Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Andrei Popescu, Peter Lammich

WS 2012/13
29. 01. 2013

Semantics of Programming Languages

Exercise Sheet 14

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.

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 14.2 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*.)

$step\ S\ (SKIP\ \{\_\}) = SKIP\ \{S\}$
$step\ S\ (x::=e\ \{\_\}) = x::=e\ \{\{s'.\ \exists s{\in}S.\ s'{=}s(x{:=}aval\ e\ s)\}\}$
$step\ S\ (c_1;\ c_2) = step\ S\ c_1;\ step\ (post\ c_1)\ c_2$
$step\ S\ (IF\ b\ THEN\ \{P_1\}\ c_1\ ELSE\ \{P_2\}\ c_2\ \{\_\}) =$
$\quad IF\ b\ THEN\ \{\{s{\in}S.\ bval\ b\ s\}\}\ step\ P_1\ c_1$
$\quad ELSE\ \{\{s{\in}S.\ \neg\ bval\ b\ s\}\}\ step\ P_2\ c_2$
$\quad \{post\ c_1 \cup post\ c_2\}$
$step\ S\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ c\ \{\_\}) =$
$\quad \{S \cup post\ c\}$
$\quad WHILE\ b\ DO\ \{\{s{:}I.\ bval\ b\ s\}\}\ step\ P\ C$
$\quad \{\{s{\in}I.\ \neg\ bval\ b\ s\}\}$

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

$c = (IF\ x < 0$
$\quad THEN\ \{A_1\}$
$\quad\quad \{A_2\}\ WHILE\ 0 < y\ DO\ \{A_3\}\ (y := y + x\ \{A_4\})\ \{A_5\}$
$\quad ELSE\ \{A_6\}\ SKIP\ \{A_7\}$
$)\ \{A_8\}$

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.

	0	1	2	3	4	5	6	7	8	9
A_1	$\emptyset$	$\{\langle -2,3\rangle\}$								
A_2	$\emptyset$	$\emptyset$								
A_3	$\emptyset$	$\emptyset$								
A_4	$\emptyset$	$\emptyset$								
A_5	$\emptyset$	$\emptyset$								
A_6	$\emptyset$	$\{\langle 1,2\rangle\}$								
A_7	$\emptyset$	$\emptyset$								
A_8	$\emptyset$	$\emptyset$								

Exercise 14.3 Substitution

Recall 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:

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

Is the following lemma also true? Proof or counterexample!

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

Homework 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 interpretor.

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 $'b \Rightarrow 'a$ 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 :: "$('a::complete\_lattice \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$"
assumes m: "*mono F*" **and** "$\bigwedge f.\ mono\ f \Longrightarrow mono\ (F\ f)$"
shows "*mono (lfp F)*"

We shall also use a binary version of monotonicity:

definition "$mono2\ f \equiv \forall\ x1\ x2\ y1\ y2.\ x1\ \le\ y1\ \wedge\ x2\ \le\ y2\ \longrightarrow f\ x1\ x2\ \le\ f\ y1\ y2$"

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 synactic operators to the fixed semantic ones (e.g., *Plus* to *PPlus*):

fun *aval* :: *"aexp ⇒ 'val state ⇒ 'val"* **where**
fun *bval* :: *"bexp ⇒ 'val state ⇒ bval"* **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)"* **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 monotonicity 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 interpretor between two semantics. The following locale fixes two generic semantics: a "concrete" one on domain *cval*, whose operator names are prefixed by "*C_*", and an "abstract" one on domain *aval*, whose operator names are prefixed by "*A_*".

It also fixes a monotone concretization function between their domains that behaves well w.r.t. the semantic operators. Thus, e.g., *PPlus_γ* 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_PPlus* is sound (via γ) w.r.t. the concrete operator *C_PPlus*.

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

locale *AI* = *C* : *SEM C_NN C_PPlus C_LLess* + *A* : *SEM A_NN A_PPlus A_LLess*

for *C_NN* :: *"int ⇒ 'cval::complete_lattice"*
and *C_PPlus* :: *"'cval ⇒ 'cval ⇒ 'cval"*
and *C_LLess* :: *"'cval ⇒ 'cval ⇒ bval"*

and *A_NN* :: *"int ⇒ 'aval::complete_lattice"*
and *A_PPlus* :: *"'aval ⇒ 'aval ⇒ 'aval"*
and *A_LLess* :: *"'aval ⇒ 'aval ⇒ bval"*
+
fixes γ :: *"'aval ⇒ 'cval"*
and α :: *"'cval ⇒ 'aval"*

assumes *α_γ*: *"cv ≤ γ (α cv)"*
and *mono_γ*: *"mono γ"*
and *NN_γ*[*simp*]: *"C_NN i ≤ γ (A_NN i)"*
and *PPlus_γ*[*simp*]: *"C_PPlus (γ av1) (γ av2) ≤ γ (A_PPlus av1 av2)"*
and *LLess_γ*[*simp*]: *"C_LLess (γ av1) (γ av2) ≤ A_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_*"-prefixed parameters, as well as those for the "*A_*"-prefixed parameters. We defined abbreviations so that you can use the same prefixes for the defined concepts too, e.g., *C_sem*, *A_sem*. For theorems, use the prefixes "*C.*" and "*A.*".

γ is extended to states as usual:

definition *γ_st* :: *"'aval state ⇒ 'cval state"* **where** *"γ_st s x ≡ γ (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_wcomb_γ*:
assumes *c*: *"mono c"* **and** *b*: *"mono b"* **and** *c'*: *"mono c'"* **and** *b'*: *"mono b'"*
and *cc'*: *"c o γ_st ≤ γ_st o c'"* **and** *bb'*: *"b o γ_st ≤ b'"*
shows *"lfp (C_wcomb b c) (γ_st s) ≤ γ_st (lfp (A_wcomb b' c') s)"*

theorem *soundness*: "$C\_sem\ c\ (\gamma\_st\ s) \leq \gamma\_st\ (A\_sem\ c\ s)$"

To get a better grasp of how the above soundness result can be used, extend α 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 α_st :: "$'cval\ state \Rightarrow 'aval\ state$"

theorem *soundness_α*: "$C\_sem\ c\ s \leq \gamma\_st\ (A\_sem\ c\ (\alpha\_st\ s))$"

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.)