Software-Modellierung und Verifikation
Informatik 2
Prof. J.-P. Katoen
RWTH Aachen

Priv.-Doz. T. Noll noll@cs.rwth-aachen.de
J. Heinen heinen@cs.rwth-aachen.de
Ch. Jansen christina.jansen@cs.rwth-aachen.de

5. Exercise sheet *Static Program Analysis 2011*

Due Mon, 30. May 2011, *before* the exercise course begins.

Exercise 5.1: **(3 points)**

Perform a *type correctness* analysis for the following Java bytecode Check that the return value is of type C and that the program is type safe. The return value is the reference that remains on the operation stack after termination of the method. The bytecode uses three classes A,B and C that are not related. Register one is initialised with type A and the second with type B.

```
1  load 1
2  iconst 1
3  invoke A m Int C
4  store 0
5  load 0
6  getfield C f Int
7  iconst 0
8  if_icmpeq 1
9  load 0
10 return
```

Exercise 5.2: **(1+3+3 points)**

In Java new objects can only be created by calling a constructor, thus object creation and initialisation is one action. However, in Java bytecode this is not the case. Creation and initialisation are independent (cf. the following code example).

Object creation and access in Java:

```
Point p = new Point(2, 3);
p.print();
```

The corresponding Java bytecode:

```
1  new A                                  // creates a new Point - object
2  dup                                    // duplicates the last stack entry
3  iconst 2                               // pushes the constant value 2 to the stack
4  iconst 3                               // pushes the constant value 3 to the stack
5  invokespecial Point <init>(int,int)    // calls the constructor Point(int, int)
6  astore 4                               // stores the reference in register 4
7  aload 4                                // loads the reference in register 4
```

Because of this fact object initialisation is not guaranteed for byte code and it is important to check that any object is initialised before it is accessed.
Develop and execute an *object initialisation analysis* verifying:

(1) An uninitialised object is never been stored to a register or field, nor used as parameter and its fields or methods are never accessed.

(2) Any object is initialised by a method of the corresponding type.

(3) No object is initialised twice.

Hint: When an object is initialised the analysis information of any stack position reference to this object has to be updated. Therefore it is necessary to distinguish uninitialised objects. Considering that the size of the stack is fixed for any label (cf. *type correctness analysis*) it is sufficient to use the label of creation for distinction.

(a) Give a complete lattice satisfying ACC suitable for *object initialization* analysis.

(b) Define the *object initialization* transition rules for the following byte code:

new C:	creates a new object of type C
iconst z:	push integer z
aconst null:	push null reference
if_cmpeq l:	pop the two topmost references from stack and jump to line l if they are equal
aload n:	push reference from register n
astore n:	pop reference and store it to register n
getfield C f τ:	pop reference and push value of field f
putfield C f τ:	pop value v and reference to object o and assign v to field f of o
invoke C f σ:	pop values $v_1, \ldots, v_n$ and reference to object, calls method M with parameters v1, . . . , v_n, and pushes return value
invokespecial C f σ:	pop values $v_1, \ldots, v_n$ and reference to object and calls a constructor (invokespecial not always refers to a constructor but we assume this here)
dup:	duplicates the last stack entry

(c) Perform an *object initialization* analysis for the following byte code. Assume that at the beginning registers and the stack contain instantiated objects only:

```
 1  new A
 2  dup
 3  aload 1
 4  aconst null
 5  if_cmpeq 9
 6  aload 1
 7  invokespecial A <init> (B)
 8  goto 14
 9  new B
10  dup
11  iconst 3
12  invokespecial B <init> (A, int)
13  invokespecial A <init> (B)
14  astore 0
```

Exercise 5.3: **(2+2 points)**

So far we restricted the type-analysis to classes, while in Java interfaces define valid object types, too. In this exercise we consider $(Typ, \sqsubseteq)$ extended by interfaces.

(a) Show that $(Typ, \sqsubseteq)$ extended by interfaces is not a complete lattice satisfying ACC.

(b) Give a sound adaption for $(Typ, \sqsubseteq)$ to a complete lattice satisfying ACC.