A. Object-oriented Programming Primer

Most modern languages provide constructs to declare classes. Main motivation is to allow real-world objects to be represented in the program as “instances” of these classes. Object-oriented programming encourages thinking in terms of these objects (or object instances) that interact with each other by sending messages. In programming languages sending messages is modeled as a method call expression. Briefly some of the important concepts in object-oriented programming are:

- Encapsulation: grouping state of an object and its behavior.
- Inheritance: extending an existing representation to enhance reuse.
- Dynamic dispatch: selecting a method call based on the dynamic type of an object.

In languages such as Java, C#, C++ support for object-orientation is provided via facilities to declare classes. The notion of a class can be thought of as the template for all objects of that class, which defines the state and the behavior of these objects. An example class is shown in Figure 1.

```
1  class Point {
2      int x; int y;
3      int setX(int x){
4          this.x = x; x
5      }
6      int setY(int y){
7          this.y = y; y
8      }
9      boolean isOrigin(){
10         if(p.x == 0 && p.y == 0){ true }
11         else { false }
12      }
13  }
```

Figure 1: An Example Class Point

This class declaration represents that all objects of the Point class will maintain its state as two fields x and y that represent cartesian coordinates of a point.

To create a point (or an object of class Point) languages typically provide facilities for object creation or instantiation. In the C++ family this support is often provided as the new operator. An example instantiation of the class Point in such languages could be written as `new Point()`, which like malloc in ‘C’ and ref in our expression language evaluates to a reference to the newly created instance. It is also typical to access fields and methods of an object. These are represented using the dot notation. For example, in Point p ...; p.y is an example of accessing field y of object p. Methods can also be called in a similar manner e.g. p.setX(0). Here, the object p whose methods are being called is also referred to as the receiver of the method call, in a style reminiscent of thinking about method calls as messages. Sometimes we would also use an alternative terminology target to refer to the receiver of the method call.

To enable reuse classes can also extend other classes. A simple example appears in Figure 2 where the class Point3D inherits from class Point and adds a new field z and a new method setZ.

Here Point is the superclass and Point3D is the subclass. While extending classes, it is also possible to redefine methods that appear in the definition of the superclass. For example, the class Point3D redeﬁnes the deﬁnition of the method isOrigin to reﬂect the true nature of origin in a three dimension cartesian coordinate system.

Now let us consider the code in Figure 3 where an object of class Point3D is initialized and assigned to a variable that is declared to be Point.
class Point3D extends Point {
    int z;
    int setZ(int z) {
        this.z = z;
    }
    boolean isOrigin() {
        if (p.x == 0 && p.y == 0 && p.z == 0) { true }
        else { false }
    }
}

Figure 2: Class Point3D inherits from Class Point

Point3D p3d = new Point3D();
p3d.x = 0; p3d.y = 0; p3d.z = 10;
Point p = p3d;
p.isOrigin()
p3d.isOrigin()

Figure 3: Example Usage of Point3D and Point

Finally the method isOrigin is called, where the target is p, and p3d. Whether the result of this call expression is true or false depends on the semantics of dynamic dispatch in the language. In C++ and C#, for example, if the method isOrigin is not declared to be virtual, the evaluation result for the call p.isOrigin() would be true and p3d.isOrigin() would be false. On the other hand, if the method isOrigin was declared virtual, the evaluation result for both calls would be false. In Java all methods are implicitly virtual.

Given this background, let us now add object-oriented features to our expression language. We will consider a minimal subset of features to study object-orientation.

B. Abstract Syntax

The enhanced syntax of our expression language (541EXP ОО) is shown below. The technical details of 541EXP ОО are inspired from several languages, in particular from the work on Rajan and Leavens on Ptolemy [7], the work of Clifton and Leavens [2, 3]. These works are in turn based on Featherweight Java [5] and Classic Java [4]. We will build 541EXP ОО in increments. First, we will add classes and objects to the language. We will then add inheritance and subtyping. Some constructs such as super, interfaces, exception handling, privacy modifiers (such as public, private, protected, etc), abstract methods, or constructors will be omitted from this core calculus of object-oriented languages. There are only two built-in value types int and boolean.

B.1 Adding the Class Declaration

A program in this language can contain one or more declarations followed by an expression. This expression, like the previous version of our language (541exp), is the entry point of the program. There is only one type of top-level declaration in this language, a class declaration.

There is only one declaration form that may appear at the top level of a 541EXP ОО program: class declaration. A class declaration may not be nested inside another class declaration. Thus a class declaration is the keyword class followed by the class name c.

A class may declare several fields (field*), and methods (meth*). Field declarations are written with a type name (t), giving the field’s type, followed by a field name. Methods also have a C++ or Java-like syntax, although their body is an expression.
prog ::= decl* e
decl ::= class c { fdecl* mdecl* }
fdecl ::= t var ;
mdecl ::= t m ( form* ) { e }
form ::= t var
t ::= c | int | boolean
e ::= n | true | false | var | loc | null
    | e == e | e != e | e > e | e < e | e >= e | e <= e | ! e | e & e | e ‘ ’ e | e + e | e - e | e * e
    | e ; e | if ( e ) { e } | else { e } | while ( e ) { e }
    | e . m ( e* ) | new c() | e . f | e . f = e ; e | form = e ; e

n ∈ N, the set of numeric, integer literals
var ∈ {this} ∪ V, V is the set of variable names
c ∈ C, the set of class names
m ∈ M, the set of method names
f ∈ F, the set of field names

Figure 4: Abstract syntax with Classes

B.2 Adding Object-oriented Expressions

We have added several standard object-oriented expressions to the language. First, we need a way to construct objects out of classes. Thus the object construction expression (new c()). We also need to access fields inside classes, and thus the field dereference expression (e.f). Furthermore, it would be useful to assign new values to a field and thus the field assignment expression (e1.f = e2).

Finally, it would be useful to remember that all methods now belong to class declarations, thus in their invocation “receiver” objects must be named. A “receiver” is the object whose methods are being called. Thus we have changed the syntax of the method call expression to e0.m(e*). Here, e0 must evaluate to an object otherwise there will be an error in the method call expression.

Exercise B.1 In 541EXP_OO we do not have the following three constructs, whereas Java does.

- Abstract methods
- Object initialization methods (commonly known as constructors)
- Interfaces

1. Does having these constructs adds additional expressive power to the language? In other words, are these constructs syntactic sugars?

2. For each construct, if you answer yes, demonstrate using a simple Java program using the construct that an equivalent program is hard to write in 541EXP_OO.

3. For each construct, if you answer no, demonstrate using a simple Java program using the construct that an equivalent program can be easily written with just the features in 541EXP_OO.

4. Furthermore, explain the rationale for adding that construct to the language? (For this part it is ok to use search engines.)
Exercise B.2  In 541EXP\textsubscript{OO} we do not have nested class declarations, whereas Java does provide nested class declarations (known as inner classes). Create a simple example Java source class with a nested inner class with fields and methods in both top-level class and inner class. A ready made Java example for inner class is also available at:

\url{http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html}

which you can also use. Using a Java compiler compile this Java source class to bytecode. Using the Java bytecode disassembler (try javap -c -private “class name” on a standard Sun Java distribution) investigate how inner classes are translated to Java bytecode. Based on your observations, are nested class declarations syntactic sugar? If you answer yes, demonstrate it by showing a desugaring strategy for nested class declarations.

C.  Extending Domain and Configurations to Include Objects

We need a way to store objects in memory. Objects are constructed from classes and their behavior depend on two parts: fields and methods. Fields define state of an object and methods define the operations that can be performed on the object. However, the operations that can be performed on the object are the same for all objects of the class. Thus it is unnecessary to store several copies of these in memory. We can always just store one copy of all the methods in a table and retrieve that copy with some key. This is precisely what we will be doing in the semantics.

We will keep a representation of all class declarations in the program in the form of a table called class table ($CT$). We will use the class names as keys in this table. So $CT(c)$ will give us the entire representation of the class with name $c$. We will assume that this class table is present before the program execution begins. Typically, it will be constructed by the compiler or the virtual machine before running the program.

The issue of name space collision, i.e. two classes in a program with the same name, will be dealt with during construction of this class table. No two class representations may have the same name. Furthermore, in our language we do not have the notion of C++ namespaces, and Java packages. These are syntactic sugar, e.g. “java.util.List” can be desugared as the class name “javutilList” and if done uniformly over fully-qualified name will always give a unique name for the class List. A fully-qualified name in languages with namespace or packages is the full name including the package/namespace name, which can be used to uniquely identify a type without importing other packages. So “java.util.List” is a fully-qualified name, whereas “List” is not. Nevertheless having namespace/packages is helpful for programmers thus most modern languages often have these features.

We will be using this class table ($CT$) to find methods that can be called with an object as the receiver object. We will also use it to find out the type of fields that make up the state of the object. However, each object ($o$) of a class ($c$) may have unique values for a field ($o.f$), which may be different from the value of the same field ($f$) for other object ($o’.f$) of the same class ($c$). Thus, we need to store fields of an object separately.

The object record ($o$) in Figure 5 shows the representation of an object in the store. First, the class name ($c$) is kept around, so that we may be able to look up the class representation from the class table ($CT$). Second, value of each field is kept as map ($F$) from field name to its value. The object record uniquely identifies an object and allow us to call methods with that object as receiver and to access the fields of the object and modify their values.

The values ($v$) in an environment ($\sigma$) are limited to integers, boolean constants, a special location representing unassigned references (null), and locations. Furthermore, only object records can be kept in the store, where they are addressed using locations. We also have a stack of environments ($\rho$) similar to the previous lecture. The stack of environment is modeled mathematically in the form $\sigma + \rho$. This makes the
\[ \Gamma := (e, \rho, \mu) \]
\[ \rho ::= \sigma + \rho | \bullet \]
\[ \sigma ::= \{ var_k : v_k \}_{k \in K}, \quad \text{where } K \text{ is finite, } K \subseteq I \]
\[ \mu ::= \{ loc_k \mapsto sv_k \}_{k \in K}, \quad \text{where } K \text{ is finite} \]
\[ v ::= n | \text{true} | \text{false} | \text{null} | loc \]
\[ sv ::= o \]
\[ o ::= [c, F] \]
\[ F ::= \{ f_k \mapsto v_k \}_{k \in K}, \quad \text{where } K \text{ is finite} \]

“Configurations”
“Stack”
“Environments”
“Stores”
“Values”
“Storable Values”
“Object Records”
“Field Maps”

Figure 5: Domains

topmost environment visible, as the topmost environment is the one that we will be using more often in our semantics to look up values.

\[ e ::= \ldots | loc | \text{under } e \]

Figure 6: Added syntax for tracking locations and frames in semantics

Finally, we will need two new kinds of expressions to keep track of locations and method frames in the semantics. These expressions are shown in Figure 6.

D. Extending Dynamic Semantics

We will now define evaluation rules for object-oriented expressions.

D.1 Object Construction

Informally, object construction proceeds as follows: first we find a unique location in the store, create a new object record at that location in the store, initialize all the fields in the object record to their default value, and finally return this new location, which from there onwards serves as the reference to the object. In practice creating a new object record will be done by allocating enough space in the memory to store the record, but here we will ignore these issues.

In the rest of this section, we will describe this in more detail. The rules makes use of two auxiliary functions defaultValOf to find default value of a type and fieldsOf to find fields of a class.

\[
\begin{align*}
\text{(DEFAULTINT)} & \quad \text{defaultValOf} \left( \text{int} \right) = 0 \\
\text{(DEFAULTBOOLEAN)} & \quad \text{defaultValOf} \left( \text{boolean} \right) = \text{false} \\
\text{(DEFAULTCLASSTYPE)} & \quad e \in \text{dom}(CT) \\
& \quad \text{defaultValOf} \left( c \right) = \text{null}
\end{align*}
\]

Figure 7: Auxiliary functions for default values.

The partial function defaultValOf is shown in Figure 7. This function is defined by splitting its domain into three disjoint sub-domains, when \( t \) is \text{int, boolean} and class type. The cases (DEFAULTINT),
(DEFAULTBOOLEAN), and (DEFAULTCLASSTYPE) handle each of these sub-domains and evaluate to appropriate default values. This could be easily extended to a full range of built-in types commonly available in modern languages such as float, double, long, short, unsigned and signed variations of integers, etc. The premise of the (DEFAULTCLASSTYPE) checks whether the type being looked up is in the domain of the class table (CT), which contains all classes in the program and their representations. For other values of t, defaultValOf is undefined.

The function fieldsOf evaluates to a set containing tuples representing the field names and their types for a class identified by class name c. The function typeOfField evaluates to a type, which is the declared type of the field f in the class declaration c. This is useful when we are only interested in the declared type of the field and not its name.

\[
CT(c) = \text{class } c \{ \text{field}_1, \ldots, \text{field}_k, \text{meth}^* \}
\]

\[
\text{fieldsOf}(c) = t_i, f_i \mid i \in \{1, \ldots, k\}
\]

\[
CT(c) = \text{class } c \{ \text{field}_1, \ldots, \text{field}_k, \text{meth}^* \} \quad \exists i \in \{1, \ldots, k\}. \text{field}_i = t f
\]

\[
\text{typeOfField}(c, f) = t
\]

Figure 8: Auxiliary functions for field lookup and type.

The rule (NEW) in Figure 9 shows how object construction works. In the premise of this rule we check whether loc is not already in the domain of the store (\(\mu\)). This ensures that we are always creating object records at unique locations.

Evaluation relation: \(\langle \Gamma \rangle \rightarrow \Gamma\)

\[
(\text{NEW}) \quad \begin{array}{c}
\text{loc} \notin \text{dom}(\mu) \\
\mu' = \mu \oplus \{ \text{loc} \mapsto \{c, \{ f \mapsto \text{defaultValOf}(t) \mid (t, f) \in \text{fieldsOf}(c)\}\}\}
\end{array}
\]

\[
\langle \text{new } c(), \rho, \mu' \rangle \leftarrow \langle \text{loc}, \rho, \mu \rangle
\]

Figure 9: Small-step operational semantics for object construction

In this rule we create a new store (\(\mu'\)), which is created from the old store (\(\mu\)) by adding a new mapping from the new location loc to an object record. This new object record has the same class name c as in the new expression. Furthermore, we create a field map by saying that for each field in the fields of the class c, create a mapping in the field map from its name to its default value. All fields in the class are obtained using the auxiliary function fieldsOf that we defined in Figure 8 and default values of the fields (depending upon their type) is obtained using the defaultValOf function defined previously in Figure 7. The resulting configuration contains the location loc as the result of evaluating the new expression and the modified store \(\mu'\). Note that unlike stack, where we keep track of whether a new environment has been pushed on the stack, we are not keeping track of the modification to the store in these rules. As a result, stores allow expressions to have side-effects that are visible to other expressions in the program.

Exercise D.3 In 541EXP ОО object construction differs from languages such as Java, C++, C#, where typically evaluation of the new expression also results in an initialization method typically called “constructor” being run. Extend 541EXP ОО to have “constructors”. A strategy that uses desugaring (although valid) is not an acceptable answer to this exercise.
Exercise D.4  In $541EXP_{OO}$ object construction always allocates an object record in the store. In some cases, it is desirable to allocate objects on the stack. Such allocation strategy, for example, allows objects to be deallocated automatically, if they are not referred to elsewhere in the program. Checking whether there are going to be references to an object elsewhere in a program can be done using a program analysis technique known as escape analysis [1, 6]. Such allocation strategies reduces the overhead of garbage collection in modern languages. Modify the syntax and semantics of $541EXP_{OO}$ to allow stack allocation of objects by introducing an alternative form of \texttt{new} expression \texttt{snew}.

Exercise D.5  An interesting addition to $541EXP_{OO}$ would be to add an expression \texttt{isolate(e)} that evaluates \(e\) in an “isolated” store. Such store is isolated in the sense that objects created in that store are not reflected in the main store when evaluation of \(e\) reduces to a value. From the perspective of concurrency \texttt{isolate(e)} will be useful because it can be done concurrently. Modify the syntax and semantics of $541EXP_{OO}$ to add \texttt{isolate(e)}.

D.2 Object-oriented Method Call Expression (call by value)

In this section, we study method calls in our object-oriented language. Much of the intuition is similar to the function calls that we have already looked at.

Evaluation relation: \(\Gamma \rightarrow \Gamma\)

(EVAL RECEIVER)

\[
\langle \epsilon_0, \rho, \mu \rangle \rightarrow \langle \epsilon'_0, \rho', \mu' \rangle \quad \langle \epsilon_0.m([e_1, \ldots, e], \rho, \mu) \rangle \rightarrow \langle \epsilon'_0.m([e_1, \ldots, e], \rho', \mu') \rangle
\]

(EVAL ARG)

\[
\text{loc} \in \text{dom}(\mu) \quad \langle e_i, \rho, \mu \rangle \rightarrow \langle e'_i, \rho', \mu' \rangle \quad \langle \text{loc}.m([e_1, \ldots, e]), \rho, \mu \rangle \rightarrow \langle \text{loc}.m([e'_1, \ldots, e]), \rho', \mu' \rangle
\]

(CALL)

\[
\text{loc} \in \text{dom}(\mu) \quad \langle e, F \rangle = \mu(\text{loc}) \quad \langle e, m(t_1 \; \text{var}_1, \ldots, t_n \; \text{var}_n)(e) \rangle = \text{lookUp}(e, m) \quad \sigma = \{\text{var}_i : v_i \mid 1 \leq i \leq n\} \cup \{\textbf{this} : \text{loc}\} \quad \rho = \sigma + \rho \\
\langle \text{loc}.m([v_1, \ldots, v_n]), \rho, \mu \rangle \rightarrow \langle \text{under} \; e, \rho', \mu \rangle
\]

(UNDER C)

\[
\langle e, \rho, \mu \rangle \rightarrow \langle e', \rho', \mu' \rangle \quad \langle \text{under} \; e, \rho, \mu \rangle \rightarrow \langle \text{under} \; e', \rho', \mu' \rangle \quad \langle \text{under} \; v, \rho, \mu \rangle \rightarrow \langle v, \rho, \mu \rangle
\]

Figure 10: Small-step operational semantics for object-oriented method call by value

The rules (EVAL RECEIVER) and (EVAL ARG) are congruence rules that say that the evaluation of a method call proceeds by first evaluating the receiver object to a location in the store (\text{loc}) and then each argument expression to value. The order of evaluation of arguments is not defined. The rule (CALL) describes the
\begin{align*}
\text{LOOKUp} & \\
\text{CT}(c) = \textbf{class} & \ c \ \{ \text{field}^* \ \text{meth}_1, \ldots, \text{meth}_k \} \quad \exists i \in \{1, \ldots, k\}. \text{meth}_i = t \ m(\text{var}_1, \ldots, \text{var}_n)\{e\}
\end{align*}

Figure 11: Auxiliary Function for Looking up Method Definitions.

evaluation of method call expression. In the premise of the rule, we first check if the location is valid location in the store ($\mu$). We then retrieve the object record $[c.F]$ stored at that location, which gives us the class name $c$ for the receiver object. We then use this class name and the method name that we already know from the starting configuration of the program to look up the body of the method. This is accomplished by the auxiliary method \textit{lookUp}. This function is defined in Figure 11.

The premise of the $\textit{lookUp}$ function checks whether the class name exists in the domain of the class table, and it is mapped to a given class representation, and in that class representation there exists a method ($\text{meth}_i$) with the same name as $m$. If the premise is true, $\textit{lookUp}$ evaluates to the 2-tuple $(c, t m(\text{var}_1, \ldots, \text{var}_n)\{e\})$. Note that current version of $\textit{lookUp}$ does not account for two methods having the same name, i.e. method overloading, which is a common and useful feature in modern object-oriented languages. Overloading of methods can be thought of as a syntactic sugar. It can be desugared as follows: for all overloaded methods with the name $m$, replace its name with an auto-generated name $m_i$, where $i$ is unique. For all calls to $m$ in the program check if number of arguments and their types agree with $m_i$ for some $i$ and replace call to $m$ with call to $m_i$. Nevertheless it is important to have this feature as an integral part of a modern language, which allows either facilities for creating libraries or classes to be dynamically loaded as in Java.

Finally in the premise of the rule (\textit{Call}), we create a new environment ($\sigma$) that contains mapping from formal parameters to actual arguments and an additional mapping from \texttt{this} (the special variable that refers to the current object) to the receiver object $loc$. This environment is added on top of the existing stack ($\rho$) to yield a new stack ($\rho'$). The conclusion of the rule says that if all of the premise conditions are true, resulting configuration of the program is $\langle \texttt{under} \ e, \rho', \mu \rangle$, where $e$ is the body expression of the method being called. The body expression is wrapped in an \texttt{under} expression to remember to pop off the stack when the evaluation of the body results in a value. This is shown in the rule (\texttt{under}).

### D.3 Object-oriented Field-related Expressions

In this section, we study field-related expressions in $541EXP_{OO}$. In particular, we give semantics to assignment to a field and retrieving the value of a field.

In Figure 12 the rules (\texttt{GetEvaluateReceiver}), (\texttt{SetEvaluateReceiver}), and (\texttt{SetEvaluateArgs}) are congruence rules. The rule (\texttt{Get}) defines dereferencing a field, which results in a value and the rule (\texttt{Get}) defines assignment to a field. In both cases, we first check if $loc$ is a valid location, find the object record stored at $loc$. The rule (\texttt{Get}) says that if the value corresponding to the name $f$ in the field map $F$ is $v$ then the resulting configuration of the program contains the value, i.e. $loc.f$ reduces to a value $v$, in place of the expression and the stack and store remain unchanged. The rule (\texttt{Set}) says that if a new store can be created, where the location $loc$ points to a new object record, where in the new object record field $f$ maps to the new value $v$, then $loc.f = v$ reduces to a value $v$ with the new store and the stack remains unchanged.
Evaluation relation: \( \leadsto : \Gamma \rightarrow \Gamma \)

\[
\begin{align*}
\text{GETEVALRECEIVER} & \quad \langle e, \rho, \mu \rangle \leadsto \langle e', \rho', \mu' \rangle \\
\text{SETEVALRECEIVER} & \quad \langle e_0, \rho, \mu \rangle \leadsto \langle e_0', \rho', \mu' \rangle \\
\text{SETEVALARG} & \quad \langle e_1, \rho, \mu \rangle \leadsto \langle e_1', \rho', \mu' \rangle \\
\text{GET} & \quad \text{loc} \in \text{dom}(\mu) \quad \text{class} \quad \mu' = \mu \cup \{\text{loc} \mapsto [c.F \oplus (f \mapsto v)]\} \\
\text{SET} & \quad \text{loc} \in \text{dom}(\mu) \quad \text{class} \quad \mu' = \mu \cup \{\text{loc} \mapsto [c.F \oplus (f \mapsto v)]\} \quad \langle \text{loc} = v; e_2, \rho, \mu \rangle \leadsto \langle e_2, \rho, \mu' \rangle
\end{align*}
\]

Figure 12: Small-step operational semantics for field-related expressions

E. Extending 541EXP\textsubscript{OO} with Inheritance

In Section A, we discussed the benefits of inheritance. It enables reuse of existing classes in programs in that it allows new classes to be defined as an increment to existing classes. In this section, we will extend 541EXP\textsubscript{OO} to have inheritance.

\[
\text{prog} ::= \text{decl}^* \ e \\
\text{decl} ::= \text{class} \ c \ \text{extends} \ d \ \{ \ f\text{decl}^* \ m\text{decl}^* \} \\
\text{e} ::= \ldots \ | \text{cast} \ c \ e
\]

\[
\begin{align*}
n & \in \mathcal{N}, \text{ the set of numeric, integer literals} \\
\text{var} & \in \{\text{this}\} \cup \mathcal{V}, \mathcal{V} \text{ is the set of variable names} \\
c, d & \in \mathcal{C}, \text{ the set of class names} \\
m & \in \mathcal{M}, \text{ the set of method names} \\
f & \in \mathcal{F}, \text{ the set of field names}
\end{align*}
\]

Figure 13: Abstract syntax with Classes and Inheritance

Figure 13 shows the abstract syntax as an increment of the syntax in Figure 4. We have now changed the class declaration. Like before, a class declaration may not be nested inside another class declaration. But now a class has exactly one superclass, named in its extends clause. Thus a class declaration is the keyword class followed by the class name \( c \) followed by the keyword extends followed by the name of the superclass \( d \). We have also added another expression for casting. The expression \text{cast} \ c \ loc \ starts\ treating\ the\ object\ record\ pointed\ to\ by\ the\ location\ as\ an\ object\ of\ type\ c.\

F. Dynamic Semantics with Subclassing

The domains of this extended language remains the same. However, we do need to keep track of the inheritance relationship in the class table (\( CT \)). Thus, we will modify class table to also store inheritance relationship.
F.1 Object Construction with Inheritance

Much of the object construction remains unchanged in this extension as well. However, in the rule (NEW) we also need to allocate the inherited fields in the object record. Note that we retrieved the fields using the auxiliary function fieldsOf. Thus changing this function to compute the entire set of fields that include those in the named class \( c \) and its super class \( d \) (recursively) will be sufficient.

\[
\begin{align*}
CT(c) &= \textbf{class } c \textbf{ extends } d \{ field_1, \ldots, field_k, \text{meth}\} \\
superFields &= \text{fieldsOf}(d) \quad \text{allFields} = \text{superFields} \uplus \{ t_i f_i \mid i \in \{1, \ldots, k\} \} \\
\text{fieldsOf}(c) &= \text{allFields}
\end{align*}
\]

**Figure 14:** Auxiliary functions for field lookup and type with subclassing.

Figure 14 shows the modified definition of this function. It says that if the fields in the current class is given and fields of the super class \( d \) can be computed, by recursively applying fieldsOf on \( d \), then fieldsOf(c) evaluates to the overriding union of the set of fields in the super class with the set of fields in the current class. Note that the definition of \( \uplus \) ensures that the most specific fields are selected in allFields. We have now defined \( \text{typeOfField} \) in terms of \( \text{fieldsOf} \), which means that it also accounts for inheritance.

F.2 Method Call with Inheritance

Similar to object construction, the evaluation rule for method call will also remain unchanged even in the presence of inheritance. However, we will have to modify the auxiliary function lookUp to retrieve us the correct method body expression.

\[
\begin{align*}
\text{(lookUp)} & \quad \text{CT}(c) = \textbf{class } c \textbf{ extends } d \{ field^* \text{meth}_1, \ldots, \text{meth}_k \} \\
& \quad \exists i \in \{1, \ldots, k\}. \text{meth}_i = t m(var_1, \ldots, var_n)\{e\} \\
& \quad \text{lookUp}(c, m) = (c, t m(var_1, \ldots, var_n)\{e\})
\end{align*}
\]

\[
\begin{align*}
\text{(lookUp)} & \quad \text{CT}(c) = \textbf{class } c \textbf{ extends } d \{ field^* \text{meth}_1, \ldots, \text{meth}_k \} \\
& \quad \nexists i \in \{1, \ldots, k\}. \text{meth}_i = t m(var_1, \ldots, var_n)\{e\} \\
& \quad \text{lookUp}(d, m) = (d', t m(var_1, \ldots, var_n)\{e\}) \\
& \quad \text{lookUp}(c, m) = (d', t m(var_1, \ldots, var_n)\{e\})
\end{align*}
\]

**Figure 15:** Auxiliary functions for looking up methods with subclassing.

The modified definition of this function is shown in Figure 15. First rule in this figure is for the case when the method \( m \) that we are currently looking up is defined in the class \( c \). In that case, this function also evaluates to the same result as the previous version of lookUp. Second case is more interesting as this applies when the method \( m \) is not defined in the current class. The condition \( \nexists i \in \{1, \ldots, k\}. \text{meth}_i = \)
In both functions \texttt{lookUp} and \texttt{fieldsOf} we have used recursive definitions. For these to terminate, checking the following two conditions must be satisfied.

1. We must ensure that there are no cycles in the inheritance hierarchy. Such cycles can be created by inheritance relationships such as \texttt{class c extends d \{..\}, class d extends e \{..\}}, and \texttt{class e extends c \{..\}}. This creates a cycle of c\rightarrow d \rightarrow e. This is usually detected during parsing of the program. Such programs are flagged as erroneous by most decent compilers of object-oriented languages.

2. We must also have the notion of a class in the language that is the super class of all classes. In the languages such as Java, C\#, Object (with varying capitalizations) is such top super class. Thus we can think of \texttt{class c \{..\}} as the syntactic sugar for \texttt{class c extends Object \{..\}} in Java and C\#. The C++ language has no such notion, i.e. it explicitly allows for multiple top-level classes. In the former case, we can add another rule to \texttt{lookUp} and \texttt{fieldsOf} that causes it to terminate when it reaches the class \texttt{Object}. In the latter case, we can another rule to \texttt{lookUp} and \texttt{fieldsOf} that handles the special case \texttt{class c \{..\}}.

With these two additional constraints adding the new \texttt{lookUp} function is sufficient to extend the dynamic semantics to handle inheritance between classes.

**F.3 Cast Expression**

Evaluation of the cast expression checks if location \texttt{loc} is in the store and the class name stored in the object record (\texttt{c'}) is a subtype of the cast type (\texttt{c}). The notation \texttt{c' \preceq c} checks the subtyping relationship. If both these conditions are true, the cast expression evaluates to \texttt{loc}. No changes to the stack and store are made as a result of the evaluation of the cast expression. The sole purpose of adding this is to help with the static semantics. We will study this in more detail later.

Evaluation relation: \[\langle \text{cast c loc, } \rho, \mu \rangle \leftarrow \langle \text{loc, } \rho, \mu \rangle\]

Figure 16: Small-step operational semantics for cast expressions

**Exercise F.6** \texttt{541EXP}_{OO} does not have static methods. Extend the syntax and semantics of \texttt{541EXP}_{OO} to add static methods.

**F.4 Exceptional Cases**

So far we have been ignoring exceptional cases in our semantics. For example, an expression of the form \texttt{true.f} or \texttt{2.f} or \texttt{null.f} does not have a corresponding rule in our dynamic semantics. Field get expressions are only valid when the receiver object is a valid location in the store. Thus our semantics will be stuck for these states. Not good!
A large number of such cases involve type mismatch. For example, \texttt{true.f} or \texttt{2.f} is clearly wrong because \texttt{true} and \texttt{2} are not reference types, thus it is clearly nonsensical to apply a get expression to them. We can rule them out statically using a type system (and we will do precisely that in the next lecture). The case \texttt{null.f} is special, since we won’t be able to rule out that statically. Thus we must explicitly account for them in our dynamic semantics. To do that we first need to add exceptions to our language. Following figure shows the added syntax. The nonterminals and terminals not defined here are same as before.

\[ e ::= ... | \texttt{ClassCastException} | \texttt{NullPointerException} \]

The dynamic semantics for such exceptional cases is shown below.

\[
\begin{align*}
\text{(NCALL)} & \quad \langle \texttt{null.m}(v_1, \ldots, v_n), \rho, \mu \rangle \mapsto \langle \texttt{NullPointerException}, \bullet, \mu \rangle \\
\text{(NGET)} & \quad \langle \texttt{null.f}, \rho, \mu \rangle \mapsto \langle \texttt{NullPointerException}, \bullet, \mu \rangle \\
\text{(NSET)} & \quad \langle \texttt{null.f = v}; e, \rho, \mu \rangle \mapsto \langle \texttt{NullPointerException}, \bullet, \mu \rangle \\
\text{(NCAST)} & \quad \langle \texttt{cast t null}, \rho, \mu \rangle \mapsto \langle \texttt{null}, \rho, \mu \rangle \\
\text{(XCAST)} & \quad [c.F] = S(loc) \quad c \not\approx t \\
\quad \langle \texttt{cast t loc}, \rho, \mu \rangle \mapsto \langle \texttt{ClassCastException}, \bullet, \mu \rangle
\end{align*}
\]

G. Evaluation Contexts

So far we have developed the semantics that explicitly includes all rules for giving an order to expression evaluation. Recall that these rules are called congruence rules. In a formal semantics complex enough, such as the one we are currently developing, these rules by themselves can take up a whole lot of space without adding any significant value to the discussion. The notion of evaluation context that we have developed before helps with that. In this section, we will express the rules for evaluating object-oriented expressions using evaluation contexts. Let us first define all such context for our language.

\[ E ::= - | E.m(e\ldots) | v.m(v\ldots E e\ldots) | \texttt{cast t E} \\
\quad | E.f | E; e | E.f=e | v.f=E | t \texttt{var=E}; e | \under E \]

This is the usual way of presenting semantics (due to Wright and Felleisen [8]), as a set of evaluation contexts \( E \) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context. This two-part presentation avoids the need for writing out standard recursive rules and has the advantage of more clearly presenting the order of evaluation. Figure 19 defines evaluation contexts.
contexts, and hence the order of evaluation. Like before, this language also uses a strict leftmost, innermost evaluation policy, which thus uses call-by-value.

We can now present the one-step reduction relations in this form. Notice the absence of congruence rules in this version of the semantics.

Evaluation relation: \( \Gamma \rightarrow \Gamma \)

\[
\begin{align*}
\text{(NEW)} & \quad \text{loc} \notin \text{dom}(\mu) & \mu' = \mu \oplus \{\text{loc} \mapsto \{c.\{f \mapsto \text{defaultValOf}(t) \mid (t, f) \in \text{fieldsOf}(e)\}\}\} \\
& \quad \langle E[\text{new} \, c()], \rho, \mu \rangle \rightarrow \langle E[\text{loc}], \rho, \mu' \rangle \\
\text{(CALL)} & \quad \text{loc} \in \text{dom}(\mu) & [c.\, F] = \mu(\text{loc}) \quad (c, m(t_1 \, \text{var}_1, \ldots, t_n \, \text{var}_n)\{e\}) = \text{lookUp}(c, m) \\
& \quad \sigma = \{\text{var}_i : v_i \mid 1 \leq i \leq n\} \uplus \{\textbf{this} : \text{loc}\} \quad \rho' = \sigma + \rho \\
& \quad \langle E[\text{loc}.m(v_1, \ldots, v_n)], \rho, \mu \rangle \rightarrow \langle E[\text{under} \, e], \rho', \mu \rangle \\
\text{(UNDER)} & \quad \langle E[\text{under} \, v], \sigma + \rho, \mu \rangle \rightarrow \langle E[v], \rho, \mu \rangle \\
\text{(GET)} & \quad \text{loc} \in \text{dom}(\mu) & [c.F] = \mu(\text{loc}) \quad v = F(f) \\
& \quad \langle E[\text{loc}.f], \rho, \mu \rangle \rightarrow \langle E[v], \rho, \mu \rangle \\
\text{(SET)} & \quad \text{loc} \in \text{dom}(\mu) & [c.F] = \mu(\text{loc}) \quad \mu' = \mu \oplus \{\text{loc} \mapsto \{c.F \oplus (f \mapsto v)\}\} \\
& \quad \langle E[\text{loc}.f = v ; e_2], \rho, \mu \rangle \rightarrow \langle E[e_2], \rho, \mu' \rangle \\
\text{(VAR)} & \quad v = \sigma(\text{var}) \\
& \quad \langle E[\text{var}], \sigma + \rho, \mu \rangle \rightarrow \langle E[v], \sigma + \rho, \mu \rangle \\
\text{(DEF)} & \quad \sigma' = \sigma \oplus \{\text{var} \mapsto v\} \\
& \quad \langle E[t \, \text{var} = v ; e], \sigma + \rho, \mu \rangle \rightarrow \langle E[e\, ; \text{under} \, e], \sigma' + \sigma + \rho, \mu \rangle \\
\text{(SKIP)} & \quad \langle E[v ; e], \rho, \mu \rangle \rightarrow \langle E[e], \rho, \mu \rangle \\
\text{(NCALL)} & \quad \langle E[\text{null}.m(v_1, \ldots, v_n)], \rho, \mu \rangle \rightarrow \langle \text{NullPointerException}, \bullet, \mu \rangle \\
\text{(NGET)} & \quad \langle E[\text{null}.f], \rho, \mu \rangle \rightarrow \langle \text{NullPointerException}, \bullet, \mu \rangle \\
\text{(NSET)} & \quad \langle E[\text{null}.f = v ; e], \rho, \mu \rangle \rightarrow \langle \text{NullPointerException}, \bullet, \mu \rangle \\
\text{(NCAST)} & \quad \langle E[\text{cast} \, t \, \text{null}], \rho, \mu \rangle \rightarrow \langle E[\text{null}], \rho, \mu \rangle \\
\text{(XCAST)} & \quad [c.F] = S(\text{loc}) \quad c \notin t \\
& \quad \langle E[\text{cast} \, t \, \text{loc}], \rho, \mu \rangle \rightarrow \langle \text{ClassCastException}, \bullet, \mu \rangle
\end{align*}
\]

Figure 20: Small-step Operational Semantics of Object-oriented Expressions
References


