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

**Exercise 14.1  Inductive Predicates**

Consider the following inductive predicate, which characterizes odd natural numbers.

\[
\text{inductive} \quad \text{odd} :: \text{"nat \Rightarrow bool" where}
\]

\[
\begin{align*}
\text{Suc}_0 & : \text{"odd (Suc 0)" | } \\
\text{Suc}_\text{Suc} & : \text{"odd n \Rightarrow odd (Suc (Suc n))"}
\end{align*}
\]

Using the induction principle for the predicate odd, it can be proven that three times any odd number is also odd:

\[
\text{lemma} \quad \text{"odd n \Rightarrow odd (n + n + n)"}
\]

\[
\text{proof (induct rule: odd.induct)}
\]

First, write down precisely what subgoals remain after performing induction. How many cases are there? Which assumptions are available, and what conclusion must be proved in each case? Next, describe how each case can be proved. Which simplification rules or introduction rules are used to prove each case?

**Exercise 14.2  Collecting Semantics**

Recall the datatype of annotated commands (type ‘a acom) and the collecting semantics (function \( \text{step :: state set \Rightarrow state set acom \Rightarrow state set acom} \)) from the lecture. We reproduce the definition of \( \text{step} \) here for easy reference. (Recall that \( \text{post c} \) simply returns the right-most annotation from command \( c \).)
In this exercise you must evaluate the collecting semantics on the example program below by repeatedly applying the `step` function.

\[
c = (\text{IF } x < 0 \text{ THEN } \{ A_1 \} \text{ ELSE } \{ A_6 \} \text{ SKIP } \{ A_7 \})
\]

Let \( S \) be \( \{\langle -2,3 \rangle, \langle 1,2 \rangle\} \). Calculate column \( n+1 \) in the table below by evaluating `step` \( S \) \( c \) with the annotations for \( c \) taken from column \( n \). For conciseness, we use "\( \langle i, j \rangle \)" as notation for the state \( <"x":=i, "y":=j> \). We have filled in columns 0 and 1 to get you started; now compute and fill in the rest of the table.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>\emptyset</td>
<td>{\langle -2,3 \rangle}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_2 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_3 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_4 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_5 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_6 )</td>
<td>\emptyset</td>
<td>{\langle 1,2 \rangle}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_7 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_8 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exercise 14.3 Substitution**

Recall the datatype for arithmetic expressions.

\[ \text{datatype } aexp = N \text{ int } | \ V \text{ vname } | \ Plus aexp aexp \]

Define a function \( \text{subst} :: aexp \Rightarrow vname \Rightarrow aexp \Rightarrow aexp \), such that \( \text{subst } a \ v \ a' \) yields the expression \( a \) where every occurrence of variable \( v \) is replaced by the expression \( a' \).

Moreover, define a function \( \text{occurs} :: aexp \Rightarrow vname \Rightarrow bool \) such that \( \text{occurs } a \ v \) is true if and only if the variable \( v \) occurs in the expression \( a \). Prove the following:

\[ \neg \text{occurs } a \ v \Rightarrow \text{subst } a \ v \ a' = a \]

Is the following lemma also true? Proof or counterexample!

\[ \neg \text{occurs} (\text{subst } a \ v \ a') \ v \]
Homework 14  A generic abstract interpreter based on denotational semantics

Submission until Tuesday, 5. 2. 2013, 10:00am. (To be done with Isabelle/HOL again)

In this homework, you will be guided through developing a generic semantics for IMP. Then, for two such semantics whose domain parameters are related by a concretization function, you will prove soundness of a generic abstract interpreter.

The framework will be mostly based on the complete_lattice type class, which you have seen in the lectures and in exercise sheet 12. Similarly to what is described in the lectures for semilattices, the complete-lattice order and operations are extended from a type 'a to 'b ⇒ 'a componentwise. We shall be interested in the least fixed points lfp F of monotone functionals F defined between complete lattices of functions. lfp F is itself a monotone function:

```
lemma lfp_pres_mono:
fixes F :: "('a::complete_lattice⇒'a) ⇒ 'a ⇒ 'a"
assumes m: "mono F" and "∀f. mono f =⇒ mono (F f)"
shows "mono (lfp F)"
```

We shall also use a binary version of monotonicity:

```
definition mono2 f ≡ ∀x1 x2 y1 y2. x1 ≤ y1 ∧ x2 ≤ y2 −→ f x1 x2 ≤ f y1 y2"
```

We work with the usual datatypes for expressions and commands, save for the fact that boolean expressions are slightly simplified:

```
datatype bexp = Bc bool | Less aexp aexp
```

As in the lectures, we shall consider a generic semantics, operating on states that store values from an unspecified domain 'val:

```
type_synonym 'val state = "vname ⇒ 'val"
```

The domain bval for booleans shall be fixed to a type slightly more flexible than bool:

```
datatype bval = Nothing | Tr | Fl | Any
```

Your first task is to organize bval as an order as follows: Tr and Fl represent the (incomparable) truth values, Nothing is the bottom and Any is the top:

```
instantiation bval :: order
```

```
bool is embeded in bval as expected:
```

```
fun BBc where "BBc True = Tr" | "BBc False = Fl"
```

Note that BBc is an operation on the domain of boolean values corresponding to the syntactic Bc operator. Next, in a locale SEM, we fix operators corresponding to the syntactic constructs for arithmetic expressions. These operators are assumed monotone.
locale SEM =  
fixes NN :: "int ⇒ 'val::complete_lattice" 
and PPlus :: "'val ⇒ 'val ⇒ 'val" 
and LLess :: "'val ⇒ 'val ⇒ bval" 
assumes mono2_PPlus: "mono2 PPlus" 
and mono2_LLess: "mono2 LLess"  
begin 

We now work in the context of this locale, meaning that we have available the indicated 
constants for which we can use the stated assumptions. Define evaluation functions 
handling variables by state lookup and mapping the syntactic operators to the fixed 
semantic ones (e.g., Plus to PPlus): 

fun aval :: "aexp ⇒ 'val state ⇒ 'val" where 
fun bval :: "bexp ⇒ 'val state ⇒ 'val" where 

The semantics is defined denotationally, assigning a function between states to each 
command. The while case requires taking a least fixed point, via the combinator wcomb. 

definition wcomb :: "('val state ⇒ bval) ⇒ ('val state ⇒ 'val state) ⇒ ('val state ⇒ 'val state) ⇒ ('val state ⇒ 'val state) ⇒ ('val state ⇒ 'val state)
⇒ ('val state ⇒ 'val state)" where 
"wcomb b c w s ≡ case b s of
  Nothing ⇒ bot
  Fl ⇒ s
  Tr ⇒ w (c s)
  Any ⇒ sup (w (c s)) s"

fun sem :: "com ⇒ 'val state ⇒ 'val state" where 
"sem SKIP s = s"
|"sem (x ::= a) s = s(x := aval a s)"
|"sem (c1 ; c2) s = sem c2 (sem c1 s)"
|"sem (IF b THEN c1 ELSE c2) s = (case bval b s of
  Nothing ⇒ bot
  Tr ⇒ sem c1 s
  Fl ⇒ sem c2 s
  Any ⇒ sup (sem c1 s) (sem c2 s))"
|"sem (WHILE b DO c) s = lfp (wcomb (bval b) (sem c)) s"

Prove that the command semantics is monotone. You will need lemmas about mono-
tonicity of the various involved operators, as well as the following, saying that wcomb 
preserves monotonicity: 

lemma pres_mono_wcomb; 
assumes b: "mono b" and c: "mono c" and w: "mono w" 
shows "mono (wcomb b c w)"

lemma mono_sem: "mono (sem c)"

We are done with defining a parameterized generic semantics. Now we move to defining 
an abstract interpreter between two semantics. The following locale fixes two generic
semantics: a “concrete” one on domain $cval$, whose operator names are prefixed by “$C_\cdot$”, and an “abstract” one on domain $aval$, whose operator names are prefixed by “$A_\cdot$.”

It also fixes a monotone concretization function between their domains that behaves well w.r.t. the semantic operators. Thus, e.g., $PPlus_\cdot \gamma$ says that adding two abstract values and then concretizing yields an approximation of the result of adding the concretized values; in other words, the abstract operator $A_\cdot \PPlus$ is sound (via $\gamma$) w.r.t. the concrete operator $C_\cdot \PPlus$.

Finally, it fixes an abstraction function $\alpha$ that can be used to obtain, for each concrete value, an abstract value that approximates it.

```
locale AI = $C : SEM C_\cdot NN C_\cdot PPlus C_\cdot LLess + A : SEM A_\cdot NN A_\cdot PPlus A_\cdot LLess$

  for $C_\cdot NN :: "int \Rightarrow 'cval::{complete_lattice}"$
  and $C_\cdot PPlus :: "'cval \Rightarrow 'cval \Rightarrow 'cval"$
  and $C_\cdot LLess :: "'cval \Rightarrow 'cval \Rightarrow bval"$

  and $A_\cdot NN :: "int \Rightarrow 'aval::{complete_lattice}"$
  and $A_\cdot PPlus :: "'aval \Rightarrow 'aval \Rightarrow 'aval"$
  and $A_\cdot LLess :: "'aval \Rightarrow 'aval \Rightarrow bval"$

  fixes $\gamma :: "'aval \Rightarrow 'cval"$
  and $\alpha :: "'cval \Rightarrow 'aval"$

  assumes $\alpha_\cdot \gamma$: "$cv \leq \gamma (\alpha cv)"$
  and $\text{mono}_\cdot \gamma$: "$\text{mono }\gamma$"
  and $\text{NN}_\cdot \gamma[$simp$]$: "$C_\cdot NN i \leq \gamma (A_\cdot NN i)"$
  and $\text{PPlus}_\cdot \gamma[$simp$]$: "$C_\cdot PPlus (\gamma av1) (\gamma av2) \leq \gamma (A_\cdot PPlus av1 av2)"$
  and $\text{LLess}_\cdot \gamma[$simp$]$: "$C_\cdot LLess (\gamma av1) (\gamma av2) \leq A_\cdot LLess av1 av2"

begin

In the context of this locale, we have available all the definitions and facts from the locale SEM for the “$C_\cdot$”-prefixed parameters, as well as those for the “$A_\cdot$”-prefixed parameters. We defined abbreviations so that you can use the same prefixes for the defined concepts too, e.g., $C_\cdot sem$, $A_\cdot sem$. For theorems, use the prefixes “$C_\cdot$” and “$A_\cdot$”.

$\gamma$ is extended to states as usual:

```
definition $\gamma_\cdot st :: "'aval state \Rightarrow 'cval state" where "\gamma_\cdot st s x \equiv \gamma (s x)"$
```

Prove that the abstract semantics is sound w.r.t. the concrete semantics. You will need lemmas about soundness of the concrete evaluation operators, as well as the following lemma which we proved for you:

```
lemma lfp_\cdot wcomb_\cdot \gamma:
  assumes $c$: "$\text{mono }c$" and $b$: "$\text{mono }b$" and $c'$: "$\text{mono }c'$" and $b'$: "$\text{mono }b'$"
  and $cc'$: "$c o \gamma_\cdot st \leq \gamma_\cdot st o c'$" and $bb'$: "$b o \gamma_\cdot st \leq b'$"
  shows "$\text{lfp } (C_\cdot wcomb b c) (\gamma_\cdot st s) \leq \gamma_\cdot st (\text{lfp } (A_\cdot wcomb b' c') s)"$
```

Prove that...
**Theorem** soundness: “\( C_{\text{sem}} c \ (\gamma_{\text{st}} \ s) \leq \gamma_{\text{st}} (A_{\text{sem}} c \ s) \)”

To get a better grasp of how the above soundness result can be used, extend \( \alpha \) to a function between states and prove the following theorem, showing how the concrete semantics is approximated by the abstract semantics on the abstracted state:

**Definition** \( \alpha_{\text{st}} :: \text{"'cval state ⇒ 'aval state"} \)

**Theorem** soundness,\( \alpha \): “\( C_{\text{sem}} c \ s \leq \gamma_{\text{st}} (A_{\text{sem}} c \ (\alpha_{\text{st}} s)) \)”

Instantiating a locale means providing defined constants for its parameters and discharging its assumptions, in return of which one gets the theorems from the locale instantiated to these constants. For 10 points extra credit, instantiate the locale AI as follows:

- the concrete domain consists of integer sets (with componentwise operations);
- the abstract domain is the parity domain in the complete-lattice form discussed in exercise sheet 12.

(See the template.)