Prof. Dr. Thomas Noll

Christian Dehnert, Benjamin Kaminski

Sheet 1 (Complexity of the Fixpoint Iteration): (3 Points)

In the lecture we saw that the fixpoint iteration requires at most $m \cdot n$ steps, where m is the height of the partial order while n is the number of program points (i.e. labels). But how fast is the iterative algorithm for a concrete analysis (here: live variables)?

a) Show that LVA has the following property:
Let $c \in Cmd$, $x \in Var_c$ and $l \in L_c$. If x is live on the exit of l, then there exists an **acyclic** path from B^l to a use of x that does not re-define x.

b) Show that (standard) fixpoint iteration requires at most $|L_c|$ steps for convergence in case of LVA.

Sheet 2 (Extending Interval Analysis): (2 Points)

The WHILE-language as presented in the lecture does not feature a division operator. In this exercise we aim to incorporate this operator in the language and adapt the interval analysis from the lecture accordingly.

a) Extend the val_δ function for interval analysis to also account for division. In the case of a division by zero, you are to assume that every value is a valid result.

b) Show that the transfer functions of interval analysis (including the division operator) are monotonic.

Sheet 3 (Assertions for Interval Analysis): (2 Points)

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

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

a) Give an evaluation function for statements $assert(b)$, $b \in BExp^-$ computing accurate intervals for each $x \in Var_c$.

b) Extend $BExp^-$ by the logical disjunction. Give a "precise", but "safe" approximation of the resulting intervals.

Sheet 4 (Type Correctness of Java Bytecode): (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, i.e. $R(0) = A$ and $R(1) = B$.

```
aload 1
iconst 1
invoke B m C(int)
astore 0
aload 0
getfield C f int
iconst 0
if_icmpeq 1
aload 0
areturn
```