2008.

[14] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[15] H. Kastenberg, A. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In *Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*. Springer, 2006.

[16] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM TOPLAS*, 28(4):619 – 695, 2006.

[17] Z. Liu, C. Morisset, and V. Stolz. rCOS: Theory and tool for component-based model driven development. Technical Report 406, UNU-IIST, P.O. Box 3058, Macau, 2009. `http://www.iist.unu.edu/www/docs/techreports/reports/report406.pdf`, to appear in LNCS.

[18] N. Minsky. Towards alias-free pointers. In *European Conference on Object-Oriented Programming*, 1996.

[19] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 40:1 of *ACM SIGPLAN Notices*, pages 247–258. ACM, New York, NY, USA, 2005.

[20] rCOS Tool. `http://rcos.iist.unu.edu/`.

[21] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002. Invited paper.

[22] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[23] L. Zhao, X. Liu, Z. Liu, and Z. Qiu. Graph transformations for object-oriented refinement. *Form. Asp. Comput.*, 21(1-2):103–131, 2009.