Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Brian Huffman, Peter Lammich

WS 2011/12
27. 01. 2012

Semantics of Programming Languages

Exercise Sheet 13

The following exercises are typical exam exercises. You are supposed to solve them on a sheet of paper, without using Isabelle/HOL.

Exercise 13.1 Verification Condition Generation

Regard the following While-program S:

```
a ::= x;
WHILE 1 < a DO
  a := a - 2
```

Your task is to show that:

$$\models \{x \geq 0\}\ S\ \{a = 0 \implies \mathsf{even}\ x\}$$

Find an invariant for the loop. Let S_{annot} be the annotated program, and $Q := \{a = 0 \implies \mathsf{even}\ x\}$ be the postcondition. Which proof obligations result when using the verification condition generator? What does $vc\ S_{\mathsf{annot}}\ Q$ and $pre\ S_{\mathsf{annot}}\ Q\ s$ look like?

Exercise 13.2 Parity analysis

Now regard the following While-program:

```
r := 11;
a := 11 + 11;
WHILE 1 < a DO
  r := r + 1
  a := a - 2;
r := a + 1
```

Add annotations for parity analysis to this program, and iterate the $step'$-function until a fixed point is reached. Document the results of each iteration in a table. Hint: Unlike sheet 12, you need to push the top-value of the lattice into the step function on each iteration!

Exercise 13.3 Abstract Interpretation For Conditionals

(To be done with Isabelle)

Regard the locale *Val_abs*. Define, analogous to *plus'*, a function *less'* :: *'av* $\Rightarrow$ *'av* $\Rightarrow$ *bool option* that approximates less expressions: *Some b* means, the result is definitely *b*, and *None* means unknown. Insert also an appropriate assumption *gamma_less'* to the locale.

Then define a function *bval'* :: *bexp* $\Rightarrow$ *'av st* $\Rightarrow$ *bool option* in the locale *Abs_Int_Fun* (analogous to aval'), and show a lemma *bval_sound* (analogous to *aval'_sound*).

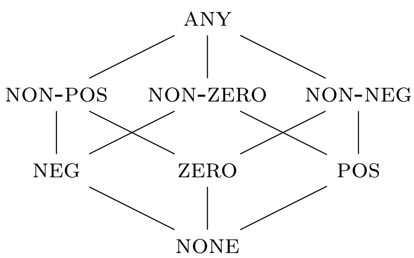
Note: You are not required to modify the *step'* function.

Homework 13 Abstract Interpretation: Sign Analysis

Submission until Wednesday, 8 February 2012, 12:00 (noon).

In this homework assignment, you must use the abstract interpretation framework (theory file *Abs_Int0.thy*) to create a *sign analysis*: For each program variable, this will calculate which signs (positive, negative, or zero) it could possibly have. (Refer to *Abs_Int0_parity.thy* to see a similar analysis for evenness/oddness. You may want to use that theory as a template for this assignment.)

First, define a type *sign* to formalize the 8-element complete lattice shown here. The elements NEG, ZERO, and POS indicate variables that are definitely known to be negative, zero, or positive, respectively. The other elements represent combinations of these.



One approach is to formalize *sign* as an 8-constructor datatype. But note that other representations are also possible!

Next, instantiate the *preord* and *SL_top* type classes: Define the ordering (*op* $\sqsubseteq$), join operator (*op* $\sqcup$), and top element ($\top$), and prove that they satisfy the class axioms.

instantiation *sign* :: *preord*
begin
  fun *le_sign* :: *"sign* $\Rightarrow$ *sign* $\Rightarrow$ *bool"* **where**
  instance
end

instantiation *sign* :: *SL_top*
begin
  fun *join_sign* :: *"sign ⇒ sign ⇒ sign"* **where**
  definition *Top_sign* :: *"sign"* **where**
  instance
end

In order to instantiate the *Val_abs* and *Abs_Int* locales, you must first define three functions that describe the meaning of the *sign* type.

The function *γ_sign* yields the set of possible integer values that correspond to each *sign*. For example, when applied to the value representing NON-NEG, it should return a set equal to $\{i.\ 0 \leq i\}$.

fun *γ_sign* :: *"sign ⇒ val set"* **where**

The function *num_sign* returns the most specific *sign* value that includes the given integer: NEG, ZERO, or POS, as appropriate.

fun *num_sign* :: *"val ⇒ sign"* **where**

The *plus_sign* function performs addition on *sign* values. It should always return the most specific element possible. For example, NON-NEG + POS = POS, and NEG + POS = ANY.

fun *plus_sign* :: *"sign ⇒ sign ⇒ sign"* **where**

Now instantiate the *Val_abs* and *Abs_Int* locales. The *Val_abs* locale requires you to supply some proofs, while *Abs_Int* does not.

interpretation *Val_abs*
  where *γ = γ_sign* **and** *num' = num_sign* **and** *plus' = plus_sign*

interpretation *Abs_Int*
  where *γ = γ_sign* **and** *num' = num_sign* **and** *plus' = plus_sign*
  defines *aval_sign* **is** *aval'* **and** *step_sign* **is** *step'* **and** *AI_sign* **is** *AI*
proof qed

Define and test the following example program as shown here. What does the analysis tell you about the values of x and y?

```
definition "test1_sign =
  ''x'' ::= N 0;
  ''y'' ::= Plus (V ''x'') (N 1);
  WHILE Less (V ''x'') (N 10) DO (
    ''x'' ::= Plus (V ''x'') (N 2);
    ''y'' ::= Plus (V ''x'') (V ''y''))"

value "show_acom_opt (AI_sign test1_sign)"
```

Finally, you must define a measure function for type *sign*, which can be used to prove that the analysis always terminates. Define a function *m_sign* and show that it satisfies the following two properties.

fun *m_sign* :: *"sign ⇒ nat"* **where**
lemma *m_sign_gt*: "$[\![x \sqsubseteq y; \neg\ y \sqsubseteq x]\!] \Longrightarrow m\_sign\ x > m\_sign\ y$"
lemma *m_sign_eq*: "$[\![x \sqsubseteq y; y \sqsubseteq x]\!] \Longrightarrow m\_sign\ x = m\_sign\ y$"