theory *ex11_solution*
imports *"~~/src/HOL/IMP/Collecting" "~~/src/HOL/IMP/VCG" "~~/src/HOL/IMP/Hoare_Total"*
begin

For each of the three programs given here, you must prove partial correctness and total correctness. For the partial correctness proofs, you should first write an annotated program, and then use the verification condition generator from *VCG*. For the total correctness proofs, use the Hoare rules from *Hoare_Total*.

Some abbreviations, freeing us from having to write double quotes for concrete variable names:

abbreviation *"aa ≡ ''a''"* **abbreviation** *"bb ≡ ''b''"* **abbreviation** *"cc ≡ ''c''"*
abbreviation *"dd ≡ ''d''"* **abbreviation** *"ee ≡ ''d''"* **abbreviation** *"ff ≡ ''f''"*
abbreviation *"pp ≡ ''p''"* **abbreviation** *"qq ≡ ''q''"* **abbreviation** *"rr ≡ ''r''"*

Some useful simplification rules:

declare *algebra_simps*[*simp*] **declare** *power2_eq_square*[*simp*]

Rotated rule for sequential composition:

lemmas *SeqTR* = *Hoare_Total.Seq*[*rotated*]

Prove the following syntax-directed conditional rule (for total correctness):

lemma *IfT*:
  assumes "$\vdash_t$ {*P1*} c_1 {*Q*}" **and** "$\vdash_t$ {*P2*} c_2 {*Q*}"
  shows "$\vdash_t$ {λ*s*. (*bval b s* ⟶ *P1 s*) ∧ (¬ *bval b s* ⟶ *P2 s*)} *IF b THEN* c_1 *ELSE* c_2 {*Q*}"
  by (*auto intro*: *assms hoaret.intros*)

A convenient loop construct:

abbreviation *For* :: *"vname ⇒ aexp ⇒ aexp ⇒ com ⇒ com"*
  (*"(FOR _/ FROM _/ TO _/ DO _)"* [*0*, *0*, *0*, *61*] *61*) **where**
  "FOR v FROM a1 TO a2 DO c ≡
    v ::= a1 ;; WHILE (Less (V v) a2) DO (c ;; v ::= Plus (V v) (N 1))"

abbreviation *Afor* :: *"assn ⇒ vname ⇒ aexp ⇒ aexp ⇒ acom ⇒ acom"*
  (*"({_}/ FOR _/ FROM _/ TO _/ DO _)"* [*0*, *0*, *0*, *0*, *61*] *61*) **where**
  "{b} FOR v FROM a1 TO a2 DO c ≡
    v ::= a1 ;; {b} WHILE (Less (V v) a2) DO (c ;; v ::= Plus (V v) (N 1))"

Multiplication. Consider the following program *MULT* for performing multiplication and the following assertions *P_MULT* and *Q_MULT*:

definition *MULT2* :: *com* **where**
  "MULT2 ≡
    FOR dd FROM (N 0) TO (V aa) DO
      cc ::= Plus (V cc) (V bb)"

definition *MULT* :: *com* **where**

"$MULT \equiv cc ::= N\ 0\ ;;\ MULT2$"

definition *P_MULT* :: "*int* ⇒ *int* ⇒ *assn*" **where**
"$P\_MULT\ i\ j \equiv \lambda s.\ s\ aa = i \wedge s\ bb = j \wedge 0 \leq i$"

definition *Q_MULT* :: "*int* ⇒ *int* ⇒ *assn*" **where**
"$Q\_MULT\ i\ j \equiv \lambda s.\ s\ cc = i * j \wedge s\ aa = i \wedge s\ bb = j$"

Define an annotated program *AMULT i j*, so that when the annotations are stripped away, it yields *MULT*. (The parameters *i* and *j* will appear only in the loop annotations.)

definition *iMULT* :: "*int* ⇒ *int* ⇒ *assn*" **where**
"$iMULT\ i\ j \equiv \lambda s.\ s\ aa = i \wedge s\ bb = j \wedge s\ cc = s\ dd * j \wedge s\ dd \leq i$"

definition *AMULT2* :: "*int* ⇒ *int* ⇒ *acom*" **where**

```
"AMULT2 i j ≡
  {iMULT i j}
  FOR dd FROM (N 0) TO (V aa) DO
    cc ::= Plus (V cc) (V bb)"
```

definition *AMULT* :: "*int* ⇒ *int* ⇒ *acom*" **where**
"$AMULT\ i\ j \equiv (cc ::= N\ 0)\ ;;\ AMULT2\ i\ j$"

lemmas *MULT_defs* = *MULT2_def MULT_def P_MULT_def Q_MULT_def iMULT_def AMULT2_def AMULT_def*

lemma *strip_AMULT*: "*strip* (*AMULT i j*) = *MULT*"
unfolding *AMULT_def MULT_def AMULT2_def MULT2_def* **by** *simp*

Once you have the correct loop annotations, then the partial correctness proof can be done in two steps, with the help of lemma *vc_sound′*.

lemma *MULT_correct*: "⊢ {*P_MULT i j*} *MULT* {*Q_MULT i j*}"
apply (*subst strip_AMULT*[**where** *i*=*i* **and** *j*=*j*, *symmetric*])
apply (*rule vc_sound′*)
apply (*auto simp*: *MULT_defs*)
done

The total correctness proof will look much like the Hoare logic proofs from Exercise Sheet 9, but you must use the rules from *Hoare_Total* instead. Also note that when using rule *Hoare_Total.While_fun′*, you must instantiate both the predicate $P :: state \Rightarrow bool$ and the measure $f :: state \Rightarrow nat$. The measure must decrease every time the body of the loop is executed. You can define the measure first:

definition *mMULT* :: "*state* ⇒ *nat*" **where**
"$mMULT \equiv \lambda s.\ nat\ (s\ aa - s\ dd)$"

lemma *MULT_totally_correct*: "$\vdash_t$ {*P_MULT i j*} *MULT* {*Q_MULT i j*}"
unfolding *MULT_def MULT2_def*
apply (*rule SeqTR*)
apply (*rule SeqTR*)

```
   apply (rule Hoare_Total.While_fun'[where P="iMULT i j" and f=mMULT])
    apply (rule SeqTR)
     apply (rule Hoare_Total.Assign)
    apply (rule Hoare_Total.Assign')
    apply (auto simp: MULT_defs)
   apply (auto simp: mMULT_def)
  apply (rule Hoare_Total.Assign)
 apply (rule Hoare_Total.Assign')
 apply auto
 done
```

Division. Define an annotated version of this division program, which yields the quotient and remainder of aa/bb in variables $''q''$ and $''r''$, respectively.

```
definition DIV1 :: com where
  "DIV1 ≡ qq ::= N 0 ;; rr ::= N 0"

definition DIV_IF :: com where
  "DIV_IF ≡
    IF Less (V rr) (V bb)
    THEN Com.SKIP
    ELSE (rr ::= N 0 ;; qq ::= Plus (V qq) (N 1))"

definition
  "DIV2 ≡ rr ::= Plus (V rr) (N 1) ;; DIV_IF"

definition DIV :: com where
  "DIV ≡ DIV1 ;; FOR cc FROM (N 0) TO (V aa) DO DIV2"

definition P_DIV :: "int ⇒ int ⇒ assn" where
  "P_DIV i j ≡ λs. s aa = i ∧ s bb = j ∧ 0 ≤ i ∧ 0 < j"

definition Q_DIV :: "int ⇒ int ⇒ assn" where
  "Q_DIV i j ≡ λ s. i = s qq * j + s rr ∧ 0 ≤ s rr ∧ s rr < j ∧ s aa = i ∧ s bb = j"

definition iDIV :: "int ⇒ int ⇒ assn" where
  "iDIV i j ≡ λ s. s cc = s qq * j + s rr ∧ 0 ≤ s rr ∧ s rr < j ∧ s cc ≤ i ∧ s aa = i ∧ s bb
= j"

definition ADIV1 :: acom where
  "ADIV1 ≡ qq ::= N 0 ;; rr ::= N 0"

definition ADIV_IF :: acom where
  "ADIV_IF ≡
    IF Less (V rr) (V bb)
    THEN SKIP
    ELSE (rr ::= N 0 ;; qq ::= Plus (V qq) (N 1))"

definition ADIV2 :: acom where
```

```
  "ADIV2 ≡ rr ::= Plus (V rr) (N 1) ;; ADIV_IF"

definition ADIV :: "int ⇒ int ⇒ acom" where
  "ADIV i j ≡ ADIV1 ;; {iDIV i j} FOR cc FROM (N 0) TO (V aa) DO ADIV2"

lemmas DIV_defs =
  DIV1_def DIV_IF_def DIV2_def DIV_def Q_DIV_def P_DIV_def iDIV_def
  ADIV1_def ADIV_IF_def ADIV2_def ADIV_def

lemma strip_ADIV: "strip (ADIV i j) = DIV"
  unfolding DIV_defs by simp

lemma DIV_correct: "⊢ {P_DIV i j} DIV {Q_DIV i j}"
  apply (subst strip_ADIV[where i=i and j=j, symmetric])
  apply (rule vc_sound')
  apply (auto simp add: DIV_defs)
  done

definition mDIV :: "state ⇒ nat" where
  "mDIV ≡ λs. nat (s aa − s cc)"

lemma DIV_totally_correct: "⊢t {P_DIV i j} DIV {Q_DIV i j}"
  unfolding DIV_defs
  apply(rule SeqTR)
   apply(rule SeqTR)
    apply(rule Hoare_Total.While_fun'[where P = "iDIV i j" and f = mDIV])
     unfolding DIV_defs mDIV_def
     apply (rule SeqTR IfT Hoare_Total.Assign Hoare_Total.Skip)+
     apply (rule Hoare_Total.Assign')
    apply simp
    apply simp
   apply (rule SeqTR IfT Hoare_Total.Assign Hoare_Total.Skip)+
  apply (rule Hoare_Total.Assign')
  apply simp
  done
```

Square roots. Define an annotated version of this square root program, which yields the square root of input *aa* (rounded down to the next integer) in output *bb*.

```
definition SQR1 :: com where
  "SQR1 ≡ bb ::= N 0 ;; cc ::= N 1"

definition SQR2 :: com where
  "SQR2 ≡
   bb ::= Plus (V bb) (N 1);;
   cc ::= Plus (V cc) (V bb);;
   cc ::= Plus (V cc) (V bb);;
   cc ::= Plus (V cc) (N 1)"
```

```
definition SQR :: com where
  "SQR ≡ SQR1 ;; (WHILE (Not (Less (V aa) (V cc))) DO SQR2)"

definition P_SQR :: "int ⇒ assn" where
  "P_SQR i ≡ λs. s aa = i ∧ 0 ≤ i"

definition Q_SQR :: "int ⇒ assn" where
  "Q_SQR i ≡ λs. s aa = i ∧ (s bb)^2 ≤ i ∧ i < (s bb + 1)^2"

definition iSQR :: "int ⇒ assn" where
  "iSQR i ≡ λs. s aa = i ∧ 0 ≤ s bb ∧ (s bb)^2 ≤ i ∧ s cc = (s bb + 1)^2"

definition ASQR1 :: acom where
  "ASQR1 ≡ bb ::= N 0 ;; cc ::= N 1"

definition ASQR2 :: acom where
  "ASQR2 ≡
   bb ::= Plus (V bb) (N 1);;
   cc ::= Plus (V cc) (V bb);;
   cc ::= Plus (V cc) (V bb);;
   cc ::= Plus (V cc) (N 1)"

definition ASQR :: "int ⇒ acom" where
  "ASQR i ≡ ASQR1 ;; ({iSQR i} WHILE (Not (Less (V aa) (V cc))) DO ASQR2)"

lemmas SQR_defs = ASQR1_def ASQR2_def ASQR_def SQR1_def SQR2_def SQR_def iSQR_def
P_SQR_def Q_SQR_def

lemma strip_ASQR: "strip (ASQR i) = SQR"
  unfolding SQR_defs by simp

lemma SQR_correct: "⊢ {P_SQR i} SQR {Q_SQR i}"
  apply (subst strip_ASQR[where i=i, symmetric])
  apply (rule vc_sound')
  apply (auto simp add: SQR_defs)
  done

definition mSQR where
  "mSQR ≡ λs. nat (s aa − s bb ∗ s bb)"

lemma SQR_totally_correct: "⊢t {P_SQR i} SQR {Q_SQR i}"
  unfolding SQR_defs
 apply (rule Hoare_Total.While_fun'[where P = "iSQR i" and f = mSQR] SeqTR Hoare_Total.Assign)+
   apply (rule Hoare_Total.Assign')
   apply (simp add: SQR_defs mSQR_def)
  apply (simp add: SQR_defs mSQR_def)
  apply (rule SeqTR Hoare_Total.Assign)+
  apply (rule Hoare_Total.Assign')
  apply (simp add: SQR_defs)
```

done

Exercise 0.1 Collecting Semantics

Recall the datatype of annotated commands (type $'a\ acom$) and the collecting semantics (function $step$ :: $state\ set \Rightarrow state\ set\ acom \Rightarrow state\ set\ acom$) 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.)

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

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

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

Let S be $\{\langle -2,3\rangle, \langle 1,2\rangle\}$, abbreviated $-2,3 \mid 1,2$. 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	10
A_1	$\emptyset$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$	$-2,3$
A_2	$\emptyset$	$\emptyset$	$-2,3$	$-2,3$	$-2,3$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1 \mid -2,-1$	$-2,3 \mid -2,1 \mid -2,-1$	$-2,3 \mid -2,1 \mid -2,-1$
A_3	$\emptyset$	$\emptyset$	$\emptyset$	$-2,3$	$-2,3$	$-2,3$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$	$-2,3 \mid -2,1$
A_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$-2,1$	$-2,1$	$-2,1$	$-2,1 \mid -2,-1$	$-2,1 \mid -2,-1$	$-2,1 \mid -2,-1$	$-2,1 \mid -2,-1$
A_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$-2,-1$	$-2,-1$
A_6	$\emptyset$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$
A_7	$\emptyset$	$\emptyset$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$
A_8	$\emptyset$	$\emptyset$	$\emptyset$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2$	$1,2 \mid -2,-1$

end