How to extend Sample to a new language

Pietro Ferrara
ETH Zurich
pietro.ferrara@inf.ethz.ch

June 21, 2010

Abstract

Sample works on a simple language aimed at being expressive in order to be applied to all the modern object oriented programming languages. This document explains the main components of this simple language, and how to translate a language into this representation.

1 Introduction

Package OORepresentation contains all the classes that have to be extended and instantiated in order to apply Sample to a language. Sample is implemented in Scala. Thus you can easily extend it to other languages using Scala but also Java, as Scala is compiled into Java bytecode and it is quite easy to interface Java code with Scala libraries.

All the components introduced in this document has been implemented and commented in Scala. The API of Sample contains all the documentation on them. In particular, you can care only about OORepresentation package.

2 Identifiers, Program Points and Types

Three identifiers, i.e. PackageIdentifier, ClassIdentifier, and MethodIdentifier, must be implemented. They are intended to represent identifiers of packages, classes, and methods (usually a string containing its name). You have to take care to implement in particular methods equals and hashCode in order to semantically represent when two identifiers are equals.
ProgramPoint represents univocally a point of the program (e.g. the line, column and class to which such program point belongs).
Type is intended to represent the type hierarchy. It requires to implement all the methods of lattice structure (e.g. lub) that represents the type hierarchy. In addition, it requires to implement some methods (i.e. isObject(), isNumericalType(), isStatic(), getName(),

isBottomExcluding(types : Set[T])) required in order to extract some information from the type system.

3 Sequential Statements

Statement is extended by all the possible existing statements. There are essentially 8 main types of statements:

- Assignment
- VariableDeclaration
- Variable
- FieldAccess
- New
- MethodCall
- NumericalConstant
- Throw

3.1 Statements as Expressions

We choose to adopt a representation as much unified as possible while preserving the expressiveness of our language. Thus we do not distinguish between expressions and statements. The common intuition is that expressions return values (e.g. $3 + \mathsf{variable}$) while statements do not (e.g. $\mathsf{x} = \mathsf{y} + \mathsf{x};$). On the other hand, there may be some cases in which a statement is also an expression (e.g. a method call that returns a value). A generalization of such approach already adopted by some programming languages (like Scala) is to consider statements as a particular case of expressions that return an "empty" value. We followed such approach when designing our simple language.

3.2 Representing Arithmetic Expressions as Method Calls

Usually, arithmetic expressions (as well as other actions like type castings) are natively defined by the programming language. This leads to more complex languages. Indeed, since we want to preserve the simplicity of our language without giving up in terms of expressiveness, we adopt a different solution following the approach of programming languages on which *everything is an object* (e.g. Scala). We represent arithmetic (and other) operators as method calls (e.g. $3 + 2$ is represented by $3.+(2)$), and numerical constants as objects.

3.3 Semantics of native methods

Since we represent all the native operators of a programming language (e.g. arithmetic operators or dynamic type castings) through method calls we also provide a way to define and implement the semantics of such statements. NativeMethodSemantics has to be implemented in order to define such semantics.

4 Control Flow Graph

The last type of statement is ControlFlowGraph. We represent conditional jumps in it. A CFG is a weighted graph whose nodes are list of statements, and where the weight of edges can be a boolean value (meaning that the condition has to be evaluated to true or false in order to cross that edge), or nothing (in the case that it is not a conditional jump, that is, it represents a goto statement). Thus each node can only have two weighted edges (one with true and the other one with false), or one not-weighted edge. In order to check a condition (usually represented by a method call, e.g. x. >= (0)), we need to put it as last statement of the node, and then add edges weighted to true and false, starting from such node and pointing to the node that has been executed if the condition is evaluated to true and false.
Note that considering the CFG as a statement is particularly expressive. For instance, in this way we support statements like x = if(B) then 1 else 2.

5 Method and Class structure

In order to represent the structure of a class, we have three major components:

- MethodDeclaration
- FieldDeclaration
- ClassDefinition

Fields and methods can be added to ClassDefinition through addField and addMethod methods.

6 Using Java

The analyzer has been implemented in Scala. Since Scala is compiled into Java bytecode, you can implement the translation from your language to our simple language using Java. The interfaces will be as follows:

- PackageIdentifier, MethodIdentifier, and ClassIdentifier are interfaces, ProgramPoint is an abstract class, all of them do not require to implement any specific method;

- Type is an interface whose definition is as follows:

```
interface Type<T extends Type<T>> extends Lattice<T>
{
  boolean isBottomExcluding(Set<T> paramSet);
  String getName();
  boolean isStatic();
  boolean isNumericalType();
  boolean isObject();
}
```

- the eight types of sequential statements are implemented as classes whose constructor signatures are as follows

```
public Assignment(ProgramPoint programpoint, Statement<T> left, Statement<T> right)
public VariableDeclaration (ProgramPoint programpoint, Variable<T> variable, T typ,
                                                                    Statement<T> right)
public Variable(ProgramPoint programpoint, String id)
public FieldAccess(ProgramPoint pp, List<Statement<T>> objs, String field, T typ)
public New(ProgramPoint pp, T typ)
public MethodCall(ProgramPoint pp, Statement<T> method, List<T> parametricTypes,
                                        List<Statement<T>> parameters, T returnedType)
public NumericalConstant(ProgramPoint pp, String value, T typ)
public Throw(ProgramPoint programpoint, Statement<T> expr)
```

  Note that all these classes are parameterized on a generic type T that is a subtype of interface Type

- ControlFlowGraph is a class whose interesting methods are the following ones:

```
public void addEdge(int x1, int x2, Option x3)
public void setNode(int x1, Object x2)
public int addNode(Object x1)
```

  Note that methods setNode and addNode receive as parameter an Object as node of the CFG, but it is intended to be a list of statement, i.e. belonging to type List < Statement >. This happens because of type erasure in Java bytecode.

- MethodDeclaration, FieldDeclaration, and ClassDefinition are classes whose constructors are defined as for Scala.

- NativeMethodSemantics is an interface that requires to implement the following method:

```
public abstract <S extends State<S, T, E>, T extends Type<T>, E extends Value<T, E>>
                                        S getSemantics(S paramS, MethodCall<T> paramMethodCall);
```