A Runtime Checker for Java Bytecode

Ersoy Bayramoğlu

December 29, 2007

1 Introduction

Java annotations allow the programmers to pass information to bytecode level without changing the runtime behavior of the program. They can be used to facilitate contracts for methods in the bytecode files. Our tool takes annotated Java bytecode files as input, and produces bytecode files with the corresponding runtime checks placed into the appropriate places. The specification language for contracts includes Java expressions (for arithmetic operations, field accesses, method calls, array dereferences) and quantification over sets. There are several advantages of operating on bytecode, instead of source code. Firstly, the runtime checker can be used for not just Java programs, but also for Scala programs. Because, Scala is also compiled into Java bytecode and Java annotations can be used in Scala programs. Another advantage is, the tool will not be affected from the changes in the language as long as JVM stays the same.

The tool consists of two parts:

- a specification compiler
- a Java program that handles bytecode instrumentation

2 Specification Compiler

The specification compiler was implemented in OCaml. It reads a pre/post conditon or a class invariant from a file, and writes the corresponding Java expression back to a file.

2.1 Specification Language

annotation ::= **expr** ;

expr ::=
  | **expr arith_op expr**
  | **expr compare expr**
  | **expr bool_op expr**
  | range(**expr**,**expr**).**set_op** (*identifier* : *type*=> **expr**)
  | reachable(**expr**,*identifier* : *type*=> List(**prmlst**))
  | **expr**.**set_op**(*identifier* : *type*=> **expr**)
  | (**expr**)
  | ! **expr**
  | **expr**.*identifier*
  | **expr**[**expr**]
  | *identifier*(**prmlist**)
  | **expr**.*identifier*(**prmlist**)

| *identifier*
| *integer*
| - *integer*
| null

arith_op ::=
| +
| -
| *
| /
| %

compare ::=
| ==
| !=
| <
| >
| <=
| >=

bool_op ::=
| &&
| ||
| ->

set_op ::=
| forall
| exists
| filter
| map

prmlst ::=
|
| **prm_aux**

prm_aux ::=
| **expr**
| **expr**,**prm_aux**

According to the grammar, non-boolean expressions can be written. However, they will be rejected by the Java compiler. Only the boolean expressions are valid annotations in this language. Method parameters must be referred by special variables according to their positions: $1, $2, etc. $0 is the special variable for *this*, and $_ is the special variable for the result of the method. If there is a package structure, type of an instance of a class is *package_name.class_name*. Set operations (forall, exists, map, filter) can only be used on HashSet objects. The operator precedence follows the rules of C++.

2.2 Translation

The specifications are translated into Java. Many of the constructs in the specification language are the same as in Java. They are left as they are.
Expressions like p -> q get translated into (!p) || (q).

Set operations (forall, exists, filter, map) are translated into function calls. For example, the expression "`$1.forall(x : Integer => x > 0);`" is translated into "`forall1($1)`". The function forall1 is produced during the translation.

```
public boolean forall1(HashSet _$1){
   java.util.Iterator it = _$1.iterator();
   while(it.hasNext()){
      Integer x = (Integer) it.next();
      boolean test = (x > 0);
      if(!test){
         return false;
      }
   }
   return true;
}
```

Set operations can be nested. The parameters of the function produced for an expression like set_name.set_op(*identifier* : *type* => *statement*) are the free variables of the *statement*, and the set variable. So, if the preconditon is `$1.forall(x:Integer => $2.exists(y:Integer => (x.intValue()>y.intValue()) && (y.intValue()<$3.intValue())));` the following two functions will be produced:
`public boolean forall1(HashSet _$1,HashSet _$2, int _$3);`
`public boolean forall2(HashSet _$2, int x, int _$3);`
The whole expression is translated into just forall1($1,$2,$3), and there will be a call to forall2 inside of forall1.
Map and filter are well know higher order functions, and they are translated in the same way as forall. They apply the given function on the elements of a HashSet and return a new HashSet.
Range expressions use a class named RangeSet.java. It is a simple implementation of the set interface of java.util. Two integer variables are kept as the boundaries of an interval. The iterator for a RangeSet object, ranges over values in this interval. (the two end points belong to the inteval) For translation, consider the following example that states, none of the elements of the array named queue is null.

```
   range(0,length-1).forall(i : Integer => queue[i.getValue()]!=null);
```

This expression will be translated into a function call forall1(), and the following function will be inserted into the file. (length is a private field in the class)

```
public static boolean forall1(){
   java.util.iterator it = new RangeSet(0,(length-1) ).iterator();
   while(it.hasNext()){
      Integer i = (Integer)it.next();
      boolean test = ((queue[i.getValue()]!=null) );
      if(!test){
         return false; }
   }
   return true;
}
```

If the instrumented method is a static method, the auxiliary methods produced will also be static methods. Note that the elements of HashSets and RangeSets cannot have primitive types, and Javassist does not allow autoboxing. So, to use a variable of type Integer, the method getValue() should be used. Reachability expressions are also translated into function calls. The name of a function corresponding to a reachability expression is again "forall"+ a unique integer. To illustrate the translation, consider the following example expression:

```
   reachable($1, x:Tree => List(x.left, x.right))
```

This evaluates to the set of all objects, reachable from $1 along the fields named right and left. The types of the fields must be the same as the type of the local variable x. During the translation, the following function will be produced:

```
public HashSet forall1(Tree _$1){
   java.util.Set set = new java.util.HashSet();
   java.util.Stack stack = new java.util.Stack();
   stack.push(_$1);
   while(!stack.isEmpty()){
      Tree x = (Tree)stack.pop();
      if(x.left!=null&&set.add(x.left)) {
         stack.push(x.left);
      }
      if(x.right!=null&&set.add(x.right)) {
         stack.push(x.right);
      }
   }
   return set;
}
```

3 Bytecode Instrumentation

Method annotations are inserted into the code using the following class named MethodAnnotation.java. Each class has a single invariant, and it can be attached to any one of the methods (preferably to the constructor.) If more than one method contains a class invariant, only the earlier one is considered.

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MethodAnnotation {
   String precond() default "true;";
   String postcond() default "true;";
   String invariant() default "true;";
}
```

We made use of Javassit[1] to carry out the instrumention. Javassist is a class library for editing bytecodes in Java. It enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it. The users can specify inserted bytecode in the form of source code, and Javassist compiles it on the fly.

The method annotations are extracted from bytecode files with the methods supplied by Javassist. After an annotation is extracted, it is written to a file and the specification compiler is called to translate the expression into Java. The specification program translates the expression, and writes the corresponding Java code to another file. The code is read from that file. The auxiliary methods are inserted directly into the class file. If the expression is a precondition, the check for the translated expression is inserted at the beginning of the method body. If it is a postcondtion, the check for the translated expression is inserted using the insertAfter method of Javassist. The checks for class invariants are placed at the beginning and at

[1]http://www.csg.is.titech.ac.jp/~chiba/javassist/

the end of each public method. Note that, because these checks are inserted into the method bodies, private fields of the class can be accessed in the specification.

4 Example

Here is an example Priority Queue class annotated with pre/postconsditions and a class invariant. An infinite loop was discovered during testing if the class.

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

class PriorityQueueElement
{
   public Object ob;
   public int key;
}

public class PriorityQueue
{
   private static PriorityQueueElement[] queue;
   private static int length;
   private static int old_length; // specvar to keep the old values of length
   private static int old_smaxlen; // specvar to keep the old values of queue.length
   private static HashSet spqueue; // specvar
   private static HashSet old_spqueue; // specvar to keep old values of spqueue

@MethodAnnotation(
   precond = "(queue==null)&&(0<=$1);",
   postcond = "(queue!=null)&&(queue.length==$1)&&(length==0)&&(spqueue.isEmpty());",
   invariant = "(queue!=null) -> ((0<=length) && (length <= queue.length) &&
(range(0,length-1).forall(i:Integer => queue[i.intValue()]!=null)) && (range(length,
queue.length-1).forall(i:Integer => queue[i.intValue()]==null))&&range(0,length-1).forall(i:
Integer => range(0,length-1).forall(j:Integer => (queue[i.intValue()]==queue[j.intValue()])->
i.intValue()==j.intValue()))&&(range(0,length-1).forall(i:Integer => (range(0,length-1).forall(j:
Integer => ((j.intValue() == i.intValue()+i.intValue()+1)|| (j.intValue()== i.intValue()+
i.intValue()+2))->queue[i.intValue()].key>=queue[j.intValue()].key))))&&(spqueue.forall(x:
PriorityQueueElement => range(0,queue.length-1).exists(i:Integer => queue[i.intValue()]!=null&&
x.equals(queue[i.intValue()])))&&range(0,queue.length-1).forall(i:Integer => (queue[i.intValue()]
!= null) -> (spqueue.contains(queue[i.intValue()])))));"
)
   public static void init(int len)
   {
      queue = new PriorityQueueElement[len];
      spqueue = new HashSet(); // specvar init
      length = 0;
   }

   @MethodAnnotation(
   precond = "queue!=null;",
   postcond = "$_==length;"
   )
   public static int currLength()
   {
      return length;
   }

   @MethodAnnotation(
   precond = "queue!=null;",
   postcond = "$_==queue.length;"
   )
   public static int maxLength()
   {
      return queue.length;
   }

   @MethodAnnotation(
   precond = "queue!=null;",
   postcond = "($_== (length==0));"
   )
   public static boolean isEmpty()
   {
      return (length == 0);
   }

   @MethodAnnotation(
   precond = "queue!=null;",
   postcond = "($_== (length==queue.length));"
   )
   public static boolean isFull()
   {
      return (length == queue.length);
   }

   @MethodAnnotation(
   precond = "$1>0;",
   postcond = "($_>=0)&&($_<$1)&&( ($1==$_+$_+1) || ($1==$_+$_+2) );"
   )
   private static int parent(int i)
   {
      return (i-1)/2;
   }

   @MethodAnnotation(
   precond = "0<=$1;",
   postcond = "($_>=0)&&($_>$1)&&($_==$1+$1+1);"
   )
   private static int left(int i)
   {
      return (2*i + 1);
   }
```

```
@MethodAnnotation(
precond = "0<=$1;",
postcond = "($_>=0)&&($_>$1)&&($_==$1+$1+2);"
)
private static int right(int i)
{
   return (2*i + 2);
}

private static HashSet remove(HashSet set, Object e) // method for functional remove operation
{
   HashSet temp = new HashSet();
   java.util.Iterator it = set.iterator();
   while(it.hasNext())
   {
      temp.add(it.next());
   }
   temp.remove(e);
   return temp;
}

private static HashSet add(HashSet set, Object e) // method for functional add operation
{
   HashSet temp = new HashSet();
   java.util.Iterator it = set.iterator();
   while(it.hasNext())
   {
      temp.add(it.next());
   }
   temp.add(e);
   return temp;
}

@MethodAnnotation(
precond = "(queue!=null)&&($1!=null)&&(length<queue.length)&&!(spqueue.contains($1));",
postcond = "(length == old_length+1) && (spqueue.equals(add(old_spqueue,$1)));"
)
public static void insert(PriorityQueueElement e)
{
   old_spqueue.clear();
   java.util.Iterator it = spqueue.iterator(); // copy spqueue to old_spqueue
   HashSet o_spqueue = new HashSet(); // local var to keep the old spqueue
   while(it.hasNext())
   {
      o_spqueue.add(it.next());
   }
   int o_length=length; // local var to keep the old value of length
   int i = length;
   length = length + 1;
   while(i > 0 && queue[parent(i)].key < e.key)
```

```
      {
         int p = parent(i);
         queue[i] = queue[p];
         i = p;
      }
      queue[i] = e;
      spqueue.add(e); // update specvar
      old_length = o_length; // update specvar
      old_spqueue=o_spqueue; // update specvar
   }

   @MethodAnnotation(
   precond= "(queue!=null)&&(length > 0);",
   postcond= "($_!=null)&&(spqueue.contains($_))&&(spqueue.forall(x:PriorityQueueElement =>
   $_.key >= x.key));"
   )
   public static PriorityQueueElement findMax()
   {
      return queue[0];
   }

   @MethodAnnotation(
   precond= "(queue!=null)&&(length>0);",
   postcond= "(length==old_length-1)&&($_!=null)&&(old_spqueue.forall(x:PriorityQueueElement
   => $_.key >= x.key))&&(spqueue.equals(remove(old_spqueue,$_)));"
   )
   public static PriorityQueueElement extractMax()
   {
      old_spqueue.clear();
      int o_length = length; // local var to keep the old value of length
      HashSet o_spqueue = new HashSet(); // local var to keep the old spqueue
      java.util.Iterator it = spqueue.iterator(); // copy spqueue to old_spqueue
      while(it.hasNext())
      {
         o_spqueue.add(it.next());
      }
      PriorityQueueElement result = queue[0];
      spqueue.remove(result); // specvar update
      length = length - 1;
      queue[0] = queue[length];
      queue[length] = null;
      if (0 < length) heapify(0);
      old_length=o_length; // update specvar
      old_spqueue=o_spqueue; // update specvar
      return result;
   }

   @MethodAnnotation(
   precond= "((queue!=null) -> ((0<=length) && (length <= queue.length) &&
(range(0,length-1).forall(i:Integer => queue[i.intValue()]!=null)) && (range(length,
queue.length-1).forall(i:Integer => queue[i.intValue()]==null))&&range(0,length-1).forall(i:
```

```
Integer => range(0,length-1).forall(j:Integer => (queue[i.intValue()]==queue[j.intValue()])->
i.intValue()==j.intValue()))&& (spqueue.forall(x:  PriorityQueueElement =>
range(0,queue.length-1).exists(i:Integer => queue[i.intValue()]!=null&& x.equals
(queue[i.intValue()]))) &&range(0,queue.length-1).forall(i:Integer => (queue[i.intValue()]
!= null) -> (spqueue.contains(queue[i.intValue()])))))) && range(0,length-1).forall(k:Integer
=> range(1,length-1).forall(j:Integer => (k.intValue()!=$1) && ((j.intValue() ==
(2*k.intValue()+1)) || (j.intValue() == (2*k.intValue()+2))) -> queue[k.intValue()].key >=
queue[j.intValue()].key))&&range(0,length-1).forall(x:Integer => (($1==x.intValue()+x.intValue()+1)
||($1==x.intValue()+x.intValue()+2))-> ((($1+$1+1 < length)-> queue[x.intValue()].key >=
queue[$1+$1+1].key) && (($1+$1+2 < length)-> queue[x.intValue()].key >= queue[$1+$1+2].key)));",
  postcond= "(queue!=null) -> ((0<=length) && (length <= queue.length) &&
(range(0,length-1).forall(i:Integer => queue[i.intValue()]!=null)) && (range(length,
queue.length-1).forall(i:Integer => queue[i.intValue()]==null))&&range(0,length-1).forall(i:
Integer => range(0,length-1).forall(j:Integer => (queue[i.intValue()]==queue[j.intValue()])->
i.intValue()==j.intValue()))&&(range(0,length-1).forall(i:Integer => (range(0,length-1).forall(j:
Integer => ((j.intValue() == i.intValue()+i.intValue()+1)|| (j.intValue()== i.intValue()+
i.intValue()+2))->queue[i.intValue()].key>=queue[j.intValue()].key))))&&(spqueue.forall(x:
PriorityQueueElement => range(0,queue.length-1).exists(i:Integer => queue[i.intValue()]!=null&&
x.equals(queue[i.intValue()])))&&range(0,queue.length-1).forall(i:Integer => (queue[i.intValue()]
!= null) -> (spqueue.contains(queue[i.intValue()])))))&&(queue!=null)&&(spqueue.equals(
old_spqueue))&&(length==old_length)&&(old_smaxlen==queue.length);"
  )
  private static void heapify(int i)
  {
    int temp_old_smaxlen = queue.length;
    old_spqueue.clear();
    int o_length = length; // local var to keep the old value of length
    java.util.Iterator it = spqueue.iterator(); // copy spqueue to old_spqueue
    HashSet o_spqueue = new HashSet(); // local var to keep the old spqueue
    while(it.hasNext())
    {
      o_spqueue.add(it.next());
    }
    int m = i;
    int l = left(i);
    if (l < length && queue[l].key > queue[i].key)
    m = l;
    int r = right(i);
    if (r < length && queue[r].key > queue[m].key)
      m = r;
    if (m != i)
    {
      PriorityQueueElement p = queue[m];
      queue[m] = queue[i];
      queue[i] = p;
      heapify(m);
    }
    old_length = o_length; // specvar update
    old_spqueue=o_spqueue; // update specvar
    old_smaxlen=temp_old_smaxlen;
  }
```

```
}
```

4.1 Overhead

Type	Function	Parameter	Duration
without checks	insert	100	178000 nanoseconds
with checks	insert	100	171919000 nanoseconds
without checks	insert	1000	163000 nanoseconds
with checks	insert	1000	56987020000 nanoseconds
without checks	extractMax	100	372000 nanoseconds
with checks	extractMax	100	385613000 nanoseconds
without checks	extractMax	1000	6578000 nanoseconds
without checks	extractMax	100	58083781000 nanoseconds

The extractMax function calls the heapify function. The measurements for the calls of extractMax with 1000, were made without the assertions on the heapify function. With the assertions, it takes more than 10 minutes.

5 How to Run

The path to the specification compiler must be written to a file named config.txt. If all the required files are in the same the folder, the program can be run using the command:

java instrument -classpath ".:./javassist.jar" [*class files to be instrumented*]