Front. Comput. Sci., 2013, 7(1): 109–134
DOI 10.1007/s11704-012-1307-8

RESEARCH ARTICLE

A graph-based generic type system for object-oriented programs

Wei KE[1], **Zhiming LIU**[2], **Shuling WANG**[3], **Liang ZHAO** (✉)[2]

1 Macao Polytechnic Institute, Macau, China
2 United Nations University – International Institute for Software Technology, Macau, China
3 The Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

Abstract We present a graph-based model of a generic type system for an OO language. The type system supports the features of recursive types, generics and interfaces, which are commonly found in modern OO languages such as Java. In the classical graph theory, we define type graphs, instantiation graphs and conjunction graphs that naturally illustrate the relations among types, generics and interfaces within complex OO programs. The model employs a combination of nominal and anonymous nodes to represent respectively types that are identified by names and structures, and defines graph-based relations and operations on types including equivalence, subtyping, conjunction and instantiation. Algorithms based on the graph structures are designed for the implementation of the type system. We believe that this type system is important for the development of a graph-based logical foundation of a formal method for verification of and reasoning about OO programs.

Keywords OO programs, type systems, generics, type graphs, recursive types

1 Introduction

Graph notations are widely used to express class structures and execution states of OO programs. For OO languages, such as Java [1], where object variables are references (pointers), it is natural to represent object identities by nodes and variables by directed edges, e.g., in UML [2] and the trace model of Hoare and He [3]. However, traditional semantic definitions and reasoning about properties of OO programs are based on the plain theory of sets and relations [4,5], without much utilization of the properties of graph structures. To close the gap between the structural representation and the reasoning, we define a graph-based type system and operational semantics for OO programs. A significant advantage of this type system is the intuitiveness to study properties and relations of complex types and executions of OO programs using concepts and properties of graphs, which, in particular those of paths, connectivity and homomorphism, articulate the formulation of properties of a program to verify, the idea of reasoning about the properties, and the design of verification and implementation [6]. Notice that we emphasize the simplicity of mathematical structure and way of thinking, instead of the actual visual representation of a program.

In software engineering processes, OO methods are popular, often with the help of formulation and analysis tools, where graph notations are commonly adopted. For such a tool to automate graph transformations, a theory that supports sophisticated OO features is required. When designing this OO type system, we always keep in mind to provide the support to the development of software engineering tools that can be practically used.

1.1 Overview of contribution

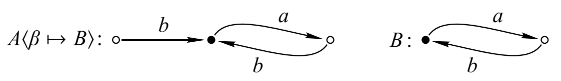
Graph-based representation of type systems with no type operations is rather straightforward. Types are represented as nodes of a *type graph*, and relations between types, including the inheritance relation, are represented as directed edges.

Received December 19, 2011; accepted September 29, 2012

E-mail: liang@iist.unu.edu

However, modern OO languages often incorporate generics, i.e., parameterized types, to strengthen the type-checking with dynamic typing. Type systems for these languages often use identified and constrained type variables in place of a much widened universal superclass to avoid unchecked downcasts [7]. For a graph-based representation of such type systems, type operations that construct new types should be realized by graph transformations that introduce new components into a type graph. How these new components are defined, and how they coexist and get related with the rest of the type graph are the main problem of the representation.

Recursive generic classes lead to further complication of the problem. That is, a generic class can be instantiated to a recursive type only for certain type arguments, and such a recursive type is actually an unfolded version of some other type. Consider an example generic class $A\langle\beta\rangle$, where β is a type variable, and one of its instantiations $A\langle\beta \mapsto B\rangle$. Let b of type β be an attribute of $A\langle\beta\rangle$ and a of type $A\langle\beta \mapsto B\rangle$ an attribute of class B, the instantiation of the recursive generic type is shown below:



In the figure, the graph on the left is an instantiation graph of A, by substituting the graph of B (on the right) for β. The white nodes all represent the type $A\langle\beta \mapsto B\rangle$, but the graphs connected to these white nodes are different. We need to relate these graphs using the equi-recursive approach [8, 9] to ensure consistency. How this approach can be formulated in terms of graphs puts another challenge to the representation.

We extend the *r*COS[1)] notation for specification and design of OO programs [4, 10] to demonstrate our type system. We focus on the representation of class and object structures which play the key role in OO programming, while minimizing the complexity of other language constructs to keep the graph notation simple. We choose *r*COS because it is designed for reasoning the structural refinement of OO programs [11]. Also, we have previously defined and implemented a graph-based operational semantics [6] with an unpublished simple type system [12]. This paper can be regarded as a revised version of the previous work [6] with an extensive type system.

We define type graphs to represent the type definitions of programs, where subgraphs with roots represent the structures of types. We treat user-defined classes as *nominal types* whose graphs are rooted at nominal nodes, and types obtained through type operations as *structural types* whose graphs are rooted at anonymous structural nodes. We define two type operations, *class instantiation* and *class conjunction*, to implement the generics with memberwise constraints. We also study an equivalence relation between types of different structures. In this way, a type system with generic types, conjunction types and subtyping is developed based on a graph notation. For the type system, we prove that

- the subtype relation is a partial order, i.e., compatible with other OO type systems,
- the class instantiation preserves type-safety, i.e., type-checking can be done prior to instantiations, and
- a type-safe program will be executed correctly, i.e., the soundness of the type system.

1.2 Related work

There is a large body of non-graph-based approaches to formalizing OO programs. An excellent piece of work emphasizing on types of general programming languages is given in [8], and a piece of classic work on types and semantics of OO programs is given in [13]. They both cover parametric polymorphism, subtyping and recursive types by using calculi of terms. They provide a solid foundation for reasoning about OO programs. Besides these works, a functional core calculus, FJ, for Java with generics is given in [14], with a type-safety proof of the key features of Java, including inheritance and dynamic typecasts. A stronger type system based on FJ is given in [15], focusing on type-safe downcasts. And a comprehensive type system of a Java-like language, Jinja, is given in [5], providing a unified model of languages, virtual machines and compilers. As one of the key issues that our graph-based type system is to address, there are quite some publications on generics with subtyping [16, 17]. While our system focuses on how to formulate the generics-related type operations in a graph notation and sticks with a simpler invariant "shallow" subtyping, the variance-based "deep" subtyping is thoroughly explored in those works. Shallow subtyping considers only the inclusion of the first level components of the superclass, while deep subtyping also considers the subtyping of the component types.

There are also graph-based approaches using graph transformation rules to formalize OO program semantics. Our graph-based operational semantic model in [6] is influenced by the trace model defined in [3], which intends for specification of state properties by location insensitive traces of object

1) *r*COS stands for "refinement of component and object systems".

pointers. In [18], classes and objects are defined as *type* and *instance hypergraphs*. The work presents an OO semantics in terms of graph transformation rules based on category theory. It does not give a formal type system, though. The work of using graph transformation systems to formalize OO programs, including execution states and program commands, appears in [19, 20]. The former is based on hypergraphs, while the latter is based on simple graphs. They both use graph transformation as a model of automation. For more applications of graph transformation systems, see also [21–23]. The common feature of these methods is that they focus on program execution semantics instead of type systems. The reason is mainly that representing advanced features of type systems by graphs, such as generic types and polymorphism, is challenging since it involves hierarchical structures. The use of graph notations is not limited to OO systems. In a type system that merges ML-style type inference with System-F polymorphism [24], graphic types are used to obtain a linear-time unification algorithm. In such a graphic type, nodes are labeled by type operations, type variables and type constants (such as bottom), while edges connect operations to operands, and also represent quantifier bindings. Although it is quite different from using nodes as types and edges as attributes in our system, the idea of benefiting from the intuitive relations between graphs is similar.

1.3 Outline of structure

We define the extension to *r*COS, called the *r*COS/g language, in Section 2, and type graphs with their relations in Section 3. In Section 4, we define the operations of class conjunction and class instantiation, and discuss how the graphs of recursive generic classes can be constructed. The type checking system is presented in Section 5, together with the proofs of the subtyping partial order and the type-safety of command instantiation. Section 6 is about environment graphs that encode the runtime type information. We show how environment graphs are constructed from type graphs. In Section 7, we study the connection between the type system and the operational semantics based on environment graphs, proving that the program execution is type-safe.

2 An object oriented language

The formal language of *r*COS is a general OO language which has a Java-like syntax and supports most of the essential OO features such as inheritance, type casting, dynamic binding and recursive objects [4]. For a generic type system, we extend *r*COS by introducing generic classes, class instantiations and class conjunctions. We also modify the syntax of method invocation, adopting the mechanism of "parameter binding by name". With such binding mechanism, the order of parameters of a method is not significant, which reduces the complexity of the graph representation. We call the new language *r*COS/g, and show its full syntax in Table 1. The syntax definition uses the overline notation $\overline{exp(s,t)}$ to represent a list of expressions $exp(s_1,t_1), exp(s_2,t_2), \ldots, exp(s_k,t_k)$. For example, $\overline{Te\ x}$ is a list $Te_1\ x_1, Te_2\ x_2, \ldots, Te_k\ x_k$ of variable declarations.

In this syntax, C ranges over class names, α denotes an arbitrary type variable, x a variable or a method parameter, a an attribute, m a method name, and l a literal of a primitive type. We assume primitive types $\mathbb{Z}$ for integers, $\mathbb{B}$ for booleans and $\mathbb{S}$ for text strings. We also define the auxiliary class $Null$ as the type of the $null$ object.

As in Java, a program consists of a list $\overline{c\text{-}decl}$ of class declarations and a $Main$ method as the entry point. The $Main$ method has a list $\overline{ext\text{-}v}$ of external variable declarations and a method body command c that accesses the external variables. Each type in $\overline{ext\text{-}v}$ is a primitive type or a class declared in $\overline{c\text{-}decl}$. A class declaration is of the form

$$\textbf{class}\ C\ [\langle\overline{\alpha\ cnstr}\rangle]\ [\textbf{ext}.\ Cc]\ \{\overline{a\text{-}def};\overline{m\text{-}def}\},$$

where C is the name of the class, $[\langle\overline{\alpha\ cnstr}\rangle]$ is an optional type parameter list for generic classes, $[\textbf{ext}.\ Cc]$ is an optional superclass specification, and $\{\overline{a\text{-}def};\overline{m\text{-}def}\}$ are definitions of *members*, i.e., attributes and methods, of the class. A type parameter is a type variable α followed by its constraints $cnstr$, involving an optional superclass constraint and an optional member constraint. An attribute definition $a\text{-}def$ consists of a visibility specification vis and an attribute typing. A method definition $m\text{-}def$ is a method signature consisting of a method name and a formal parameter typing list, followed by a command as the method body.

A type is specified by a type expression Te, which is either a primitive type or a *class expression*. Class expressions Ce are formed from classes, type variables, class conjunctions and class instantiations. Among class expressions, classes and class instantiations are called *concrete classes*, denoted as Cc. Only concrete classes can have instances, thus we can only invoke the **new** command on concrete classes. A class conjunction C_n is a collection of member signatures of a set of concrete classes. Class conjunctions are used as class interfaces to describe memberwise requirements and assumptions. A class instantiation C_j represents a new instance of a generic class, in the form of the generic class name followed by a

Table 1 *r*COS/g abstract syntax

$prog ::= \overline{c\text{-}decl} \bullet Main$	Program	$c ::=$	Command
$c\text{-}decl ::= \mathbf{class}\ C$	Class declaration	$\mathbf{skip}$	
$[\langle\overline{\alpha\ cnstr}\rangle]$	Type parameters	$\mid \mathbf{var}\ \overline{Te\ x}$	Variable declaration
$[\mathbf{ext}.\ Cc]$	Direct superclass	$\mid \mathbf{end}\ \overline{x}$	Variable undeclaration
$\{\overline{a\text{-}def};\overline{m\text{-}def}\}$	Members	$\mid Cc.\mathbf{new}(le)$	Object creation
$cnstr ::=$	Constraints	$\mid le := e$	Assignment
$[\mathbf{ext}.\ Cu]$	Superclass constraint	$\mid e.m(\overline{ve} : \overline{x} : \overline{re})$	Method invocation
$[\mathbf{impl}.\ C_n]$	Member constraint	$\mid c;c \mid c \lhd e \rhd c \mid e * c$	Structural commands
$a\text{-}def ::= vis\ Te\ a$	Attribute definition	$ve ::= e$	Value argument
$vis ::= \mathbf{priv}. \mid \mathbf{prot}. \mid \mathbf{pub}.$	Visibility	$re ::= le$	Result argument
$m\text{-}def ::= m(\overline{Te\ x})\ \{c\}$	Method definition	$e ::=$	Expression
$Cu ::= Cc \mid \alpha$	Class upper bound	$le \mid self \mid null \mid l$	
$Te ::=$	Type expression	$\mid (Ce)e$	Type cast
$\mathbb{Z} \mid \mathbb{B} \mid \mathbb{S}$	Primitive types	$\mid f(\overline{e})$	Built-in function application
$\mid Ce$		$le ::=$	l-expression
$Ce ::= Cc \mid C_n \mid \alpha$	Class expressions	x	
$Cn ::= \Cap\overline{Cc}$	Class conjunction	$\mid e.a$	Object attribute
$Cc ::= C \mid C_j$	Concrete classes	$ext\text{-}v ::= Te\ x$	External variable declaration
$C_j ::= C\langle\overline{\alpha \mapsto Te}\rangle$	Class instantiation	$Main ::= \{\overline{ext\text{-}v};c\}$	Main method definition

substitution of type variables with type expressions.

A command c in *r*COS/g is similar to those in common imperative languages. We provide structural constructors of commands including sequential compositions $c;c$, conditionals $c \lhd e \rhd c$ and while-loops $e * c$. We require that commands, including assignments, object creations and method invocations, should not occur in expressions, so that the evaluation of expressions has no side-effects. We explicitly formulate variable declarations and undeclarations as commands for local scope introductions and cleanups, respectively.

A method invocation specifies the method name and the object upon which the method is invoked, as well as how the value arguments $\overline{ve}$ are passed to and how the result arguments $\overline{re}$ are retrieved from the formal parameters $\overline{x}$. Upon entering the method, the value of a value argument is copied to the corresponding formal parameter, and upon leaving the method, the value of a formal parameter is copied to the corresponding result argument. For example, the method invocation

$$o.m(true\,, 4 : x, y : p.a, q.b)$$

invokes method m upon object o, passing $true$ to x, 4 to y when entering, and retrieving $p.a$ from x, $q.b$ from y when leaving.

Expressions are formed from variables, attributes and literals through applications of attribute navigations $e.a$, typecasts $(Ce)e$ and primitive-type functions $f(\overline{e})$. We denote the null object by *Null*, and use the special variable *self* to point to the currently active object. The set of *l-expressions*, ranged over by le, includes variables and attribute *navigation paths*, which are allowed to occur on the left-hand side of an assignment.

To simplify our discussion, all variables, parameters and attributes are initialized with the default values of their types, specifically, 0 for integers, *false* for Booleans, ε (empty) for text strings and *Null* for all class types.

3 Type graphs

The class declarations $\overline{c\text{-}decl}$ of an OO program define the structure of the objects of the program in terms of their types and relations. This structure can be formally represented by a directed and labeled graph [11], called a *type graph*. In this paper, we develop a graph-based type system for *r*COS/g, stretching the model in [11] to capture the extended features of the language.

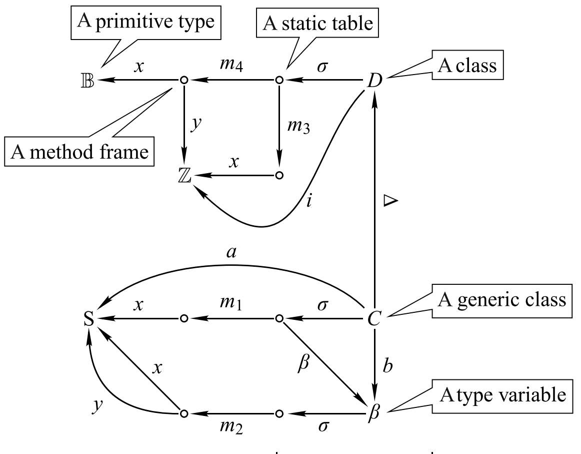
3.1 Types and structures

In an *r*COS/g program, the primitive types, classes and type variables are called *nominal types*, each of which has a name. Nominal types with different names are different, even if they have the same set of members. Class instantiations and conjunctions are types obtained from type operations, and they are called *structural types*. Two structural types are the same if they have the same set of members. Both nominal and structural types are represented in a type graph.

The *static table* of a class lists the methods of the class, and the *frame* of a method consists of the parameters of the method. These structures are also characterized in a type graph. In fact, each type, static table or method frame of a program is represented as a subgraph of the type graph of the program. There is a designated node in such a subgraph, called the *root* of the graph of the type, static table or method frame. For a nominal type, the root of its graph is the name of the type, while for a structural type, a static table or a method frame, the root is *anonymous*. We call the root of a nominal type a *nominal node* and an anonymous root a *structural node*.

We use $\mathcal{D}$ to denote the primitive type (data type) names and $\mathcal{U}$ the class (user type) names, and $\mathcal{C} = \mathcal{D} \cup \mathcal{U}$ the type constants. Let $\mathcal{X}$ be the type variables. Thus, $\mathcal{C} \cup \mathcal{X}$ is the set of nominal nodes. Using $\mathcal{S}$ to denote the structural nodes and assuming all these sets are mutually disjoint, we have the set $\mathcal{N} = \mathcal{C} \cup \mathcal{X} \cup \mathcal{S}$ of nodes of type graphs.

For the labels of edges, let $\mathcal{A}$ be the set of the names of variables, attributes, methods and parameters, which is disjoint with $\mathcal{N}$. We introduce a special label "$\rhd$" for the class inheritance relation, and another special symbol "σ" to label an incoming edge to the root of a static table. If a class contains a type variable α, the class is generic. A generic class can be instantiated by mapping α to some actual type u. We use an edge labeled by the type variable α and targeting u to record the mapping $\alpha \mapsto u$. We put such an edge to the static table of each generic class and class instantiation, see Fig. 1. Thus, $\mathcal{A}^+ = \mathcal{A} \cup \mathcal{X} \cup \{\rhd, \sigma\}$ is the set of all labels of edges.



class $C\langle\beta$ impl. ⋒ $I\rangle$ ext. D	class I	class D
$\mathbb{S}\ a$;	$m_2(\mathbb{S}\ x, \mathbb{S}\ y)$;	$\mathbb{Z}\ i$;
$\beta\ b$;		$m_3(\mathbb{Z}\ x)$;
$m_1(\mathbb{S}\ x)$;		$m_4(\mathbb{B}\ x, \mathbb{Z}\ y)$;

Fig. 1 A type graph and the corresponding class declarations

Definition 1 (Type graph) A type graph is a directed and labeled graph $\Gamma = \langle N, E\rangle$, where $N \subseteq \mathcal{N}$ is the set of nodes, and $E : N \times \mathcal{A}^+ \to N$ is the set of edges.

Notice that E is a function implies that labels of the outgoing edges of a node are distinct. Thus the names of attributes and methods of a class are different, and parameters of a method are different. In particular, there is no multiple inheritances allowed in this model.

Figure 1 shows a type graph and the corresponding class declarations. In the graph, C is a generic class, with a type variable β and a superclass D. Notice the β-labeled edge targeting at the node β. If β is instantiated to an actual type, the β node is replaced by the node of the actual type. C has two attributes a and b of types $\mathbb{S}$ and β, respectively, and a method m_1 with a parameter x of type $\mathbb{S}$. The type variable β implementing class I has a method m_2 with two parameters x and y both of type $\mathbb{S}$, meaning that any replacement of β in an instantiation of C must define such a method. D is a non-generic class with an attribute i and two methods m_3 and m_4, where m_3 has a parameter x and m_4 has two parameters x and y.

We use $s \xrightarrow{a} t$ to represent an a-labeled edge from s to t, and call s the *source node* and t the *target node* of the edge. We use $s \xrightarrow{a} \cdot$ to represent an a-labeled outgoing edge of s and $\cdot \xrightarrow{a} t$ an a-labeled incoming edge to t. For a given graph $\Gamma = \langle N, E\rangle$, we use $\Gamma.ns$ to denote the set N of nodes, and $\Gamma.es$ the set E of edges.

A path p is a sequence of consecutive edges, denoted as $n_0 \xrightarrow{a_1} n_1 \cdots n_{k-1} \xrightarrow{a_k} n_k$ or simply $n_0 \xrightarrow{a_1 a_2 \cdots a_k} n_k$, if we are not concerned with its intermediate nodes. We also overload the notation of set membership "$\in$", using $n \in \Gamma$, $e \in \Gamma$ and $p \in \Gamma$ to denote that a node n, an edge e and a path p are in graph Γ, respectively.

For a path p, let *source* (p) and *target* (p) be the starting and ending node, respectively. We say that a node n_2 is *reachable* from a node n_1, denoted as $n_1 \to^* n_2$, if there is a path from n_1 to n_2. A path $n_1 \xrightarrow{a} \cdots \xrightarrow{a} n_2$ is simply denoted as $n_1 \xrightarrow{a}{}^* n_2$.

3.2 Rooted graphs

In the type graph Γ of a program, an individual type T, say a class, is represented by a node r of Γ, the nodes reachable from r and the edges between these nodes. These nodes and edges form a subgraph of Γ with r being designated as the root. This rooted graph represents the type T.

Definition 2 (Rooted graph) Given a node r of a graph Γ, the rooting operation $\Gamma \odot r$ returns the subgraph $G = \langle N, E, r\rangle$ of Γ such that

$$N = \{n \mid r \to^* n \in \Gamma\},$$

$$E = \{n_1 \xrightarrow{a} n_2 \in \Gamma \mid n_1, n_2 \in N\}.$$

This subgraph is called a rooted graph, where r is the root, denoted as $G.\odot$.

Notice that for each node $r \in \Gamma$, $\Gamma \odot r$ is unique. For a rooted graph $G = \langle N, E, r\rangle$, the graph $\langle N, E\rangle$ is called the *base graph* of G, denoted as $G.base$. A path in G from the root $r \xrightarrow{\overline{a}} n$ is denoted as $\odot \xrightarrow{\overline{a}} n$. We extend the rooting operation to rooted graphs, and define $G \odot n \mathrel{\hat{=}} G.base \odot n$.

The rooting operation performs garbage collection. For example, a removal of a set es of edges from a rooted graph $G = \langle N, E, r\rangle$ is defined as

$$G \setminus es \mathrel{\hat{=}} \langle N, E \setminus es \rangle \odot r. \tag{1}$$

The removal of edges may disconnect a subgraph from the root, causing additional nodes and edges to be removed.

3.3 Type-equivalence

Relations between types can now be studied by comparing the structures of their rooted graphs. Graph *homomorphisms* are the instruments for this purpose. In addition to the conventional graph homomorphism, which preserves outgoing edges of each node, root nodes are also preserved in our case.

Definition 3 (Homomorphism) Given two rooted graphs $G = \langle N, E, r\rangle$ and $G' = \langle N', E', r'\rangle$, a function $f : N \to N'$ is a homomorphism from G to G', denoted as $f : G \to G'$, if

$$s \xrightarrow{a} t \in E \implies f(s) \xrightarrow{a} f(t) \in E', \quad \text{and} \quad f(r) = r'.$$

A node, the name of the node in particular, together with the entire set of its outgoing edges is considered the structure of the node. If this structure of a set of nodes does not change under a homomorphism, we say the structure is preserved by the homomorphism.

Definition 4 (Structural preservation) Let G and G' be two rooted graphs. A homomorphism $f : G \to G'$ is a structural preservation on a set $P \subseteq G.ns$ of nodes, if $\forall n, n' \in P$ such that

$$n \in \mathcal{C} \cup \mathcal{X} \implies f(n) = n, \qquad f(n) = f(n') \implies n = n',$$
$$n \in \mathcal{S} \implies f(n) \in \mathcal{S}, \quad f(n) \xrightarrow{a} \cdot \in G' \implies n \xrightarrow{a} \cdot \in G.$$

Notice that a structural preservation is different from a subgraph isomorphism, in that the latter does not consider those edges going outside the subgraph. Obviously, a structural preservation on the entire node set of a graph is a graph isomorphism preserving nominal nodes.

Definition 5 (Structural equivalence) Two rooted graphs G and G' are structural equivalent, denoted as $G \simeq G'$, if there exists a structural preservation $f : G \to G'$ on $G.ns$.

However, the structural equivalence relation does not handle the equivalence between a recursive structural type, in which an attribute refers back to its owner type, and the unfolded variants of the recursive type, as shown in Fig. 2. For this, we define the type reduction relation, which allows structural nodes with the same set of outgoing edges to be merged, so as to have a graph possibly mapped to its subgraph. That is, we remove the 1-1 property for structural nodes from the conditions of Definition 4.

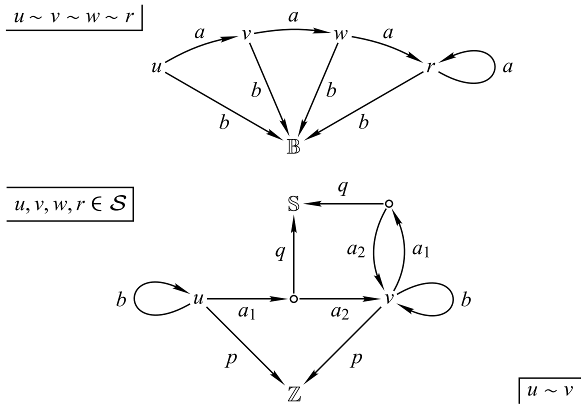


Fig. 2 Equivalent recursive types

Definition 6 (Reduction) A rooted graph G is reducible to a rooted graph G', denoted as $G \rightsquigarrow G'$, if there exists a homomorphism $f : G \to G'$ such that $\forall n \in G$,

$$n \in \mathcal{C} \cup \mathcal{X} \implies f(n) = n,$$
$$n \in \mathcal{S} \implies f(n) \in \mathcal{S}, \quad f(n) \xrightarrow{a} \cdot \in G' \implies n \xrightarrow{a} \cdot \in G.$$

We call such f a reduction function.

Definition 7 (Type-equivalence) Two rooted graphs G and G' are type-equivalent, denoted as $G \sim G'$, if there exists a rooted graph G_r such that both G and G' are reducible to G_r.

It is straightforward to verify that $\sim$ is an equivalence relation. In addition, $G \sim G'$ holds if and only if there exist G_1 and G'_1 such that G is reducible to G_1, G' is reducible to G'_1, and G_1 and G'_1 are structural equivalent, i.e., $G_1 \simeq G'_1$.

In general, we use $\Gamma \vdash n \diamond n'$ to denote that two rooted graphs $\Gamma \odot n$ and $\Gamma \odot n'$ are related by a relation $\diamond$, i.e., $\Gamma \odot n \diamond \Gamma \odot n'$. We also say the two nodes n and n' have relation $\diamond$. In particular, two nodes n and n' in a type graph Γ are called *type-equivalent* if $\Gamma \vdash n \sim n'$.

It is worth pointing out that if two types T and T' are equivalent, attributes of T and T' with the same names also refer to equivalent types. Restricting a reduction function to a subgraph, we have

$$G \rightsquigarrow_f G' \wedge n \in G \implies G \odot n \rightsquigarrow_{f_0} G' \odot f(n), \qquad (2)$$

where $\rightsquigarrow_f$ denotes the reduction defined by f, and $f_0 = f\mid_{(G\odot n).ns}$. Together with Definition 3, we have the following lemma.

Lemma 1 If $G \sim G'$ and $\odot \xrightarrow{\overline{a}} n \in G$, there exists $\odot \xrightarrow{\overline{a}} n' \in G'$ and $G \odot n \sim G' \odot n'$.

Proof By induction on the length of $\overline{a}$. □

On the other hand, for any pair of type-equivalent graphs, we can make an equivalence-preserving extension by linking them to another pair of type-equivalent graphs, respectively.

Lemma 2 If $G_1 \sim G_1'$ and $G_2 \sim G_2'$, also $G_1.ns \cap G_2.ns = \emptyset$ and $G_1 \odot n \sim G_1' \odot n'$, let a be an edge label and

$$\begin{aligned} N &= G_1.ns \cup G_2.ns, & N' &= G_1'.ns \cup G_2'.ns, \\ d &= n \xrightarrow{a} G_2.\odot, & d' &= n' \xrightarrow{a} G_2'.\odot, \\ E &= G_1.es \cup G_2.es \cup \{d\}, & E' &= G_1'.es \cup G_2'.es \cup \{d'\}, \end{aligned}$$

then $G = \langle N, E, G_1.\odot\rangle \sim G' = \langle N', E', G_1'.\odot\rangle$.

We extend the transitive closure of the $\rhd$ relation, as the subclass relation $\unrhd$, to type-equivalent rooted graphs.

Definition 8 (Subclass) A rooted graph G' is a subclass of a rooted graph G, denoted as $G' \unrhd G$, if

$$\exists \odot \xrightarrow{\rhd}{}^{*} n \in G' \bullet G' \odot n \sim G.$$

3.4 Checking type-equivalence

Given two rooted graphs X and Y, we check if X and Y are type-equivalent by using the function $t\text{-}eq(X, Y)$. The function traverses the two graphs and generates a set $Q = \overline{(s,t)}$ of pairs. Each (s,t) of the pairs consists of two sets s and t of nodes of X and Y, respectively. Starting with $(\{X.\odot\}, \{Y.\odot\})$, each next step checks the targets of the outgoing edges, which have identical labels, of nodes in each pair of sets already generated. If the node of X (or Y) being visited was visited before in step i, the corresponding nodes of Y (or X, respectively) being visited are added to the set of nodes of Y (or X, respectively) in the pair generated in step i; otherwise, a new pair of sets containing these nodes respectively is generated. The traverse terminates when there exists a pair (s,t) for that the set of labels of the outgoing edges of s is different from that of t. In this case, the checking function returns *false*. Otherwise, the checking terminates when all nodes are visited and the checking function returns *true* along with the generated set Q of pairs, if and only if, for any pair (s,t) in Q, s and t contain only structural nodes, or are both singletons and equal.

Table 2 shows the definition of the auxiliary function $t\text{-}eq_0$. The definition is given by a list of inference rules, evaluated sequentially. If, for a rule, all the conditions above the line hold, the evaluation reduces to the part of the rule below the line, otherwise the next rule is further evaluated. We define the $t\text{-}eq$ function based on $t\text{-}eq_0$ as

$$t\text{-}eq(X,Y) \mathrel{\hat{=}} t\text{-}eq_0(X, Y, \{(\{X.\odot\}, \{Y.\odot\})\}). \qquad (3)$$

If $t\text{-}eq(X,Y)$ returns (Q, true), X and Y are type-equivalent and we can construct the graph G_r from Q that both X and Y are reducible to. For a pair $(\{C\}, \{C\})$ in Q, where C is nominal, G_r has node C. And for a pair of sets (s,t) of structural nodes in Q, G_r has a corresponding structural node n that represents all the equivalent nodes in the pair. The outgoing edges of n correspond to the outgoing edges of the equivalent nodes in s and t.

Table 2 Type-equivalence

$$t\text{-}eq_0(X,Y,Q) \mathrel{\hat{=}} \quad \frac{\begin{array}{c}(s,t)\in Q \quad x\in s \quad y \in t \\ x \xrightarrow{a} x' \in X \quad x' \notin Q.s\end{array}}{\dfrac{y \xrightarrow{a} y' \in Y \quad y' \in t' \quad (s',t') \in Q}{t\text{-}eq_0(X,Y,Q \setminus \{(s',t')\} \cup \{(s' \cup \{x'\}, t')\})} \quad \dfrac{y \xrightarrow{a} y' \in Y \quad y' \notin Q.t}{t\text{-}eq_0(X,Y,Q \cup \{(\{x'\},\{y'\})\})} \quad \dfrac{}{\mathit{false}}} \qquad \frac{\begin{array}{c}(s,t)\in Q \quad x\in s \quad y \in t \\ y \xrightarrow{a} y' \in Y \quad y' \notin Q.t\end{array}}{t\text{-}eq_0(Y,X,\{(v,u) \mid (u,v) \in Q\})}$$

$$\frac{\begin{array}{c}(s,t),(s_1',t_1'),(s_2',t_2') \in Q \quad (s_1',t_1') \neq (s_2',t_2') \\ x_1,x_2 \in s \quad x_1 \xrightarrow{a} x_1', x_2 \xrightarrow{a} x_2' \in X \quad x_1' \in s_1' \quad x_2' \in s_2'\end{array}}{t\text{-}eq_0(X,Y,Q \setminus \{(s_1',t_1'),(s_2',t_2')\} \cup \{(s_1' \cup s_2', t_1' \cup t_2')\})} \qquad \frac{\begin{array}{c}(s,t),(s_1',t_1'),(s_2',t_2') \in Q \quad (s_1',t_1') \neq (s_2',t_2') \\ y_1,y_2 \in t \quad y_1 \xrightarrow{a} y_1', y_2 \xrightarrow{a} y_2' \in Y \quad y_1' \in t_1' \quad y_2' \in t_2'\end{array}}{t\text{-}eq_0(X,Y,Q \setminus \{(s_1',t_1'),(s_2',t_2')\} \cup \{(s_1' \cup s_2', t_1' \cup t_2')\})}$$

$$\frac{(s,t)\in Q \quad n_1,n_2 \in s \cup t \quad n_1 \neq n_2 \quad \{n_1,n_2\} \not\subseteq \mathcal{S}}{\mathit{false}} \qquad \frac{(s,t)\in Q \quad x \in s \quad y \in t \quad \{a \mid x \xrightarrow{a} \cdot \in X\} \neq \{b \mid y \xrightarrow{b} \cdot \in Y\}}{\mathit{false}} \qquad \frac{}{(Q,\mathit{true})}.$$

Q is a set of pairs and each pair consists of two sets of equivalent nodes, $Q.s = \bigcup_{(s,t)\in Q} s$ and $Q.t = \bigcup_{(s,t)\in Q} t$

4 Type operations

In an *r*COS/g program, types are specified by type expressions. Besides atomic type expressions, such as primitive types, classes and type variables, a type expression can also consists of type operations.

We have two type operations, class conjunctions and class instantiations. Since types are represented by rooted graphs, type operations are defined in terms of graph operations. In this section, we define conjunction graphs and instantiation graphs for class conjunctions and class instantiations, respectively. These graph operations share a common idea — the result graph is constructed by introducing new structural nodes with new outgoing edges pointing into the operand graphs without changing any existing rooted graphs.

4.1 Shallow structures

Unlike in C++ where an attribute of an object is a nested subobject, an attribute of an object in *r*COS/g is a reference to another object. Therefore, we say the attribute structures are *shallow*. For a rooted graph that represents a type, the outgoing edges of the root represent the attribute structure. To compare the attribute structures of two types, we define the *shallow cover* relation between two rooted graphs that the root-originated edges of one graph are "covered" by those of the other.

Definition 9 (Shallow cover) A rooted graph G' is a shallow cover of a rooted graph G, denoted as $G' \hat{\otimes} G$, if for each outgoing edge of the root of G, there is an outgoing edge of the root of G' with the same label and type-equivalent target:

$$\odot \xrightarrow{a} n \in G \implies \exists \odot \xrightarrow{a} n' \in G' \bullet G \odot n \sim G' \odot n'.$$

By making a structural shallow copy of the root node and its outgoing edges, we obtain the *shallow structure* of a rooted graph.

Definition 10 (Shallow structure) A rooted graph G' is a shallow structure of a rooted graph G, denoted as $G' \otimes G$, if

1. the root of G' is a structural node: $G'.\odot \in \mathcal{S}$,
2. the two rooted graphs shallow cover each other:

$$G' \hat{\otimes} G \wedge G \hat{\otimes} G'.$$

Figure 3 shows a class B and its shallow structure rooted at v. We will see later in this section that the shallow structure of the graph of a generic class C is used as the *template* of instantiations for C. Note that the graph of C itself has a nominal root and cannot be used directly to generate instantiations of C whose roots are structural.

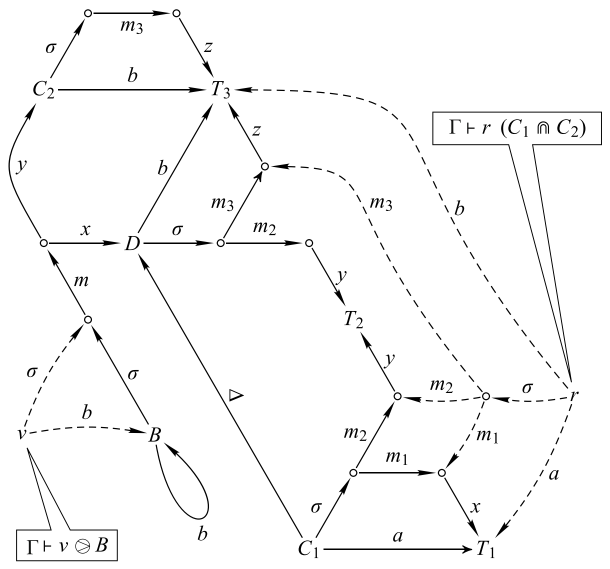


Fig. 3 A shallow structure and a conjunction graph

Given a type graph Γ and a node n, function *shlo* introduces a new structural node n', and new edges from n' to the adjacent nodes of n. The new graph rooted at n' is a shallow structure of $\Gamma \odot n$. Formally, $shlo(\Gamma, n) \mathrel{\hat{=}} (\Gamma', n')$, where

$$\begin{aligned} \Gamma'.ns &= \Gamma.ns \cup \{n'\}, \\ \Gamma'.es &= \Gamma.es \cup \{n' \xrightarrow{a} t \mid n \xrightarrow{a} t \in \Gamma.es\}, \qquad (4) \\ n' &\in \mathcal{S} \setminus \Gamma.ns. \end{aligned}$$

4.2 Conjunction graphs

A class contains not only the members defined in the class itself, but also the members of its superclasses. Inductively along the inheritance hierarchy, the *effective* member structure of a class is a *conjunction* of all the members of its superclasses. The effective member structure is also called the *interface* of the class.

The notion of conjunction is actually general, and can be applied to a set of arbitrary classes, provided they are *consistent*: members with the same name in different classes have equivalent types. We often use a conjunction to combine a number of interfaces so as to define a structural requirement that is to be implemented by a concrete class.

To formalize conjunctions, we introduce a notion of *memberwise cover*. It is an extension of shallow cover with the consideration of methods, and members from superclasses.

Definition 11 (Memberwise cover) A rooted graph G' is a memberwise cover of a rooted graph G, denoted as $G' \hat{\supset} G$, if

1. all effective attributes in G are included in G':

$$\odot \xrightarrow{\rhd}{}^{*}\, t \in G \implies G' \mathrel{\hat{\circledcirc}} G \odot t \setminus \{t \xrightarrow{a} n \mid a \notin \mathcal{A}\},$$

2. all effective method signatures in G are included in G':

$$\odot \xrightarrow{\rhd}{}^{*} \xrightarrow{\sigma} s \in G \implies G'_s \mathrel{\hat{\circledcirc}} G \odot s \setminus \{s \xrightarrow{a} n \mid a \notin \mathcal{A}\},$$

where $\odot \xrightarrow{\sigma} s' \in G'$ and $G'_s = G' \odot s'$.

A conjunction of a set of classes *minimally* memberwise covers all of these classes.

Definition 12 (Conjunction graph) Given a list of rooted graphs $\overline{G}$, a rooted graph G' is a conjunction graph of $\overline{G}$, denoted as $G' \sim \Cap \overline{G}$, if

$$\begin{aligned} G'.\odot \in \mathcal{S}, \qquad & \odot \xrightarrow{a} \cdot \in G' \implies a \in \mathcal{A} \cup \{\sigma\}, \\ & \odot \xrightarrow{\sigma.m} \cdot \in G' \implies m \in \mathcal{A}, \\ \overline{G' \mathrel{\hat{\supset}} G}, \qquad & \overline{G'' \mathrel{\hat{\supset}} G} \implies G'' \mathrel{\hat{\supset}} G'. \end{aligned}$$

Figure 3 shows a conjunction graph rooted at r, of two classes C_1 and C_2. A conjunction graph can be considered as a shallow structure of multiple classes.

In a type graph Γ, two nodes n and n' may lead to type-equivalent nodes t and t' through the same navigation path $\overline{a}$. We can thus choose either t or t' for the corresponding type. Formally, let $\overline{n}$ be a list of nodes of Γ and $\overline{a}$ a navigation path, we define *co-target* $(\Gamma, \overline{n}, \overline{a}) \mathrel{\hat{=}} ts_1$, where

$$\begin{aligned} ts_1 &= \begin{cases} \{t\}, & \text{if } \exists t \in ts \bullet t' \in ts \implies \Gamma \vdash t \sim t', \\ \emptyset, & \text{otherwise, if } ts = \emptyset, \end{cases} \\ ts &= \{t \mid n \xrightarrow{\rhd}{}^{*} \xrightarrow{\overline{a}} t \in \Gamma, n \in \overline{n}\}. \end{aligned} \tag{5}$$

Based on function *co-target*, we define a function *conj* to construct a conjunction graph. Formally, $conj(\Gamma, \overline{n}) \mathrel{\hat{=}} (\Gamma', r)$, where $r, s \in \mathcal{S} \setminus \Gamma.ns$, $r \neq s$ and

$$\begin{aligned} \Gamma'.ns &= \Gamma.ns \cup \{r, s\}, \\ \Gamma'.es &= \Gamma.es \cup \{r \xrightarrow{\sigma} s\} \\ &\quad \cup \{r \xrightarrow{a} t \mid t \in co\text{-}target(\Gamma, \overline{n}, a), a \in \mathcal{A}\} \\ &\quad \cup \{s \xrightarrow{m} t \mid t \in co\text{-}target(\Gamma, \overline{n}, \sigma.m), m \in \mathcal{A}\}. \end{aligned} \tag{6}$$

Function $conj(\Gamma, \overline{n})$ actually provides an approach to constructing a conjunction graph of the type graphs rooted at $\overline{n}$. It is straightforward to prove the following lemma, i.e., such construction is correct.

Lemma 3 If $(\Gamma', r) = conj(\Gamma, \overline{n})$ is defined, then $\Gamma' \vdash r \sim \Cap \overline{n}$.

In particular, the conjunction graph of a single rooted graph G is the effective member structure of G, called the *memberwise closure graph*, denoted as $clo(G)$. Formally,

$$clo(G) \mathrel{\hat{=}} \Gamma' \odot r, \text{ where } (\Gamma', r) = conj(G.base, G.\odot). \tag{7}$$

With the closure graph, we retrieve the attributes and methods of a class directly, without walking through any $\rhd$-path. In particular, we define a *memberwise closure cover* relation $\hat{\supset}_{clo}$ to determine if the rooted graph of an inherited class includes the effective members of another graph. Formally,

$$G' \mathrel{\hat{\supset}_{clo}} G \mathrel{\hat{=}} clo(G') \mathrel{\hat{\supset}} G. \tag{8}$$

For a method m in a rooted graph G, we define a function $mfr(G, m)$ to get the edges of the method frame, which comprise all the parameters of the method. Formally,

$$mfr(G, m) \mathrel{\hat{=}} \{n \xrightarrow{a} t \mid \odot \xrightarrow{\sigma.m} n \xrightarrow{a} t \in clo(G)\}. \tag{9}$$

4.3 Class constraints

A type variable can be declared with constraints that the actual type should satisfy. We have two kinds of constraints: member constraints and superclass constraints. A member constraint lists the minimal set of members for a class, while a superclass constraint requires a class to have a specific superclass. A type variable is possible to have both constraints.

We encode the constraints on a type variable as a graph rooted at the type variable. This allows us to assume the members and superclass of the type variable by its graph. When the type variable is replaced by an actual type satisfying the constraints, the type-safety is retained.

For some types $\overline{S}$ and T in a program, let Γ be a type graph of the program, $\overline{n}$ the nodes representing types $\overline{S}$ and t the node representing type T. A member constraint α **impl.** $\Cap \overline{S}$ on a type variable α is encoded as a conjunction graph of $\overline{\Gamma \odot n}$, with the root of the graph changed to α. A superclass constraint α **ext.** T on α is encoded as an edge $\alpha \xrightarrow{\rhd} t$.

The satisfaction of a member constraint is formalized as a memberwise closure cover, between the rooted graph of the actual type and the rooted graph of the type variable.

If a rooted graph G' contains an $\rhd$-chain from the root to a node n, G' is a subclass of the subgraph rooted at n of G'. We call G' a superclass cover of a rooted graph that encodes a superclass constraint $\odot \xrightarrow{\rhd} n$.

Definition 13 (Superclass cover) A rooted graph G' is a superclass cover of a rooted graph G, denoted as $G' \mathrel{\hat{\rhd}} G$, if an edge $\odot \xrightarrow{\rhd} n$ in G implies that G' is a subclass of n:

$$\odot \xrightarrow{\rhd} n \in G \implies G' \trianglerighteq G \odot n.$$

4.4 Instantiation graphs

Given a generic class $C\langle\overline{\alpha}\rangle$ with a list $\overline{\alpha}$ of type variables, $C\langle\overline{\alpha \mapsto Te}\rangle$ is a class instantiation of $C\langle\overline{\alpha}\rangle$, where $\overline{Te}$ is a list of type expressions. We call $\langle\overline{\alpha \mapsto Te}\rangle$ the *type substitution* of the instantiation. The root of the graph of the generic class $C\langle\overline{\alpha}\rangle$ is named with C. However, the root of the graph of an instantiation $C\langle\overline{\alpha \mapsto Te}\rangle$ is a structural node. And in this instantiation graph, all the subgraphs rooted at the type variables $\overline{\alpha}$ in the graph of $C\langle\overline{\alpha}\rangle$ are replaced by the corresponding graphs of the actual types $\overline{Te}$. Let $\overline{u}$ be the roots of the graphs of $\overline{Te}$. We also call $\langle\overline{\alpha \mapsto u}\rangle$ the *type substitution* of the instantiation of the graph of $C\langle\overline{\alpha}\rangle$ with the graph of $C\langle\overline{\alpha \mapsto Te}\rangle$.

Generally, any rooted graph containing type variables is generic, and can be used as a *template* for instantiation. Type variables only appear in the graph of a generic class, or an instantiation graph resulted by using type variables as actual types. We unify the two cases by using the shallow structure of a generic class as the template for the instantiation of the generic class. In fact, the shallow structure is type-equivalent to an instantiation graph of the generic class over an *identity type substitution* $\langle\overline{\alpha \mapsto \alpha}\rangle$. Moreover, since a generic class itself cannot be used as a type, there is no edge pointing to the root of its graph. Thus, in a template graph, nodes on a path to a type variable can only be structural nodes or type variables.

To use a homomorphism to define the instantiation of a template G over a type substitution $\langle\overline{\alpha \mapsto u}\rangle$, we define the subgraph of G terminating at the $\overline{\alpha}$-nodes. For a list $\overline{n}$ of nodes in a rooted graph G, a subgraph of G *terminating* at $\overline{n}$ is obtained from G by removing all the outgoing edges of $\overline{n}$,

$$G \oslash \overline{n} \triangleq G \setminus \{n \xrightarrow{a} t \in G \mid n \in \overline{n}\}. \qquad (10)$$

Notice that $G \oslash \overline{n}$ has the same root as G.

Definition 14 (Instantiation graph) Given a type graph Γ, two nodes $t, j \in \Gamma$ and a type substitution $s = \langle\overline{\alpha \mapsto u}\rangle$, let $G = \Gamma \odot t \oslash \overline{\alpha}$ and $J = \Gamma \odot j$. J is an instantiation graph of t over s if there is a graph homomorphism $f : G \to J$ such that

1. type variables $\overline{\alpha}$ are respectively mapped to their replacements $\overline{u}$: $\overline{f(\alpha) = u}$,
2. f is a structural preservation on the set $G.ns \setminus \overline{\alpha}$ of other nodes, as defined in Definition 4.

We call f the *instantiation function*, and j an *instantiation node* of t. By projecting the instantiation function f to a subgraph of G, we have a property that $f(n)$ is an instantiation node of a node $n \in G$.

Figure 4 (left) shows a generic class C and two instantiation graphs of C, one over $\langle\alpha \mapsto \mathbb{S}, \beta \mapsto \mathbb{S}\rangle$ as a subgraph, the other over $\langle\alpha \mapsto \mathbb{Z}, \beta \mapsto C\langle\alpha \mapsto \mathbb{S}, \beta \mapsto \mathbb{S}\rangle\rangle$. Figure 4 (right) shows two mutually recursive generic classes A and B. Each graph rooted at a double circle is type-equivalent to an instantiation graph of A. The left one in A is over $\langle\alpha \mapsto \alpha\rangle$, the middle two are over $\langle\alpha \mapsto \mathbb{Z}\rangle$, and the right one in B is over $\langle\alpha \mapsto \beta\rangle$.

In a type graph, instantiations of a template over the same type substitution are structural equivalent.

Lemma 4 For a type graph Γ, a node $t \in \Gamma$, a type substitution $s = \langle\overline{\alpha \mapsto u}\rangle$, and two instantiation graphs J_1 and J_2 of t over s, we have $J_1 \simeq J_2$.

Proof Let $G = \Gamma \odot t \oslash \overline{\alpha}$, f_1 and f_2 be the instantiation functions from G to J_1 and J_2, respectively, and $U = \{u \mid \alpha \in G, \alpha \mapsto u \in s\}$. Notice that U excludes those mappings in s of type variables not in G. We can construct the homomorphism $h : J_1 \to J_2$ as follows:

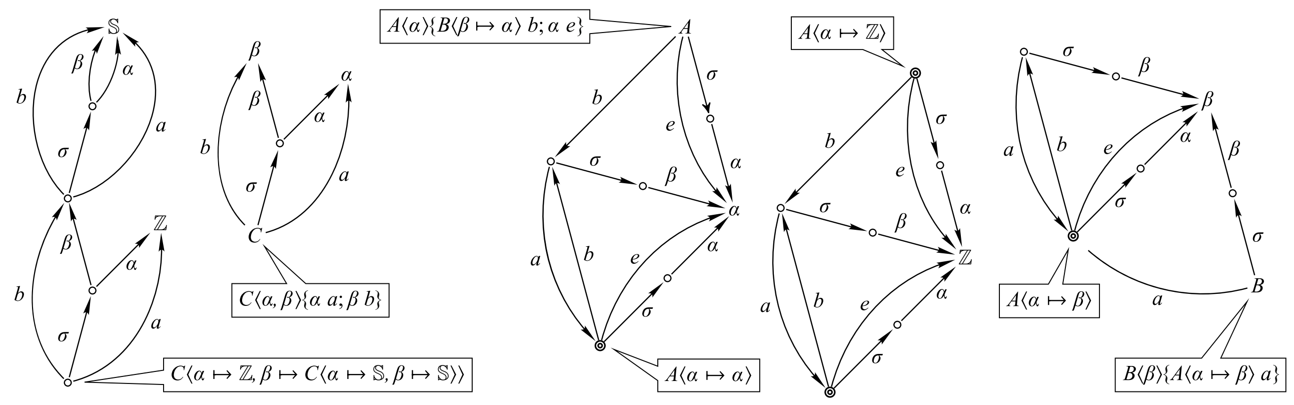

$\mathbb{S}$
β
α
b
a
σ
$\mathbb{Z}$
$C\langle\alpha \mapsto \mathbb{Z}, \beta \mapsto C\langle\alpha \mapsto \mathbb{S}, \beta \mapsto \mathbb{S}\rangle\rangle$
C
$C\langle\alpha, \beta\rangle\{\alpha\ a; \beta\ b\}$
$A\langle\alpha\rangle\{B\langle\beta \mapsto \alpha\rangle\ b; \alpha\ e\}$
A
e
$A\langle\alpha \mapsto \alpha\rangle$
$A\langle\alpha \mapsto \mathbb{Z}\rangle$
$A\langle\alpha \mapsto \beta\rangle$
B
$B\langle\beta\rangle\{A\langle\alpha \mapsto \beta\rangle\ a\}$

Fig. 4 Generic classes and class instantiations

- $h(n) = n$, if $n \in U$, since n must be in both J_1 and J_2,
- $h(n) = f_2(f_1^{-1}(n))$, if $n \notin U$, since both f_1 and f_2 are injective on nodes not in $\overline{\alpha}$.

We can show that if $t \in \overline{\alpha}$, $h(J_1.\odot) = J_1.\odot = f_1(t) = f_2(t) = J_2.\odot$, otherwise, $h(J_1.\odot) = f_2(f_1^{-1}(J_1.\odot)) = f_2(t) = J_2.\odot$. Thus, h is a structural preservation on all nodes. □

For a node $t \in \Gamma$, we denote an instantiation node j of t over a type substitution s as $\Gamma \vdash j \simeq t\langle s\rangle$. By definition, an instantiation function cannot map a class name to a different node. Thus, we cannot directly instantiate from a generic class. Instead, for a generic class C, if j is the instantiation node of a shallow structure of C, i.e., $\Gamma \vdash n \circledcirc C, j \simeq n\langle s\rangle$, we also say that j is an instantiation node of C over s in Γ, denoted as $\Gamma \vdash j \simeq C\langle s\rangle$.

Given that $\Gamma \vdash j \simeq t\langle s\rangle$, a node n that is type-equivalent to j in Γ is called a *class instantiation* of t over s, denoted as $\Gamma \vdash n \sim t\langle s\rangle$.

We now show that for a type graph Γ, a node $t \in \Gamma$ and a type substitution s, there is always an extension Γ' of Γ with a partial function f from Γ to Γ' such that $\Gamma' \vdash f(t) \simeq t\langle s\rangle$.

We define the extending function as ins_0 in Table 3. The function takes a type graph Γ, a set ts of the root nodes of templates and a map f as the arguments, and returns an updated type graph with an updated map. Initially, ts is a singleton set of the node t, and f is the type substitution s. For each structural node $r \in ts$, the function introduces a new structural node j, adds a mapping $r \mapsto j$, then recursively constructs the instantiation nodes for the adjacent nodes of r, finally adds new edges from j to these instantiation nodes, corresponding to the edges from r to its adjacent nodes. For each nominal node r in ts but not in $\overline{\alpha}$, the function adds an identity mapping $r \mapsto r$. The map f in the function is also used as the "visited" flags, where a node in the domain of f is considered visited.

We define the ins_T function as a wrapper of ins_0 to pass in the required arguments. Also, we define the ins_{Ge} function to instantiate from a generic class by first obtaining its shallow structure as the template. Formally,

$$\begin{aligned} ins_T(\Gamma, t, s) &\hat{=} (\Gamma', f(t)), \\ ins_{Ge}(\Gamma, t, s) &\hat{=} ins_T(\Gamma'', n, s), \end{aligned} \tag{11}$$

where $(\Gamma', f) = ins_0(\Gamma, \{t\}, s)$ and $(\Gamma'', n) = shlo(\Gamma, t)$.

Let $\overline{\alpha}$ be the list of type variables in the type substitution of an instantiation. The following lemma shows that the extending function ins_0 always returns a type graph containing the instantiation graph, provided that each node on a path from the root of the template to a node in $\overline{\alpha}$ is either in $\mathcal{S}$ or in $\overline{\alpha}$.

Lemma 5 Given a type graph Γ, a node $t \in \Gamma$ and a type substitution $s = \langle\overline{\alpha \mapsto u}\rangle$, let $(\Gamma', f) = ins_0(\Gamma, \{t\}, s)$ and $G = \Gamma \odot t \oslash \overline{\alpha}$, if

$$\forall \alpha \in \overline{\alpha}, t \to^* n \to^* \alpha \in G \bullet n \in \mathcal{S} \cup \{\overline{\alpha}\},$$

then $\Gamma' \vdash f(t) \simeq t\langle s\rangle$, and

$$g = f \cup \{n \mapsto n \mid n \in G.ns \setminus \mathrm{dom}(f)\},$$

is the instantiation function.

Proof We need only to show that g is the instantiation function. First, g is a homomorphism, since each new node in Γ' is a structural node in $\mathcal{S}$, and is an image under f; for every $x \in \mathrm{dom}(f)$, each of its outgoing edges $x \xrightarrow{a} y$ is mapped to $f(x) \xrightarrow{a} f(y)$; for all other nodes, their outgoing edges are not changed. Second, on a path from t with only structural nodes to a nominal node, including those in $\overline{\alpha}$, all the structural nodes are mapped to the new nodes; all the other nodes except $\overline{\alpha}$ are mapped to themselves, thus the combined map is injective. Finally, $\overline{\alpha}$ are mapped according to the type substitution. Therefore, g is the instantiation function. □

Type variables of a generic class can be used as actual types within type expressions that occur in the generic class declaration. These type expressions need to be instantiated when the generic class is instantiated. We have two kinds of type expressions, class instantiations and class conjunctions. An instantiation $t\langle s_1\rangle\langle s_2\rangle$ of a class instantiation $t\langle s_1\rangle$ is equivalent to the instantiation of the template t over the composition $s_1 \circ s_2$ of the type substitutions s_1 and s_2.

Lemma 6 Given a type graph Γ, a node $t \in \Gamma$ and two type substitutions s_1, s_2, if $\Gamma \vdash t' \simeq t\langle s_1\rangle$ and $\Gamma \vdash t'' \simeq t'\langle s_2\rangle$, then $\Gamma \vdash t'' \simeq t\langle s_1 \circ s_2\rangle$.

Table 3 Construction of instantiations

$$ins_0(\Gamma, ts, f) \hat{=} \quad \frac{r \in ts \setminus \mathcal{S} \setminus \mathrm{dom}(f)}{ins_0(\Gamma, ts, f \cup \{r \mapsto r\})} \qquad \frac{\begin{array}{c} r \in ts \setminus \mathrm{dom}(f) \quad \langle N, E\rangle = \Gamma \quad j \in \mathcal{S} \setminus N \\ (\langle N', E'\rangle, f') = ins_0(\langle N \cup \{j\}, E\rangle, \{n \mid r \xrightarrow{a} n \in E\}, f \cup \{r \mapsto j\}) \end{array}}{ins_0(\langle N', E' \cup \{j \xrightarrow{a} f'(n) \mid r \xrightarrow{a} n \in E\}\rangle, ts, f')} \qquad \frac{}{(\Gamma, f)}.$$

Proof By Definition 14 and the definition of substitution composition, we can directly establish the required structural preservation. □

In addition, an instantiation of a conjunction of a set of classes can be distributed over the conjunction.

Lemma 7 Given a type graph Γ, nodes $\overline{t} \in \Gamma$ and a type substitution s, if for nodes $\overline{j}, j', r, r'$,

$$\Gamma \vdash \overline{j \simeq t\langle s\rangle},\ r' \sim \Cap \overline{j},\ r \sim \Cap \overline{t},\ j' \simeq r\langle s\rangle,$$

then $\Gamma \vdash r' \sim j'$.

Proof For a navigation path $\overline{a}$, and a path $j' \xrightarrow{\overline{a}} u \in \Gamma$, by instantiation, there is either a path $r \xrightarrow{\overline{a}} u \in \Gamma$, possibly in term of structural equivalence, or a path $r \xrightarrow{\overline{a}} \alpha \in \Gamma$ if $\alpha \mapsto u \in s$. Thus, by conjunction, there is a node $t_0 \in \overline{t}$ such that $t_0 \xrightarrow{\rhd}{}^{*}\xrightarrow{\overline{a}} u \in \Gamma$, or $t_0 \xrightarrow{\rhd}{}^{*}\xrightarrow{\overline{a}} \alpha \in \Gamma$. Let $j_0 \in \overline{j}$ be the instantiation node of t_0. There is a path $j_0 \xrightarrow{\rhd}{}^{*}\xrightarrow{\overline{a}} u_0 \in \Gamma$ such that $\Gamma \vdash u_0 \simeq u$, either directly mapped, or by the substitution $\alpha \mapsto u$. Thus there is a path $r' \xrightarrow{\overline{a}} u_0' \in \Gamma$ such that $\Gamma \vdash u_0' \sim u$. We have proven that for such a navigation path $\overline{a}$, $j' \xrightarrow{\overline{a}} u \in \Gamma \implies \exists r' \xrightarrow{\overline{a}} u_0' \bullet \Gamma \vdash u_0' \sim u$. The opposite is similar. We then apply Lemma 2 to establish the type-equivalence. □

4.5 Recursive generic classes

For a non-recursive generic class $D\langle\overline{\beta}\rangle$, we construct the graph of $D\langle\overline{\beta}\rangle$ directly from its textual definition. Based on this graph, we can construct any instantiation of $D\langle\overline{\beta}\rangle$ in the way stated in the previous subsection. However, for a recursive generic class $C\langle\overline{\alpha}\rangle$, a class instantiation $C\langle s\rangle$, where s is a type substitution, may be used as the type of an attribute of $C\langle\overline{\alpha}\rangle$. In this case, there is an edge from the node C to the node representing $C\langle s\rangle$, thus the graph of $C\langle\overline{\alpha}\rangle$ depends on the graph of $C\langle s\rangle$. For the same reason the graph of $C\langle s\rangle$ depends on the graph of $C\langle s^2\rangle$, and so on, where $s^k \mathrel{\hat{=}} s^{k-1} \circ s$. Notice that Lemma 6 ensures that the notation of instantiation $C\langle s^k\rangle$ is well defined for a positive number k. Because of the recursive instantiation, a generic class $C\langle\overline{\alpha}\rangle$ with an attribute of type $C\langle s\rangle$ is defined only if the type substitution s is *convergent*.

A type substitution s is called *convergent* if the set $\{s^i \mid i > 0\}$ is finite, otherwise it is called *divergent*. For example, $\langle\alpha \mapsto \alpha\rangle$ and $\langle\alpha \mapsto \beta, \beta \mapsto \alpha\rangle$ are convergent substitutions, while $\langle\alpha \mapsto C\langle\alpha \mapsto \alpha\rangle\rangle$ and $\langle\alpha \mapsto \gamma, \gamma \mapsto D\langle\beta \mapsto \alpha\rangle\rangle$ are divergent ones. A type substitution s is divergent if and only if there is a list of mappings $\alpha_1 \mapsto Te_1, \alpha_2 \mapsto Te_2, \ldots, \alpha_k \mapsto Te_k$ $(k \geqslant 1)$ in s such that α_{i+1} occurs as an actual type in Te_i for $1 \leqslant i < k$ and α_1 occurs as an actual type in a class instantiation in Te_k.

When s is convergent, the set $\{C\langle s^i\rangle \mid i > 0\}$ of instantiations that the generic class $C\langle\overline{\alpha}\rangle$ depends on is finite, i.e., the graph of $C\langle\overline{\alpha}\rangle$ is finite. Since the types $C\langle s^i\rangle$ mutually refer to each other, we construct the graph of $C\langle\overline{\alpha}\rangle$ and these instantiations $C\langle s^i\rangle$ of $C\langle\overline{\alpha}\rangle$ altogether. We first identify the nodes of the graph, each of which represents a type expression that $C\langle\overline{\alpha}\rangle$ refers to, directly or indirectly. This can be done by using a function to associate each type expression with a unique node. Then, we add the edges between these nodes based on the relations between their corresponding type expressions.

Let F be an injective function from type expressions to $\mathcal{N}$ such that $F(t) = t$ for $t \in \mathcal{C} \cup \mathcal{X}$, and $F(t) \in \mathcal{S}$ otherwise. For a list $\overline{C}$ of classes, containing a recursive generic class $C_0\langle\overline{\alpha}\rangle$ and all the classes that C_0 refers to, generic or non-generic, the graph of $\overline{C}$ is constructed by the following procedure through F,

- for a superclass **ext**. Cc defined in a class C, we add an edge $C \xrightarrow{\rhd} F(Cc)$,
- for an attribute $Te\ a$ defined in a class C, we add an edge $C \xrightarrow{a} F(Te)$,
- for a parameter $Te\ x$ of method m defined in a class C, we add a path $C \xrightarrow{\sigma.m.x} F(Te)$,
- for each edge $C \xrightarrow{a} F(Te)$ and each instantiation $C\langle s\rangle$ of C, we add an edge $F(C\langle s\rangle) \xrightarrow{a} F(Te\langle s\rangle)$,
- for each path $C \xrightarrow{\sigma.m.x} F(Te)$ and each instantiation $C\langle s\rangle$ of C, we add an edge $F(C\langle s\rangle) \xrightarrow{\sigma.m.x} F(Te\langle s\rangle)$,
- for each class conjunction $Te = \Cap \overline{Cc}$ or type variable α with constraint **impl**. $\Cap \overline{Cc}$, we construct the conjunction graph G by *conj* and change the root of G to $F(Te)$ or α,
- for each type variable α with constraint **ext**. Cu, we add an edge $\alpha \xrightarrow{\rhd} F(Cu)$,

where C is a generic or non-generic class in $\overline{C}$. An instantiation of a type expression $Te\langle s\rangle$ is defined as,

$$Te\langle s\rangle \mathrel{\hat{=}} \begin{cases} s(\alpha), & \text{if } Te = \alpha \in \mathrm{dom}(s), \\ C\langle s' \circ s\rangle, & \text{if } Te = C\langle s'\rangle, \\ \Cap \overline{Cc\langle s\rangle}, & \text{if } Te = \Cap \overline{Cc}, \\ Te, & \text{otherwise.} \end{cases} \tag{12}$$

In fact, this procedure can be generally used to construct the graph of any class, not only recursive generic classes.

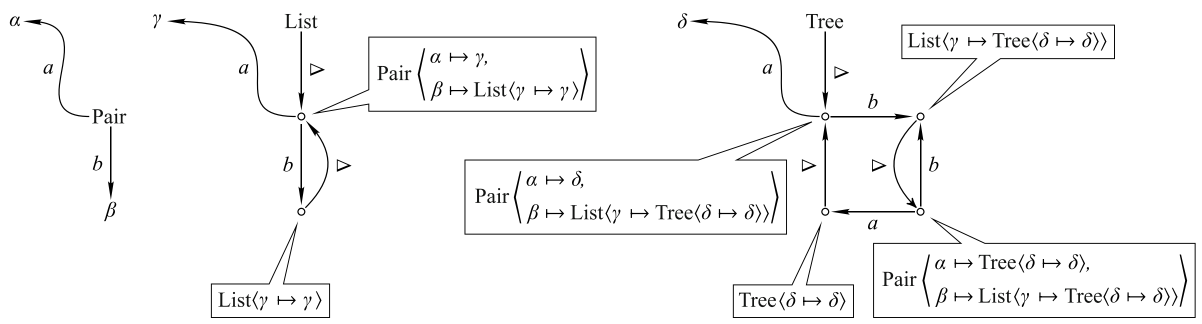

α
a
Pair
b
β
γ
List
a
▷
Pair⟨α ↦ γ, β ↦ List⟨γ ↦ γ⟩⟩
b
▷
List⟨γ ↦ γ⟩
δ
Tree
a
▷
b
List⟨γ ↦ Tree⟨δ ↦ δ⟩⟩
Pair⟨α ↦ δ, β ↦ List⟨γ ↦ Tree⟨δ ↦ δ⟩⟩⟩
▷
▷
b
a
Tree⟨δ ↦ δ⟩
Pair⟨α ↦ Tree⟨δ ↦ δ⟩, β ↦ List⟨γ ↦ Tree⟨δ ↦ δ⟩⟩⟩

Fig. 5 Recursive generic classes

Figure 5 shows the graphs of three generic classes, $Pair\langle\alpha,\beta\rangle$, $List\langle\gamma\rangle$ and $Tree\langle\delta\rangle$. Among the three, *List* and *Tree* are recursive. A *List* of γ extends a *Pair* of an element type γ and a *List* of γ. A *Tree* of δ extends a *Pair* of an element type δ and a *List* of *Tree* of δ. In the figure, the type expression *Te* in the callout box of a structural node n indicates there is a mapping $Te \mapsto n$ in F.

The soundness of the above construction procedure can be proved by the type-equivalence between an instantiation graph of a recursive generic class $C\langle\overline{\alpha}\rangle$, and the subgraph of $C\langle\overline{\alpha}\rangle$ corresponding to the class instantiation.

Lemma 8 Let F be an injective function from type expressions to $\mathcal{N}$ such that $F(t) = t$ for $t \in \mathcal{C} \cup \mathcal{X}$, and Γ be the type graph containing the graph of a generic class $C_0\langle\overline{\alpha}\rangle$ constructed in the above procedure through F. For any node j in Γ and type substitution $\langle\overline{\alpha \mapsto Te}\rangle$ such that $\Gamma \vdash j \simeq C_0\langle\overline{\alpha \mapsto F(Te)}\rangle$, we have $\Gamma \vdash j \sim F(C_0\langle\overline{\alpha \mapsto Te}\rangle)$.

Proof Let f be the instantiation function from $\Gamma \odot C_0 \otimes \overline{\alpha}$ to $\Gamma \odot j$. We can construct the reduction function g from $\Gamma \odot j$ to $\Gamma \odot F(C_0\langle\overline{\alpha \mapsto Te}\rangle)$ as $g(n) \mathrel{\hat{=}} n'$, where

$$n' = \begin{cases} F(C_0\langle\overline{\alpha \mapsto Te}\rangle), & \text{if } n = j, \\ n, & \text{if } \exists \alpha \in \overline{\alpha} \bullet f(\alpha) = n, \\ F(F^{-1}(f^{-1}(n))\langle\overline{\alpha \mapsto Te}\rangle), & \text{otherwise.} \end{cases}$$

□

5 Type system

The type system checks whether a program is well-typed statically based on a *well-formed* type graph of the program. The type-checking is done in a bottom-up way, from the checking of type expressions, expressions and commands to that of methods and programs. During the type-checking, commands are translated to their *environment dependent* counterparts, where type expressions are replaced by their corresponding type nodes in the type graph.

For example, the well-typedness of the command "$C\langle\alpha \mapsto \mathbb{B}\rangle$.**new**$(o); o.a := 1$" requires the type graph to have a node t representing the class instantiation $C\langle\alpha \mapsto \mathbb{B}\rangle$ which has an outgoing edge a to $\mathbb{Z}$. The type-checking procedure then translates the command to "$\lceil t \rfloor$.**new**$(o); o.a := 1$", which is environment dependent, where $\lceil\,\rfloor$ encloses a type node.

After the type-checking, an *environment graph* is constructed from the type graph to store the type information for program execution. The types in the environment graph are referred to by the type nodes $\lceil t \rfloor$ in the environment dependent commands.

5.1 Well-formed type substitutions

A type substitution is considered well-formed in a type graph if the constraints of each type variable are satisfied by its corresponding actual type. The rooted graph of the actual type must memberwise and superclass cover the *instantiated* shallow structure of the rooted graph of the type variable.

An instantiation over a well-formed type substitution ensures that if a member access of an object, or an assignment is type-safe for a template, it is also type-safe for the instantiation. For example, "$C\langle\alpha$ **ext**. $\beta,\beta\rangle\{\alpha\ a;\beta\ b\}$" is a generic class with a superclass constraint on the type variable α, in a method of C, "$b := a$" is type-safe. For a type substitution $\langle\alpha \mapsto A, \beta \mapsto B\rangle$ to be well-formed, A must be a subclass of B. Then, for an instantiation over $\langle\alpha \mapsto A, \beta \mapsto B\rangle$, we have "$\{A\ a; B\ b\}$", thus "$b := a$" is also type-safe.

Given a type graph Γ, a type variable α, a node u in Γ and a type substitution s, for a mapping $\alpha \mapsto u \in s$, we say *u satisfies* the constraints on α over s in Γ, if there exist nodes n and j such that

$$\Gamma \vdash n \otimes \alpha,\ j \simeq n\langle s\rangle,\ u \mathrel{\hat{\supset}_{clo}} j,\ u \mathrel{\hat{\triangleright}} j. \tag{13}$$

Definition 15 (Well-formed type substitution) A type sub-stitution s is well-formed in a type graph Γ, denoted as $\Gamma \vdash \langle s\rangle$, if for all mappings $\alpha \mapsto u \in s$, type u satisfies the constraints on α over s in Γ.

5.2 Well-formed type graphs

The well-formedness of type graphs is defined upon the notion of well-formed nodes. The nodes of a type graph Γ are called *well-formed*, if they can be partitioned into the following nine subsets:

1. Primitive types: nodes in $\mathcal{D}$, with a σ-labeled edge to a non-generic static table, which is a leaf.
2. Method frames: nodes in $\mathcal{S}$, with

   2† $\mathcal{A}$-labeled edges to types.
3. Non-generic static tables: nodes in $\mathcal{S}$, with

   3† $\mathcal{A}$-labeled edges to method frames.
4. Generic static tables: nodes in $\mathcal{S}$, with 3† and $\mathcal{X}$-labeled edges to types.
5. Generic classes: nodes $C \in \mathcal{U}$, with 2†, a σ-labeled edge to a generic static table, and

   5† an optional $\rhd$-labeled edge to a concrete class, such that there is a path $C \xrightarrow{\sigma.\alpha} \alpha$ for any type variable α reachable from C.
6. Non-generic classes, or classes for short: nodes in $\mathcal{U}$ that reach no type variable, with 2†, 5†, and

   6† a σ-labeled edge to a non-generic static table.
7. Type variables: nodes in $\mathcal{X}$, with 2†, 6†, and an optional $\rhd$-labeled edge to a concrete class or a type variable.
8. Class conjunctions: nodes in $\mathcal{S}$, with 2† and 6†.
9. Class instantiations: nodes $j \in \mathcal{S}$, such that $\Gamma \vdash j \sim C\langle s\rangle$ for some generic class C and well-formed type substitution s.

Here, a *class type* means a node that belongs to either the subset 6, 7, 8, or 9, while a *type* means either a class type or a primitive type. A *concrete class* means a node in either the subset 6 or 9. In addition, a *static table* means either a non-generic static table or a generic static table.

Notice that we add an empty static table to each of the primitive types, so that we can treat primitive types and class types uniformly in the type system. It is also worth pointing out that the above partitioning is unique, since each node belongs to at most one specific subset, decided by its own outgoing edges.

Definition 16 (Well-formed type graph) A well-formed type graph consists of only well-formed nodes and satisfies:

1. there is no cycle consisting of only $\rhd$-labeled edges;
2. any two generic classes have two disjoint sets of type variable labels outgoing from their static tables.

5.3 Environment dependent commands

The type information needed for execution of a program is encoded in an environment graph. For this purpose, we need to translate type expressions in the commands to type nodes of the environment graph. A translated command is called an *environment dependent command*. Through the type nodes stored in the environment dependent commands, we can lookup the type information in the corresponding environment graph. The syntax of environment dependent commands is the same as that in Table 1, except for the following primitives involving types:

$$\begin{array}{rll} c ::= \ldots & \text{other commands} & \\ \mid \mathbf{var}\ \overline{\lceil t \rfloor\ x} & \text{variable declaration} & \\ \mid \lceil t \rfloor.\mathbf{new}(le) & \text{object creation} & \\ e ::= \ldots & \text{other expressions} & (14) \\ \mid (\lceil t \rfloor)e & \text{typecast} & \\ ext\text{-}v ::= \lceil t \rfloor\ x & \text{external variable declaration} & \\ \lceil t \rfloor & \text{type node} & \end{array}$$

5.4 Type graph completeness

Given a program, the type graph Γ of the program must include the rooted graphs of all the possible type expressions used in the program. The graphs of the type expressions to define class attributes and method parameters are constructed by the procedure described in Section 4.5. For the type expressions used in commands and expressions, we construct the conjunction and instantiation graphs by applying the *shlo*, *conj*, ins_T and ins_{Ge} functions on the graphs of the operands of the type operations, and extend the type graph to include the resulted graphs. We do this by extending the type graph Γ to a new type graph Γ'. As mentioned in the beginning of Section 4, we only introduce new nodes with new edges pointing into the old graph Γ. The following lemma ensures that any existing rooted graph in Γ remains unchanged when the type graph is extended.

Lemma 9 If a type graph Γ' is the result of an application of *shlo*, *conj*, ins_T, or ins_{Ge} to a type graph Γ, then $\Gamma \odot n = \Gamma' \odot n$ for all nodes $n \in \Gamma$.

Proof By the definitions of *shlo* by Eq. (4), *conj* by Eq. (6), ins_T and ins_{Ge} by Eq. (11). □

Let f be one of the *shlo*, *conj*, ins_T, and ins_{Ge} functions,

Γ a type graph, and $\overline{X}$ the rest of the arguments of f. We can always extend Γ to Γ' without changing any rooted graphs in Γ, provided that $(\Gamma', n') = f(\Gamma, \overline{X})$ is defined. To simplify the type-checking rules, we require all the rooted graphs of type expressions to be already in Γ, and abbreviate

$$\exists n \in \Gamma \bullet \Gamma' \odot n' \sim \Gamma \odot n, \text{ where } (\Gamma', n') = f(\Gamma, \overline{X}),$$

as $\Gamma \vdash n \sim f(\overline{X})$.

5.5 Type contexts

A type graph Γ alone is not enough to decide whether an expression or a command is well-typed. For example, a variable x is well-typed inside a local scope **var** $T\ x; \cdots;$ **end** x, but it is undefined with respect to Γ. To address this issue, we introduce a graph notation of *type contexts* that describes the scopes of local variables. A type context graph consists of nodes representing scopes, and edges labeled by the variables from their scope nodes to the corresponding type nodes in the underlying type graph.

Let $\mathcal{O}$ be a new infinite set of nodes disjoint with $\mathcal{N}$. Besides the scope nodes of type contexts, the nodes in $\mathcal{O}$ will also be used to represent the scopes and objects of program execution states later in the paper.

Definition 17 (Type context) A type context is a rooted, directed and labeled graph $\Delta = \langle N, E, r \rangle$, where

- $N \subseteq \mathcal{N} \cup \mathcal{O}$ is the set of nodes, containing *type nodes* $N \cap \mathcal{N}$ and *scope nodes* $N \cap \mathcal{O}$,
- $E : N \times (\mathcal{A} \cup \{\$, self\}) \to N$ is the set of edges,
- $r \in N \cap \mathcal{O}$ is the root of the graph.

A type context $\Delta = \langle N, E, r \rangle$ is well-formed if the following conditions hold,

1. each outgoing edge of a scope node is labeled either by a variable (or *self*) and targets a type node, or by \$ and targets another scope node,
2. all the scope nodes are on a path starting from the root r, with edges labeled by \$, and
3. except the root r, which has no incoming edge, each scope node has exactly one incoming edge.

In Fig. 6, the dashed subgraphs are examples of type contexts.

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 each scope at that time. 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 scope is pushed onto the top of the stack. When the type-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.

For a list $\overline{x}$ of variables and a list $\overline{n}$ of nodes, $push\,(\Delta, \overline{x}, \overline{n})$ adds a new scope to a type context Δ with outgoing edges labeled by $\overline{x}$ and pointing to $\overline{n}$, respectively. Formally, $push\,(\Delta, \overline{x}, \overline{n}) \mathrel{\hat{=}} \langle N, E, r \rangle$, where

$$\begin{aligned} r \in \mathcal{O} \setminus \Delta.ns\,, \quad & N = \Delta.ns \cup \{r, \overline{n}\}, \\ & E = \Delta.es \cup \{r \xrightarrow{\$} \Delta.\odot, \overline{r \xrightarrow{x} n}\}. \end{aligned} \tag{15}$$

As shown in Fig. 6, ending a scope pops the root out of the stack by simply removing it, along with all its outgoing edges, from the type context, and the next node on the stack becomes the root. Formally,

$$pop\,(\Delta) \mathrel{\hat{=}} \Delta \odot r_{next}, \quad \text{provided} \odot \xrightarrow{\$} r_{next} \in \Delta. \tag{16}$$

A type context only adds and removes scope nodes, pointing into the underlying type graph, and does not introduce any other change.

Let Δ be a type context, n one of its scope nodes and w a variable (or *self*). We define a partial function $search\,(\Delta, n, w)$ that searches for the edge labeled by w in Δ from n node-by-node down the stack. Formally, $search\,(\Delta, n, w) \mathrel{\hat{=}} d$, where

$$d = \begin{cases} n \xrightarrow{w} t, & \text{if } n \xrightarrow{w} t \in \Delta, \\ search\,(\Delta, n', w), & \text{otherwise, if } n \xrightarrow{\$} n' \in \Delta. \end{cases} \tag{17}$$

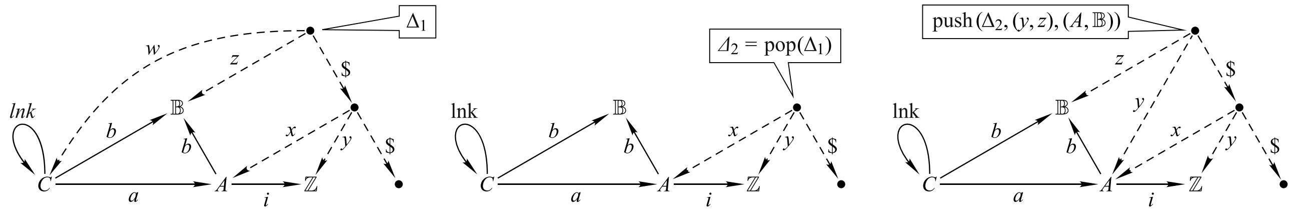


Fig. 6 Stack push and pop

And $search(\Delta, w) \mathrel{\hat=} search(\Delta, \Delta.\odot, w)$ searches from the root. Notice that the recursion always terminates as there is no loop formed by \$-edges.

Based on *search*, the function

$$\Delta(w) \mathrel{\hat=} target(search(\Delta, w)) \tag{18}$$

returns the type of w in Δ. $\Delta(w)$ exists if and only if there is an edge in Δ 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 Δ, $size(\Delta, n)$ the length of the \$-path starting from n in Δ, and $depth(\Delta, n)$ the length of the \$-path from the root to n in Δ. Formally,

$$\begin{aligned} var(\Delta, n) &\mathrel{\hat=} \{w \mid t \in \mathcal{N}, n \xrightarrow{w} t \in \Delta\},\\ size(\Delta, n) &\mathrel{\hat=} \begin{cases} 1 + size(\Delta, n'), & \text{if } n \xrightarrow{\$} n' \in \Delta,\\ 0, & \text{otherwise,}\end{cases} \\ depth(\Delta, n) &\mathrel{\hat=} size(\Delta, \Delta.\odot) - size(\Delta, n).\end{aligned} \tag{19}$$

We abbreviate $var(\Delta, \Delta.\odot)$ as $var(\Delta)$, i.e., the set of variables in the current scope, and $size(\Delta, \Delta.\odot)$ as $size(\Delta)$, i.e., the length of the entire \$-path.

The type-checking possibly updates the type context, and translates type expressions to their corresponding nodes in the type graph. Generally, the type-checking is a quintuple

$$\Gamma, \quad \Delta \to \Delta' \vdash X \to X'$$

meaning that in the type-checking, X is translated to X', X' is well-typed under Γ and Δ, and Δ is updated to Δ' after the checking.

5.6 Type-checking of type expressions

For a type graph Γ, a type expression *Te* and a node t, we use $\Gamma \vdash Te \to \lceil t \rfloor$ to denote that *Te* is translated to $\lceil t \rfloor$, and t is well-typed under Γ. The well-typedness of a type expression is independent of type contexts. For a class instantiation, we require that the type substitution contains exactly the type variables specified in the generic class. We define a function $t\text{-}var(\Gamma, t)$ to return the set of type variables of a node t in Γ. Formally,

$$t\text{-}var(\Gamma, t) \mathrel{\hat=} \{\alpha \mid t \xrightarrow{\sigma.\alpha} \cdot \in \Gamma\}, \quad \text{provided } t \in \Gamma. \tag{20}$$

We also define a function $ge\text{-}cls(\Gamma, j)$ to return the generic class of a class instantiation j in Γ. Formally, $ge\text{-}cls(\Gamma, j) \mathrel{\hat=} C$, provided that

$$j \in \mathcal{S}, \quad C \in \mathcal{C}, \quad t\text{-}var(\Gamma, C) = t\text{-}var(\Gamma, j) \neq \emptyset. \tag{21}$$

Table 4 lists the type-checking rules for type expressions.

Given a program and its type graph Γ, we define a function $a\text{-}vis(\Gamma, t, a)$, in Table 5, to return the visibility of an attribute a of type t. We say that a type t' *publicizes* a type t, denoted as $\Gamma \vdash t' \text{pubs}\ t$, if any attribute of t is also a public attribute of t', i.e.,

$$a \in \mathcal{A} \wedge \odot \xrightarrow{a} \cdot \in clo(\Gamma \odot t) \implies a\text{-}vis(\Gamma, t', a) = \mathbf{pub}. \tag{22}$$

Since all of the attributes in a class conjunction are assumed public, to conform to the visibility rule, in a memberwise cover $\Gamma \vdash t' \hat{\sqsupset} t$, we also require $\Gamma \vdash t' \text{pubs}\ t$. For simplicity, this requirement is implied in the type system.

Based on the visibility of an attribute a of a type u, whether u can navigate through a in a method m is also determined by the class C in which m is defined. The class C is recorded as the type of *self* in the type context Δ. Given a type graph Γ and a type context Δ, we denote the accessibility of a from u as $\Gamma, \Delta \vdash u \text{ sees } a$, defined by the ($V$-Priv), ($V$-Prot) and ($V$-Pub) rules in Table 5.

Table 4 Type-checking of type expressions

$$(T\text{-Name})\dfrac{T \in \mathcal{C} \cup \mathcal{X} \quad t\text{-}var(\Gamma, T) = \emptyset}{\Gamma \vdash T \to \lceil T \rfloor} \qquad (T\text{-Conj})\dfrac{\Gamma \vdash \overline{Te} \to \overline{\lceil t \rfloor} \quad \Gamma \vdash n \sim \Cap \overline{t}}{\Gamma \vdash \Cap \overline{Te} \to \lceil n \rfloor} \qquad (T\text{-Ins})\dfrac{\begin{array}{c} C \in \mathcal{C} \quad \{\overline{\alpha}\} = t\text{-}var(\Gamma, C) \quad \Gamma \vdash \overline{Te} \to \overline{\lceil t \rfloor} \\ \Gamma \vdash \langle \overline{\alpha \mapsto t} \rangle \quad \Gamma \vdash j \sim C\langle \overline{\alpha \mapsto t} \rangle \end{array}}{\Gamma \vdash C\langle \overline{\alpha \mapsto Te} \rangle \to \lceil j \rfloor}.$$

Table 5 Visibility of attributes

$$a\text{-}vis(\Gamma, t, a) \mathrel{\hat=} \quad \dfrac{\text{“}\mathbf{class}\ C\ \{vis\ \ Te\ a\}\text{”} \quad t = C}{\dfrac{t \xrightarrow{\triangleright} t' \xrightarrow{\triangleright}{}^{*}\xrightarrow{a} \cdot \in \Gamma}{\dfrac{vis \geqslant a\text{-}vis(\Gamma, t', a) > \mathbf{priv}.}{vis}} \quad \dfrac{}{vis}} \qquad \dfrac{t \in \mathcal{S} \quad t \xrightarrow{a} \cdot \in \Gamma}{\dfrac{C = ge\text{-}cls(\Gamma, t)}{a\text{-}vis(\Gamma, C, a)} \quad \dfrac{}{\mathbf{pub}.}} \qquad \dfrac{\begin{array}{c} t \in \mathcal{C} \quad t \xrightarrow{\triangleright} t' \in \Gamma \\ a\text{-}vis(\Gamma, t', a) > \mathbf{priv}. \end{array}}{a\text{-}vis(\Gamma, t', a)} \qquad \dfrac{\begin{array}{c} t \in \mathcal{X} \\ t \xrightarrow{a} \cdot \in \Gamma \end{array}}{\mathbf{pub}.},$$

$$(V\text{-Priv})\dfrac{a\text{-}vis(\Gamma, u, a) = \mathbf{priv}. \quad \Gamma \vdash \Delta(self) \sim u}{\Gamma, \Delta \vdash u \text{ sees } a} \qquad (V\text{-Prot})\dfrac{a\text{-}vis(\Gamma, u, a) = \mathbf{prot}. \quad \Gamma \vdash \Delta(self) \trianglerighteq u}{\Gamma, \Delta \vdash u \text{ sees } a} \qquad (V\text{-Pub})\dfrac{a\text{-}vis(\Gamma, u, a) = \mathbf{pub}.}{\Gamma, \Delta \vdash u \text{ sees } a}.$$

Note: the order of visibilities is defined as **pub**. > **prot**. > **priv**.

5.7 Subtype relation

The compatibility of types should be checked in various occasions, including typecasts, assignments and method invocations. The compatibility is governed by the subtype relation between type expressions. Since we have translated type expressions to type nodes, we define the subtype relation between type nodes.

Given a type graph Γ, a node t' is a subtype of a node t, denoted as $\Gamma \vdash t' \preccurlyeq t$, if either t' is a subclass of t, or t is a class conjunction and t' is a memberwise closure cover of t. The rules for the subtype relation are given in Table 6.

Theorem 1 (Subtyping partial order) The subtype relation given in Table 6 is a partial order.

Proof Since $\rhd$-cycles are not allowed, and by Definitions 9, 11 and 12, we can verify that two well-formed class conjunctions memberwise covering each other are type-equivalent, so the subtype relation is antisymmetric.

We can also verify that the $\hat{\supset}_{clo}$ relation is transitive, and furthermore, by Definitions 12, 8 and Lemma 1, $G' \trianglerighteq G$ implies $G' \hat{\supset}_{clo} G$, hence the subtype relation is transitive. □

5.8 Type-checking of commands

For a type graph Γ, a type context Δ, an expression e and a type node t, we use $\Gamma, \Delta \vdash e \rightarrow e' : t$ to denote that e is translated to e', e' is well-typed under Γ and Δ, and e' is of type t in Γ. The type-checking of an expression does not update the type context, although it looks up variables in the type context.

For constant literals l, including those of the primitive types and the *null* object, we use $\mathbf{T}(l)$ to denote their types. For example, $\mathbf{T}(5) = \mathbb{Z}$, $\mathbf{T}(\mathit{false}) = \mathbb{B}$ and $\mathbf{T}(\mathit{null}) = \mathit{Null}$. The type-checking rules for expressions are given in Table 7.

A well-typed command must satisfy certain conditions by its own. For example, the types of the two expressions on the left-hand side and right-hand side of an assignment should be compatible, and the types of actual arguments in a method invocation should be consistent with the method signature. Such kind of requirements is called *local well-typedness*. Furthermore, a locally well-typed command c under Γ and Δ is well-typed, denoted as $\Gamma, \Delta \vdash c \rightarrow c'$, if its variable declarations and undeclarations always match. This means the original type context is never under-broken, and restored afterwards, during the type-checking.

Besides checking whether the stack sizes of the starting type context $\mathit{size}(\Delta)$ and ending type context $\mathit{size}(\Delta')$ are the same, we also need to distinguish commands such as $c_1 = (\textbf{var}\ T\ x; \textbf{end}\ x)$ and $c_2 = (\textbf{end}\ x; \textbf{var}\ T\ x)$. Both of them are locally well-typed, but the latter is not well-typed as a whole. For a type graph Γ, a type context Δ and a command c, we define the local well-typedness as $\Gamma, \Delta \rightarrow \Delta' : \rho \vdash c \rightarrow c'$, to denote that c is translated to its environment dependent version c', c' is locally well-typed under Γ and Δ, Δ is updated to Δ' after the checking, and the smallest size of the type context during the checking is ρ, where ρ is a natural number.

Now, it is clear that $\Gamma, \Delta \vdash c \rightarrow c'$ if and only if $\Gamma, \Delta \rightarrow \Delta : \mathit{size}(\Delta) \vdash c \rightarrow c'$, i.e., the command c maintains the size of the type context to be no less than the original and eventually restores the original type context.

For the object creation command $Cc.\textbf{new}(le)$, we check if Cc can be reduced to a concrete class node. A concrete class node is either a non-generic class name or a structural node that represents an instantiation. We define a predicate $\mathit{is}\text{-}\mathit{Cc}(\Gamma, t)$ to determine if t is a concrete class node in Γ,

$$\mathit{is}\text{-}\mathit{Cc}(\Gamma, t) \mathrel{\hat{=}} (X = \emptyset \wedge t \in \mathcal{U}) \vee (X \neq \emptyset \wedge t \in \mathcal{S}), \qquad (23)$$

where $X = t\text{-}\mathit{var}(\Gamma, t)$. We give the type-checking rules for commands in Table 9.

Table 6 Subtyping of type nodes

$$(\preccurlyeq\text{-}\textsc{Nul})\frac{t \notin \boldsymbol{D} \quad t \xrightarrow{\sigma} \cdot \in \Gamma}{\Gamma \vdash \mathit{Null} \preccurlyeq t} \qquad (\preccurlyeq\text{-}\textsc{Isa})\frac{\Gamma \vdash t' \trianglerighteq t}{\Gamma \vdash t' \preccurlyeq t} \qquad (\preccurlyeq\text{-}\textsc{Cover})\frac{t \in \mathcal{S} \quad t\text{-}\mathit{var}(\Gamma, t) = \emptyset \quad \Gamma \vdash t' \hat{\supset}_{clo} t}{\Gamma \vdash t' \preccurlyeq t}.$$

Table 7 Type-checking of expressions

$$(T\text{-}\textsc{Lit})\frac{\mathbf{T}(l) = B}{\Gamma, \Delta \vdash l \rightarrow l : B} \qquad (T\text{-}\textsc{Op})\frac{f : \overline{B} \rightarrow B' \quad \Gamma, \Delta \vdash \overline{e} \rightarrow \overline{e' : B}}{\Gamma, \Delta \vdash f(\overline{e}) \rightarrow f(\overline{e'}) : B'} \qquad (T\text{-}\textsc{Self})\frac{\Delta(\mathit{self}) = u}{\Gamma, \Delta \vdash \mathit{self} \rightarrow \mathit{self} : u}$$

$$(T\text{-}\textsc{Var})\frac{\Delta(x) = t}{\Gamma, \Delta \vdash x \rightarrow x : t} \qquad (T\text{-}\textsc{Attr})\frac{\Gamma, \Delta \vdash e \rightarrow e' : u \quad \odot \xrightarrow{a} t \in \mathit{clo}(\Gamma \odot u) \quad \Gamma, \Delta \vdash u \text{ sees } a}{\Gamma, \Delta \vdash e.a \rightarrow e'.a : t}$$

$$(T\text{-}\textsc{UCast})\frac{\Gamma \vdash Ce \rightarrow \lceil u \rfloor \quad \Gamma, \Delta \vdash e \rightarrow e' : u' \quad \Gamma \vdash u' \preccurlyeq u}{\Gamma, \Delta \vdash (Ce)e \rightarrow e' : u} \qquad (T\text{-}\textsc{DCast})\frac{\Gamma \vdash Ce \rightarrow \lceil u \rfloor \quad \Gamma, \Delta \vdash e \rightarrow e' : u' \quad \Gamma \vdash u \preccurlyeq u'}{\Gamma, \Delta \vdash (Ce)e \rightarrow (\lceil u \rfloor)e' : u}.$$

5.9 Instantiation of commands

After the type-checking of commands, type expressions in the commands have been replaced by their corresponding nodes in the type graph. An instantiation of a command over a type substitution is to replace the type nodes in the command with their instantiation nodes over the type substitution.

For expressions, type nodes occur only in typecast expressions. For commands, type nodes can occur in nested expressions, variable declarations and object creations. Given a type graph Γ and a type substitution s, we follow the structure of expressions, commands and type contexts to define the instantiation functions ins_E, ins_K, and ins_Δ, in Table 8, to respectively replace the type nodes in an expression e, a command c and a type context Δ with the instantiation nodes over the type substitution s.

We show in Lemma 10 that over a well-formed type substitution, such an instantiation preserves the subtype relations between any two type nodes in the command, and in Lemma 11 that a replacement node provides all the attributes and methods of a replaced node. Hence, the command instantiation is type-safe.

Lemma 10 Given a type graph Γ, two type nodes $t_1, t_2 \in \Gamma$ and a well-formed type substitution s, if $\Gamma \vdash j_1 \simeq t_1\langle s\rangle$ and $\Gamma \vdash j_2 \simeq t_2\langle s\rangle$, then

$$\Gamma \vdash t_1 \preccurlyeq t_2 \implies \Gamma \vdash j_1 \preccurlyeq j_2.$$

Proof Let $G_1 = \Gamma \odot t_1$, $G_2 = \Gamma \odot t_2$, $J_1 = \Gamma \odot j_1$ and $J_2 = \Gamma \odot j_2$. By the subtyping rules in Table 6, we need to prove

$$G_1 \trianglerighteq G_2 \implies J_1 \trianglerighteq J_2,$$
$$t_2 \in \mathcal{S} \wedge G_1 \hat{\supset}_{clo} G_2 \implies J_1 \hat{\supset}_{clo} J_2.$$

By Definition 14, a non-generic class remains the same in the instantiation, while in a structural type, only the type variables are replaced in the instantiation. Over the same type substitution, if two paths lead to the same type variable, they must lead to the same node after the instantiations. Furthermore, a type variable is nominal and only equivalent to itself. Thus, the equivalence is preserved for all nodes, the subclass relation and memberwise cover are preserved for nodes not in dom(s).

For a replaced type variable, if it covers a structural node, so does its shallow structure. By Definition 15, the replacement covers the instantiated shallow structure of the type variable, thus covers the instantiated structural node.

In the case that t_1 is a type variable subclassing t_2, by Definition 8, $G_1 \trianglerighteq G_2$ implies there exists a path $t_1 \overset{\triangleright}{\rightarrow}{}^{*}\, n \in G_1$, such that $G_1 \odot n \sim G_2$. By induction on the length k of the $\triangleright$-path, if $k = 0$, we have shown the equivalence case that $G_1 \sim G_2$ implies $J_1 \sim J_2$. If $k \geqslant 1$, there exists a path $t_1 \overset{\triangleright}{\rightarrow} n_0 \overset{\triangleright}{\rightarrow}{}^{*}\, n \in G_1$ and an edge $\odot \overset{\triangleright}{\rightarrow} j_0$ in the instantiation graph J'_1 of the shallow structure of t_1 over s. Let $G_0 = G_1 \odot n_0$ and $J_0 = J'_1 \odot j_0$. By Definition 15 and 13, $J_1 \trianglerighteq J_0$. By the induction hypothesis, we have that $G_0 \trianglerighteq G_2$ implies $J_0 \trianglerighteq J_2$, hence $J_1 \trianglerighteq J_2$. □

Lemma 11 Given a type graph Γ, a type node $t \in \Gamma$ and a well-formed type substitution s, if $\Gamma \vdash j \simeq t\langle s\rangle$, let $G = clo\,(\Gamma \odot t)$ and $J = clo\,(\Gamma \odot j)$, then

$$a \neq \sigma \wedge \odot \xrightarrow{a} n \in G \implies \exists \odot \xrightarrow{a} n' \in J \bullet \Gamma \vdash n' \sim n\langle s\rangle,$$
$$\odot \xrightarrow{\sigma.m} n \in G \implies \exists \odot \xrightarrow{\sigma.m} n' \in J \bullet \Gamma \vdash n' \sim n\langle s\rangle.$$

Proof By the definition of clo by Eq. (7), the properties of the instantiation function in Definition 14 and Definition 15. □

From the above two lemmas, an instantiation of a well-typed command is also well-typed. We instantiate commands and type contexts by the functions ins_K and ins_Δ, respectively.

Table 8 Instantiation of expressions, commands, and type contexts

$$ins_E(\Gamma, e, s) \mathrel{\hat{=}} \frac{e = (\lceil t \rfloor)e' \quad \Gamma \vdash t' \sim ins_T(t, s)}{(\lceil t' \rfloor)ins_E(\Gamma, e')} \qquad \frac{e = e'.a}{ins_E(\Gamma, e').a} \qquad \frac{e = f(\overline{e'})}{f(\overline{ins_E(\Gamma, e')})} \qquad \frac{}{e},$$

$$ins_K(\Gamma, c, s) \mathrel{\hat{=}} \frac{c = \lceil u \rfloor.\mathbf{new}(le) \quad \Gamma \vdash u' \sim ins_T(u, s)}{\lceil u' \rfloor.\mathbf{new}(ins_E(\Gamma, le, s))} \qquad \frac{c = \mathbf{var}\ \overline{\lceil t \rfloor}\ x \quad \Gamma \vdash \overline{t' \sim ins_T(t, s)}}{\mathbf{var}\ \overline{\lceil t' \rfloor}\ x} \qquad \frac{c = le := e \quad (le', e') = ins_E(\Gamma, (le, e), s)}{le' := e'}$$

$$\frac{c = e.m(\overline{ve} : \overline{x} : \overline{re}) \quad (e', \overline{ve'}, \overline{re'}) = ins_E(\Gamma, (e, \overline{ve}, \overline{re}), s)}{e'.m(\overline{ve'} : \overline{x} : \overline{re'})} \qquad \frac{c = c_1 ; c_2 \quad (c'_1, c'_2) = ins_K(\Gamma, (c_1, c_2), s)}{c'_1 ; c'_2}$$

$$\frac{c = c_1 \lhd e \rhd c_2 \quad e' = ins_E(\Gamma, e, s) \quad (c'_1, c'_2) = ins_K(\Gamma, (c_1, c_2), s)}{c'_1 \lhd e' \rhd c'_2} \qquad \frac{c = e * c_1 \quad e' = ins_E(\Gamma, e, s) \quad c'_1 = ins_K(\Gamma, c_1, s)}{e' * c'_1} \qquad \frac{}{c},$$

$$ins_\Delta(\Gamma, \Delta, s) \mathrel{\hat{=}} \frac{t \in \mathcal{N} \quad d = n \xrightarrow{w} t \in \Delta \quad \Delta' = ins_\Delta(\Gamma, \Delta \smallsetminus \{d\}, s) \quad \Gamma \vdash t' \sim ins_T(t, s)}{\langle \Delta'.ns \cup \{t'\}, \Delta'.es \cup \{n \xrightarrow{w} t'\}, \Delta'.\odot \rangle} \qquad \frac{}{\Delta}.$$

Theorem 2 (Type-safety of command instantiations) Given a type graph Γ and a well-formed type substitution s, if $\Gamma, \Delta_1 \rightarrow \Delta_2 : \rho \vdash c$, let

$$c' = ins_K(\Gamma, c, s), \quad \Delta_1' = ins_\Delta(\Gamma, \Delta_1, s) \quad \text{and} \quad \Delta_2' = ins_\Delta(\Gamma, \Delta_2, s),$$

then $\Gamma, \Delta_1' \rightarrow \Delta_2' : \rho \vdash c'$.

Proof The theorem is proven by replacing all the type nodes with their corresponding instantiation nodes in the type-checking rules for expressions and commands, and then applying Lemma 5–7, 9–11. □

5.10 Type-checking of programs

Based on the type-checking of commands, we are able to check the well-typedness of a program. A program is well-typed if each method, including the *Main* method, defined in the program is well-typed, and a method definition is well-typed if its body command is well-typed according to its formal parameters.

In the type context of the checking of a method body command, the *self* reference points to the class defining the method. If the class is generic, *self* must point to an instantiation of the generic class over an identity type substitution, i.e., a shallow structure of the generic class. We define a function $self(\Gamma, t)$ to return the shallow structure when needed. Formally,

$$self(\Gamma, t) \mathrel{\hat{=}} \begin{cases} shlo(\Gamma, t), & \text{if } t\text{-}var(\Gamma, t) \neq \emptyset, \\ (\Gamma, t), & \text{otherwise.} \end{cases} \tag{24}$$

We start by creating the empty type context with only a root node,

$$\Delta_\emptyset = \langle \{r\}, \emptyset, r \rangle, \quad \text{where } r \in O. \tag{25}$$

For a method definition, we push the method frame onto $\Delta_\emptyset$, so that the result type context $\Delta_\emptyset'$ contains all the parameters respectively pointing to their types. We check whether the body command is well-typed under $\Delta_\emptyset'$. The following lemma ensures that if a parameter can be found in the method frame pushed onto the empty type context, it can also be found in the method frame pushed on any type context. Thus a well-typed method can always be invoked.

Lemma 12 Given a type graph Γ, a type context Δ and a command c, let

$$\Delta_\emptyset' = push(\Delta_\emptyset, \overline{x}, \overline{n}) \text{ and } \Delta' = push(\Delta, \overline{x}, \overline{n}),$$

if $\Gamma, \Delta_\emptyset' \vdash c$, then $\Gamma, \Delta' \vdash c$ and for all variables y occurred in c and all $\Delta_1 \rightarrow \Delta_2$ occurred in the deduction of $\Gamma, \Delta' \vdash c$ such that $search(\Delta_1, y) \notin \Delta$.

Proof We first show that if a variable is well-typed in $\Delta_\emptyset'$, it is also well-typed in Δ' and the variable cannot be in Δ. Then, we prove that the lemma holds for a general command by induction on the structure of commands and expressions.

If a variable y in c is well-typed, by (T-VAR), y is found by *search*, hence, y is either introduced by c above the stack of $\Delta_\emptyset'$, or in $\Delta_\emptyset'$. If y is in $\Delta_\emptyset'$, it must be in $\overline{x}$, because $\Delta_\emptyset$ contains no edge. So the edge for y is either in Δ' or introduced in command c, but not in Δ. □

We extract all the class methods $C :: m$ as a list $\overline{C :: m\{c\}}$, and the *Main* method as $\{\overline{Te\ x}; c\}$. In this way, a program is denoted as

$$P = \overline{C :: m\{c\}} \bullet \{\overline{Te\ x}; c\}.$$

Definition 18 (Well-typed program) Let Γ be a type graph and P a program. If $\Gamma \vdash P \rightarrow P'$, P' is a well-typed program under type graph Γ.

After the type-checking, if a program P' is well-typed under Γ, P' contains the environment dependent commands for all the methods. The type-checking rules for methods and programs are also given in Table 9.

6 Execution environments

The environment of a program stores the structure of the initial object plus the method body commands for each concrete class. An initial object in the environment is represented by a type node in the environment graph. The outgoing edges of this node are labeled by the attributes of the class and the targets of these edges are the default (or initial) values of the corresponding attributes. It also has an outgoing edge labeled by σ to the static table of the class, represented as a node with outgoing edges labeled by method names of the class to the corresponding body commands. The outgoing edges, the targeted literals and the static table of the class constitute the structure of the initial object of the class, called a *class frame*. There is also a set of nodes associated with the static table of each class, representing its supertypes.

Based on an environment graph, the execution of a program looks up methods, creates class instances and checks typecasts at runtime without the need of a type graph.

6.1 Environment graphs

Let $\mathcal{L}$ be the set of literals, including the *null* object and the

Table 9 Type-checking of commands, methods, and programs

$$(T\text{-Skip})\ \Gamma,\Delta \vdash \mathbf{skip} \to \mathbf{skip} \qquad (T\text{-Invk})\frac{\Gamma,\Delta \vdash e, \overline{ve}, \overline{re} \to e' : u, \overline{ve' : t}, \overline{re' : t'} \quad \overline{\{\cdot \xrightarrow{x} t''\}} = mfr(\Gamma \odot u, m) \quad \overline{\Gamma \vdash t \preccurlyeq t'' \preccurlyeq t'}}{\Gamma,\Delta \vdash e.m(\overline{ve} : \overline{x} : \overline{re}) \to e'.m(\overline{ve'} : \overline{x} : \overline{re'})}$$

$$(T\text{-Assign})\frac{\Gamma,\Delta \vdash le, e \to le' : t, e' : t' \quad \Gamma \vdash t' \preccurlyeq t}{\Gamma,\Delta \vdash le := e \to le' := e'} \qquad (T\text{-New})\frac{\Gamma \vdash Cc \to \lceil u \rfloor \quad is\text{-}Cc(\Gamma, u) \quad \Gamma,\Delta \vdash le \to le' : u' \quad \Gamma \vdash u \preccurlyeq u'}{\Gamma,\Delta \vdash Cc.\mathbf{new}(le) \to \lceil u \rfloor.\mathbf{new}(le')}$$

$$(T\text{-Decl})\frac{\Gamma \vdash \overline{Te} \to \overline{\lceil t \rfloor} \quad \rho = size(\Delta)}{\Gamma,\Delta \to push(\Delta, \overline{x}, \overline{t}) : \rho \vdash \mathbf{var}\ \overline{Te\ x} \to \mathbf{var}\ \overline{\lceil t \rfloor\ x}} \qquad (T\text{-End})\frac{var(\Delta) = \{\overline{x}\} \quad \rho = size(\Delta) > 1}{\Gamma,\Delta \to pop(\Delta) : \rho - 1 \vdash \mathbf{end}\ \overline{x} \to \mathbf{end}\ \overline{x}}$$

$$(T\text{-Seq})\frac{\Gamma,\Delta \to \Delta'' : \rho_1 \vdash c_1 \to c_1' \quad \Gamma,\Delta'' \to \Delta' : \rho_2 \vdash c_2 \to c_2'}{\Gamma,\Delta \to \Delta' : \min(\rho_1,\rho_2) \vdash c_1; c_2 \to c_1'; c_2'} \qquad (T\text{-If})\frac{\Gamma,\Delta \vdash e, c_1, c_2 \to e' : \mathbb{B}, c_1', c_2'}{\Gamma,\Delta \vdash c_1 \lhd e \rhd c_2 \to c_1' \lhd e' \rhd c_2'}$$

$$(T\text{-While})\frac{\Gamma,\Delta \vdash e, c \to e' : \mathbb{B}, c'}{\Gamma,\Delta \vdash e * c \to e' * c'} \qquad \dagger \qquad (T\text{-Meth})\frac{\Gamma \vdash j \sim self(u) \quad \overline{\{\cdot \xrightarrow{x} t\}} = mfr(\Gamma \odot j, m) \quad \Gamma, push(\Delta_\emptyset, (self, \overline{x}), (j, \overline{t})) \vdash c \to c'}{\Gamma \vdash u :: m\{c\} \to u :: m\{c'\}}$$

$$(T\text{-Main})\frac{\Gamma \vdash \overline{Te} \to \overline{\lceil t \rfloor} \quad \Gamma, push(\Delta_\emptyset, \overline{x}, \overline{t}) \vdash c \to c'}{\Gamma \vdash \{\overline{Te\ x}; c\} \to \{\overline{\lceil t \rfloor\ x}; c'\}} \qquad \dagger \qquad (T\text{-Prog})\frac{\Gamma \vdash \overline{md}, Md \to \overline{md'}, Md'}{\Gamma \vdash \overline{md} \bullet Md \to \overline{md'} \bullet Md'}.$$

values of primitive types, and $\mathcal{K}$ the set of environment dependent commands. We assume $\mathcal{L}$, $\mathcal{K}$, $\mathcal{O}$ and $\mathcal{N}$ are disjoint.

Definition 19 (Environment graph) An environment graph is a directed and labeled graph $R = \langle N, E, S \rangle$, where

- $N \subseteq \mathcal{N} \cup \mathcal{L} \cup \mathcal{K}$, is the set of type nodes, static tables, literals and commands,
- $E : (N \cap \mathcal{N}) \times (\mathcal{A} \cup \{\sigma\}) \to N$ is the set of edges,
- $S : \mathcal{S} \to 2^{\mathcal{N}}$ is a partial function mapping each static table in N to a set of superclass nodes, denoted as $R.sm$.

An environment graph $R = \langle N, E, S \rangle$ is well-formed if

1. for a method body command $k \in N \cap \mathcal{K}$, there is at least one incoming edge to k in R and every incoming edge to k is from a node in dom(S),
2. for a static table $s \in$ dom(S), there is exactly one incoming edge labeled σ to s in R and all the nodes adjacent to s is in $\mathcal{K}$,
3. for a type node $t \notin \mathcal{K} \cup$dom$(S) \cup \mathcal{L}$, there is no incoming edge to t in R and there is an outgoing edge of t labeled σ in R, and
4. each literal $l \in \mathcal{L}$ is a leaf in R.

Let R be an environment graph. An edge $C \xrightarrow{a} l$ in R with l in $\mathcal{L}$ means that type C has an attribute a with initial value l. A node u in set $R.sm(s)$ means that an object having static table s can be cast to type u. An edge $s \xrightarrow{m} k$ with k in $\mathcal{K}$ means that an object having static table s has a method m with k as the body command.

Figure 7 shows an environment graph. In the graph, u represents the class frame of a class instantiation u, which is a structural node; D and C represent the class frames of a subclass D of u and a subclass C of D, respectively. Each class frame contains two attribute edges i, j (two additional a, b for C) to their initial values, and a σ-edge to the static table. Each static table contains a method edge m_1 leading to the corresponding body command. We can see that D does not override the m_1 method inherited from u, that they share the same command, while C overrides m_1 with a different command. Also, the superclass map S maps the static table of each class to the set of superclasses (including the class itself).

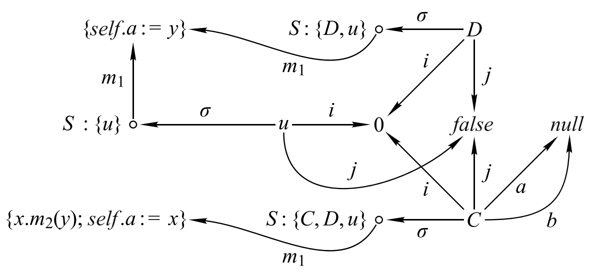


Fig. 7 An environment graph

6.2 Construction of environment graphs

Based on the type graph of a well-typed program, the construction of the environment graph of the program is straightforward. We copy all the nodes of concrete classes in the type graph along with their attributes and static tables, point the attributes to their initial values. For each method, we lookup its body command in the environment dependent program, and instantiate the command when needed, which may bring in more class instantiations. We also associate the static table of each type with the set of its supertypes.

For a type graph Γ and a program P, we construct the environment graph by

$$cons_R(P, \Gamma, \langle \emptyset, \emptyset, \emptyset \rangle), \tag{26}$$

Table 10 Construction of environment graphs

$$cons_R(P,\Gamma,\langle N,E,S\rangle) \mathrel{\hat{=}} \quad (R\text{-}\textsc{Attr})\frac{t \xrightarrow{\sigma} s \in \Gamma \quad \textit{is-Cc}(\Gamma,t) \quad t \notin N \quad es = \{t \xrightarrow{a} \textit{ini}(n) \mid \odot \xrightarrow{a} n \in \textit{clo}(\Gamma \odot t), a \neq \sigma\} \quad ns = \{l \mid t \xrightarrow{a} l \in es\}}{cons_R(P,\Gamma,\langle N \cup \{t,s\} \cup ns, E \cup \{t \xrightarrow{\sigma} s\} \cup es, S\rangle)}$$

$$(R\text{-}\textsc{Meth})\frac{t \xrightarrow{\sigma} s \in N \quad s \xrightarrow{m} \cdot \in \textit{clo}(\Gamma \odot t) \quad s \xrightarrow{m} \cdot \notin N \qquad k = \textit{ins}_K(\Gamma, \textit{mk}(P,\Gamma,t,m), \textit{ssj}(\Gamma,t))}{cons_R(P,\Gamma,\langle N \cup \{k\}, E \cup \{s \xrightarrow{m} k\}, S\rangle)} \qquad (R\text{-}\textsc{Super})\frac{t \xrightarrow{\sigma} s \in N \quad s \notin \mathrm{dom}(S) \qquad ts = \{t' \mid \Gamma \vdash t \preccurlyeq t'\}}{cons_R(P,\Gamma,\langle N,E,S \cup \{s \mapsto ts\}\rangle)} \qquad \frac{}{\langle N,E,S\rangle}.$$

where the function $cons_R$ is defined in Table 10.

In (R-Attr), the function $cons_R$ builds the class frame that represents the structure of the initial object for each concrete class node t in Γ. It adds to the environment graph the type nodes representing t and its static table s, the default values of all attributes of t, and the edges outgoing from t labeled by σ to the static table and by the attributes to their default values. We use $\textit{ini}(t')$ to denote the default value of a type node t', i.e., $\textit{ini}(\mathbb{Z}) = 0$, $\textit{ini}(\mathbb{B}) = \textit{false}$, $\textit{ini}(\mathbb{S}) = \varepsilon$ and $\textit{ini}(C) = \textit{null}$ for any class type C. In (R-Meth), $cons_R$ constructs outgoing edges of static tables labeled by method names, targeting the instantiated body commands of the methods respectively. Finally, $cons_R$ associates each static table s of t with the set of supertype nodes of t, as shown in (R-Super).

In order to get the instantiated method body commands, we introduce two functions. First, for a class instantiation $j \in \Gamma$, when we instantiate a command for j, the type substitution can be recovered from the edges labeled by type variables in the generic static table of j. The function $\textit{ssj}$ returns the type substitution. Formally,

$$\textit{ssj}(\Gamma, j) \mathrel{\hat{=}} \langle \alpha \mapsto t \mid \alpha \in \mathcal{X}, j \xrightarrow{\sigma.\alpha} t \in \Gamma\rangle. \tag{27}$$

Second, for method body commands, we define a function $\textit{mk}$ to lookup in program P for body command k_0 of method m of type t. If t is a class instantiation, we track for its template generic class in Γ. Formally, $\textit{mk}(P,\Gamma,t,m) \mathrel{\hat{=}} k_0$, where

$$k_0 = \begin{cases} k_0', & \text{if } \exists t :: m\{k_0'\} \in P, \\ \textit{mk}(P,\Gamma,t',m), & \text{if } \exists t' = \textit{ge-cls}(\Gamma,t), \\ \textit{mk}(P,\Gamma,t',m), & \text{otherwise, if } \exists t \xrightarrow{\rhd} t' \in \Gamma. \end{cases} \tag{28}$$

When t is directly defined in P, $\textit{mk}$ returns the body command by the method definition, otherwise it looks up the command in the template generic class or the superclass. Thus, for a type node t, we can instantiate the body command k_0 for a method m of t over the type instantiation extracted from $\textit{ssj}(\Gamma,t)$, and finally add the instantiated commands to the environment graph.

7 Operational semantics

We now proceed to define a small-step operational semantics for *r*COS/g, based on state graphs and their well-typedness. We prove the *strong type soundness* following the standard approach [25], that a well-typed program can always *progress* (or terminate) on a well-typed state graph, *preserving* the well-typedness.

7.1 State graphs

A state of a program at runtime consists of objects, values (states) of their attributes, as well as stacked (local) scopes and edges representing variables from scopes to objects and literal values. Each small-step of the execution of the program is to change the state by creating a new object, assigning to an attribute or a variable, creating a new scope with new variables, or destroying the current scope.

Such a program state can be represented by a rooted graph, with scopes, objects and literal values being the nodes, variables and attributes being the edges.

Definition 20 (State graph) Given an environment graph R, a state graph is a rooted, directed and labeled graph $G = \langle N, E, r\rangle$, where

- $N \subseteq \mathcal{O} \cup \mathcal{L} \cup \mathcal{S}$ is the set of nodes,
- $E : N \times (\mathcal{A} \cup \{\sigma, \textit{self}, \$\}) \to N$ is the set of edges,
- $r \in N \cap \mathcal{O}$ is the root of the graph.

A state graph $G = \langle N, E, r\rangle$ is well-formed, representing a proper program state, if it satisfies that

1. starting from r, the \$-edges, if there are any, form a \$-path (stack) such that except r, which has no incoming edge, each node on the path has only one incoming edge,
2. a node is either an *object node* in $\mathcal{O}$ but not on the \$-path, a literal value in $\mathcal{L}$, a static table in $\mathcal{S}$, or a *scope node* in $\mathcal{O}$ and on the \$-path,
3. a literal value is a leaf, having no outgoing edges,

4. the source of an edge labeled by *self* is a scope node,
5. an object node has an outgoing edge labeled σ pointing to a static table, and
6. a static table in G must also be a static table in $\mathrm{dom}(R.sm)$.

A state graph G is shown as solid lines in Fig. 8.

```
var A x, ℤ y; A.new(x); x.b := true;
var C w, 𝔹 z; C.new(w); z := x.b; C.new(w.lnk); w.a := x;
```

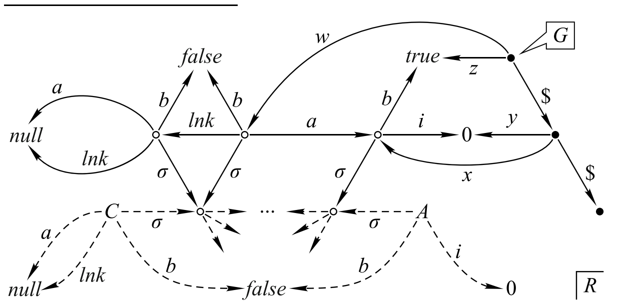


Fig. 8 A state graph and the corresponding environment graph

Besides extending the stack operations on type contexts to state graphs, we define the *swing* operation to change the targets of a list of edges in a state graph. Formally, given $G = \langle N, E, r\rangle$, $\overline{d = n' \xrightarrow{a} \cdot} \in E$, and $\overline{n} \in N \cup \mathcal{L}$, we define $swing\,(G, \overline{d}, \overline{n}) \mathrel{\hat{=}} \langle N', E'\rangle \odot r$, where

$$N' = N \cup \{\overline{n}\}, \quad E' = E \oplus \overline{n' \xrightarrow{a} n}. \tag{29}$$

The $M \oplus \overline{e}$ map operation sequentially adds mappings $\overline{e}$ to M and overrides the existing mappings. Formally,

$$M \oplus (a \mapsto b, \overline{e'}) \mathrel{\hat{=}} (\{x \mapsto y \in M \mid x \neq a\} \cup \{a \mapsto b\}) \oplus \overline{e'}. \tag{30}$$

7.2 Execution environment lookup

The static table nodes connect objects to their runtime class information in the environment graph. Via static tables, we can lookup superclasses, method names and method body commands. A static table is introduced to a state graph by an object creation, which can be simplified as shallow copying all the outgoing edges of a type node in the environment graph to the state graph with the substitution of a new object node for the type node.

Besides the static table of an object, we can also access the environment graph through the environment dependent $\lceil t \rfloor$.**new** command. Given an environment graph R, a type node $t \in R$, a state graph G and an edge $d \in G$, we define a function to add a new instance of t to the state graph and swing d to the instance. Formally, let $G = \langle N, E, r\rangle$ and $d = n \xrightarrow{a} \cdot \in E$, we define $new\,(R, G, t, d) \mathrel{\hat{=}} \langle N', E'\rangle \odot r$, where

$$\begin{aligned} o \in \mathcal{O} \setminus N, \quad N' &= N \cup \{o\} \cup \{l \mid t \xrightarrow{b} l \in R\}, \\ E' &= E \cup \{o \xrightarrow{b} l \mid t \xrightarrow{b} l \in R\} \oplus n \xrightarrow{a} o. \end{aligned} \tag{31}$$

An environment graph R, which corresponds to a state graph G, is shown as dashed lines in Fig. 8.

7.3 Well-typed state graphs

A state graph G is well-typed if it is consistent with a type graph Γ and a type context Δ. For an object node $n \in G$, n must correspond to a concrete class node in Γ, denoted as $t = \Gamma(G, n)$, such that

$$is\text{-}Cc(\Gamma, t), \quad t \xrightarrow{\sigma} s \in \Gamma, \quad n \xrightarrow{\sigma} s \in G, \tag{32}$$

for some static table s. For a literal value l, l corresponds to its type $\mathrm{T}(l)$. So, we define $\Gamma(G, l)$ as $\mathrm{T}(l)$. For a scope node $n_s \in G$ satisfying $depth\,(n_s) < size\,(\Delta)$, there must exist a scope node in Δ, denoted as $t_s = \Delta(G, n_s)$, such that $depth\,(G, n_s) = depth\,(\Delta, t_s)$.

Definition 21 (Well-typed state graph) Let Γ be a type graph and Δ a type context. A state graph G is well-typed w.r.t. $\Gamma \bullet \Delta$, if

1. for an object node $n \in G$ and $a \in \mathcal{A}$,
$$n \xrightarrow{a} n' \in G \iff \Gamma(G, n) \xrightarrow{\triangleright}{}^{*} \xrightarrow{a} t' \in \Gamma,$$
$$\text{such that } \Gamma \vdash \Gamma(G, n') \preccurlyeq t',$$
2. for a scope node $n_s \in G$ with $depth\,(G, n_s) < size\,(\Delta)$ and $w \in \mathcal{A} \cup \{self\}$,
$$n_s \xrightarrow{w} n' \in G \iff \Delta(G, n_s) \xrightarrow{w} t' \in \Delta,$$
$$\text{such that } \Gamma \vdash \Gamma(G, n') \preccurlyeq t',$$
3. $size\,(G) \geqslant size\,(\Delta) - 1$.

The -1 offset on $size\,(\Delta)$ is a result of the approach of building type contexts upon $\Delta_\emptyset$. The state graph G shown in Fig. 8 is well-typed w.r.t. the type graph and the type context Δ_1 in Fig. 6. In the state graph, there are two scopes with local variables on the stack. The outer scope (lower on the stack) has variables x and y, leading to an object of class A and integer 0, respectively. The inner scope (upper on the stack) has variables w and z, leading to an object of class C and boolean value *true*, respectively. The object referred to by w has an attribute *lnk* leading to another object of class C.

7.4 Semantic rules

Given the environment graph R constructed from a type graph

Table 11 Operational semantics of commands and evaluation of expressions

$$(\textsc{Lit})\frac{l \in \mathcal{L}}{eval\,(G, l) \mathrel{\hat{=}} l} \qquad (\textsc{Op})\frac{\overline{n = eval\,(G, e)} \in \mathcal{L}}{eval\,(G, f(\overline{e})) \mathrel{\hat{=}} f(\overline{n})} \qquad (\textsc{Cast})\frac{n = eval\,(G, e) \quad n \xrightarrow{\sigma} s \in G \quad t \in R.sm\,(s)}{eval\,(G, (\lceil t \rfloor)e) \mathrel{\hat{=}} n}$$

$$(\textsc{Var})\frac{w \in \mathcal{A} \cup \{self\} \cup \{x^* \mid x \in \mathcal{A}\}}{eval\,(G, w) \mathrel{\hat{=}} G(w)} \qquad (\textsc{Attr})\frac{eval\,(G, e) \xrightarrow{a} n \in G}{eval\,(G, e.a) \mathrel{\hat{=}} n} \quad \dagger \quad (l\text{-}\textsc{Var})\frac{x \in \mathcal{A}}{l\text{-}eval\,(G, x) \mathrel{\hat{=}} search\,(G, x)}$$

$$(l\text{-}\textsc{Attr})\frac{q = eval\,(G, e) \quad a \in \mathcal{A} \quad q \xrightarrow{a} n \in G}{l\text{-}eval\,(G, e.a) \mathrel{\hat{=}} q \xrightarrow{a} n} \quad \dagger \quad (p\text{-}\textsc{Var})\frac{x \in \mathcal{A}}{po\,(G, x) \mathrel{\hat{=}} null} \qquad (p\text{-}\textsc{Attr})\frac{q = eval\,(G, e) \quad a \in \mathcal{A}}{po\,(G, e.a) \mathrel{\hat{=}} q}$$

$$(r\text{-}\textsc{Var})\frac{q = null \quad x \in \mathcal{A}}{spo\,(G, x, q) \mathrel{\hat{=}} l\text{-}eval\,(pop\,(G), x)} \qquad (r\text{-}\textsc{Attr})\frac{q \neq null \quad q \xrightarrow{a} n \in G}{spo\,(G, e.a, q) \mathrel{\hat{=}} q \xrightarrow{a} n} \quad \dagger \quad (\textsc{Skip})\,\langle \mathbf{skip}, G\rangle \to G$$

$$(\textsc{Declare})\langle \mathbf{var}\ \overline{\lceil t \rfloor\ x}, G\rangle \to push\,(G, \overline{x}, \overline{ini\,(t)}) \qquad (\textsc{End})\langle \mathbf{end}\ \overline{x}, G\rangle \to pop\,(G) \qquad (\textsc{Assign})\frac{d = l\text{-}eval\,(G, \leqslant) \quad n = eval\,(G, e)}{\langle \leqslant := e, G\rangle \to swing\,(G, d, n)}$$

$$(\textsc{New})\frac{d = l\text{-}eval\,(G, \leqslant)}{\langle \lceil t \rfloor.\mathbf{new}(\leqslant), G\rangle \to new\,(R, G, t, d)} \qquad (\textsc{Invk})\frac{o = eval\,(G, e) \quad o \xrightarrow{\sigma} s \in G \quad s \xrightarrow{m} k \in R}{\langle e.m(\overline{ve} : \overline{x} : \overline{re}), G\rangle \to \langle \boldsymbol{enter}(o, \overline{ve}, \overline{x}, \overline{re}); k; \boldsymbol{leave}(\overline{x}, \overline{re}), G\rangle}$$

$$(\textsc{Enter})\frac{\overline{n = eval\,(G, ve)} \quad \overline{q = po\,(G, re)}}{\langle \boldsymbol{enter}(o, \overline{ve}, \overline{x}, \overline{re}), G\rangle \to push\,(G, (self, \overline{x}, \overline{x^*}), (o, \overline{n}, \overline{q}))} \qquad (\textsc{Leave})\frac{\overline{d = spo\,(G, re, eval\,(G, x^*))} \quad \overline{n = eval\,(G, x)}}{\langle \boldsymbol{leave}(\overline{x}, \overline{re}), G\rangle \to pop\,(swing\,(G, \overline{d}, \overline{n}))}$$

$$(\textsc{Seq})\frac{\langle c_1, G\rangle \to \langle c_1', G'\rangle}{\langle c_1; c_2, G\rangle \to \langle c_1'; c_2, G'\rangle} \qquad (\textsc{Seq-})\frac{\langle c_1, G\rangle \to G'}{\langle c_1; c_2, G\rangle \to \langle c_2, G'\rangle} \qquad (\textsc{If}\text{-}f)\frac{eval\,(G, b) = false}{\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_2, G\rangle}$$

$$(\textsc{If}\text{-}t)\frac{eval\,(G, b) = true}{\langle c_1 \lhd b \rhd c_2, G\rangle \to \langle c_1, G\rangle} \qquad (\textsc{While}\text{-}f)\frac{eval\,(G, b) = false}{\langle b * c, G\rangle \to G} \qquad (\textsc{While}\text{-}t)\frac{eval\,(G, b) = true}{\langle b * c, G\rangle \to \langle c; b * c, G\rangle}.$$

Γ, the small-step operational semantics is given in Table 11, including the evaluation of expressions, l-expressions and the execution of commands.

Given a state graph G, an expression e is evaluated by $eval\,(G, e)$ to an object node in G or a literal. By contrast, an l-expression le is evaluated by $l\text{-}eval\,(G, le)$ to an edge in G that can be swung, representing an *l-value*, whose target node is not significant.

The execution of commands is defined by the transition relation $\to$ between *configurations*. There are two kinds of configurations:

1. a *non-terminated configuration* is a pair $\langle c, G\rangle$, representing a state graph G with a command c to be executed, and
2. a *terminated configuration* is a state graph G, representing the completion of the execution of a command.

The transition rules for assignment, object creation, variable declaration and undeclaration are defined by the simple graph operations of edge swing, new instance adding, stack push and pop. By contrast, the rule for method invocation is more dedicate and deserves some explanation.

In a method invocation, a result argument re is an l-expression. We have to remember the l-value of re initially before it is possibly changed during the invocation. If $re = e.a$ is a navigation path, we call the object referred to by e the *parent object* of the attribute a. Our approach is to assign this parent object $po\,(G, re)$ to an auxiliary variable x^* when entering a method. And when exiting the method, we recover the initial l-value of re by $spo\,(G, re, q)$ that returns the outgoing edge labeled by a of the parent object q retrieved from x^*. For a variable w, the notion of parent object is not significant since its l-value cannot be changed. For unification, we define the parent object of w as the $null$ object.

Figure 8 shows on top the state graph G a command sequence that can result the state from the initial state $\langle \{r\}, \emptyset, r\rangle$, where $r \in O$, provided the environment graph R.

7.5 Type-safety of programs

The type-safety of a program ensures that a well-typed expression can be evaluated and a well-typed command can be executed. We reason about the type-safety by showing that given a well-typed state graph, there exists a semantic rule which applies, and that the resulted state graph of the semantic rule is also well-typed.

There are exceptional cases when an expression cannot be evaluated, or a command cannot be executed.

Exceptional case 1 (Null reference) The evaluation of an expression $e.a$ or the execution of a command $e.m(\ldots)$ fails, if e is evaluated to $null$.

Exceptional case 2 (Illegal downcast) The evaluation of an expression $(\lceil t \rfloor)e$ fails, if e is evaluated to a node n and the type of n is not a subtype of t.

In general, these two exceptional cases can not be checked statically [8,26].

Lemma 13 Let Γ be a type graph, Δ a type context, e an expression, le an l-expression, and G a well-typed state graph w.r.t. $\Gamma \bullet \Delta$. Unless one of the exceptional cases happens, we have

1. if $\Gamma, \Delta \vdash e : t$, $n = eval(G, e)$ exists and $\Gamma \vdash \Gamma(G, n) \preccurlyeq t$,
2. if $\Gamma, \Delta \vdash le : t$, $l\text{-}eval(G, le)$ exists.

Proof The proof is given by induction on the structure of an expression. We only show two demonstrative cases.

A well-typed variable $\Gamma, \Delta \vdash w : t$ can only be deduced from (T-Var) and (T-Self). Thus we have $\Delta(w) = t$. Because G is well-typed w.r.t. $\Gamma \bullet \Delta$, $n = G(w)$ exists and $\Gamma \vdash \Gamma(G, n) \preccurlyeq t$. Because $\Delta(w) = target(search(\Delta, w))$ and $G(w) = target(search(G, w))$, $search(G, w)$ also exists.

A well-typed attribute $\Gamma, \Delta \vdash e.a : t$ can only be deduced from (T-Attr). Thus we have $\Gamma, \Delta \vdash e : u$ and $u \overset{\triangleright}{\rightarrow}{}^{*} \xrightarrow{a} t \in \Gamma$. By the induction hypothesis, $eval(G, e) = q$ and $\Gamma \vdash \Gamma(G, q) \preccurlyeq u$. By the subtyping rules, $\Gamma(G, q) \overset{\triangleright}{\rightarrow}{}^{*} \xrightarrow{a} t \in \Gamma$. Because G is well-typed w.r.t. $\Gamma \bullet \Delta$, $q \xrightarrow{a} n \in G$ and $\Gamma \vdash \Gamma(G, n) \preccurlyeq t$. □

Theorem 3 (Type-safety of commands) Let Γ be a type graph, R an environment graph derived from Γ, Δ and $\Delta_\diamond$ two type contexts, c a command, and G a well-typed state graph w.r.t. $\Gamma \bullet \Delta$. Unless one of the exceptional cases happens, if $\Gamma \vdash u :: m\{k\}$ for all $u \xrightarrow{\sigma.m} k \in R$, and $\Gamma, \Delta \rightarrow \Delta_\diamond \vdash c$, either

1. there exists a well-typed state graph $G_\diamond$ w.r.t. $\Gamma \bullet \Delta_\diamond$ such that $\langle c, G \rangle \rightarrow G_\diamond$, or
2. there exists a configuration $\langle c', G' \rangle$ and a type context Δ' with G' being a well-typed state graph w.r.t. $\Gamma \bullet \Delta'$ such that $\langle c, G \rangle \rightarrow \langle c', G' \rangle$ and $\Gamma, \Delta' \rightarrow \Delta_\diamond \vdash c'$.

Proof The proof is in general by induction on the structure of the command. We only give the proof for method invocation.

A well-typed method invocation $\Gamma, \Delta \vdash e.m(\overline{ve} : \overline{x} : \overline{re})$ is deduced from (T-Invk). Thus, we have $\Gamma, \Delta \vdash e : u$ and $mfr(\Gamma \odot u, m)$ defined. Hence, $o = eval(G, e)$ exists and $\odot \xrightarrow{\sigma.m} \cdot \in clo(\Gamma \odot u)$. Since $\Gamma \vdash u' = \Gamma(G, o) \preccurlyeq u$, we have $\odot \xrightarrow{\sigma.m} \cdot \in clo(\Gamma \odot u')$. By ($R$-Meth) and ($T$-Attr) in Table 10, and the type of object nodes in Eq. (32), we have $u' \xrightarrow{\sigma} s \xrightarrow{m} k \in R$ and $o \xrightarrow{\sigma} s \in G$. Therefore, by (Invk), the method invocation can be reduced to $\langle \boldsymbol{enter}(o, \overline{ve}, \overline{x}, \overline{re}); k; \boldsymbol{leave}(\overline{x}, \overline{re}), G \rangle$.

Also, we have $\Gamma, \Delta \vdash \overline{ve : t}, \overline{x : t''}, \overline{re : t'}, t \preccurlyeq t''$. It implies that (a) $\overline{n = eval(G, ve)}$ exist and $\Gamma \vdash \overline{\Gamma(G, n) \preccurlyeq t}$, and (b) $\overline{re}$ must be l-expressions, either variables or attributes. Hence, $\overline{q = po(G, re)}$ exist. Therefore, by auxiliary command $\boldsymbol{enter}$, G is transformed to $G_1 = push(G, (self, \overline{x}, \overline{x^*}), (o, \overline{n}, \overline{q}))$, which is well-typed w.r.t. $\Gamma \bullet \Delta_1 = push(\Delta, (self, \overline{x}), (u, \overline{t''}))$. Notice that Definition 21 of well-typed states permits extra auxiliary variables with names not in $\mathcal{A} \cup \{self\}$.

According to the premise, $\Gamma \vdash u :: m\{k\}$, and by ($T$-Meth), we have $\Gamma, push(\Delta_\emptyset, (self, \overline{x}), (u, \overline{t''})) \vdash k$. By Lemma 12, we also have $\Gamma, \Delta_1 \vdash k$, and k cannot see any variables in Δ, including those in $\overline{re}$. Also, the auxiliary variables $\overline{x^*}$ are used only by $\boldsymbol{enter}$ and $\boldsymbol{leave}$ and cannot be seen by k.

By the induction hypothesis, if $\langle k, G_1 \rangle \rightarrow G'_1$, we have that a state graph G'_1 is well-typed w.r.t. $\Gamma \bullet \Delta_1$. It implies that $pop(G'_1)$ is well-typed w.r.t. $\Gamma \bullet \Delta$, hence, we can perform the spo on $\overline{re}$, including the case that $\overline{re}$ contain variables, to retain the original edges $\overline{d}$ for $\overline{re}$. According to (T-Var), we maintain $\Gamma, \Delta_1 \vdash \overline{x : t''}$. Thus $\overline{n = eval(G'_1, x)}$ exist and $\Gamma \vdash \overline{\Gamma(G'_1, n) \preccurlyeq t''}$. Therefore, the $\boldsymbol{leave}$ command swings $\overline{d}$ to $\overline{n}$ which are of subtypes of $\overline{t'}$, for $\Gamma \vdash t'' \preccurlyeq t'$ being a premise of (T-Invk). Then it pops the graph. The resulted state graph $G_\diamond$ differs from $pop(G'_1)$ only in the update of $\overline{d}$, thus is still well-typed w.r.t. $\Gamma \bullet \Delta$.

If $\langle k, G_1 \rangle \rightarrow \langle k', G'_1 \rangle$, we have $\Gamma, \Delta'_1 \rightarrow \Delta_1 \vdash k'$ and G'_1 well-typed w.r.t. $\Gamma \bullet \Delta'_1$. In order to deduce $\Gamma, \Delta'_1 \rightarrow \Delta \vdash (k'; \boldsymbol{leave}(\overline{x}, \overline{re}))$, we only need to setup an auxiliary typing rule $\Gamma, \Delta \rightarrow pop(\Delta) \vdash \boldsymbol{leave}(\overline{x}, \overline{re})$. This is possible, since we have shown that a $\boldsymbol{leave}$ command in a method invocation expansion is always type-safe, and $\boldsymbol{leave}$ pops a state graph, meaning that the type context also needs to be popped. □

Based on Lemma 13, Theorem 2 and 3, we can prove the type-safety of programs.

Theorem 4 (Type-safety of programs) Let $Md = \{\overline{\lceil t \rfloor\, x}; c\}$ be a main method, $P = \overline{md} \bullet Md$ a program, Γ a type graph, R an environment graph derived from Γ, $\Delta_1 = push(\Delta_\emptyset, \overline{x}, \overline{t})$, and G_1 a well-typed state graph w.r.t. $\Gamma \bullet \Delta_1$. Unless one of the exceptional cases happens, if $\Gamma \vdash P$, either

1. there exists a well-typed state graph $G_\diamond$ w.r.t. $\Gamma \bullet \Delta_1$ such that $\langle c, G_1 \rangle \rightarrow G_\diamond$, or
2. there exists a configuration $\langle c', G' \rangle$ and a type context Δ' with G' being a well-typed state graph w.r.t. $\Gamma \bullet \Delta'$ such that $\langle c, G_1 \rangle \rightarrow \langle c', G' \rangle$ and $\Gamma, \Delta' \rightarrow \Delta_1 \vdash c'$.

Proof We only need to show that the well-typed program P

implies that all methods in the environment graph R are well-typed, and then apply Theorem 3 to the body command c of the main method Md.

This is true by Theorem 2 that an instantiation of a well-typed command is also well-typed, and by (R-METH) in Table 10, the body commands of the methods in R are all instantiations of the body commands of $\overline{md}$ defined in P. □

Theorem 3 and 4 state that the execution of a well-typed program either terminates or progresses correctly, given a well-typed initial state graph with external objects of concrete classes, unless one of the two exceptional cases happens. Thus, the graph-based generic type system is sound.

8 Conclusion

We give a formal definition of OO type graphs with generics, and provide algorithms for related graph operations. We establish a type system purely based on the graph model, and prove its soundness with respect to a graph-based operational semantics[2]. A distinct nature of our type system is the representation of advanced type features of OO systems, including parameterized types and conjunction types, by simple edge-labeled graphs. Based on such representation, the whole procedure of type checking at compile-time and program execution at runtime are implemented by elementary graph transformations. This provides a systematic graph-based model to help users to understand and analyze the types and semantics of OO programs.

With this formalization, it is possible to express ideas in and base analyses on the intuitive graph notation without sacrificing rigor. With the graph notation, one can spot the relation between two structures intuitively, while it is not so obvious when using symbolic notations. For example, Fig. 5 shows the graph of type *Tree* which is composed in a functional style type expression by type constructors *Pair* and *List*. If we merge the nodes linked by the ⊳-edge (inheritance), it is clear that this structure is identical to the usual OO style definition of the tree class shown in Fig. 9.

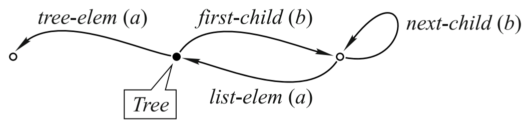


Fig. 9 A graph of the *Tree* class

Formalization is also essential to tool development. Indeed, tool development is one of our motives for this type system. With the support of the advanced type features of OO languages, it is possible to develop tools based on the type system that aim at solving practical problems.

Future work is listed here. The graph algorithms have been used to derive a concrete implementation to parse, type-check and interpret *r*COS/g programs from the source form. This has extended the implementation of the graph-based operational semantics in [6, 12]. Further development can lead to a full-featured tool set to animate the type-checking and interpreting processes of OO programs written in *r*COS/g. Notice that the type graphs and state graphs are likely to be complex for programs with advanced type features. For the tool set to be practically used to show graphs in a human-manageable scale, it is important to be able to render a graph partially, with the hints from the type system and semantics.

Objects that have variable number of attributes, such as arrays, cannot be represented in the current graph model yet, but they are convenient in programming. It is also helpful if we can setup regions in the graphs and reason about the external properties of a region without being concerned with its internals. Besides, simplicity is always one of the key issues of the graph-based model to improve.

Acknowledgements This paper is the result of the research work supported by the project GAVES funded by Macao S&TD Fund, National Natural Science Foundatime of China (Grant Nos. 90718014, 60970031, 91018012, and 60736017).

References

1. Gosling J, Joy B, Steele G, Bracha G. Java Language Specification: The Java Series. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000
2. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, 2007. Object Management Group
3. Hoare C A R, He J. A trace model for pointers and objects. In: Proceedings of the 13th European Conference on Object-Oriented Programming, LNCS 1628. 1999, 1–17
4. Jifeng H, Li X, Liu Z. *r*COS: a refinement calculus of object systems. Theoretical Computer Science, 2006, 365(1): 109–142
5. Klein G, Nipkow T. A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems, 2006, 28(4): 619–695
6. Ke W, Liu Z, Wang S, Zhao L. A graph-based operational semantics of OO programs. Formal Methods and Software Engineering, 2009, 347–366
7. Bracha G. Generics in the Java programming language. Sun Microsystemsm, 2004, 1–23
8. Pierce B. Types and Programming Languages. The MIT Press, 2002
9. Gauthier N, Pottier F. Numbering matters: first-order canonical forms

[2] A type checker and runtime system of *r*COS/g has been implemented in Haskell, following the definitions in this paper. Full source of the implementation can be obtained at "http://a319-101.ipm.edu.mo/~ wke/ggts/impl/".

for second-order recursive types. In: ACM SIGPLAN Notices, 2004, 39(9): 150–161

10. Ke W, Li X, Liu Z, Stolz V. rCOS: a formal model-driven engineering method for component-based software. Frontiers of Computer Science, 2012, 6(1): 17–39
11. Zhao L, Liu X, Liu Z, Qiu Z. Graph transformations for object-oriented refinement. Formal Aspects of Computing, 2009, 21(1): 103–131
12. Ke W, Liu Z, Wang S, Zhao L. Graph-based type system, operational semantics and implementation of an object-oriented programming language. Technical Report 410, UNU-IIST, Macau, China, 2009. www.iist.unu.edu/www/docs/techreports/reports/report410.pdf
13. Abadi M, Cardelli L. A theory of objects. Springer, 1996
14. Igarashi A, Pierce B, Wadler P. Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 2001, 23(3): 396–450
15. Wang S, Long Q, Qiu Z. Type safety for FJ and FGJ. Theoretical Aspects of Computing-ICTAC 2006, 2006, 257–271
16. Rémy D. From classes to objects via subtyping. Programming Languages and Systems, 1998, 200–220
17. Igarashi A, Viroli M. On variance-based subtyping for parametric types. In: Proceedings of the 16th European Conference on Object-Oriented Programming. 2002, 441–469
18. Ferreira A, Foss L, Ribeiro L. Formal verification of object-oriented graph grammars specifications. Electronic Notes in Theoretical Computer Science, 2007, 175(4): 101–114
19. Corradini A, Dotti F, Foss L, Ribeiro L. Translating Java code to graph transformation systems. Graph Transformations, 2004, 171–174
20. Kastenberg H, Kleppe A, Rensink A. Defining object-oriented execution semantics using graph transformations. Formal Methods for Open Object-Based Distributed Systems, 2006, 186–201
21. Heckel R, Küster J, Taentzer G. Confluence of typed attributed graph transformation systems. Graph Transformation, 2002, 161–176
22. Wermelinger M, Fiadeiro J. A graph transformation approach to software architecture reconfiguration. Science of Computer Programming, 2002, 44(2): 133–155
23. Ehrig H, Ehrig K, Prange U, Taentzer G. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. Fundamenta Informaticae, 2006, 74(1): 31–61
24. Rémy D, Yakobowski B. A graphical presentation of ML F types with a linear-time unification algorithm. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. 2007, 27–38
25. Wright A, Felleisen M. A syntactic approach to type soundness. Information and Computation, 1994, 115(1): 38–94
26. Flanagan C, Leino K, Lillibridge M, Nelson G, Saxe J, Stata R. Extended static checking for Java. ACM SIGPLAN Notices, 2002, 37(5): 234–245

Wei Ke is a researcher and lecturer of Macao Polytechnic Institute, China. He received his PhD from School of Computer Science and Engineering, Beihang University, China. His research interests include programming languages, functional programming, formal methods and tool, support for object-oriented and component-based engineering and systems. His recent research focuses on programming tools, environments, and program analysis.

Zhiming Liu is a senior research fellow of UNU-IIST and the head of Information Engineering and Technology in Health Programme (IETH). He was formerly a university lecturer at the University of Leicester and a research Fellow at the University of Warwick. He holds a PhD from the University of Warwick. He is internationally known for his work on the transformational approach to fault-tolerance and real-time computing, and the *r*COS Method of model-driven design of component software. The research of IETH extends and applies these methods to human and environmental health care. Among a number of international conferences he founded, Zhiming Liu is one of founders of the International Symposium on Foundations of Health Information Engineering and Systems (FHIES).

Shuling Wang is an assistant research professor at Institute of Software, Chinese Academy of Scieneces (ISCAS). She received her PhD in 2008 from School of Mathematical Sciences, Peking University, China. In the following years, she was a postdoctoral research fellow at UNU-IIST and ISCAS. Her current research interests are hybrid systems and object-oriented programs, both related to formal modeling, semantics, and verification.

Liang Zhao is currently a postdoctoral fellow of United Nations University-International Institute for Software Technology (UNU-IIST). His research interests include semantics and type systems of programming languages, graph transformations, and formal methods for object-oriented development.