An Imperative Core Calculus for Java *

Baojian Hua
Department of Computer Science and Technology, University of Science and Technology of China
Hefei 230027, China
Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China
Suzhou 215123, China
huabj@mail.ustc.edu.cn

Abstract
This technical report presents the specification of a programming language Cool: the Core Object-Oriented Language. Cool is a small language but retains many of the features of modern class-based object-oriented languages, including primitive types, classes, inheritance, objects, instance variables and methods, dynamic method binding, null and arrays, etc. The specification for Cool in this report is formal. First, this paper gives syntactic specifications for Cool; second, we define static semantics for Cool via typing rules; and finally, we define its operational semantics. In stark contrast to any previous research effort in this area, we introduce side conditions into Cool's type system, which express important safety obligations.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs Specification Techniques; D.2.4 [Software Engineering]: Software/Program Verification—safety proofs, formal methods

General Terms Languages, Security, Semantics, Compilers

Keywords software safety, language design, object-oriented programming, type system, operational semantics

1. Introduction
The recent decade has seen the popularity of object-oriented programming paradigm. Part of reasons for its popularity are the combination of several important language features. Taking Java [2] as an example, the design of the Java programming language combines the experience of several object-oriented languages, in particular Smalltalk, C++ and CLOS. The "kernel" of the language, i.e., the sequential subset is designed to be small and straightforward, including features only with well understood semantics. So in order to understand the the language as a whole and to show the safety aspects, it's of great importance to formally define the language semantics.

The previous research efforts to define semantics for object-oriented languages, such as Java, could be classified into two categories. The first one is called object encoding [] which translates source languages features into more primitive core formal calculus, such as the $\lambda$ calculus or the System F. These encodings help in understanding the way objects are translated into low-level language features, and in understanding the interaction of objects and other languages features. However, one significant drawback of this method is that it's difficult, if not impossible, to translate some languages features, such as exceptions etc. And this method is not suitable for language design and semantics documentation. And proving the language safe involves not only proving the target language safe, but also the translation process.

The second method to formalize object-oriented languages is called primitive objects [], in which objects and classes and primitive language mechanisms. Rather than translating language features into more primitive low-level core calculus, this method gives a direct account of language features. This method enables us to reason about the static and dynamic semantics of language feature at a high-level, and is more suitable to language design and documentation. While recent research efforts have make significant progress on understanding the semantics issues involved, the theoretical underpinnings for object-oriented programming paradigm still lags well behind practical implementations and applications. Specifically, there remain at least the following two challenges:

1. Previous studies have not shown how to incorporate the value-related safety constrains with type system. As a result, these type system could not express these constraints.

2. The safety theorems proved in previous research are rather weak. Well-typed programs also raise exceptions which could otherwise be detected by static discipline.

The primary goal of this paper is to define a clear and general semantics framework, which capture crucial value-related constrains. To be specific, we adopt the primitive objects method, and design a formal imperative core calculus Cool: the Core Object-Oriented Language. Cool is a small language but retains many of the features of modern class-based object-oriented languages including classes, objects, inheritance, etc.. Essentially, Cool is very close to a subset of Java [2], and is also very close to other popular class-based object-oriented language, such as C# [1].

To incorporate the value-related constraints with the type system, we adopt the method of side conditions previous proposed for the PointerC [3] programming language. A

---

*This research is in part supported by the National Natural Science Foundation of China under grant No. 60673126, and a grant from the Intel China Research Center (ICRC). Any opinions, findings and conclusions contained in this document are those of the author(s) and do not necessarily reflect the views of these agencies.
side condition is very much like dependent type [4] in that they both express value constraints. However, side conditions are not part of the type system, and is not guaranteed by traditional type checker.

To summarize, the primary technical highlights and contribution of this paper are:

- We have formalized an imperative core calculus Cool for the Java programming languages.
- We have illustrated the combination of typing rules and side conditions which expresses more refined safety obligations.
- We have proved the Cool safe, and the proof for this theorem follows the standard progress and preservation.

The rest of the paper is organized as follows. Section 2 introduces the syntactic rules for Cool; section 3 presents operational semantics; section 4 presents a formal type system; section section 5 concludes.

2. Syntax

The Syntax of Cool is given in Figure 1. The program $p$

\[
\begin{align*}
\text{Program} & \quad p ::= cs \\
\text{Classes} & \quad cs ::= c cs \\
\text{Class Dec.} & \quad c ::= \text{class } x\{ vs ms \} \\
& \quad \quad | \quad \text{class } x \text{ extend } x\{ vs ms \} \\
\text{Var. Dec. List} & \quad vs ::= t x ; vs \mid e \\
\text{Method Dec. List} & \quad ms ::= m ms \mid e \\
\text{Method Dec.} & \quad m ::= \text{public } t(x fs) \{ vs ss \text{ return}(e); \} \\
\text{Formals} & \quad fs ::= t \text{ id } ft \mid e \\
\text{Formal Rest} & \quad ft ::= . t \text{ id } ft \mid e \\
\text{Type} & \quad t ::= \text{int} \mid \text{boolean} \mid \text{int}[] \mid x \\
\text{Statements} & \quad ss ::= s ss \mid e \\
\text{Statement} & \quad s ::= \{ ss \} \\
& \quad | \quad \text{if } (e) \mid \text{else } s \\
& \quad | \quad \text{while } (e) \mid s \\
& \quad | \quad x = e; \\
& \quad | \quad x = \text{new } x(); \\
& \quad | \quad x = \text{new } \text{int}[e]; \\
& \quad | \quad x = e.\text{charAt}(e); \\
& \quad | \quad x[e] = e; \\
\text{Expression} & \quad e ::= e \oplus e \\
& \quad | \quad e[e] \\
& \quad | \quad e.\text{length} \\
& \quad \text{true} \\
& \quad \text{false} \\
& \quad \text{x} \\
& \quad \text{this} \\
& \quad \text{null} \\
& \quad \text{l} \\
& \quad \text{e} \\
\text{Exp. List} & \quad es ::= e \text{ et } | e \\
\text{Exp. Rest} & \quad et ::= . e \text{ et } | e \\
\text{Operator} & \quad + | - | \ast | / | \% \\
\end{align*}
\]

Figure 1. Syntax of Cool

consists of a bunch of class declarations $cs$. Every class $c$ is either inherits from the default object (which is omitted) or from some other declared class $x$. A group of instances variables and methods could be defined in class body. For clarity, we omit the variable modifiers and all methods are public.

Representative Java types are retained in Cool, such as int, boolean, and class type etc. The principle here is that we would only include a minimal set of necessary types to present the key ideas. Of course, adding more types constructors is possible, but by doing so, only more engineering efforts are required.

Cool includes a bunch of basic assignment and control flow statements. The assignment could change the value of store variables, objects, and array element.

The syntactic forms of expressions $e$ include binary operations, array indexing, and this etc. One key point to note is that all expressions are pure, they do not produce side-effect during evaluation. To achieve this, the side-effect expressions are lifted to the level of statements.

3. Operational Semantics

The Cool programming language has heap-allocated objects and arrays, along with atomic values, such as integers and booleans. So the operational semantics are given by defining an abstract machine model. And the program execution is modeled by the transition relation of this machine model. For any well-typed programs, the machine will start from a initial state, and at each program point, there exists at most one transition rule that applies. There is a run-time error, if there is no such transition rule could apply.

This section presents the operational semantics of Cool. First, a machine model is designed to model the abstract program state; and then the evaluation rules are given as a transition relation on the abstract machine model.

3.1 The Abstract Machine Model

The abstract machine model for Cool is given in Figure 2. Cool has instance variables, method local variables etc.,

\[
\begin{align*}
\text{Machine} & \quad M ::= (S, H, K, C) \\
\text{Store} & \quad S ::= x \mapsto v, S | . \\
\text{Heap} & \quad H ::= l \mapsto hv, H | . \\
\text{Value} & \quad v ::= n \mid \text{true} \mid \text{false} \mid l \\
\text{Heap Value} & \quad hv ::= \{ \text{tag} \mapsto x, h_1 \mapsto v, \ldots, h_n \mapsto v \} \\
& \quad | \quad [k, n_1, \ldots, n_k] \\
\text{Heap Address} & \quad l ::= l_1 | l_2 | \ldots \\
\text{Continuation} & \quad K ::= s, K | . \\
\text{Class Table} & \quad C ::= (x, y) \mapsto (A, L, s), C | . \\
\end{align*}
\]

Figure 2. The Abstract Machine Model

so the abstract machine model $M$ consists of four parts $(S, H, K, C)$, where $S : x \mapsto v$ is a store mapping any variable $x$ local to a method to a value $v$; $H : l \mapsto hv$ is a heap, which maps any heap address $l$ to a heap value $hv$, note that these heap address $ls$ are abstract in the sense that they are abstract labels and no address arithmetic are allowed on them.

Value $v$ only consists of four forms: integer literal $n$, booleans true, false, and address $l$. Note also that these values are all atomic, in the sense that on a concrete machine, these values could all fit into a 32-bit address or a machine register. This design decision reflects the fact that though the definition is given at high level, this abstract machine models some key aspects of the low-level concrete machine honestly, and it’s relatively easy to be translate this model to a low representation.
Heap value \( hv \) has two syntactic forms. The first form \( \{ \text{tag} = x, h_1 \mapsto v_1, \ldots, h_n \mapsto v \} \) stands for an object. One object consists of a sequence of fields. The first field \( \text{tag} \) is a type tag, and is initialized by the new \( x() \) statement at object creation time. Field labels \( h_1, \ldots, h_n \) denotes object fields, and have values \( v_1, \ldots, v_n \) respectively. Also note that each of these fields could only has value \( v \), instead of heap value \( hv \). The second form \( [k, n_1, \ldots, n_4] \) is an array. An array consists of a sequence of element, and the first element is the array length.

Unlike values, heap values are structured value and big, which reflects the fact that on a concrete machine, they cannot be stored into registers, and thus have to be allocated on the stack or dynamic heap.

A control \( R \) is a sequence of statement list \( ss \), and its behavior is like a stack. The primary goal of \( R \) is to model the execution of the while statement, that is, when the loop condition holds, then the loop itself is put onto the top of \( R \), and after the execution of loop body finished, the top element of \( R \) is fetched and executed. The purpose of introducing a continuation \( K \) is very similar. When method invocation occurs, the remaining statements after the invocation point in the caller in put onto the top of \( K \), and after the call returns, the top element of \( K \) is fetched and executed. All these details will be further explained in operational semantics rules.

Finally, the class table \( C \) maps every tuple \( (x, y) \) to a triple \( (A, L, s) \). The first element \( x \) of the tuple \( (x, y) \) is a class identifier, while the second \( y \) is a method identifier. The triple \( (A, L, s) \) consists of three parts: both the first one \( A \) and the second one \( L \) are variable list, stand for arguments and local variables of the method \( y \); and \( s \) is the method body. It’s important to note that this kind of class table representation hides many implementation details in a production compiler. For instance, a real compiler may organize the class table using a linked list, instead of the current flat one. But by using this relative abstract class table representation, not only the presentation and discussion of properties will be more straightforward, but also give the compiler writers freedom to choose whatever representation strategies.

### 3.2 Evaluation Rules

The evaluation rules are presented in a call-by-value small-step style, and are defined by a transition relation \( \leadsto \):

\[(S, H, K, C) \triangleright a \leadsto (S', H', R', K', C') \triangleright b\]

At the left hand side of \( \leadsto \) is a pair separated by \( \triangleright \). The first part of the pair is a machine state \((S, H, K, C)\), and the second part is some syntactic form \( a \). The pair as a whole indicates that we are evaluating a under the current machine state \((S, H, K, C)\). And the result of this evaluation is a new machine state \((S', H', R', K', C')\) and a new evaluation candidate \( b \), just as the right hand side of \( \leadsto \) shows.

As the class table \( C \) is a runtime constant and never changes during the program execution, in the following, it’s omitted from the abstract machine model for clarity.

\[
\begin{align*}
(S, H, K) \triangleright v_1 \oplus v_2 & \leadsto (S, H, K) \triangleright v & \quad (\text{E-}\oplus 1) \\
(S, H, K) \triangleright e_1 \leadsto (S, H, K) \triangleright e'_1 & \quad (\text{E-}\oplus 2) \\
(S, H, K) \triangleright e_1 \oplus e_2 \leadsto (S, H, K) \triangleright e'_1 \oplus e_2 & \quad (\text{E-}\oplus 3) \\
\end{align*}
\]

\[
\begin{align*}
H(l) = [k, v_0, \ldots, v_{k-1}] & \quad 0 \leq i \leq k - 1 & \quad (\text{E-ARRAY1}) \\
(S, H, K) \triangleright l[i] \leadsto v_i & \quad (\text{E-ARRAY2}) \\
(S, H, K) \triangleright e_1 \leadsto (S, H, K) \triangleright e'_1 & \quad (\text{E-ARRAY3}) \\
(S, H, K) \triangleright l[e_2] \leadsto (S, H, K) \triangleright l[e'_2] & \quad (\text{E-ARRAY1}) \\
(S, H, K) \triangleright \l.\text{length} \leadsto (S, H, K) \triangleright k & \quad (\text{E-LENGTH1}) \\
(S, H, K) \triangleright e \cdot l.\text{length} \leadsto (S, H, K) \triangleright e' \cdot l.\text{length} & \quad (\text{E-LENGTH2}) \\
\end{align*}
\]

Statement evaluation rules:

\[
\begin{align*}
S[x \mapsto v] = S' & \quad R = s \circ R' & \quad (\text{E-ASNVAR1}) \\
(S, H, K) \triangleright x = v \leadsto (S', H', R', K) \circ s & \quad (\text{E-ASNVAR2}) \\
(S, H, K) \triangleright e \leadsto (S, H, K) \triangleright e' & \quad (\text{E-ASNOBJ1}) \\
(S, H, K) \triangleright e_1, x = e_2 \leadsto (S, H, K) \triangleright e'_1, x = e_2 & \quad (\text{E-ASNOBJ2}) \\
(S, H, K) \triangleright l.x = e_2 \leadsto (S, H, K) \triangleright l.x = e_2 & \quad (\text{E-ASNOBJ3}) \\
\end{align*}
\]

\[
\begin{align*}
\text{newloc}(l) & \quad S[x \mapsto l] = S' & \quad (\text{E-NEW}) \\
H[l \mapsto \{\text{tag} = y, h_1 \mapsto \ldots, h_n \mapsto ?\}] = H' & \quad (\text{E-NEW}) \\
(S, H, K) \triangleright x = \text{new int}[k] \leadsto (S', H', R', K) \triangleright & \quad (\text{E-NEWARRAY}) \\
\end{align*}
\]
\[(S, H, K) \vdash e' \rightarrow (S, H, K) \vdash e'(\text{E-INVOC1})\]
\[(S, H, K) \vdash x = \text{e}.f(\text{es}) \rightarrow (S, H, K) \vdash x = \text{e}.f(\text{es})\]
\[(S, H, K) \vdash e_i \rightarrow (S, H, K) \vdash e_i' \text{ (E-INVOC2)}\]
\[(S, H, K) \vdash x = l.f(v_1, \ldots, v_n) \rightarrow (S, H, K) \vdash x = l.f(v_1, \ldots, v_n)\]
\[C(H().\text{tag}, f) = (a_1, \ldots, a_n, b_1, \ldots, b_m, s)\]
\[S[\text{this} \mapsto l, a_1 \mapsto v_1, \ldots, a_n \mapsto v_n, R \mapsto K'] = \text{s}'\]
\[(S, H, K) \vdash x = l.f(v_1, \ldots, v_n) \rightarrow (S', H, \cdot, K') \vdash s\]
\[(S, H, K) \vdash x[i] = e_1 \rightarrow (S, H, K) \vdash x[i] = e_2 \text{ (E-ASNARRAY1)}\]
\[(S, H, K) \vdash e_1 \rightarrow (S, H, K) \vdash e_2\]
\[(S, H, K) \vdash x[i] = e_1 \rightarrow (S, H, K) \vdash x[i] = e_2\]
\[(S, H, K) \vdash e_2 \rightarrow (S, H, K) \vdash e_2'\]
\[\text{if (true) } s_1 \text{ else } s_2 \rightarrow (S, H, K) \vdash s_1\]
\[\text{if (false) } s_1 \text{ else } s_2 \rightarrow (S, H, K) \vdash s_2\]
\[(S, H, K) \vdash \text{if (e) } s_1 \text{ else } s_2 \rightarrow (S, H, K) \vdash \text{if (e) } s_1 \text{ else } s_2\]
\[(S, H, K) \vdash \text{while (e) } s \rightarrow (S, H, K) \vdash \text{while (e) } s\]
\[(S, H, K) \vdash \text{true} \rightarrow (S, H, K) \vdash \text{true}\]
\[(S, H, K) \vdash \text{while (e) } s \rightarrow (S, H, K) \vdash \text{while (e) } s\]
\[\Delta, \Gamma; C \vdash e : \text{int[]}\]
\[\Delta, \Gamma; C \vdash \theta : \text{int[]}\]

### 4. Type System

This section presents the type system of Cool. These typing rules are largely standard and are very close to those of Java. And in order to make the task of studying the formal properties easier, these typing rules have been designed in a manner that are very much like those for λ-calculus.

One novel feature of Cool’s typing rules is that some of them include explicit side conditions. Just like the case for a C-like imperative language [?], a side condition is a logic proposition, and expresses the explicit requirements posed on variables values or pointer states. In order to distinguish these side conditions from ordinary typing premises in typing rules, side conditions are enclosed by a pair of braces, and put at the left hand side of the corresponding rules. For instance, in the typing rule for array length operation

\[\Delta, \Gamma; C \vdash e : \text{int}[]\]

The normal typing premise \(\Delta, \Gamma; C \vdash e : \text{int}[]\) expresses that \(e\) is an array with elements of type \(\text{int}\), and the side condition \(e \in \text{effective}\) expresses that the value of \(e\) should be an effective pointer, rather than a null pointer.

One key point to note is that the typing rules only make clear the necessary side conditions, instead of defining rules on how to prove these sides conditions. To prove these side conditions statically, we’ve designed a pointer logic. For the proof methodology, the interested readers are invited to refer to the pointer logic paper.

#### 4.1 Types

The definitions for types and typing environment are given in Figure 3. The type \(\tau\) includes basic Cool type constructors: \(\text{int}\), \(\text{boolean}\) and \(\text{int}[]\); identifier \(x\) stands for a class name. The last two types \(\text{unit}\) and \(\text{ns}\) themselves are not valid types, but are used as internal types. The type \(\text{unit}\) is used to type statements (which only produce side effect, rather than interesting values), and the type \(\text{ns}\) are used to type the constant \(\text{null}\).

The local environment \(\Gamma\) is a list composed of variable and type tuples, which maps every method argument or local variable \(x\) to its type \(\tau\).

A class environment \(\Delta\) maps every class name \(x\) to its type \((\Sigma, \Theta, y)\) or \((\Sigma, \Theta, \cdot)\). The third element of the triple could either be \(y\) or \(\cdot\), which expresses that the super class of \(x\) could either be another class \(y\) or be the \text{Object}. And in the second case, the Object is omitted for clarity, and the super of class of \(x\) is just written as a dot \(\cdot\). The first element of this triple is a class variable environment \(\Sigma\), which maps instance every variable \(x\) to its type \(\tau\). And the second element of this triple is a class method environment \(\Theta\), which maps every instance method \(m\) to its signature \((x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau\), where \(x_1, \ldots, x_n\) are arguments and \(\tau\) is the return type.

\[
\begin{align*}
\text{Type} & : \quad \tau ::= \text{int} | \text{boolean} | \text{int}[] \\
\text{Local Env.} & : \quad \Gamma ::= x : \tau, \Gamma' | \cdot \\
\text{Class Var. Env.} & : \quad \Sigma ::= x : \tau, \Sigma' |
\text{Class Mtd. Env.} & : \quad \Theta ::= m : (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau, \Theta | \cdot \\
\text{Class Env.} & : \quad \Delta ::= x : (\Sigma, \Theta, y), \Delta | x : (\Sigma, \Theta, \cdot), \Delta |
\end{align*}
\]

Figure 3. Types and Typing Environment
4.2 Auxiliary Definitions

In the process of type checking a class C under a given class environment \( \Delta \), one instance variable \( x \) may appear in C itself or in C’s super class. So we define a judgement \( \vdash S(\Delta, C, x) = \tau \) to indicate that under the given class environment \( \Delta \), the type of the variable \( x \) in the class \( C \) is \( \tau \). And this judgement is given by the following two rules:

\[
\Delta(C) = (\Sigma, \Theta, C') \quad \Sigma(x) = \tau \\
\vdash S(\Delta, C, x) = \tau
\]

\[
\Delta(C) = (\Sigma, \Theta, C') \quad x \notin \text{Dom}(\Sigma) \\
\vdash S(\Delta, C', x) = \tau
\]

Similarly, to type a method \( m \) in a class \( C \), we would need the judgement \( \vdash F(\Delta, C, m) = (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau \), and it’s given by the following two rules:

\[
\Delta(C) = (\Sigma, \Theta, C') \quad \Theta(m) = (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau \\
\vdash F(\Delta, C, m) = (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau
\]

\[
\Delta(C) = (\Sigma, \Theta, C') \quad m \notin \text{Dom}(\Theta) \\
\vdash F(\Delta, C', m) = (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau
\]

And we also want to search when a type \( x \) is a subtype of another type \( y \), and which is given by the assertion \( T(\Delta, x, y) \), where \( \Delta \) is a class environment and \( x \) and \( y \) are two class names:

\[
\Delta(x) = (\Sigma, \Theta, y) \\
\vdash T(\Delta, x, y)
\]

\[
\Delta(x) = (\Sigma, \Theta, z) \quad y \neq z \\
\vdash T(\Delta, z, y)
\]

The subtype relation is given in Figure 4. And the assertion

\[
\Delta \vdash \tau_1 <_s \tau_2
\]

\[
\Delta \vdash \tau <_s \tau
\] (SUB-REFL)

\[
\vdash T(\Delta, C_1, C_2) \quad C_1 \neq C_2 \\
\Delta \vdash C_1 <_s C_2
\] (SUB-ENV)

Figure 4. Rules for Subtype Relation

4.3 Typing Rules

\[
\Delta; \Gamma; C \vdash e : \tau
\]

\[
\Delta; \Gamma; C \vdash e_1 : \text{int} \\
\Delta; \Gamma; C \vdash e_2 : \text{int}
\]

\[
\Delta; \Gamma; C \vdash e_1 \oplus e_2 : \text{int}
\] (⊕)

\[
\Delta; \Gamma; C \vdash e_1 : \text{int}[] \\
\Delta; \Gamma; C \vdash e_2 : \text{int}
\]

\[
\Delta; \Gamma; C \vdash e_1[e_2] : \text{int}
\] \( \{e_1 \in \text{effective} \} \) (ARRAY)

\[
\Delta; \Gamma; C \vdash e : \text{int}[] \\
\Delta; \Gamma; C \vdash e.length : \text{int}
\] \( \{e \in \text{effective} \} \) (LENGTH)

\[
\Delta; \Gamma; C \vdash n : \text{int}
\] (INTEGER)

\[
\Delta; \Gamma; C \vdash \text{true} : \text{boolean}
\] (TRUE)

\[
\Delta; \Gamma; C \vdash \text{false} : \text{boolean}
\] (FALSE)

\[
\Delta; \Gamma; C \vdash e : \text{boolean}
\] (NOT)

\[
\Delta; \Gamma; C \vdash \text{this} : C
\] (THIS)

\[
\Delta; \Gamma; C \vdash \text{null} : \text{ns}
\] (NULL)

\[
\Delta; \Gamma; C \vdash e : \text{unit}
\] (PAREN)

\[
\Delta; \Gamma; C \vdash s : \text{unit}
\]

\[
\Delta; \Gamma; C \vdash e : \text{boolean} \\
\Delta; \Gamma; C \vdash s_1 : \text{unit}
\]

\[
\Delta; \Gamma; C \vdash \text{if}(e) s_1 \text{ else } s_2 : \text{unit}
\] (If)

\[
\Delta; \Gamma; C \vdash e : \text{boolean} \\
\Delta; \Gamma; C \vdash s : \text{unit}
\]

\[
\Delta; \Gamma; C \vdash \text{while}(e) s : \text{unit}
\] (While)

\[
\Delta; \Gamma; C \vdash x : \tau_1 \\
\Delta; \Gamma; C \vdash e : \tau_2
\]

\[
\Delta; \Gamma; C \vdash x <_s \tau_1 \quad \Delta; \Gamma; C \vdash e <_s \tau_2
\]

\[
\Delta; \Gamma; C \vdash x : \tau \quad \Delta; \Gamma; C \vdash C <_s \tau
\]

\[
\Delta; \Gamma; C \vdash x = \text{e} : \text{unit}
\] (AsnExp)

\[
\Delta; \Gamma; C \vdash x = \text{new} C() : \text{unit}
\] (AsnNew)

\[
\Delta; \Gamma; C \vdash x : \text{int}[] \\
\Delta; \Gamma; C \vdash \text{e} : \text{int}
\]

\[
\Delta; \Gamma; C \vdash x = \text{new int[e]} : \text{unit}
\] (AsnNewArr)

\[
\Delta; \Gamma; C \vdash x : \tau \\
\Delta; \Gamma; C \vdash e : C'
\]

\[
\vdash F(\Delta, C', y) = (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau_i
\]

\[
\Delta; \Gamma; C \vdash x = \text{e}.y(e_1, \ldots, e_n) : \text{unit} \\
\{e \in \text{effective} \}
\] (Call)

\[
M \triangleq F(\Delta, C', y) = \tau_i \quad \forall 1 \leq i \leq n.\Delta; \Gamma; C \vdash e_i : \tau_i \quad \tau_i \tau_1 \ldots \tau_n <_s \tau_s
\]
Below are environments formation rules:

\[ \Delta, \Gamma \vdash C \vdash x : \text{int}[()] \]
\[ \Delta, \Gamma \vdash e_1 : \text{int} \]
\[ \Delta, \Gamma \vdash e_2 : \text{int} \]
\[ \Delta; \Gamma \vdash x[e_1] = e_2 : \text{unit} \quad \{ x \in \text{effective} \} \quad \text{(ASSARRAY)} \]

\[ \Delta; \Gamma \vdash s : \text{unit} \]
\[ \Delta, \Gamma \vdash s; ss : \text{unit} \quad \text{(STMCONS)} \]
\[ \Delta, \Gamma \vdash \vdash : \text{unit} \quad \text{(STMEMPTY)} \]

\[ \Delta; \Gamma \vdash \text{localVarDec} : \Delta; \Gamma \vdash \]
\[ x \notin \text{Dom}(\Gamma) \quad \Delta; \Gamma, x : \tau, C \vdash xs : \Delta, \Gamma', C \]
\[ \Delta; \Gamma \vdash C \vdash x, xs : \Delta, \Gamma', C \quad \text{(LOCALVARDECCONS)} \]
\[ \Delta; \vdash : \Delta; \Gamma \vdash \quad \text{(LOCALVARDECEMPHY)} \]

\[ \Delta; C \vdash f : \text{unit} \]
\[ \Delta; C \vdash fs : \Delta; \Gamma \vdash C \quad \Delta; C \vdash e : \tau \quad \tau < : t \]
\[ \Delta; C \vdash \text{public } t x(f s) \{ \text{vs } ss \text{ return } e; \} : \text{unit} \quad \text{(MTD)} \]
\[ \Delta; C \vdash f : \text{unit} \quad \Delta; C \vdash fs : \text{unit} \quad \text{(MTDS)} \]
\[ \Delta; C \vdash : \text{unit} \quad \text{(MTDEMPHY)} \]

\[ \Delta \vdash c : \text{unit} \]
\[ \Delta; C \vdash mtds : \text{unit} \quad \text{(CLASS)} \]
\[ \Delta \vdash \text{class } C \{ \text{vs } ms \} \quad \text{(CLASSCONS)} \]
\[ \Delta \vdash c : \text{unit} \quad \Delta \vdash cs : \text{unit} \quad \text{(CLASSEMPTY)} \]
\[ \Delta \vdash : \text{unit} \quad \text{(CLASSEMPY)} \]

Below are environments formation rules:

\[ \Sigma \vdash \text{classVar} : \Sigma \]
\[ x \notin \text{Dom}(\Sigma) \quad \Sigma; x : \tau \vdash xs : \Sigma' \quad \text{(CLASSVARCONS)} \]
\[ \Sigma \vdash : \Sigma \quad \text{(CLASSEMPY)} \]

\[ \Theta \vdash f : \Theta \]

\[ f \notin \text{Dom}(\Theta) \quad \Theta; f : \tau_1 \cdots \tau_n \rightarrow t \vdash ms : \Theta' \]
\[ \Theta \vdash \text{public } t f(x_1 : \tau_1, \ldots, x_n : \tau_n) \{ \text{vs } ss \text{ return } e; \} : \text{ms} : \Theta' \quad \text{(MTDFORMCONS)} \]
\[ \Theta \vdash : \Theta \quad \text{(MTDFORMEMPTY)} \]

\[ x \notin \text{Dom}(\Delta) \quad \vdash vs : \Sigma \quad \vdash ms : \Theta \quad \Delta, x : (\Sigma, \Theta, \cdot) \vdash cs : \Delta' \quad \text{(CLASSFORMCONS1)} \]
\[ x \notin \text{Dom}(\Delta) \quad \vdash vs : \Sigma \quad \vdash ms : \Theta \quad \Delta, x : (\Sigma, \Theta, C') \vdash cs : \Delta' \quad \text{(CLASSFORMCONS2)} \]
\[ \Delta \vdash : \Delta \quad \text{(CLASSEMPY)} \]

\[ \vdash p : \Delta \quad \Delta \vdash p : \text{unit} \quad \text{(PROG1)} \]

5. Conclusion

This paper presents an imperative core calculus called Cool for the Java programming language. The syntax, static semantics and dynamic semantics closely follow those of Java. One novelty of Cool is that its typing rules are accompanied with side conditions, which express value-related constraints. We argue that this is an important step to formalize some Java aspects that traditionally handled by runtime methods.

There are several directions for our future research. First, we are extending the language features that could be handled, especially, we are investigating exceptions, which will not affect the static semantics, but will alter the dynamic semantics drastically. Second, we are investigating the relationship between the object encoding and primitive objects with side conditions, especially the translation from the latter into the former one.

References