CS422 - Programming Language Design

Defining a Typed Object-Oriented Language

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We start investigating the relationship between objects and types in programming languages. The main idea is that an object `o` can be thought of having a type `c`, if an only if `c` is either the class for which `o` was created as an instance or a superclass of that class.

In order the make our OO language more interesting from the type checking point of view, we extend `OO-FUN` with several features which can be encountered in many other OO languages, such as

- *type/class declarations*,
- *abstract classes*,
- *casting*, and
- *subtype polymorphism*.

As usual, before defining our type checker rigorously we need to first understand all these features and also how the type checking procedure will work.

An Example in Typed `OO-FUN`

The following OO program implements trees that can have binary nodes and leafs storing integers, together with methods `sum` and `equal`.

Both classes `node` and `leaf` will extend the following *abstract class*:

```
abstract class tree {
  method int initialize() { 1 }
  abstractmethod int sum()
  abstractmethod bool equal(tree t)
}
```

Classes which are not abstract are called *concrete*. The major difference between abstract classes and concrete classes is that *objects can be created only for concrete classes*. An abstract class can contain *abstract methods*, which do not have a body, but can also contain *concrete* methods, which do have a body. All methods, both abstract and concrete, declare the “types” of their arguments as well as the types of their returned values.

The role of abstract classes is to provide static information about their future concrete subclasses. For example, the tree abstract class says that each concrete subclass of `tree` is supposed to override the methods `sum` and `equal`.

Let us now declare a first concrete subclass of `tree`, staying for binary nodes, which has two fields `left` and `right`:

```
class node inherits tree {
  field tree left
  field tree right
```

Notice that the fields are declared with a type, which is the abstract class `tree`. This says that in any instance of the class `node`, the two fields are expected to contain objects that are instances of `tree` (this is not possible though because `tree` is an abstract class so it cannot have instance objects) ... or subclasses of it. This leads us to the following very important concept in OO programming languages:

> *Subtype polymorphism:* An object whose creation class is `c` can be used in any context where an object of class `c` or any of its superclasses is expected.

Therefore, any objects of subclasses of `tree`, including `node`, will be allowed to be assigned to these fields.

Let us next define the methods of `node`. When one creates a node object, one most likely wants to also initialize its `left` and `right` fields to other objects. Thus, one would want to have a binary `initialize` method as follows:

```
method void initialize(tree l, tree r) {
  left := l ;
  right := r
}
```

Due to subtype polymorphism, `initialize` can be called with any objects of subclasses of `tree` as arguments. Notice, however, that `initialize` in `node` has a different arity than the same method in its superclass, `tree`! We will therefore allow the `initialize` method to have different types across different classes in the hierarchy, and this creates a type safety leak as we will see later, but this will be the only exception. And there will be no safety leak if one *does not use* `initialize` explicitly (but only implicitly, when creating new objects), which can be easily checked statically.

The following three methods are straightforward:

```
method tree getleft() { left }
method tree getright() { right }
method int sum() {
  (send left sum()) + (send right sum())
}
```

Note that `sum` overrides the corresponding abstract method in `tree`. The next method overrides the other abstract method in `tree`, called `equal`, whose role is to test whether two trees are equal.

Each subclass of `tree` is expected to define its own `equal` concrete method, because what it means for two trees to be equal is dependent on the particular structure of the tree:

```
    method bool equal(tree t) {
      if (t instanceOf node)
      then if send left equal(send (cast t to node) getleft())
           then send right equal(send (cast t to node) getright())
           else false
      else false
    }
  } // end of class tree
```

There are two new language constructs in the method above, namely `instanceOf` and `cast`. Their behavior is as follows:

- `<Exp> instanceOf <Name>` evaluates the `<Exp>` to an object and then returns true if and only if that object is an instance of `<Name>` or of any of its subclasses.

- `cast <Exp> to <Name>` evaluates the `<Exp>` to an object and then returns *that object* when it is an instance of `<Name>` or of any of its subclasses, and a runtime error otherwise.

Casting objects is, in my view, a *slightly misleading terminology*. This is because any object has a definite class, set when it was instantiated, and nothing can change that class during the lifetime of the object.

Similarly, we can now define the `leaf` concrete subclass of `tree`:

```
class leaf inherits tree {
  field int value
  method void initialize(int v) { value := v }
  method int sum() { value }
  method int getvalue() { value }
  method bool equal(tree t) {
    if (t instanceOf leaf)
    then value eq (send (cast t to leaf) getvalue())
    else false
  }
}
```

One can now evaluate the following in the context of these classes:

```
main
  let o1 = new node(new node(new leaf(3),
                             new leaf(4)),
                    new leaf(5))
  in [send o1 sum(), if (send o1 equal(o1)) then 100 else 200]
```

A definition of this OO language extending `OO-FUN` can be very easily obtained from the definition of `OO-FUN`. One should first make sure that an object is never created for an abstract class, by checking whether the class specified in a `new` statement is concrete or not. Then one should add syntax and semantics for `<Exp> instanceOf <Name>` and `cast <Exp> to <Name>`.

Exercise 1 *Modify the definition of* `OO-FUN` *to support type declarations and the other features described above. You do not need to check the typing policy at runtime; just make sure that your definition can evaluate programs correctly.*

Type Checking the OO Language

We next focus on the principles underlying our type checking tool that will be formally defined in the next lecture. The role of the type checker is to *ensure that certain kinds of errors will never occur at runtime*. Besides the other errors considered by the existing type checker of `FUN`, we are also considering:

- sending a message to an object which does not provide the corresponding method;
- sending a message with the wrong number and type of arguments;
- creating an object for an abstract class;
- creating an object for a concrete subclass of an abstract class, in which one or more of the abstract methods were not overridden by concrete methods.

Technique: Abstract Interpretation

As we have seen so far in the class, all our definitions have almost the same structure, in the sense that they *give meaning* to the syntactic programming language constructs. One particular kind of meaning that one can associate to an expression is, of course, its value. More generally, the *concrete meaning of an expression can be viewed as a (partial) function* from concrete values associated to its free names into a concrete value associated to itself:

$$\mathit{concreteMeaning}(E) := [\mathit{free}(E) \to \mathit{Value}] \to \mathit{Value}$$

We have rigorously defined the meaning of each expression inductively, where the map $\mathit{free}(E) \to \mathit{Value}$ was provided implicitly as part of the state infrastructure.

In the context of our typed `FUN`, what we did was to think of a type as "the value" of an expression. And what we really defined inductively over the structure of expressions was a type specific meaning of expressions:

$$\mathit{typeMeaning}(E) := [\mathit{free}(E) \to \mathit{Type}] \to \mathit{Type}$$

Therefore, the *concrete values bound to names were abstracted away by their types*. It is then not surprising that we used the same definitional structure for defining a type checker as we used for defining the semantics of the language. And in general, almost if not all software analysis tools should follow the same pattern.

The technique underlying all these definitions is called *abstract interpretation*, It is typically defined quite mathematically, and this will be done in CS522 next semester, but in this course we keep it simple and just describe it intuitively in the context of our programming languages:

- Select a set of *abstract values* that you are interested to analyze;
- Define the *abstract meaning* of a program as a map

$$absMeaning(E) := [free(E) \to absValue] \to absValue$$

Once one fixes the abstract values of interest and understands *how they propagate*, then one can state it formally following the same pattern used to define the semantics of the programming languages. The problem may sometimes be quite non-trivial, because in order to apply the abstraction on a certain language construct one may need to do quite involved reasoning (think of type inference, for example).

Types of Objects

So we now have to define the abstract values of our new analysis tool, the OO type checker. While the types of the other expressions are like before, that is, elements in the infinite set of well formed terms over the type constructors, the types of objects are to be considered differently.

What is the type of an object? The class that the object was created as an instance for? The answer is certainly no, because otherwise we could not handle subtype polymorphism. The type of an object, which is its *abstract value* in what follows, is a *set of possible classes*, namely all those for which it can be an instance of (in the sense of `instanceOf`). Thus, we say that an object `o` *has type* `c` if and only if its instance class is either `c` or a subclass of `c`.

What we will do is to follow the structure of the executable semantics and adapt it to “evaluate” expressions to OO types.

Processing Classes

The typing policy that must be enforced when processing the class declarations is the following:

- Each *concrete class has only concrete methods*;
- Each method overriding a superclass method *has the same type* as the overridden method. This is because one cannot know statically to which method in the chain of classes of an object a method invocation refers to (because of subtype polymorphism), so the best one can do is to enforce that overriding methods preserve the types (modulo subtype polymorphism, of course). One exception here will be the `initialize` method, which is allowed to have different types from superclasses to subclasses.

Processing Methods

When processing methods, one must check that the following typing policy is not violated:

- Nothing to worry about in the case of abstract methods;
- The concrete methods have to be type checked to make sure that they have their declared type. In order to do it, the typed parameters are added to the environment and then the body of the method is evaluated. The obtained type must be either the declared type or a subtype of it.

Processing Expressions

Nothing changes with respect to type checking the already known expressions, except that one should now consider the *subtype polymorphism* aspect overall. More precisely, the declared type of a bound name can be equal to or a supertype of the actual type obtained by type checking (or evaluating) the bound expression.

Processing the `new` Construct

When processing `new`, we will check the following policy:

- The corresponding class should have been declared;
- The corresponding class is concrete;
- The initialization is *type safe*, that is, the number and the types of arguments match the declared ones, modulo the policy of subtype polymorphism;
- The returned type after evaluating the `new` construct is the class for which the instance object was created.

Processing Method Invocation

The typing policy for method invocation is the following:

- The expression to which the message is sent must evaluate to an object type, that is, its corresponding class;
- After retrieving that class' info, check that the method's name is accessible from that class; we don't have to worry about that method being abstract, because we already know that we are not in an abstract class and that each abstract method has been overridden;
- After retrieving the corresponding method declaration, check that its declared type matches the type of the argument expressions, by type checking them; this has also to be done modulo the subtype polymorphism.

Processing `instanceOf` and `cast`

In the case of `instanceOf`, the only thing one needs to do is to check that its first argument evaluates to an object type and that its second argument is defined as a class.

The situation is more problematic for `cast`, because it is *impossible* to guarantee statically that it will never generate a runtime error! Why? The best we can do is to *reject casts that will always fail*.

How can we detect those casts that will always fail? This is relatively easy, by *intersecting* the set containing the type class of the object to which its first argument evaluates (if it does not evaluate to an object then issue a type error) together with its subclasses, with the set containing the second class and its subclasses. If the intersection is *empty* then the cast will always fail, so issue a type error. Otherwise we cannot say anything.