apl. Prof. Dr. Thomas Noll
Christina Jansen, Christoph Matheja

General Remarks

- Please hand in your solutions in groups of 3. Either hand in your solutions at the beginning of the exercise class or put them into the box at the chair.
- If you have questions regarding the exercises and/or lecture, feel free to write us an email or visit us at our office.

Exercise 1 (Interval Analysis): (5 Points)

Consider the following program:

```
x := -1;
y := -2 * x;
while (y - 1 > x) do
    y := y - 1;
    if (x > -2) then
        x := x - 1;
    end;
end;
```

a) Add the appropriate assertions to the program.

b) Perform an interval analysis using widening (with ∇ as presented in the lecture) on the program extend by assertions using the worklist algorithm. It is sufficient to provide the result of each step.

c) Use narrowing to possibly improve the results obtained in **b)**. You may give the results of narrowing without actually performing the iteration.

Exercise 2 (Improving Assertion Evaluation): (1 Points)

Consider the interval analysis using assertions. Let us now restrict the Boolean expressions to the following subset $BExp^-$ of $BExp$:

$$b \coloneqq true \mid false \mid x_1 = x_2 \mid x_1 < x_2 \text{ with } x_1, x_2 \in Var_c$$

Give an evaluation function for statements $assert(b)$, $b \in BExp^-$ computing intervals more precise than in the lecture for each $x \in Var_c$. Choose the evaluation function in a way that ensures the monotonicity of the transfer function for interval analysis.

Exercise 3 (Object Initialisation Analysis): (4 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, e.g.

Object creation and access in Java:

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

The corresponding Java bytecode:

```
1  new Point                          // 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 being stored to a register or field, nor used as parameter and its fields or methods are never accessed.

(2) every 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 reference to this object on the stack 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 program label at creation for the 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 instructions:

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
```