Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2021/22
Feb 1

Semantics of Programming Lectures

Exercise Sheet 13

Exercise 13.1 Inverse Analysis

Consider a simple sign analysis based on this abstract domain:

datatype *sign* = *None* | *Neg* | *Pos0* | *Any*

fun γ :: *"sign ⇒ val set"* **where**
"γ None = {}" |
"γ Neg = {i. i < 0}" |
"γ Pos0 = {i. i ≥ 0}" |
"γ Any = UNIV"

Define inverse analyses for "+" and "<" and prove the required correctness properties:

fun *inv_plus′* :: *"sign ⇒ sign ⇒ sign ⇒ sign ∗ sign"*
lemma
"$\llbracket$ *inv_plus′ a a1 a2 = (a1′,a2′); i1 ∈ γ a1; i2 ∈ γ a2; i1+i2 ∈ γ a* $\rrbracket$
$\Longrightarrow$ *i1 ∈ γ a1′ ∧ i2 ∈ γ a2′* "
fun *inv_less′* :: *"bool ⇒ sign ⇒ sign ⇒ sign ∗ sign"*
lemma
"$\llbracket$ *inv_less′ bv a1 a2 = (a1′,a2′); i1 ∈ γ a1; i2 ∈ γ a2; (i1<i2) = bv* $\rrbracket$
$\Longrightarrow$ *i1 ∈ γ a1′ ∧ i2 ∈ γ a2′*"

The following is an old exam exercise:

Exercise 13.2 Command Equivalence

Recall the notion of *command equivalence*:

$c_1 \sim c_2 \equiv (\forall\, s\ t.\ (c_1,\ s) \Rightarrow t \ \longleftrightarrow\ (c_2,\ s) \Rightarrow t)$

1. Define a function *is_SKIP* :: *com ⇒ bool* which holds on commands equivalent to *SKIP*. The function *is_SKIP* should be as precise as possible, but it should not analyse arithmetic or boolean expressions.
   Prove: *is_SKIP c* $\Longrightarrow$ *c* $\sim$ *SKIP*

2. The following command equivalence is wrong. Give a counterexample in the form of concrete instances for b_1, b_2, c_1, c_2, and a state s.

$$\begin{array}{l} \textit{WHILE } b_1 \textit{ DO IF } b_2 \textit{ THEN } c_1 \textit{ ELSE } c_2 \\ \sim \textit{IF } b_2 \textit{ THEN } (\textit{WHILE } b_1 \textit{ DO } c_1) \textit{ ELSE } (\textit{WHILE } b_1 \textit{ DO } c_2) \end{array} \qquad (*)$$

3. Define a condition P on b_1, b_2, c_1, and c_2 such that the previous statement $(*)$ holds, i.e. $P\ b_1\ b_2\ c_1\ c_2 \Longrightarrow (*)$

   Your condition should be as precise as possible, but only using:
   - *lvars* :: *com* ⇒ *vname set* (all left variables, i.e. written variables),
   - *rvars* :: *com* ⇒ *vname set* (all right variables, i.e. all read variables),
   - *vars* :: *bexp* ⇒ *vname set* (all variables in a condition), and
   - boolean connectives and set operations

Homework 13.1 A generic abstract interpreter based on denotational semantics

Submission until Sunday, Feb 6, 23:59pm.

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 exercise sheet 12. This class is defined in the theory *Complete_Lattices*.

Similarly to what is described in the lectures for semilattices, the lattice order operations are extended from any 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:

$[\![\textit{mono } F;\ \bigwedge f.\ \textit{mono } f \Longrightarrow \textit{mono } (F\ f)]\!] \Longrightarrow \textit{mono } (\textit{lfp } F)$

We shall also use a binary version of monotonicity:

$\textit{mono2 } f \equiv \forall\, x1\ x2\ y1\ y2.\ x1 \le y1 \wedge x2 \le y2 \longrightarrow f\ x1\ x2 \le f\ y1\ y2$

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*

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) boolean values, *Nothing* is the bottom and *Any* is the top:

instantiation *bval* :: *order*
begin

definition *less_eq_bval* :: "*bval* ⇒ *bval* ⇒ *bool*"
definition *less_bval* :: "*bval* ⇒ *bval* ⇒ *bool*"
instance
end

Show the following for your definitions:

lemma *not_less_eq_bval*[*simp*]:
"$a \leq Nothing \longleftrightarrow a = Nothing$" "$\neg\ Any \leq Fl$" "$\neg\ Tr \leq Fl$" "$\neg\ Any \leq Tr$" "$\neg\ Fl \leq Tr$"

bool is embeded in *bval* as expected:

$BBc\ True = Tr$

$BBc\ False = Fl$

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*"
fun *bval* :: "*bexp* ⇒ '*val state* ⇒ *bval*"

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_wcomb*: **assumes** *c*: *"mono c"*
  shows *"mono (wcomb b c)"*
lemma *mono_sem*: *"mono (sem c)"*

end

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_*", 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
```

setup so that abbreviations are printed nicely:

```
abbreviation "C_aval ≡ C.aval" abbreviation "C_bval ≡ C.bval"
abbreviation "C_wcomb ≡ C.wcomb" abbreviation "C_sem ≡ C.sem"
abbreviation "A_aval ≡ A.aval" abbreviation "A_bval ≡ A.bval"
abbreviation "A_wcomb ≡ A.wcomb" abbreviation "A_sem ≡ A.sem"
```

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 lemmas which we proved for you:

```
lemma wcomb_γ:
  assumes mw: "mono w"
    and w: "w o γ_st ≤ γ_st o w'" and c: "c o γ_st ≤ γ_st o c'" and b: "b o γ_st ≤ b'"
  shows "(C_wcomb b c w) o γ_st ≤ γ_st o (A_wcomb b' c' w')"
proof (subst le_fun_def, standard)
  have 0: "⋀s. w (c (γ_st s)) ≤ γ_st (w' (c' s))"
      using mw w c order_trans unfolding mono_def comp_def le_fun_def by blast
  fix s show " (C_wcomb b c w ∘ γ_st) s ≤ (γ_st ∘ A_wcomb b' c' w') s"
  proof (cases "b' s")
    case Nothing
    hence "b (γ_st s) = Nothing"
      using b unfolding comp_def le_fun_def
      by (cases "b (γ_st s)") (metis bval.simps not_less_eq_bval)+
    thus ?thesis using Nothing unfolding C.wcomb_def A.wcomb_def by auto
  next
    case Tr
    hence "b (γ_st s) = Nothing ∨ b (γ_st s) = Tr"
      using b unfolding comp_def le_fun_def
      by (cases "b (γ_st s)") (metis not_less_eq_bval)+
    thus ?thesis using Tr 0 unfolding C.wcomb_def A.wcomb_def by auto
  next
```

```
    case Fl
    hence "b (γ_st s) = Nothing ∨ b (γ_st s) = Fl"
      using b unfolding comp_def le_fun_def
      by (cases "b (γ_st s)") (metis not_less_eq_bval)+
    thus ?thesis using Fl 0 unfolding C.wcomb_def A.wcomb_def by auto
  next
    case Any
    thus ?thesis
      unfolding C.wcomb_def A.wcomb_def
      by (auto split: bval.splits)
        (smt 0 γ_st_def le_fun_def le_sup_iff mono_γ mono_sup order_trans sup_fun_def)+
  qed
qed

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)"
proof -
  let ?F = "C_wcomb b c"
  let ?F' = "A_wcomb b' c'"
  have F: "mono ?F" and F': "mono ?F'"
    using C.mono_wcomb[OF c] A.mono_wcomb[OF c'] by auto
  have "mono (lfp ?F) ∧ lfp ?F ∘ γ_st ≤ γ_st ∘ lfp ?F'"
  proof (induction rule: lfp_ordinal_induct[OF F])
    case 1 then show ?case
      using wcomb_γ[OF _ _ cc' bb', of _"lfp ?F'"] C.pres_mono_wcomb[OF b c]
      unfolding lfp_unfold[symmetric, OF F'] by blast
  next
    case (2 A)
    then have "mono (Sup A)"
      using mono_Sup by fast
    moreover have "Sup A ∘ γ_st ≤ γ_st ∘ lfp ?F'"
      unfolding comp_def using 2 by (auto simp: le_fun_def intro: SUP_least)
    ultimately show ?case by blast
  qed
  thus ?thesis by (simp add: le_fun_def)
qed

lemma soundness: "C_sem c (γ_st s) ≤ γ_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 ⇒ 'aval state"*

lemma *soundness_*α: *"C_sem c s* $\leq$ γ*_st (A_sem c (*α*_st s))"*