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

WS 2011/12
20. 01. 2012

Semantics of Programming Languages

Exercise Sheet 12

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

Exercise 12.1 Inductive Predicates

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

inductive *odd* :: *"nat ⇒ bool"* **where**
  Suc_0: *"odd (Suc 0)"* |
  Suc_Suc: *"odd n ⟹ odd (Suc (Suc n))"*

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

lemma *"odd n ⟹ odd (n + n + n)"*
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 12.2 Collecting Semantics

Recall the datatype of annotated commands (type *'a acom*) and the collecting semantics (function *step* :: *state set ⇒ state set acom ⇒ state set acom*) from the lecture. We reproduce the definition of *step* here for easy reference. (Recall that *post c* simply returns the right-most annotation from command *c*.)

```
  step S (SKIP {_}) = SKIP {S}
  step S (x::=e {_}) = x::=e {{s'. ∃s∈S. s'=s(x:=aval e s)}}
  step S (c₁; c₂) = step S c₁; step (post c₁) c₂
  step S (IF b THEN c₁ ELSE c₂ {_}) =
       IF b THEN step {s∈S. bval b s} c₁
       ELSE step {s∈S. ¬ bval b s} c₂ {post c₁ ∪ post c₂}
  step S ({I} WHILE b DO c {_}) =
       {S ∪ post c} WHILE b DO (step {s∈I. bval b s} c) {{s∈I. ¬ bval b s}}
```

In this exercise you must evaluate the collecting semantics on the example program below by repeatedly applying the *step* function.

```
c = (IF x < 0 THEN
        {A₁} WHILE 0 < y DO (
          y := y + x {A₂}
        ) {A₃}
        ELSE SKIP {A₄}
     ) {A₅}
```

Calculate column $n{+}1$ in the table below by evaluating *step S c* with the value of S and 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.

	0	1	2	3	4	5	6	7
S	$\{\langle -2,3\rangle,\langle 1,2\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
A_1	$\emptyset$	$\{\langle -2,3\rangle\}$						
A_2	$\emptyset$	$\emptyset$						
A_3	$\emptyset$	$\emptyset$						
A_4	$\emptyset$	$\{\langle 1,2\rangle\}$						
A_5	$\emptyset$	$\emptyset$						

Exercise 12.3 Substitution

Reconsider the datatype for arithmetic expressions.

datatype *aexp* = *N int* | *V vname* | *Plus aexp aexp*

Define a function $subst{::}aexp \Rightarrow vname \Rightarrow aexp \Rightarrow aexp$, such that $subst\ a\ v\ a'$ yields the expression a where every occurence of variable v is replaced by the expression a'. Moreover, define a function $occurs{::}aexp \Rightarrow vname \Rightarrow bool$ such that $occurs\ a\ v$ is true if and only if the variable v occurs in the expression a.

Prove the following lemma:

$\neg\ occurs\ a\ v \Longrightarrow subst\ a\ v\ a' = a$

Is the following lemma also true? Proof or counterexample!

$\neg occurs\ (subst\ a\ v\ a')\ v$

Homework 12 Complete Lattices and the Kleene Fixed Point Theorem

Submission until Wednesday, 1 February 2012, 12:00 (noon). (To be done with Isabelle/HOL again)

Have a look at the *complete_lattice* typeclass of Isabelle. (The class definition can be found in *src/HOL/Complete_Lattices.thy*, and the definitions of the lattice and semilattice super-classes can be found in *src/HOL/Lattices.thy* in the distribution.) It provides the standard characterization of complete lattices in Isabelle, where the carrier set of the lattice is assumed to be all elements of the type.

As mentioned in the lecture, the operations ≤ and *Sup* already uniquely determine the other operations (<,*top*, *bot*, *Inf*, *inf*, *sup*).
Provide and prove suitable characterizations for *bot*, *top*, and *Inf*:

```
lemma "(bot::'a::complete_lattice) = XXX" oops
lemma "(top::'a::complete_lattice) = XXX" oops
lemma "Inf (X::'a::complete_lattice set) = XXX" oops
```

Note: The only lattice operations that XXX may contain are ≤ and *Sup*.
Hint: The relevant lemmas are:

```
thm order_refl order_antisym order_trans Sup_least Sup_upper
thm bot_least bot_unique top_greatest top_unique Inf_lower Inf_greatest
```

Next, you shall prove the Kleene fixed point theorem. We first introduce some auxiliary definitions:

A chain is a set such that any two elements are comparable. For the purposes of the Kleene fixed-point theorem, it is sufficient to consider only countable chains. It is easiest to formalize these as ascending sequences. (We can obtain the corresponding set using the function *range* :: ('a ⇒ 'b) ⇒ 'b set.)

```
definition chain :: "(nat ⇒ 'a::complete_lattice) ⇒ bool"
  where "chain C ⟷ (∀n. C n ≤ C (Suc n))"
```

A function is continuous, if it commutes with least upper bounds of chains.

```
definition continuous :: "('a::complete_lattice ⇒ 'b::complete_lattice) ⇒ bool"
  where "continuous f ⟷ (∀C. chain C ⟶ f (Sup (range C)) = Sup (f ` range C))"
```

The following lemma may be handy:

```
lemma continuousD: "⟦continuous f; chain C⟧ ⟹ f (Sup (range C)) = Sup (f ` range C)"
  unfolding continuous_def by auto
```

As first exercise, show that any continuous function is monotonic:

```
lemma cont_imp_mono:
  fixes f :: "'a::complete_lattice ⇒ 'b::complete_lattice"
  assumes "continuous f"
  shows "mono f"
```

Hint: The relevant lemmas are

```
thm mono_def monoI monoD
```

Finally show the Kleene fixed point theorem. Note that this theorem is important, as it provides a way to compute least fixed points by iteration.

```
theorem kleene_lfp:
  fixes f:: “'a::complete_lattice ⇒ 'a”
  assumes CONT: “continuous f”
  shows “lfp f = Sup (range (λi. (f^^i) bot))”
proof -
```

We propose a proof structure here, however, you may deviate from this and use your own proof structure:

```
  let ?C = “λi. (f^^i) bot”
  note MONO=cont_imp_mono[OF CONT]

  have CHAIN: “chain ?C”
  show ?thesis
  proof (rule antisym)
    show “Sup (range ?C) ≤ lfp f”
  next
    show “lfp f ≤ Sup (range ?C)”
  qed
qed
```

Hint: Some relevant lemmas are

```
thm lfp_unfold lfp_lowerbound Sup_subset_mono range_eqI
```