Sequential μJava: Formal Foundations[⋆]

Qiu Zongyan, Wang Shuling and Quan Long

LMAM and Department of Informatics,
School of Mathematical Sciences, Peking University
{qzy,joycy,longquan}@math.pku.edu.cn

Abstract. In recent years, many researchers in the programming language and formal methods communities pay much attentions on various problems related to object-oriented (OO) programs. We develop in this article a theoretical foundation for a core sequential OO language, μJava. The language covers most important object-oriented features including reference types, subtypes, inheritance, dynamic binding, and sharing based parameters for methods. The static environment, type system, and operational semantics are defined to capture the essential features of object-orientation. We have proved *Type Soundness* theorem in the sense that the successfully terminated execution of a well-formed command will transform a state to another state, while both conform to the static environment. The language and the formalization form a foundation for investigating various aspects of OO languages and systems.

keywords: Object Orientation, Types, Semantics

1 Introduction

In both software development and programming languages fields, object-orientation (OO) is and will remain a main-stream technology. The languages supporting OO concepts give a level of abstraction that separates the view of what a software component does from the details of how it does, and a high-level flexibility. OO languages and development methods bring many important benefits for software developing, maintaining, and component reusing. Certain features of object-orientation, say, inheritance, object references, and dynamic binding are essential for these.

There have been extensive formal studies on OO languages and systems. For instances, America and de Boer presented a logic for a parallel language POOL [2], an imperative language with object sharing, but no subtyping and method overriding. Abadi and Leino defined an axiomatic semantics for an imperative OO language with object sharing [1], which lacks dynamic binding of method invocation. Cavalcanti and Naumann adopted value model for variables and attributes in their work [4], thus could not deal with phenomena related to object sharing. He *et al.* proposed a calculus for an OO language [6], which covers most of important features of sequential Java using Hoare and He's Unifying Theories of Programming [7], but no clear division of the static and dynamic features proposed. Furthermore, some authors (i.e., [4, 6]) adopt an Ada-like parameter model for methods, which is not used commonly in the practical OO languages.

⋆ Supported by NNSF of China (No. 60573081).

In this article, we introduce a core language, sequential μJava, as a simplified model of OO languages. It supports reference type, or object sharing, with other important features including subtypes, inheritance, dynamic binding, and sharing parameters for methods. The language is sufficiently close to the sequential part of Java. It is rich enough for covering most of the important OO features in the imperative setting (with state and state transition), thus can be used in meaningful case studies to capture central challenges in modeling OO programs. On the other hand, it is also simple enough for the deep theoretical work.

We define the syntax of the language, and a procedure to build the static environment for programs. We define a type system that can judge the well-formedness of programs statically. To make a complete formal model, an operational semantics for the core language is given. For defining the semantics, we use a storage model composed of a pair of a store and an *Object Pool*, Opool for short. The basic unit in Opool is the *reference-attribute-reference* tuple, while the references can be shared by not only variables in programs but also attributes of *objects*. In our model, every entity is an object and every variable (and every attribute in an object) is a reference. So what we defined is a language with a pure object model, and a pure reference semantics rather than the value semantics as studied in many other articles, e.g. [12, 4], etc. This makes the study more useful for the further understanding on practical OO languages and programs.

We prove that our type system for μJava is sound, in the sense that if the execution of any command from a state conforming to the static environment terminates successfully, then it will reach a state conforming to the static environment. Type soundness for object-oriented languages has been proved in other literatures, such as [5, 9, 4] and so on. However, most of the works deal with a purely functional core of object-oriented languages, or with a value semantics, or with a very limited language.

Other contributions of this work are as follows: We handle the dynamic binding issue by fixing the code of a certain method to its class. According to our knowledge, the existing work on the formal semantics of non-trivial OO languages either did not handle it [1], or handled it by looking up the method body during execution [4, 6]. Furthermore, different from many existing work, for instances [1, 4, 6], we have constructor here which makes the language closer to the real languages. Thus many OO concepts and techniques could be formalized more clearly.

The rest of this paper is organized as follows. We define the storage model used here in Section 2, then the core language, sequential μJava in Section 3. The building of the static environments and typing rules are given in Section 4. In Section 5, we define the operational semantics for μJava, and prove the conformance properties between static rules/environments and the semantics in Section 6. We give an example in Section 7 as an illustration. In the last section, the article is concluded and some future research directions are discussed.

2 The Storage Model

For supporting important OO features, most practical OO languages adopts the reference model, in which the value of a variable is a reference to an object in the object pool, and the value of every attribute of an object is also a reference to some object[1]. A special case is that these references can be null to mean referring to no object. In this storage model, there is more possibility of sharing: In addition to that different variables can share the same reference in the object pool, different attributes of objects can share the same reference, and can also have sharing with variables too.

Additionally, we need to find the type of an object in the execution for supporting various activities based on Run-Time-Type-Identification, e.g., type-casting.

We define that a state of the OO programs includes a store and an object pool. For the definition of the storage model, we have two basic sets:

- Name: An infinite set of names, which are used as the variable names or the attribute names in programs, where there is a special name $\mathsf{type} \notin \mathrm{Name}$.
- Ref: An infinite set of references which are used as the addresses of objects. There is a special and distinguishable reference rnull which never refers to any object.

We define two sets (sets of mappings) as follows:

$$\begin{aligned}\text{Store} &\mathrel{\widehat{=}} \mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref}\\ \text{Opool} &\mathrel{\widehat{=}} \mathrm{Ref} \rightharpoonup_{\mathsf{fin}} ((\mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref}) \cup (\{\mathsf{type}\} \to \mathrm{Name}))\end{aligned}$$

We define $\mathrm{Name}^+ \mathrel{\widehat{=}} \mathrm{Name} \cup \{\mathsf{type}\}$.

We will use meta-variables σ and O, possibly with subscript, to denote elements of Store or Opool (object pool), respectively. A store σ maps (variable) names to the references they record, and an Opool O maps references to functions from (attribute) names to references. We assume a special name **null** $\in \mathrm{Name}$, with σ**null** $= \mathsf{rnull}$ holds for every σ.

An element of O is a triple $\langle r, n, r' \rangle$, where r is a reference to some object o, $n \in \mathrm{Name}^+$ is the name of an attribute of o or type, and r' is the reference recorded in attribute n of o, or a name of a type (class) when n is type. We will sometimes use o, possibly with subscripts, to indicate an object.

For any $r \in \mathsf{dom}\, O$, $O(r)$ is a finite function which can be represented as a finite set of pairs:

$$O(r) = \{\langle n, r_1 \rangle \,|\, \langle r, n, r_1 \rangle \in O\}$$

where each pair associates an attribute name and a reference, or type with a type name. In fact, the set (of pairs) $O(r)$ represents the object referred by r. For benefiting the discussion, we introduced a notation $\{n_1 \triangleright r_1, \ldots, n_k \triangleright r_k\}$ to represent the object with attributes $n_1, \ldots, n_k$ and corresponding values $r_1, \ldots, r_k$ respectively. We define two projects:

$$\begin{aligned}O(r)_1 &\mathrel{\widehat{=}} \{n \,|\, \langle n, r' \rangle \in O(r) \text{ for some } r'\}\\ O(r)_2 &\mathrel{\widehat{=}} \{r' \,|\, \langle n, r' \rangle \in O(r) \text{ for some } n\}\end{aligned}$$

where $O(r)_1$ is the set of the first element of the pairs in $O(r)$, and $O(r)_2$ is the set of the second elements. In our interpretation, $O(r)_1$ is the set of attribute names of object $O(r)$, and $O(r)_2$ the corresponding references (their "values").

[1] One exception might be the variable or attribute of primitive type. Many OO languages use the value model for them under the consideration of efficiency.

For the relation between $\mathrm{Ref} \rightharpoonup (\mathrm{Name} \rightharpoonup \mathrm{Ref})$ and $\mathrm{Ref} \times \mathrm{Name} \rightharpoonup \mathrm{Ref}$, when we say the domain of O, we might sometimes want to mean a subset of Ref which is associated with a set of objects, or sometimes a subset of $\mathrm{Ref} \times \mathrm{Name}$ associated with a set of values (references). We will use $\mathsf{dom}\, O$ for the first case, and $\mathsf{dom}_2\, O$ especially for the second, where an element in $\mathsf{dom}_2\, O$ is a pair of the form $\langle r, a \rangle$.

We assume that from an object in O, its type (its class) can be determine directly in the run-time. For this becoming possible, we can take the way used in practice, to include a special attribute in each object to record its type. Because this feature does not have critical effect on our discussion, we will omit the treatment, and simply assume a function $\mathit{type}(r)$ to get the type of the object referred by r.

The store is modified by assignments to variables. Assume the current store is σ. Assignment $x := y$ will make a new store σ' with all values of variables the same as σ, except that $\sigma' x = \sigma y$. Assignments to attributes change the Opool. If x refers to an object o in current Opool O, i.e., $O(\sigma x) = o$. After the assignment $x.a := y$, we will have $o(a) = \sigma y$, that is, $O'(\sigma x)(a) = \sigma y$, where O' is the updated Opool.

Most commands keep the domain of O unchanged, except object creations. If we want to create an object o and let variable x refer to it, the system will take a fresh reference r, and let $O' = O \oplus \{r \mapsto o\}$ and $\sigma' x = r$. Here we use $\oplus$ to denote the standard set overriding. We do not have release operation, and assume a garbage collector to take case the unreachable object and reduce the Opool. Formal treatment of garbage collection is also an interesting topic. In [8], the authors defined a semantics for an object-oriented language that models garbage collection explicitly.

To state the properties locally, we borrow some notations from Separation Logic, and use $O_1 \perp O_2$ to indicate that two Opools O_1 and O_2 have disjoint domains, i.e.,

$$O_1 \perp O_2 \overset{\text{def}}{=} \mathsf{dom}_2\, O_1 \cap \mathsf{dom}_2\, O_2 = \emptyset$$

We use $O_1 * O_2$ to indicate the union of O_1 and O_2 when $O_1 \perp O_2$.

A state is a pair of a store and an Opool, i.e., of the form (σ, O), thus we have

$$\text{State} = \{(\sigma, O) \mid \sigma \in \text{Store} \wedge O \in \text{Opool}\}$$

3 Syntax of the Sequential μJava

The language investigated in this article, μJava, can be seen as a sequential subset of Java. What we consider mainly is the essential OO features relating to object sharing, updating, and creation. The syntax of expressions and commands is as follows:

$$\begin{aligned}
v &::= \textbf{this} \mid x \\
e &::= \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid v \\
b &::= \textbf{true} \mid \textbf{false} \mid e = e \mid \neg b \mid b \wedge b \mid b \vee b \\
c &::= \textbf{skip} \mid x := e \mid v.a := x \mid x := v.a \mid x := (C)v \\
&\quad \mid x := v.m(\overline{e}) \mid x := \textbf{new}\, C(\overline{e}) \mid \textbf{return}\, e \\
&\quad \mid c; c \mid \textbf{if}\, b\ c\, \textbf{else}\, c \mid \textbf{while}\, b\, \textbf{do}\, c
\end{aligned}$$

Here x denotes a variable name, C denotes a class name, while a and m an attribute name and a method name respectively.

Here are some explanations about the language:

– Similar to [1], to keep the semantics simple and clear, we have boolean the only primitive type, while **true** and **false** the only primitive values. For making a pure reference model, we assume two implicit objects representing the truth-values with references rtrue and rfalse respectively, and σ**true** $=$ rtrue and σ**false** $=$ rfalse for any store σ. Other primitive types can be added without substantial difficulties.
– We adopt the restricted forms of expressions, with values depending only on the store. The expressions do not touch inside the Opool. The more complex expressions can be encoded with the help of assignments and auxiliary variables. For instance, one can let $x = y.a$ and then refer to $x.a'$ as a replacement of $y.a.a'$. The form of assignments is also restricted to a number of special forms, that are also enough to encode other general cases. These simplifications are not substantial.
– We consider the cast as a part of commands rather than expressions. This is also a non-essential restriction. And further, we can give the semantics of cast related assignments in the operational rules.
– Command of the form $x := \textbf{new}\, C(\overline{e})$ creates a new object initiated with parameter $\overline{e}$ and assigns its reference to x.

Similar to Java, we consider $main()$ (if existing) a special method where the program starts. So we do not have special type **program** (as what introduced in [4]) in typing. The syntax for classes and programs is as follows.

$$\begin{array}{ll}
T & ::= \textbf{Bool} \,|\, \textbf{Object} \,|\, C \\
md & ::= T\ m(\overline{T\ x})\{\overline{T\ x};\ c\} \\
cd & ::= \textbf{class}\, C : C\{\overline{T\ a}; C(\overline{T\ x})\{\overline{\textbf{this}.a := x}\}; \overline{md}\} \\
cds & ::= cd \,|\, cd; cds \\
prog & ::= cds
\end{array}$$

where m denotes a method name, C denotes a class name. We don't include complex access control mechanism in the language, because this mechanism is not essential, and it is not hard to modify the framework to include it.

A program is a sequence of class declarations where each offers a group of methods as services to other code. Here are some explanations about the definition:

– We assume a pre-defined class **Object** as the super class of all user-defined classes. Thus, each user-defined class has a direct super class. The (only) primitive type **Bool** have two literals **true** and **false**. It is not a supertype or subtype of any other types. We assume an internal type **Null** as the type of **null**, which is the subtype of any classes. **Null** is not a feature of the language, thus cannot be used in programs. It is used only in the definitions for the type system and the semantics.
– A special method $C(\overline{T\ x})\{\overline{\textbf{this}.a := x}\}$ in each class C services as the constructor, which must have the same name with the class. It takes exactly as many parameters as its attributes (including inherited ones), and its body is a sequence of assignments from the parameters to all the attributes. To extend the language by taking a more general form for constructors will introduce no substantial difficulties, but only some redundancy into the formalization, we will not do it here.
– We assume that all references to attributes in the methods are decorated with **this**, to make the attribute references uniformly with the form $v.a$. We can get rid of this

restriction without any real problem, but thus, some more rules and more conditions of the rules should be added.

– We do not have structures for visibility rules in our model, because these rules are vary from language to language. We use predicate $visible(C, m, C'.a)$ to mean that attribute a of class C' is visible in the body of method m in class C. The visibility cannot be determined locally in C. It depends on other class declarations of the program. Anyway, this relation can be determined statically in any practical language. We use $visible$ to hide language details in this formal study.
– We will use sometimes $\overline{para}$, $\overline{local}$ to denote the parameter and local variable declarations of a method, respectively, when the details are not important. We use function $dtype$ to get the declared type of a parameter or a local variable, and promote it to the parameter and variable lists. We use function FV to extract the set of free variables from a piece of code, that is easy to defined recursively.
– There can be at most one class containing a method named $main$ in a program as the start point of execution. The class with $main$ method should not have any attribute or other method, and no constructor. The $main$ method should have no parameter and return **Bool** to report the finishing status. A program without main serves as a class library. It has no independent execution, but all other aspects of our definitions are applicable to it.
– We do not permit redefinition of attributes with the same name in subclasses. We permit method overriding but *not* method overloading in class declarations. This problem can be solved with a more sophisticated static analysis and an enhanced static environment, which have not many thing to do with our focus here.

4 Static Environments and Typing

The authors of [4] developed a static environment to localize and simplify the definition of semantics. What we do here is similar. Our static environment consists of two components: Γ and Θ, where Γ stands for the typing environment, recording information about the structure of the class declarations, and Θ is a method lookup environment. These environments are established by scanning the program before execution.

4.1 Typing Environment Γ

The typing environment Γ_{cds} records the static structural information of class declarations cds with which the types of expressions or predicates are derived, and the well-formedness of commands can be checked. We will abbreviate it as Γ when there is no confusion. Formally, Γ_{cds} is a tuple of the form:

$$\langle \mathsf{cname}, \mathsf{super}, \mathsf{method}, \mathsf{attr}, \mathsf{locvar} \rangle$$

where all the elements are relations over the classes, methods, and attributes, except for cname which is a simple set:

– cname: The set of all the class names appearing in the class declarations cds, with the predefined types **Object**, **Null** and **Bool**.

– super: A relation mapping a class to its immediate superclass, thus $\mathsf{super}(C_1, C_2)$ means C_2 is the immediate superclass of C_1.
– method: A relation connecting each class to its method signatures, where $\mathsf{method}(C, m(\overline{para}) : T)$ stands for the fact that $m(\overline{para}) : T$ is a method with this signature declared in C or inherited from the superclass of C. We include also all the constructors in method, which takes a special form $C(\overline{para})$ without the return type.
– attr: A relation from classes to their attributes and the corresponding type, where $\mathsf{attr}(C, a : T)$ means that the attribute named a of type T is defined in C. We will use $\mathit{\Gamma}.\mathsf{attr}(C)$ to denote the set of all attributes of class C.
– locvar: A relation over classes, methods and variables. $\mathsf{locvar}(C, m, x : T)$ means that x of type T is a parameter or local variable in method m of class C, thus is visible in the body of m.

4.2 Construction of $\mathit{\Gamma}$

We present the rules for the construction of $\mathit{\Gamma}$ in this subsection. Because $\mathit{\Gamma}$ is a tuple with components cname, super, method, attr, and locvar, its construction is the construction of each of these components on the way to scan the program text.

Initially we have $\mathit{\Gamma}_0 = \langle\{\textbf{Object}, \textbf{Bool}, \textbf{Null}\}, \emptyset, \emptyset, \emptyset, \emptyset\rangle$. Following the scanning, $\mathit{\Gamma}$ is built step-wisely. The construction rules are defined below, where the primed $\mathit{\Gamma}'$ is used to denote the typing environment after the modification. When a component of $\mathit{\Gamma}$ is not modified in a step, we well not show it in the rule.

Class name and super class:

$$\frac{\textbf{class}\, C_1 : C_2\{...\} \quad C_2 \in \mathit{\Gamma}.\mathsf{cname}}{\mathit{\Gamma}'.\mathsf{cname} = \mathit{\Gamma}.\mathsf{cname} \cup \{C_1\} \qquad \mathit{\Gamma}'.\mathsf{super} = \mathit{\Gamma}.\mathsf{super} \cup \{(C_1, C_2)\}}$$

Attribute:

$$\frac{\textbf{class}\, C_1 : C_2\{...T\ a...\}}{\mathit{\Gamma}'.\mathsf{attr} = \mathit{\Gamma}.\mathsf{attr} \cup \{(C_1, a : T)\}}$$

Attribute Inheritance:

$$\frac{\textbf{class}\, C_1 : C_2\{...\} \quad \{(C_2, a : T)\} \in \mathit{\Gamma}.\mathsf{attr}}{\mathit{\Gamma}'.\mathsf{attr} = \mathit{\Gamma}.\mathsf{attr} \cup \{(C_1, a : T)\}}$$

Constructor:

$$\frac{\textbf{class}\, C_1 : C_2\{...C_1(\overline{para})...\}}{\mathit{\Gamma}'.\mathsf{method} = \mathit{\Gamma}.\mathsf{method} \cup \{(C_1, C_1(\overline{para}))\}}$$

Method:

$$\frac{\textbf{class}\, C_1 : C_2\{...T\ m(\overline{para})...\}}{\mathit{\Gamma}'.\mathsf{method} = \mathit{\Gamma}.\mathsf{method} \cup \{(C_1, m(\overline{para}) : T)\}}$$

Method Inheritance:

$$\frac{\textbf{class}\, C_1 : C_2\{...\} \quad (C_2, m(\overline{para}) : T) \in \mathit{\Gamma}.\mathsf{method}}{\mathit{\Gamma}'.\mathsf{method} = \mathit{\Gamma}.\mathsf{method} \cup \{(C_1, m(\overline{para}) : T)\}}$$

Parameter and Local Variable:

$$\frac{\textbf{class}\,C\ldots\{\ldots C(\overline{T\ x})\{\ldots\}\ldots\}}{\Gamma'.\mathsf{locvar} = \Gamma.\mathsf{locvar} \cup \mathit{local}(C, C, \{\overline{(T, x)}\}) \cup \{(C, C, \textbf{this} : C)\}}$$

$$\frac{\textbf{class}\,C\ldots\{\ldots T\ m(\overline{T_1\ x})\{\overline{T_2\ y};\ldots\}\ldots\}}{\begin{array}{l}\Gamma'.\mathsf{locvar} = \Gamma.\mathsf{locvar} \cup \mathit{local}(C, m, \{\overline{(T_1\ x)}\}) \cup \mathit{local}(C, m, \{\overline{(T_2\ y)}\}) \\ \qquad \cup \{(C, m, \textbf{this} : C), (C, m, \textbf{res} : T)\}\end{array}}$$

where $\mathit{local}(C, m, S) \overset{\text{def}}{=} \{(C, m, x : T) \,|(T\ x) \in S\}$, which expands the definitions to a set of the correct form. The local variable **this** is introduced with the type C, and **res** is an internal variable with correct type for recording the return value of the method.

Please notice that we do not permit method overload, hence there is no problem of processing order of rule "Method" and "Method Inheritance", because if both of them exist, their signature will be the same.

We define an extended subclass relation $\Gamma \vdash T_1 \preceq T_2$ as the transitive closure of relation super under environment Γ:

$$\frac{C \in \Gamma.\mathsf{cname}}{\Gamma \vdash C \preceq C} \quad \frac{\Gamma.\mathsf{super}(C_1, D) \quad \Gamma \vdash D \preceq C_2}{\Gamma \vdash C_1 \preceq C_2} \quad \frac{C \in \Gamma.\mathsf{cname} \quad C \neq \textbf{Bool}}{\Gamma \vdash \textbf{Null} \preceq C}$$

Please note that, the only relation with boolean type is **Bool** $\preceq$ **Bool**. In the following, we will often omit Γ and write $T_1 \preceq T_2$ directly when it makes no confusion.

4.3 Type Judgement of Expressions

We use typing statement $\Gamma, C, m \vdash e : T$ to mean that expression e is of the type T in method m of class C under the typing environment Γ. Typing rules for expressions are given below. If there is no premise, we will omit the line above the consequence:

true *and* **false***:*

$$\Gamma, C, m \vdash \textbf{true} : \textbf{Bool} \qquad \Gamma, C, m \vdash \textbf{false} : \textbf{Bool}$$

this *and* **null***:*

$$\Gamma, C, m \vdash \textbf{this} : C \qquad \Gamma, C, m \vdash \textbf{null} : \textbf{Null}$$

Parameter or Local Variable x in a Method:

$$\frac{\Gamma.\mathsf{locvar}(C, m, x : T)}{\Gamma, C, m \vdash x : T}$$

Boolean:

$$\frac{\Gamma, C, m \vdash e_1 : C_1 \quad \Gamma, C, m \vdash e_2 : C_2 \quad C_1 \preceq C_2 \vee C_2 \preceq C_1}{\Gamma, C, m \vdash e_1 = e_2 : \textbf{Bool}}$$

$$\frac{\Gamma, C, m \vdash b : \textbf{Bool}}{\Gamma, C, m \vdash \neg b : \textbf{Bool}}$$

Premise $C_1 \preceq C_2$ or $C_2 \preceq C_1$ covers the case when both of these types are **Bool**. The rules for $b_1 \wedge b_2$, $b_1 \vee b_2$ are similar to the rule for negation, and omitted here.

4.4 Well-Formedness of Commands

We use statements of the form $\Gamma, C, m \vdash c : \textbf{com}$ to mean that command c is well-formed in the scope of m in C under typing environment Γ.

Skip: A **skip** is always well-formed.

$$\Gamma, C, m \vdash \textbf{skip} : \textbf{com}$$

Assignment: A simple assignment is well-formed if its variable and expression are well-typed, and the right hand side is a subtype of the left hand side.

$$\frac{\Gamma, C, m \vdash x : C_1 \quad \Gamma, C, m \vdash e : C_2 \quad C_2 \preceq C_1}{\Gamma, C, m \vdash x := e : \textbf{com}}$$

The attribute mutation and looking up are similar to the simple assignments, thus their typing rules are similar, except with more complicated premises:

$$\frac{\begin{array}{c}\Gamma, C, m \vdash v : N \quad (N, a : T) \in \Gamma.\mathsf{attr}\\ \mathit{visible}(C, m, N.a) \quad \Gamma, C, m \vdash x : C_2 \quad C_2 \preceq T\end{array}}{\Gamma, C, m \vdash v.a := x : \textbf{com}}$$

$$\frac{\begin{array}{c}\Gamma, C, m \vdash x : C_1 \quad \Gamma, C, m \vdash v : N\\ (N, a : T) \in \Gamma.\mathsf{attr} \quad \mathit{visible}(C, m, N.a) \quad T \preceq C_1\end{array}}{\Gamma, C, m \vdash x := v.a : \textbf{com}}$$

Assignment by Cast: An assignment by cast is well-formed if (1) The casing type is a subtype of the type of the assigned variable, and (2) The type of the casted variable is a subtype or supertype of the casting type.

$$\frac{\begin{array}{c}\Gamma, C, m \vdash x : C_1 \quad \Gamma, C, m \vdash v : C_2\\ N \in \Gamma.\mathsf{cname} \quad N \preceq C_1 \quad N \preceq C_2 \vee C_2 \preceq N\end{array}}{\Gamma, C, m \vdash x := (N)v : \textbf{com}}$$

Please note that, because our definition of relation $C_1 \preceq C_2$, all the rules for assignments are applicable to the Boolean cases. The same is true for the rules about method invocations and object creations.

Method Invocation: A method invocation is well-formed if the corresponding method does exist, and the actual parameters are of the subtypes of formal parameters, and the return type of the method is subtype of the type of the assigned variable.

$$\frac{\begin{array}{c}\Gamma, N, m_0 \vdash v : C \quad (C, m(\overline{para}) : T) \in \Gamma.\mathsf{method}\\ \Gamma, N, m_0 \vdash \overline{e} : \overline{D} \quad \overline{D} \preceq \mathit{dtype}(\overline{para}) \quad \Gamma, N, m_0 \vdash x : T_1 \quad T \preceq T_1\end{array}}{\Gamma, N, m_0 \vdash x := v.m(\overline{e}) : \textbf{com}}$$

The premises imply that the invocation should be consistent with definition of the corresponding method in both number and types of the parameters.

Object Creation: An object creation is well-formed if the actual parameters are well-typed, and their types are subtypes of the formal parameters of the constructor, and the variable assigned has the same type as the created object.

$$\frac{C \in \Gamma.\mathsf{cname} \quad (C, C(\overline{para})) \in \Gamma.\mathsf{method} \qquad \Gamma, N, m \vdash x : C \quad \Gamma, N, m \vdash \overline{e} : \overline{D} \quad \overline{D} \preceq \mathit{dtype}(\overline{para})}{\Gamma, N, m \vdash x := \textbf{new}\, C(\overline{e}) : \textbf{com}}$$

Return: The type of the expression in a return statement must be compatible with the return type of the method:

$$\frac{(C, m(\overline{para}) : T) \in \Gamma.\mathsf{method} \quad \Gamma, C, m \vdash e : T_1 \quad T_1 \preceq T}{\Gamma, C, m \vdash \textbf{return}\, e : \textbf{com}}$$

Sequential Composition: A sequential composition is well-formed if all its components are well-formed.

$$\frac{\Gamma, C, m \vdash c_i : \textbf{com} \ \ (i = 1, 2)}{\Gamma, C, m \vdash c_1; c_2 : \textbf{com}}$$

Choice:

$$\frac{\Gamma, C, m \vdash c_i : \textbf{com} \ \ (i = 1, 2) \qquad \Gamma, C, m \vdash b : \textbf{Bool}}{\Gamma, C, m \vdash \textbf{if}\, b\ c_1\ \textbf{else}\, c_2 : \textbf{com}}$$

While Loop:

$$\frac{\Gamma, C, m \vdash c : \textbf{com} \qquad \Gamma, C, m \vdash b : \textbf{Bool}}{\Gamma, C, m \vdash \textbf{while}\, b\ \textbf{do}\, c : \textbf{com}}$$

Now we define the well-formedness for the higher level structures, including constructors, method declarations and class declarations. A method declaration is well-formed with respect to Γ and its class C, if all types for its parameters, local variables and return are defined in Γ, and its body is well-formed. The body of a constructor must be a sequence of well-formed assignments from parameters to all the attributes (including inherited ones). The class declaration is well-formed in Γ if all types occurring in it are defined in Γ, and, the constructor and all methods declared in it are well-formed.

Method Declaration:

$$\frac{T, \overline{T_1}, \overline{T_2} \in \Gamma.\mathsf{cname} \quad \Gamma, C, m \vdash c : \textbf{com} \quad \overline{x}, \overline{y} \text{ are diff. names}}{\Gamma, C \vdash T\ m(\overline{T_1\ x})\{\overline{T_2\ y}; c\} : \textbf{ok}}$$

Constructor:

$$\frac{\overline{T_2} \in \Gamma.\mathsf{cname} \quad \Gamma.\mathsf{attr}(C) = \{\overline{a : T}\} \quad \overline{T_2} \preceq \overline{T} \quad \overline{x} \text{ are diff. names}}{\Gamma, C \vdash C(\overline{T_2\ x})\{\overline{\textbf{this}.a := x}\} : \textbf{ok}}$$

Method Declaration Sequence:

$$\frac{m_1, \ldots, m_k \text{ are diff. names} \quad \Gamma, C \vdash T_i\ m_i(\overline{T_{i1}\ x_i})\{\overline{T_{i2}\ y_i}; c_i\} : \textbf{ok} \quad i \in 1..k}{\Gamma, C \vdash T_1\ m_1(\overline{T_{11}\ x_1})\{\overline{T_{12}\ y_1}; c_1\}\ \ldots\ T_k\ m_k(\overline{T_{k1}\ x_k})\{\overline{T_{k2}\ y_k}; c_k\} : \textbf{ok}}$$

Class Declaration:

$$\frac{\begin{array}{c} C, D, \overline{T_1} \in \mathit{\Gamma}.\mathsf{cname} \quad \overline{a} \text{ are diff. names} \\ \mathit{\Gamma}, C \vdash \overline{md} : \textbf{ok} \quad \mathit{\Gamma}, C \vdash C(\overline{T_2\ x})\{\overline{\textbf{this}.a := x}\} : \textbf{ok} \end{array}}{\mathit{\Gamma} \vdash \textbf{class}\, C : D\{\overline{T_1\ a}; C(\overline{T_2\ x})\{\overline{\textbf{this}.a := x}\}; \overline{md}\} : \textbf{ok}}$$

Based on these definitions, we can also define the well-formedness of a class declaration sequence, and thus of a program. It is routine and omitted here.

4.5 Method Body Lookup

Now we introduce our approach of method body lookup. The system will fix the method body code to the corresponding class before the execution of the main method.

We use $\mathit{\Theta}$ to denote the environment for method body lookup, which is composed of triples with the form $(C, m, \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\})$, where C is a class name in cname, m is a method name of C, and c is the body of m, which is a well-formed command. The informal meaning of $(C, m, \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\})$ is that the invocation of m from an object of type C is equivalent to the execution of c after initializing the parameters $\overline{x}$ by the real arguments, with the local variables $\overline{y}$. Because we deal with only the dynamic behavior of well-typed programs without the consideration of the overriding, we do not need to record the type information in the method lookup environment.

For intuitional concern, we use $\mathit{\Theta}, C \vdash m \twoheadrightarrow \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\}$ to represent the fact that $(C, m, \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\}) \in \mathit{\Theta}$. In fact, our approach mirrors the *vtable* technique used in the OO practice, that is, each class copies all the inherited method bodies from its direct superclass (either a reference or the code. For simplicity of the discussion, we copy the code here). This idea is embodied by the following two rules:

The method is defined immediately[2]:

$$\frac{\textbf{class}\ C : N\ \{\ldots m(\overline{T\ x})\{\overline{T\ y}; c\}\ldots\} \qquad \mathit{\Gamma}, C, m \vdash c : \textbf{com}}{\mathit{\Theta}, C \vdash m \twoheadrightarrow \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\}}$$

Copy from immediate superclass:

$$\frac{\textbf{class}\ C : N\ \{...\} \qquad \textit{undefined}(\mathit{\Theta}, C, m) \quad \mathit{\Theta}, N \vdash m \twoheadrightarrow \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\}}{\mathit{\Theta}, C \vdash m \twoheadrightarrow \lambda(\overline{x})\{\textbf{var}\,\overline{y}; c\}}$$

where predicate *undefined*$(\mathit{\Theta}, C, m)$ expresses that there is not a method named m in C in current state of $\mathit{\Theta}$.

One might argue that, to ensure that a particular super class is done before its subclasses, a linear order of classes for applying above rules should be given. The answer is "yes". Commonly, the written order of the class declarations satisfies this requirement, because a subclass can be defined only when its direct superclass is already there. Furthermore, we can also consider the typing as a fixed point calculating procedure, to relax the forced order on the class declarations.

[2] In fact, we need another law for constructors. But it is almost the same as this. Thus, we consider this law covers the cases for constructors. The same assumption will be adopted in the rest of this paper.

4.6 Properties

In this subsection, we show that an expression or a command typing does determine a derivation-uniqueness of typing derivation.

Theorem 1. *For a typing of the form $\Gamma, C, (m) \vdash e : T$ or $\Gamma, C, (m) \vdash b :$* **Bool***, there is at most one derivation.*

Proof. By induction on the structure of the expressions. Instead we can prove that, for each syntactic construct there is exactly one typing rule.

– *Case* **this**, **null**, **true**, **false**. The proof is trivial.
– *Case* v. We know that v can only be **this** or a local variable, or a parameter of some method m of a class C. If we have $\Gamma, C, m \vdash v : T$, then it must be derived by the exact rule for local variable or parameter in a method, or from the rule for **this**, thus, we can get the conclusion.
– *Case* e. Immediately from all of the cases of e.
– *Case* b. By induction on the structures of b. □

Theorem 2. *For each typing $\Gamma, C, m \vdash c :$* **com***, there is at most one derivation.*

Proof. By **Theorem** 1, and induction on typing derivations for commands, the result is easy to prove. □

5 Operational Semantics

The operational semantics of μJava programs is defined as a mapping from configurations to configurations. We consider here only the well-formed programs, for getting rid of many well-formedness conditions which are guaranteed by rules in Section 4. A configuration is either a tuple $\langle c, s\rangle$ consisting of a command and a state, or a terminated state $s = (\sigma, O)$. The semantics is defined as a transition relation $\leadsto$:

$$\begin{aligned} &\text{Configuration} \mathrel{\widehat{=}} \langle \text{Command}, \text{State}\rangle \cup \text{State} \\ &\leadsto \qquad\qquad\quad \mathrel{\widehat{=}} \text{Configuration} \rightharpoonup_{\mathsf{fin}} \text{Configuration} \cup \{\mathsf{abort}\} \end{aligned}$$

Here Command is the set of program texts. A configuration consisting of only a state is a terminal state, which represents that the execution of a (piece of) program has completed successfully, while abort represents that the program goes wrong in execution, because of memory faults, wrong type casts, and so on. We define $\leadsto^*$ as a finite transition closure of $\leadsto$. The semantics for commands in μJava is defined by the following inference rules, with the help of static environments Γ and Θ.

5.1 Semantic Rules

Now we give the semantics by a set of deduction rules. We omit here the rules for the evaluation of expressions, because they are routine, and the definitions will not have any effect on our discussion below. The rule for **skip** is trivial which keeps the configuration unchanged. We do not list it here either.

The first group includes rules for various assignments. The plain form $x := e$ is independent of Opool. It updates x in σ to current value of e. Both *Update* and *Lookup* operations look into the Opool. They will go abort when they dereference an attribute of an object out of the Opool, that includes the cases where v or x has a rnull value.

Assignment:

$$\frac{}{\langle x := e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\}, O)}$$

Mutation:

$$\frac{\langle \sigma v, a\rangle \in \mathsf{dom}_2\, O}{\langle v.a := x, (\sigma, O)\rangle \rightsquigarrow (\sigma, O \oplus \{\langle \sigma v, a\rangle \mapsto \sigma x\})}$$

$$\frac{\langle \sigma v, a\rangle \notin \mathsf{dom}_2\, O}{\langle v.a := x, (\sigma, O)\rangle \rightsquigarrow \mathsf{abort}}$$

Lookup:

$$\frac{\langle \sigma v, a\rangle \in \mathsf{dom}_2\, O}{\langle x := v.a, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto O(\sigma v)(a)\}, O)}$$

$$\frac{\langle \sigma v, a\rangle \notin \mathsf{dom}_2\, O}{\langle x := v.a, (\sigma, O)\rangle \rightsquigarrow \mathsf{abort}}$$

Assignment by Cast: The assignment by cast $x := (N)v$ needs to check whether $(N)v$ is a correct cast in the run-time, to ensure the type of the object referred by v to be a subclass to N. If this condition does not hold, the execution fails, which reveals a wrong downcast. The upcast in the well-typed program is always allowed by the rule.

$$\frac{\mathit{type}(\sigma v) \preceq N}{\langle x := (N)v, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma v\}, O)}$$

$$\frac{\mathit{type}(\sigma v) \not\preceq N}{\langle x := (N)v, (\sigma, O)\rangle \rightsquigarrow \mathsf{abort}}$$

Method Invocation: The rule for method invocation captures the dynamic binding feature. As seen, the method body to execute is determined by the type of the object referred by v in the run time. Because the method lookup environment Θ corresponding to the whole program is created statically, it is always usable.

In the rule, the body command c of method m executes from a new store created locally, with all parameters initialized using the real arguments of the invocation, and the local variables initialized by nil values (represented as nil in the rule) according to their types, that is, rfalse for **Bool** type and rnull for the class types. According to our assumption, all attribute references in method body are decorated with **this**. We bind **this** to the object referred by v during the execution of c. The internal variable **res** is initialized by the nil value. The rule for **return** statement will assign it the return value. With this variable, the case with multiple **return** statements can be dealt with naturally.

When the method returns, x is updated by value of **res** taken from the final local store.

$$\frac{\begin{array}{c} type(\sigma v) = C \qquad \Theta, C \vdash m \twoheadrightarrow \lambda(\overline{y})\{\textbf{var}\,\overline{z}; c\} \\ \langle c, (\{\overline{y \mapsto \sigma e}, \overline{z \mapsto nil}, \textbf{this} \mapsto \sigma v, \textbf{res} \mapsto nil\}, O)\rangle \rightsquigarrow^* (\sigma', O') \end{array}}{\langle x := v.m(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma' \textbf{res}\}, O')}$$

We should have another rule for the case that the execution of method body is stuck resulting an abort. It is similar to this one and simpler, thus is omitted here.

Object Creation: The object creation command $x := \textbf{new}\, C(\overline{e})$ creates a new object of class C, and then initiates its attributes (including inherited ones) with $\overline{e}$ and let x refer to it. Here r is a fresh reference not contained in the domain of O. We assume infinite available references, thus a fresh one can always be found. The selection of r is non-deterministic.

$$\frac{\Gamma.\mathsf{attr}(C) = \{\overline{a : T}\}}{\langle x := \textbf{new}\ C(\overline{e}), (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto r\}, O \oplus \{r \mapsto \{\overline{a \triangleright \sigma e}\}\})}\ r \notin \mathsf{dom}\, O$$

Here $\overline{a}$ are the attribute sequence of class C including the inherited ones.

Return: The **return** e statement records the value of e in **res**. As said, **res** is introduced for recording the return value, and it is assumed to have always the suitable type.

$$\frac{}{\langle \textbf{return}\ e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{\textbf{res} \mapsto \sigma e\}, O)}$$

The operational rules for structural commands are defined inductively. The abort during the execution will stop the execution, and reach the abort termination directly. In these rules, θ represents a terminal state or abort.

Sequential:

$$\frac{\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O'),\ \ \langle c_2, (\sigma', O')\rangle \rightsquigarrow^* \theta}{\langle c_1; c_2, (\sigma, O)\rangle \rightsquigarrow^* \theta}$$

$$\frac{\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* \mathsf{abort}}{\langle c_1; c_2, (\sigma, O)\rangle \rightsquigarrow^* \mathsf{abort}}$$

Choice:

$$\frac{\sigma b = \mathsf{rtrue},\ \ \langle c_1, (\sigma, O)\rangle \rightsquigarrow^* \theta}{\langle \textbf{if}\, b\, c_1\, \textbf{else}\, c_2, (\sigma, O)\rangle \rightsquigarrow^* \theta}$$

$$\frac{\sigma b = \mathsf{rfalse},\ \ \langle c_2, (\sigma, O)\rangle \rightsquigarrow^* \theta}{\langle \textbf{if}\, b\, c_1\, \textbf{else}\, c_2, (\sigma, O)\rangle \rightsquigarrow^* \theta}$$

While:

$$\frac{\sigma b = \mathsf{rfalse}}{\langle \textbf{while}\ b\ \textbf{do}\ c, (\sigma, O)\rangle \rightsquigarrow (\sigma, O)}$$

$$\frac{\sigma b = \mathsf{rtrue},\ \ \langle c; \textbf{while}\ b\ \textbf{do}\ c, (\sigma, O)\rangle \rightsquigarrow^* \theta}{\langle \textbf{while}\ b\ \textbf{do}\ c, (\sigma, O)\rangle \rightsquigarrow^* \theta}$$

Obviously, a program might fail to terminate for falling into an infinite iteration, or infinite recursive calls. In this case, the deduction cannot terminate too.

5.2 Properties

Theorem 3 (Uniqueness of Deduction). *In the environment Γ, Θ, given a well-formed command c and a state (σ, O), then there is one and at most one derivation route for $\langle c, (\sigma, O)\rangle$ with respect to our operational rules.*

Proof. Based on the structure of the command c, we can prove the conclusion inductively according to the transition rules. A special case to be noted here is that, the choice of the fresh reference value for the new object in the object creation operation is nondeterministic. The only condition is that the value is not active in the current object pool. However, the different choices of the reference have no influence on the deduction using the operational rules (thus, on the semantics of programs), because no language structure can distinguish the different choices. So the conclusion holds. $\square$

Having given the operational semantic rules for all commands, we introduce the following definitions that will be employed later.

Definition 1. *In the environments Γ and Θ, for any command c, store σ and Opool O:*

- *c gets* stuck *from (σ, O), if $\langle c, (\sigma, O)\rangle \rightsquigarrow^* \mathsf{abort}$;*
- *c is* safe *from (σ, O), if there exists a terminated configuration (σ', O') such that $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$.*
- *c is* divergent *from (σ, O), if there is an infinite transition sequence beginning from $\langle c, (\sigma, O)\rangle$.* $\square$

A well-typed program determines its environments Γ and Θ. Sometimes we will use them implicitly in the discussion, but not mention them in the text.

Proposition 1. *For a command c, a store σ and an Opool O, $\langle c, (\sigma, O)\rangle$ will either get stuck, or be safe, or be divergent.* $\square$

From the rules and Theorem 3, this proposition is obviously true.

The operational rules for commands given above take a global view for the Opool. We can restrict it to the footprints (as called by O'Hearn [10, 11]) of the command under consideration, that is, the object pool actually used by the command. For any command c, if c starts execution from a state with an object pool including its footprint, then it will never get memory faults. This brings us a kind of locality.

Definition 2 (Footprint of Command). *Footprint of command c, denoted by $fp(c)$, defines the object pool that is actually used by c in execution. The definition is by induction on the structure of command c:*

- *$fp(\textbf{skip}) = fp(x := e) = fp(x := (C)v) = fp(x := \textbf{new}\, C(\overline{e})) = \emptyset$;*
- *$fp(v.a := x) = fp(x := v.a) = \{\langle \sigma v, a, \text{-}\rangle\}$;*
- *$fp(x := v.m(\overline{e})) = fp(c[v/\textbf{this}, \overline{e}/\overline{y}])$, when $type(\sigma v) = C$, and $\Theta, C \vdash m \twoheadrightarrow \lambda(\overline{y})\{\textbf{var}\, \overline{z}; c\}$;*
- *$fp(c_1; c_2) = fp(\textbf{if}\ b\ c_1\ \textbf{else}\ c_2) = fp(c_1) \cup fp(c_2)$;*
- *$fp(\textbf{while}\ b\ \textbf{do}\ c) = fp(c)$.* $\square$

If we extend the object pool, any command c which has successful execution will behavior the same, in the sense that the extra part of the object pool keeps unchanged. Additionally, the divergence of commands will keep on extending the object pool. We have the following properties.

Proposition 2. *Suppose $O = O_1 * O_2$, where O, O_1, O_2 are object pools, and O_c is the footprint of the command c under consideration, we have*

- *If c gets stuck from state (σ, O_c), then c will get stuck from any (σ, O) where $O_c \subseteq O$. In this case, c meets null dereference or fault downcast in its execution.*
- *If c is safe (is divergent) from (σ, O_c), then c is safe (is divergent) from any (σ, O) where $O_c \subseteq O$.*
- *If c gets stuck from (σ, O), then c gets stuck from (σ, O_1).*
- *If c is safe from (σ, O_1), i.e., $\langle c, (\sigma, O_1)\rangle \rightsquigarrow^* (\sigma', O_1')$ for some store σ' and object pool O_1', then c is safe from (σ, O), and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O_1' * O_2)$.*
- *If c is safe from (σ, O), i.e., $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$ for some store σ' and object pool O', then either c gets stuck from (σ, O_1), or c is safe from (σ, O_1) with $\langle c, (\sigma, O_1)\rangle \rightsquigarrow^* (\sigma', O_1')$ and in this case, we must have $O' = O_1' * O_2$.* □

6 Consistency between Static and Dynamic Semantics

In this section, we investigate the relation between the semantics and the type system. We will prove the main theorem about the well-behavior property of programs - Typing Soundness Theorem, which says that if a well-formed program successfully terminates, then it must produce a state conforming to the static environment.

6.1 Conforming Environments and States

For the discussion on the relation between typing system and the operational semantics, we need a concept about the conformance of the state to the typing environment. In the first, we define the concept that a value (reference) conforms to a type:

Definition 3 (Conforming Value). *A reference (value) r in a state (σ, O) conforms to type T with respect to an environment Γ, if and only if*

- $r = \mathsf{rnull}$*, and T is* **Object** *or a user-defined class type;*
- $r \in \{\mathsf{rfalse}, \mathsf{rtrue}\}$ *and T is* **Bool***;*
- $r \notin \{\mathsf{rnull}, \mathsf{rfalse}, \mathsf{rtrue}\}$ *and $\Gamma \vdash \mathit{type}(r) \preceq T$.* □

Please note that $\mathit{type}(r)$ gives the type of the object referred by r, thus it depends on the state implicitly. Additionally, $\mathit{type}(r) \preceq T$ implies that $r \in O$, because only then $\mathit{type}(r)$ has a valid value.

For a state conforming to Γ, we require that variables in the store contain values of appropriate types, and objects in the object pool have the correct layout according to their classes, and each of their attributes records a reference value with suitable type. We define this locally with respect to a specific method in a specific class.

Definition 4 (Conforming State). *A state (σ, O) conforms to an environment Γ in a method m of class C, if and only if* $\mathsf{dom}\,\sigma = \{x \,|\, (C, m, x : T) \in \Gamma.\mathsf{locvar}\}$*, and*

- *for each variable $x \in \mathsf{dom}\,\sigma$, σx conforms to T with respect to Γ;*
- *for each $r \in \mathsf{dom}\,O$, $r \notin \{\mathsf{rtrue}, \mathsf{rfalse}, \mathsf{rnull}\}$, suppose $O(r) = \{f_i \triangleright r_i\}_{i=1}^{n}$, we must have $\Gamma.\mathsf{attr}(type(r)) = \{f_i : T_i\}_{i=1}^{n}$, and for each $i \in \{1, \cdots, n\}$, we must have also that r_i conforms to T_i in Γ.* $\square$

When the typing environment are clear from the context, we often omit to mention it explicitly, and say, e.g., r conforms to T, etc.

6.2 Well-Behavior of Programs

Now we state the main theorem that, if the execution of a well-formed command from a state conforming to its environment terminates successfully, then it will produce a state also conforming to the environment in the end. This theorem tells us that the type system and the operational semantics defined here are consistent with each other. It is sometimes thought also as the soundness theorem of the type system.

Every command here executes in a specific method m of a specific class C. For the simplicity, in the following discussion, we might not mention the context m and C explicitly when this will not cause confusion.

Theorem 4 (Typing Soundness). *Take an environment Γ produced by a well-formed program, a well-formed command c with $\Gamma, C, m \vdash c :$ **com**, and a state (σ, O) conforming to Γ in method m of class C. Then if there exists (σ', O') such that $\langle c, (\sigma, O)\rangle \leadsto^* (\sigma', O')$, then (σ', O') conforms to Γ in method m of C.*

Proof. Firstly, the condition that $\mathsf{dom}\,\sigma = \{x \,|\, (C, m, x : T) \in \Gamma.\mathsf{locvar}\}$ in the end state holds, providing it is true before execution of each command, because no command changes the domain of σ. We need only mention this condition in the proof for *Method Invocation*, in which a local store is created.

By induction on the structure of command c. We assume $\langle c, (\sigma, O)\rangle \leadsto^* (\sigma', O')$ for each command in the following proof.

- **Case skip**: **skip** do nothing. The conclusion holds trivially.
- **Case** $x := e$: From the operational rule for $x := e$, we have $\sigma' = \sigma \oplus \{x \mapsto \sigma e\}$ and $O' = O$ in the terminal configuration. Suppose we have $\Gamma, C, m \vdash x : T_1$ and $e : T_2$, the well-formedness of the command guarantees that $T_2 \preceq T_1$. We need only to prove that $\sigma'(x)$ conforms to T_1 in Γ, because the other parts of (σ', O') are the same as (σ, O). If $type(\sigma' x) =$ **Bool**, then $T_2 =$ **Bool**, the only possibility is that $T_1 = T_2 =$ **Bool**, the conclusion holds trivially. Otherwise, we have $type(\sigma' x) = type(\sigma e) \preceq T_2 \preceq T_1$. The conclusion holds too.
- **Case** $v.a := x$: From the operational rule for $v.a := x$, we have $\sigma' = \sigma$ and $O' = O \oplus \{\langle \sigma v, a\rangle \mapsto \sigma x\}\}$. Suppose $\Gamma, C, m \vdash v : N$ and $(N, a : T_1) \in \Gamma$ and $\Gamma, C, m \vdash x : T_2$, the well-formedness tell us that $T_2 \preceq T_1$. We need only to check that $O'(\sigma v)(a)$ conforms to T_1 in Γ, because the other part of (σ', O') is unchanged. If $type(O'(\sigma v)(a)) =$ **Bool**, then $type(\sigma x) =$ **Bool**, so T_2 must be **Bool**, which leads to that T_1 is **Bool** and the conclusion holds. Otherwise, we have $type(O'(\sigma v)(a)) = type(\sigma y) \preceq T_2 \preceq T_1$. The conclusion holds too.

- **Case** $x := v.a$: From the operational rule for $x := v.a$, we have $\sigma' = \sigma \oplus \{x \mapsto O(\sigma v)(a)\}$ and $O' = O$. Suppose $\Gamma, C, m \vdash v : N$ and $(N, a : T_1) \in \Gamma$ and $\Gamma, C, m \vdash x : T_2$, then we have $T_1 \preceq T_2$. We need only to check that $\sigma' x$ conforms to T_2 in Γ. If $\mathit{type}(\sigma' x) = \textbf{Bool}$, then $\mathit{type}(O(\sigma v)(a)) = \textbf{Bool}$ to, so T_1 must be **Bool**, which leads to T_2 is **Bool** and the conclusion holds. If $\mathit{type}(\sigma' x) \neq \textbf{Bool}$, we have $\mathit{type}(\sigma' x) = \mathit{type}(O(\sigma v)(a)) \preceq T_1 \preceq T_2$. The conclusion holds.
- **Case** $x := (N)v$: The argument is similar to that for $x := e$.
- **Case** $x := v.m_1(\overline{e})$: Suppose we have

$$\frac{\mathit{type}(\sigma v) = C_1 \qquad \Theta, C_1 \vdash m_1 \twoheadrightarrow \lambda(\overline{y})\{\textbf{var}\,\overline{z}; c\} \quad \langle c, (\{\overline{y} \mapsto \sigma\overline{e}, \overline{z} \mapsto \overline{nil}, \textbf{this} \mapsto \sigma v, \textbf{res} \mapsto nil\}, O)\rangle \rightsquigarrow^* (\sigma', O')}{\langle x := v.m_1(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma'\textbf{res}\}, O')}$$

  where nil represents the nil value according to the type of the local variables and **res**. Now we need to prove that $(\sigma \oplus \{x' \mapsto \sigma\textbf{res}\}, O')$ conforms to Γ.
  Firstly, we can see that $\{\overline{y} \mapsto \sigma\overline{e}, \overline{z} \mapsto \overline{nil}, \textbf{this} \mapsto \sigma v, \textbf{res} \mapsto nil\}$ conforms to Γ in method m_1 of class C_1, because: (1) Its domain contains exactly the parameters, local variables of method m_1, plus **this** and **res**; (2) the well-formedness of the invocation statement ensures that arguments $\overline{e}$ are suitable for parameters $\overline{y}$, that makes each value of the parameters $\overline{y}$ conforms to Γ; (3) **this** denotes the object referred by v, thus has the same type, thus conforms to Γ; and (4) z and **res** conform to Γ trivially because they have the suitable nil value.
  By induction, the execution of c ends in a state conforming to Γ, i.e., (σ', O') conforms to Γ in method m_1 of class C_1. Suppose we have $\Gamma, C, m \vdash x : T_1$, and the return type of m_1 is T_2, then the well-formedness tells us that $T_2 \preceq T_1$ and $\sigma'\textbf{res} \preceq T_2$ (because the well-formedness of **return** statement), so $\sigma'\textbf{res} \preceq T_1$. In store $\sigma \oplus \{x \mapsto \sigma'\textbf{res}\}$, we have $\mathit{type}(x) = \mathit{type}(\sigma'\textbf{res}) \preceq T_1$. Thus, $(\sigma \oplus \{x \mapsto \sigma'\textbf{res}\}, O')$ conforms to Γ in method m of class C.
- **Case** $x := \textbf{new}\ C_1(\overline{e})$: From the operational rule for this case, we have $\sigma' = \sigma \oplus \{x \mapsto r\}$ and $O' = O \oplus \{r \mapsto \{\overline{f \triangleright e}\}\}$, where r is a fresh reference. Suppose $\Gamma, C, m \vdash x : T'$, we need to prove that r conforms to T', and $\overline{e}$ conforms to $\overline{T_1}$ in Γ, where $\overline{T_1}$ are types of attributes of C_1. The type checking guarantees $\mathit{type}(r) = C_1 \preceq T'$, thus the first condition holds. Suppose $\Gamma, C, m \vdash \overline{e : T_2}$, from the well-formedness rule, we know $\overline{T_2 \preceq T_1}$, thus we have the conclusion.
- **Case return** e: The operational rule for **return** e gives $\sigma' = \sigma \oplus \{\textbf{res} \mapsto \sigma e\}$ and $O' = O$. Suppose the return type of m is T_1, and $\Gamma, C, m \vdash e : T_2$, then the well-formedness condition guarantees that $T_2 \preceq T_1$. It is easy to see that $\sigma\textbf{res}$ conforms to T_1 in Γ, and thus $(\sigma \oplus \{\textbf{res} \mapsto \sigma e\}, O)$ is fine. The conclusion holds.

The proof for the structural commands is trivial by the induction on the basic ones given above. Here we will not list them in details. □

Theorem 5 (Well-Behavior of Programs). *Suppose P is a well-formed program in μJava with a main method. If the execution of the command of the main method, beginning from the initial state consisted of the local context of main with all nil values to the local variables and the empty object pool, terminates successfully, it will go though a series of states with each of these states conforms to the typing environment Γ_P.* □

Class A **extends Object**$\{$	**Class** B **extends** $A\{$	**Class** C **extends Object**$\{$
Bool a; **Bool** b;	$B($**Bool** $c,$ **Bool** $d)$ $\{\ldots\}$	**Bool** $main\ ()\{$
$A($**Bool** $c,$ **Bool** $d)$ $\{\ldots\}$	**Bool** $m()\{$	$B\ z1, A\ z2,$ **Bool** $z3$;
Bool $m()\{$	**Bool** y;	$z1 :=$ **new** $B($**false**, **true**$)$;
Bool x;	$y :=$ **this**$.b$;	$z2 := (A)z1; z3 := z2.m()$;
$x :=$ **this**$.a$; **return** x;	**return** y;	**return** $z3$;
$\}$	$\}$	$\}$
$\}$	$\}$	$\}$

Table 1. An Example in μJava

7 An Example in μJava

We consider a sample program in μJava, as listed in Table 1. We use it to demonstrate some interesting features of the language such as reference types, inheritance, method dynamic binding, and type system, semantics, and so on. With the definitions above, first we are going to build the environment Γ for the program and consider the type checking, then show the states produced by the operational semantics. We check the soundness property in the last.

- **Build the typing environment Γ.** Each of its components is defined step by step following the rules, and the final result is as follows:

$$\begin{aligned}
\Gamma.\mathsf{cname} &= \{\mathbf{Object}, \mathbf{Bool}, \mathbf{Null}, A, B, C\}\\
\Gamma.\mathsf{super} &= \{(A, \mathbf{Object}), (B, A), (C, \mathbf{Object})\}\\
\Gamma.\mathsf{method} &= \{(A, A(\mathbf{Bool}\ c, \mathbf{Bool}\ d)), (A, m() : \mathbf{Bool}), (B, B(\mathbf{Bool}\ c, \mathbf{Bool}\ d)),\\
&\quad\ (B, m() : \mathbf{Bool}), (C, main() : \mathbf{Bool})\}\\
\Gamma.\mathsf{attr} &= \{(A, a : \mathbf{Bool}), (A, b : \mathbf{Bool}), (B, a : \mathbf{Bool}), (B, b : \mathbf{Bool})\}\\
\Gamma.\mathsf{locvar} &= \{(A, m, \mathbf{this} : A), (A, m, \mathbf{res} : \mathbf{Bool}), (A, A, c : \mathbf{Bool}),\\
&\quad\ (A, A, d : \mathbf{Bool}), (A, m, x : \mathbf{Bool}), (B, m, \mathbf{this} : B),\\
&\quad\ (B, m, \mathbf{res} : \mathbf{Bool}), (B, B, c : \mathbf{Bool}), (B, B, d : \mathbf{Bool}),\\
&\quad\ (B, m, y : \mathbf{Bool}), (C, main, \mathbf{res} : \mathbf{Bool}), (C, main, z1 : B),\\
&\quad\ (C, main, z2 : A), (C, main, z3 : \mathbf{Bool})\}
\end{aligned}$$

  Because class C has no local state, we omit **this** in $main$ here and below.
- **Typing in Γ.** Here we list only the type judgement for a part of commands in the program. We use the extended $\preceq$ relation for Γ here.
  For command $x :=$ **this**$.a$ in method m in class A, by the typing rule:

$$\frac{\Gamma, A, m \vdash x : \mathbf{Bool} \quad \Gamma, A, m \vdash \mathbf{this} : A \quad (A, a : \mathbf{Bool}) \in \Gamma.\mathsf{attr} \quad visible(A, m, A.a) \quad \mathbf{Bool} \preceq \mathbf{Bool}}{\Gamma, A, m \vdash x := \mathbf{this}.a : \mathbf{com}}$$

  For method m in class A, we have:

$$\frac{(A, m() : \mathbf{Bool}) \in \Gamma.\mathsf{method} \quad \Gamma, A, m \vdash x : \mathbf{Bool} \quad \mathbf{Bool} \preceq \mathbf{Bool}}{\Gamma, A, m \vdash \mathbf{return}\ \ x : \mathbf{com}}$$

$$\frac{\textbf{Bool} \in \Gamma.\mathsf{cname} \quad \Gamma, A, m \vdash x := \textbf{this}.a : \textbf{com}, \textbf{return}\ x : \textbf{com}}{\Gamma, A \vdash \textbf{Bool}\ m()\ \{\cdots\} : \textbf{ok}}$$

With the typing rules, we can get the result that all commands, method declarations, constructors, class declarations in this program are well-formed.

- **Operational Semantics for commands in** $main$ **method**. The $main$ method starts at the initial state. According to the operational semantics, we have the following results, in which method dynamic binding has been well dealt with.

$$(\{\langle z1, \mathsf{rnull}\rangle, \langle z2, \mathsf{rnull}\rangle, \langle z3, \mathsf{rfalse}\rangle, \langle \textbf{res}, \mathsf{rfalse}\rangle\}, \emptyset)$$
$$z1 := \textbf{new}\ B(\textbf{true}, \textbf{false});$$
$$(\{\langle z1, r1\rangle, \langle z2, \mathsf{rnull}\rangle, \langle z3, \mathsf{rfalse}\rangle, \langle \textbf{res}, \mathsf{rfalse}\rangle\}, \{\langle r1, a, \mathsf{rfalse}\rangle, \langle r1, b, \mathsf{rtrue}\rangle\})$$
$$z2 := (A)z1;$$
$$(\{\langle z1, r1\rangle, \langle z2, r1\rangle, \langle z3, \mathsf{rfalse}\rangle, \langle \textbf{res}, \mathsf{rfalse}\rangle\}, \{\langle r1, a, \mathsf{rfalse}\rangle, \langle r1, b, \mathsf{rtrue}\rangle\})$$
$$z3 := z2.m();$$
$$(\{\langle z1, r1\rangle, \langle z2, r1\rangle, \langle z3, \mathsf{rtrue}\rangle, \langle \textbf{res}, \mathsf{rfalse}\rangle\}, \{\langle r1, a, \mathsf{rfalse}\rangle, \langle r1, b, \mathsf{rtrue}\rangle\})$$
$$\textbf{return}\ z3;$$
$$(\{\langle z1, r1\rangle, \langle z2, r1\rangle, \langle z3, \mathsf{rtrue}\rangle, \langle \textbf{res}, \mathsf{rtrue}\rangle\}, \{\langle r1, a, \mathsf{rfalse}\rangle, \langle r1, b, \mathsf{rtrue}\rangle\})$$

- **Well behavior**. As the statement of Theorem 5, all the states through the execution above conform to the environment Γ in method $main$ of class C.

8 Conclusion

In this paper, we defined a storage model for OO program, and studied a simple OO language, sequential μJava. We presented the building of the static environment by scanning the program text. Then we defined a type system to judge the well-formedness of programs in μJava, which is enlightened by, and similar to, but more powerful than the one proposed in [4]. On the other hand, we adopt the reference model (other than value model in [4], etc.), that is commonly adopted by the practical OO languages. We defined an operational semantics for the well-formed programs. The notable features of the semantics include that the minimal store is adopted, and the context switching is dealt with accurately in the rules for the semantics.

We presented and proved some important properties of typing system and the semantics. We proved the type soundness theorem that the successful execution of the well-formed programs in μJava will go through a series of states conforming to the static environment. At last, we presented the building of static environment, typing procedure and the semantics by an example.

Although simple, μJava captures enough crucial features of general OO programming language, including reference types, subtypes, inheritance, dynamic binding, and sharing based parameters passing for methods, constructor, etc. Many more general structures in sequential OO programs can be encoding in this language with the help of some auxiliary variables etc. Thus, it can be used to model and study problems related to practical OO programs. We are working on a weakest-precondition semantics for μJava, thus we can give the notion of refinement that can be used to support OO based development. Also we have an on-going work on the confinement problems of OO programs based on this language. Other possible working direction is to extend our model with other features such as static method, exception handling and multi-threads.

References

1. M. Abadi and R. Leino. A logic of object-oriented programs. In *TAPSOFT '97, LNCS 1214*, pages 682–696. Springer, 1997.
2. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.
3. A.L.C. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *FME'99*, LNCS 1709, pages 1439–1459. Springer, 1999.
4. A.L.C. Cavalcanti and D. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. on Software Engineering*, 26(8):713–728, 2000.
5. Sophia Drossopoulou and Susan Eisenbach. Towards an Operational Semantics and Proof of Type Soundness for Java. In *Formal Syntax and Semantics of Java, LNCS 1523*. Springer, 1999.
6. J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *APLAS'04, LNCS 3302*. Springer, 2004.
7. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
8. Rob Hunter and Shriram Krishnamurthi. The semantics of garbage collection in OO languages. In *Foundations of Object-Oriented Languages*, 2003.
9. Atsushi Igarashi, Benjamin C.Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. In *Proceedings of OOPSLA'99*, pages 132–146. ACM, 1999.
10. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215C244, 1999.
11. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, volume 2142 of Lecture Notes in Computer Science*, pages 1–19, 2001.
12. Cees Pierik and Frank S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems*, 2003.