Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D., Dr. Peter Lammich
Johannes Hölzl

WS 2014/15
13. 1. 2015

Semantics of Programming Languages

Exercise Sheet 11

There is a template available on the courses homepage

Exercise 11.1 Forward Assignment Rule

Think up and prove a forward assignment rule, i.e., a rule of the form $\vdash \{P\}\ x{::=}a\ \{\dots\}$, where $\dots$ is some suitable postcondition. Hint: To prove this rule, use the completeness property, and prove the rule semantically.

lemmas *fwd_Assign' = weaken_post[OF fwd_Assign]*

Redo the proofs for *MAX* and *MUL* from the previous exercise sheet, this time using your forward assignment rule.

```
definition MAX :: com where
  "MAX ≡
  IF (Less (V ''a'') (V ''b'')) THEN
    ''c''::=V ''b''
  ELSE
    ''c''::=V ''a''"
```

```
lemma "⊢ {λs. True} MAX {λs. s ''c'' = max (s ''a'') (s ''b'')}"
definition MUL :: com where
  "MUL ≡
  ''z''::=N 0;;
  ''c''::=N 0;;
  WHILE (Less (V ''c'') (V ''y'')) DO (
    ''z''::=Plus (V ''z'') (V ''x'');;
    ''c''::=Plus (V ''c'') (N 1))"
```

```
lemma "⊢ {λs. 0 ≤ s ''y''} MUL {λs. s ''z'' = s ''x'' * s ''y''}"
```

Exercise 11.2 Using the VCG

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

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*]

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

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

definition *Q_MULT* :: *"int ⇒ int ⇒ assn"* **where**
"Q_MULT i j ≡ λs. s cc = i ∗ j ∧ s aa = i ∧ 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.)

Hint: The program *AMULT i j* will be essentially *MULT* with an invariant annotation *iMULT i j* at the FOR loop, which you have to define:

definition *iMULT* :: *"int ⇒ int ⇒ assn"* **where**
definition *AMULT* :: *"int ⇒ int ⇒ acom"* **where**
  "AMULT i j ≡
  (cc ::= N 0) ;;
  {iMULT i j} FOR dd FROM (N 0) TO (V aa) DO
    cc ::= Plus (V cc) (V bb)"

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

lemma *strip_AMULT*: "*strip* (*AMULT i j*) = *MULT*"

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*: "$\vdash$ {*P_MULT i j*} *MULT* {*Q_MULT i j*}"

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 DIV :: com where "DIV ≡
  qq ::= N 0 ;;
  rr ::= N 0 ;;
  FOR cc FROM (N 0) TO (V aa) DO (
    rr ::= Plus (V rr) (N 1) ;;
    IF Less (V rr) (V bb) THEN
      Com.SKIP
    ELSE (
      rr ::= N 0 ;;
      qq ::= Plus (V qq) (N 1))
  )"

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

definition ADIV :: "int ⇒ int ⇒ acom" where "ADIV i j ≡
  qq ::= N 0 ;;
  rr ::= N 0 ;;
  {iDIV i j} FOR cc FROM (N 0) TO (V aa) DO (
    rr ::= Plus (V rr) (N 1) ;;
    IF Less (V rr) (V bb) THEN
      SKIP
    ELSE (
      rr ::= N 0 ;;
      qq ::= Plus (V qq) (N 1)
    )
  )"

lemma strip_ADIV: "strip (ADIV i j) = DIV"
lemma DIV_correct: "⊢ {P_DIV i j} DIV {Q_DIV i j}"
```

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 *SQR* :: *com* **where** *"SQR ≡*
  bb ::= N 0 ;;
  cc ::= N 1 ;;
  WHILE (Not (Less (V aa) (V cc))) DO (
    bb ::= Plus (V bb) (N 1);;
    cc ::= Plus (V cc) (Plus (V bb) (Plus (V bb) (N 1)))
)"

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"

Homework 11.1 Program Verification

Submission until Tuesday, 20 January 2015, 10:00am.

Define an annotated command *Cdiff* that subtracts x from y and prove

lemma "$\vdash \{\lambda s.\ s\ ''x'' = x \wedge s\ ''y'' = y \wedge 0 \leq x\}$ *strip*(*Cdiff* $x\ y$) $\{\lambda t.\ t\ ''y'' = y - x\}$"

Homework 11.2 Hoard Logic OR

Submission until Tuesday, 20 January 2015, 10:00am.

Extend IMP with a new command c_1 *OR* c_2 that is a nondeterministic choice: it may execute either c_1 or c_2. Add the constructor

```
Or com com   ("_ OR/ _" [60, 61] 60)
```

to datatype *com* in theory *Com*, adjust the definition of the big-step semantics in theory *Big_Step*, add a rule for *OR* to the Hoare logic in theory *Hoare*, and adjust the soundness and completeness proofs in theory *Hoare_Sound_Complete*.

All these changes should be quite minimal and very local if you have got the definitions right.