MP 4 – Evaluation Semantics

CS 477 – Spring 2018

Revision 1.0

Assigned March 30, 2018
Due April 6, 2018, 9:00 PM
Extension extend48 hours (penalty 20% of total points possible)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to test the student's understanding of

- Natural semantics evaluation, transition semantics evaluation, and program transition systems

Another purpose of MPs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP problems.

3 Turn-In Procedure

A skeleton version of the file `mp4.thy` for this assignment should be found in the `assignments/mp4/` subdirectory of your svn directory for this course. You should put code answering each of the problems below in the file `mp4.thy`. Your completed `mp4.thy` file should be put in the `assignments/mp4/` subdirectory of your svn directory (where it was originally found) and committed as follows:

```
svn commit -m "Turning in mp4"
```

Please read the *Instructions for Submitting Assignments* in

`http://courses.engr.illinois.edu/cs477/mps/index.html`

You may find it helpful to refer to Chapters 4, 7 and 12 of Concrete Symantics, which you can find in you svn repository at `resources/concrete_semantics.pdf`.

4 Syntax and Semantics for a Simple Imperative Programming Language (SIMP)

4.1 Syntax for SIMP

I have isolated the syntax of SIMP here so that it could be shared between the Hoare Logic theory, and the operational semantics theories. I have omitted *SKIP* so that the same language is used for

Hoare Logic, Natural Semantics and Transition Semantics. Note that we used the same "lifted" shallowly embedded syntax for arithmetic and boolean expressions as we did int he assignment on Hoare Logic.

datatype *'data command =*
  AssignCom var-name 'data exp (- ::= - [*1000*, *61*] *61*)
| *SeqCom 'data command 'data command* (-;;/ - [*60*, *61*] *60*)
| *CondCom 'data bool-exp 'data command 'data command*
  (*IF -/ THEN -/ ELSE -/ FI* [*0*,*0*,*0*] *60*)
| *WhileCom 'data bool-exp 'data command* (*WHILE -/ DO -/ OD* [*0*,*0*] *62*)

Here are a couple of somewhat silly example programs:

IF $ ''x'' [=] k 0 THEN ''y'' ::= k 2 ELSE ''y'' ::= k 3 FI

WHILE $ ''x'' [=] k 0 DO ''y'' ::= k 2;; ''x'' ::= $ ''x'' [−] k 1 OD

The first is a program that assigns `y` the value 2 if `x` has a value of 0, and assigns `y` a value of 3 otherwise. The second program checks if `x` has a value of 0, and if so, assigns `y` the value 2, decrements `x`, and returns to its start again.

4.2 Natural Semantics for SIMP

The rules below give the Natural Semantics for SIMP that were given in class, except that we are using the shallow embedding of arithmetic and boolean expressions to treat expressions as their values in a state.

inductive *eval* (**infixl** ⇓ *55*)
  where
  AsgEval:
    (*x* ::= *e*, *m1*) ⇓ *m1*(*x* := (*e m1*))

| *SeqEval*:
    ⟦ (*C1*, *m1*) ⇓ *m2*; (*C2*, *m2*) ⇓ *m3*⟧
    ⟹
    (*C1* ;; *C2*, *m1*) ⇓ *m3*

| *CondTrueEval*:
    ⟦*B m1*; (*C1*, *m1*)⇓ *m2*⟧
    ⟹
    (*IF B THEN C1 ELSE C2 FI*, *m1*) ⇓ *m2*

| *CondFalseEval*:
    ⟦¬(*B m1*); (*C2*, *m1*)⇓ *m2*⟧
    ⟹
    (*IF B THEN C1 ELSE C2 FI*, *m1*) ⇓ *m2*

| *WhileTrueEval*:
    ⟦*B m1*; (*C*, *m1*)⇓ *m2*; (*WHILE B DO C OD*, *m2*) ⇓ *m3*⟧
    ⟹
    (*WHILE B DO C OD*, *m1*) ⇓ *m3*

| *WhileFalseEval*:
    $[\![\neg(B\ m1)]\!]$
    $\Longrightarrow$
    $(WHILE\ B\ DO\ C\ OD,\ m1) \Downarrow m1$

4.3 Facts rephrasing the Natural Semantics rules

Below are a collection of lemmas, some of which may be useful in the problems later on, that restate facts about the evaluation of SIMP command in forms that may be more useful for drawing conclusions about the final state after the evaluation is done. The proofs have been done in Isar style, as those proofs tend to be more readable.

First we will give a couple of alternate rules for evaluation where all restrictions on the result state have been moved from the conclusions to equations in the hypotheses. This will facilitate using the rules in a computional manner that allows tracking of intermediate sates is a readable form.

Following the first two lemmas rephrasing two of the evaluation rules, are a collection of rules drawing conclusions under the assumption that evaluation occur on a command of a given form. For all the theorems, except those about the *WHILE*, the proof begins by a cases analysis of what form of evaluation rule could have been used the conclude the assumption, and this is driven by what form of command we are using. In most cases, only one rule can possibly apply. An exception to this is with the *IF* command, where we must consider both when the boolean guard is true, and when it is false.

lemma *AsgEvalAlt*:
  assumes *Same*: $m2 = m1(x := e(m1))$
  shows $(x ::= e,\ m1) \Downarrow m2$
  using *Same* **and** *AsgEval* **by** *simp*

lemma *WhileFalseEvalAlt*:
  assumes *FalseGuard*: $\neg(B\ m1)$
    and *Same*: $m2 = m1$
   shows $(WHILE\ B\ DO\ C\ OD,\ m1) \Downarrow m2$
  using *FalseGuard* **and** *Same* **and** *WhileFalseEval* **by** *simp*

lemma *assign*:
  assumes *Cmd*: $(x ::= e,\ m1) \Downarrow m2$
  shows $m2 = m1(x := e(m1))$
  using *Cmd*
  proof *cases*
   case *AsgEval* **then show** *?thesis* **by** *assumption*
  qed

lemma *sequence*:
  assumes *Cmd*: $(C;;C',\ m) \Downarrow m'$
  shows $\exists\ m''.\ (C,\ m) \Downarrow m'' \wedge (C',\ m'') \Downarrow m'$
  using *Cmd*
proof *cases*
  case $(SeqEval\ m'')$
  then show *?thesis* **by** *auto*
qed

```
lemma ifthenelse:
  assumes Cmd: (IF B THEN C ELSE C′ FI, m) ⇓ m′
  shows ((B m) ∧ (C, m) ⇓ m′) ∨ ((¬(B m)) ∧ (C′, m) ⇓ m′)
  using Cmd
    proof cases
      case CondTrueEval
      then
      show ?thesis by simp
    next
      case CondFalseEval
      then show ?thesis by simp
    qed
```

Proving facts about the *WHILE* command generally requires more than just an analysis of what rule was the last to be used. Because one of the rules has a recursive call to the *WHILE*, we need to use the induction principle inherent in the definition of evaluation as a family of inductive rules. The theorem is named *eval.induct* and its statement is as follows:

```
⟦x1 ⇓ x2; ⋀x e m1. P (x ::= e, m1) (m1(x := e m1));
 ⋀C1 m1 m2 C2 m3.
   ⟦(C1, m1) ⇓ m2; P (C1, m1) m2; (C2, m2) ⇓ m3; P (C2, m2) m3⟧
   ⟹ P (C1;; C2, m1) m3;
 ⋀B m1 C1 m2 C2.
   ⟦B m1; (C1, m1) ⇓ m2; P (C1, m1) m2⟧
   ⟹ P (IF B THEN C1 ELSE C2 FI, m1) m2;
 ⋀B m1 C2 m2 C1.
   ⟦¬ B m1; (C2, m1) ⇓ m2; P (C2, m1) m2⟧
   ⟹ P (IF B THEN C1 ELSE C2 FI, m1) m2;
 ⋀B m1 C m2 m3.
   ⟦B m1; (C, m1) ⇓ m2; P (C, m1) m2; (WHILE B DO C OD, m2) ⇓ m3;
    P (WHILE B DO C OD, m2) m3⟧
   ⟹ P (WHILE B DO C OD, m1) m3;
 ⋀B m1 C. ¬ B m1 ⟹ P (WHILE B DO C OD, m1) m1⟧
⟹ P x1 x2
```

```
lemma while-done-aux [rule-format]:
cm ⇓ m ⟹ (λ cm::('data command × (string ⇒ 'data)). λ m::string ⇒ 'data.
  (∃ C m1. (cm = (WHILE B DO C OD, m1))) ⟶ (¬ B m)) cm m
by (rule-tac P =
    (λ cm::('data command × (string ⇒ 'data)). λ m::string ⇒ 'data.
      (∃ C m1. (cm = (WHILE B DO C OD, m1))) ⟶ (¬ B m))
   in eval.induct, simp-all, auto)

lemma while-done:
⟦(WHILE B DO C OD, m1) ⇓ m2⟧ ⟹ (([¬] B) m2)
  by (simp add: not-b-def, erule while-done-aux, auto)

lemma while-inv-aux [rule-format]:
⟦cm ⇓ m′⟧ ⟹
  (λ cm. λ m′::string ⇒ 'data.
   ((∃B C m. (cm = ((WHILE B DO C OD),m)) ∧ P m ∧
     (∀ m1 m2. (P m1 ∧ B m1 ∧ (C, m1) ⇓ m2) ⟶ P m2)) ⟶ P m′)) cm m′
```

by (*rule-tac P* =
(λ *cm*. λ *m'*::*string* $\Rightarrow$ *'data*.
(($\exists$ *B C m*. (*cm* = ((*WHILE B DO C OD*),*m*)) $\wedge$ *P m* $\wedge$
($\forall$ *m1 m2*. (*P m1* $\wedge$ *B m1* $\wedge$ (*C*, *m1*) $\Downarrow$ *m2*) $\longrightarrow$ *P m2*)) $\longrightarrow$ *P m'*))
in *eval.induct*, *auto*)

lemma *while-inv*:
$[\![$(*WHILE B DO C OD*, *m*) $\Downarrow$ *m'*; $\forall$ *m1 m2*. *P m1* $\wedge$ *B m1* $\wedge$ (*C*, *m1*) $\Downarrow$ *m2* $\longrightarrow$ *P m2*; *P m*$]\!]$ $\Longrightarrow$
P m'
by (*erule-tac cm* = (*WHILE B DO C OD*, *m*) **and** *m'* = *m'* **in** *while-inv-aux*, *auto*)

lemma *while*:
$[\![$(*WHILE B DO C OD*, *m*) $\Downarrow$ *m'*; $\forall$*m1 m2*. *P m1* $\wedge$ *B m1* $\wedge$ (*C*, *m1*) $\Downarrow$ *m2* $\longrightarrow$ *P m2*; *P m* $]\!]$
$\Longrightarrow$ (*P* [$\wedge$] ([$\neg$] *B*)) *m'*
by (*simp add*: *and-b-def*, *rule conjI*, *erule*(*2*) *while-inv*, *erule while-done*)

lemma *while-unfold*:
assumes *Cmd*: (*WHILE B DO C OD*, *m*) $\Downarrow$ *m'*
shows (($\neg$ (*B m*)) $\wedge$ (*m'* = *m*)) $\vee$
((*B m*) $\wedge$ ($\exists$ *m''*. ((*C*, *m*) $\Downarrow$ *m''*) $\wedge$ (*WHILE B DO C OD*, *m''*) $\Downarrow$ *m'*))
using *Cmd*
proof *cases*
fix *m''*
assume *WUA1*: *B m*
and *WUA2*: (*C*, *m*) $\Downarrow$ *m''*
and *WUA3*: (*WHILE B DO C OD*, *m''*) $\Downarrow$ *m'*
case (*WhileTrueEval m''*)
then show *?thesis*
by (*simp*, *rule-tac x* = *m''* **in** *exI*, *simp*)
next
assume *WUA4*: *m'* = *m*
and *WUA5*: $\neg$ *B m*
case *WhileFalseEval*
from *WUA4* **and** *WUA5*
show *?thesis* **by** *simp*
qed

5 Transition Semantics for SIMP

In this subsection, we give the rules for small-step (transition) semantics for SIMP. We do again is in the Natural Semantics and use the shallow embedding of arithmetic and boolean expression to equate expressions with their value in the given state, and thus no transition steps are used in evaluating them.

As we did in class, we will transition a "configuration" to a "configuration". Configurations will either be a pair of a command and a state use to evaluate the command, or just a final state.

Because the language SIMP has no *SKIP* command, we can not transition a *WHILE* command to an *IF* command because we can not put a *SKIP* in the *ELSE* branch. However, because we are evaluating boolean expressions in zero steps, we can transition in one step to either what would have been the *THEN* branch, or we can transition in one step to being done.

datatype *'data configuration* =

Intermediate 'data command 'data state (⟨/ -, / -/ ⟩ [*60*, *61*] *60*)
| *Finished 'data state* (⟨⟨/ - / ⟩⟩ [*60*] *60*)

inductive *step* (**infixl** → *55*)
where
AsgStep:
⟨*x* ::= *e*, *m1*⟩ → ⟨⟨*m1*(*x* := (*e m1*))⟩⟩

| *FirstSeqStep*:
⟨*C1*, *m1*⟩ → ⟨*C1a*, *m2*⟩
⟹
⟨*C1*;;*C2*, *m1*⟩ → ⟨*C1a*;;*C2*, *m2*⟩

| *FirstSeqDone*:
⟨*C1*, *m1*⟩ → ⟨⟨*m2*⟩⟩
⟹
⟨*C1*;;*C2*, *m1*⟩ → ⟨*C2*, *m2*⟩

| *CondTrueStep*:
B m1
⟹
⟨*IF B THEN C1 ELSE C2 FI*, *m1*⟩ → ⟨*C1*, *m1*⟩

| *CondFalseStep*:
¬(*B m1*)
⟹
⟨*IF B THEN C1 ELSE C2 FI*, *m1*⟩ → ⟨*C2*, *m1*⟩

| *WhileTrueStep*:
B m1
⟹
⟨*WHILE B DO C OD*, *m1*⟩ → ⟨*C* ;; *WHILE B DO C OD*, *m1*⟩

| *WhileFalseStep*:
¬(*B m1*)
⟹
⟨*WHILE B DO C OD*, *m1*⟩ → ⟨⟨*m1*⟩⟩

It is somewhat interesting to note that in transition semantics, the only recursion in the rules is in the rule for sequences. Once you remove steps for evaluation of arithmetic and boolean expressions, evaluating the pieces of a sequence of commands is the only thing that involves evaluation a sub-entity.

As we did with Natural Semantics, we will prove a collection of alternate rules fro transition steps, where all constraints on the resultant configuration have been moved to equations in the assumptions. This time, we will need an alternate rule for every type of step.

lemma *AsgStepAlt*:
assumes *Same*: *conf* = ⟨⟨*m1*(*x* := (*e m1*))⟩⟩
shows ⟨*x* ::= *e*, *m1*⟩ → *conf*
using *Same* **and** *AsgStep* **by** *simp*

lemma *FirstSeqStepAlt*:
assumes *Step*: ⟨*C1*, *m1*⟩ → ⟨*C1a*, *m2*⟩

and *Same*: *conf* = ⟨*C1a*;;*C2*, *m2*⟩
shows ⟨*C1*;;*C2*, *m1*⟩ → *conf*
using *Same* **and** *Step* **and** *FirstSeqStep* **by** *simp*

lemma *FirstSeqDoneAlt*:
assumes *Step*: ⟨*C1*, *m1*⟩ → ⟨⟨*m2*⟩⟩
and *Same*: *conf* = ⟨*C2*, *m2*⟩
shows ⟨*C1*;;*C2*, *m1*⟩ → *conf*
using *Same* **and** *Step* **and** *FirstSeqDone* **by** *simp*

lemma *CondTrueStepAlt*:
assumes *Guard*: *B m1*
and *Same*: *conf* = ⟨*C1*, *m1*⟩
shows ⟨*IF B THEN C1 ELSE C2 FI*, *m1*⟩ → *conf*
using *Guard* **and** *Same* **and** *CondTrueStep* **by** *simp*

lemma *CondFalseStepAlt*:
assumes *Guard*: ¬(*B m1*)
and *Same*: *conf* = ⟨*C2*, *m1*⟩
shows ⟨*IF B THEN C1 ELSE C2 FI*, *m1*⟩ → *conf*
using *Guard* **and** *Same* **and** *CondFalseStep* **by** *simp*

lemma *WhileTrueStepAlt*:
assumes *Guard*: *B m1*
and *Same*: *conf* = ⟨*C* ;; *WHILE B DO C OD*, *m1*⟩
shows ⟨*WHILE B DO C OD*, *m1*⟩ → *conf*
using *Guard* **and** *Same* **and** *WhileTrueStep* **by** *simp*

lemma *WhileFalseStepAlt*:
assumes *Guard*: ¬(*B m1*)
and *Same*: *conf* = ⟨⟨*m1*⟩⟩
shows ⟨*WHILE B DO C OD*, *m1*⟩ → *conf*
using *Guard* **and** *Same* **and** *WhileFalseStep* **by** *simp*

6 Examples using Natural and Transition Semantics

In this section we will give examples of evaluating programs by Natural Semantics and Transition Semantics, and examples of proving results using Natural Semantics similar to what we did for Hoare Logic. The proofs in this section are all in apply style to make them patterns for what you will do for your problems.

lemma *ex1*:
(''*x*'' ::= $ ''*z*'' [+] *k 1*, λ *y*. *if y* = ''*z*'' *then 4 else 0*)
⇓
(λ *y*. *if y* = ''*z*'' *then 4 else if y* = ''*x*'' *then 5 else 0*)
— We want to use the rule *AsgEval*, but our resultant state
— is not expressed in the form of an update. We want to prove the theorem
— first using another form for the resultant state and then prove the two
— resultant states equal. We can do this using the theorem
— *AsgEvalAlt*: *m2* = *m1*(*x* := *e m1*) ⟹ (*x* ::= *e*, *m1*) ⇓ *m2*
— instead.

apply (*rule AsgEvalAlt*)

— And that leaves us with showing the computed resultant state is equal
— to the given one. However, states are functions from variable names to
— values, integers in this case. TO show two functions are equal, we
— will use extensionality to show that they are equal on arbitrary input.
apply (*rule ext*)
— This leaves us with expanding out lifter express notation and basic
— arithmetic that *simp* can handle.
by (*simp add*: *plus-e-def rev-app-def k-def*)

lemma *ex2*:
$(''x'' ::= \$\ ''z''\ [+]\ k\ 1,\ m1) \Downarrow m2 \Longrightarrow$
$(m2\ ''x'' = (m1\ ''z'' + 1)) \wedge ((y \neq ''x'') \longrightarrow m1\ y = m2\ y)$
— We want to begin by drawing a conclusion from the fact that we can evaluate
— an assignment. We want to derive a new assumption from one we currently have
— using the theorem *assign*. The proof methods *drule* and
— *drule_tac* allow us to do exactly that.
apply (*drule assign*)
— The theorem *assign* replaces the assumption that the assignment
— command evaluates to a state with the relation between the start and end states.
— This together is enough, together with expanding out the lifted operators, for
— *simp* once again to finish the proof.
by (*simp add*: *plus-e-def rev-app-def k-def*)

The proof of *ex3* will go muct as that of *ex1*.

lemma *ex3*: $\langle ''x'' ::= \$\ ''z''\ [+]\ k\ 1,\ (\lambda\ y.\ \mathit{if}\ y = ''z''\ \mathit{then}\ 4\ \mathit{else}\ 0)\rangle \rightarrow$
$\langle\langle(\lambda\ y.\ \mathit{if}\ y = ''z''\ \mathit{then}\ 4\ \mathit{else}\ \mathit{if}\ y = ''x''\ \mathit{then}\ 5\ \mathit{else}\ 0)\rangle\rangle$

apply (*rule AsgStepAlt*)
— We want to show two final configurations are the same. To do that, we
— need to show the underlying states (functions) are the same.
apply *simp*
— To show the functions are the same, we will use extensionality to show
— they produce the same value on an arbitrary input.
apply (*rule ext*)
by (*simp add*: *k-def rev-app-def plus-e-def*)

Proofs for *IF* require chaining a few steps of reasoning together.

lemma *ex4*:
$(\mathit{IF}\ (\$''y''\ [<]\ \$''z'')\ \mathit{THEN}\ (''x'' ::= \$\ ''z''\ [+]\ k\ 1)$
$\mathit{ELSE}\ (''x'' ::= \$\ ''z''\ [-]\ k\ 1)\ \mathit{FI},$
$(\lambda\ y.\ \mathit{if}\ y = ''z''\ \mathit{then}\ 4\ \mathit{else}\ 0))$
$\Downarrow$
$(\lambda\ y.\ \mathit{if}\ y = ''z''\ \mathit{then}\ 4\ \mathit{else}\ \mathit{if}\ y = ''x''\ \mathit{then}\ 5\ \mathit{else}\ 0)$
— In our input memory, *"y"* has a value of 0 and *"z"* has a
— value of 4, so the boolean guard is true, and we need to use the rule *CondTrueEval*.
apply (*rule CondTrueEval*)
apply (*simp add*: *less-b-def rev-app-def*)
— From here it is the same proof as in *ex1*.
apply (*rule AsgEvalAlt*)
apply (*rule ext*)
by (*simp add*: *k-def rev-app-def plus-e-def*)

```
lemma ex5:
 ⟦(IF ($''y'' [<] $''z'') THEN (''x'' ::= $ ''z'' [+] k 1)
   ELSE (''x'' ::= $ ''z'' [−] k 1) FI, m1) ⇓ m2; m1 ''y'' < m1 ''z''⟧ ⟹
  m2 ''x'' > m2 ''z''
  apply (drule ifthenelse)
  apply (simp add: rev-app-def k-def less-b-def plus-e-def)
  apply (drule assign)
  by simp
```

```
lemma ex6:
 ⟨IF ($''y'' [<] $''z'') THEN (''x'' ::= $ ''z'' [+] k 1)
  ELSE (''x'' ::= $ ''z'' [−] k 1) FI, (λ y. if y = ''z'' then 4 else 0)⟩
  →
  ⟨(''x'' ::= $ ''z'' [+] k 1),(λ y. if y = ''z'' then 4 else 0)⟩
   apply (rule CondTrueStepAlt)
   apply (simp add: less-b-def rev-app-def)
  by simp
```

In Example *ex7a*, we do a computation of the results of evaluating a fairly simple (and stupid) while loop. We repeatedly apply the rule that applies to the topmost structure of the program in the top goal, and solve the expression constraints as they come up. This way of doing things means we have to determined the truth of the boolean guard from an increasingly complex expression for the state.

```
lemma ex7a:
(WHILE $ ''i'' [<] k 2 DO
   (''i'' ::= $ ''i'' [+] k 2;;
    ''i'' ::= $ ''i'' [−] k 1)
   OD, (λ s. 0)) ⇓ (λ s. if s = ''i'' then 2 else 0)
  apply (rule WhileTrueEval)
    apply (simp add: less-b-def k-def rev-app-def)
   apply (rule SeqEval)
    apply (rule AsgEval)
   apply (rule AsgEval)
  apply (rule WhileTrueEval)
    apply (simp add: rev-app-def k-def less-b-def plus-e-def minus-e-def )
   apply (rule SeqEval)
    apply (rule AsgEval)
   apply (rule AsgEval)
   apply (rule WhileFalseEvalAlt)
   apply (simp add: rev-app-def k-def plus-e-def minus-e-def less-b-def)
  apply (rule ext)
  by (simp add: rev-app-def k-def plus-e-def minus-e-def less-b-def)
```

As an alternate approach, I will do the same computation, but using the *Alt* versions of the rules for assignment, allowing me to compute a "simplified" version of the state after each update through theorem proving.

```
lemma ex7b:
(WHILE $ ''i'' [<] k 2 DO
   (''i'' ::= $ ''i'' [+] k 2;;
    ''i'' ::= $ ''i'' [−] k 1)
   OD, (λ s. 0)) ⇓ (λ s. if s = ''i'' then 2 else 0)
  apply (rule WhileTrueEval)
    apply (simp add: less-b-def k-def rev-app-def)
```

```
  apply (rule SeqEval)
   apply (rule AsgEvalAlt)
 apply (simp add: k-def rev-app-def plus-e-def)
  apply (rule AsgEvalAlt)
 apply (simp add: k-def rev-app-def plus-e-def minus-e-def)
 apply (rule WhileTrueEval)
   apply (simp add: rev-app-def k-def less-b-def plus-e-def minus-e-def)
  apply (rule SeqEval)
   apply (rule AsgEvalAlt)
   apply (simp add: rev-app-def k-def less-b-def plus-e-def minus-e-def)
  apply (rule AsgEvalAlt)
  apply (simp add: rev-app-def k-def minus-e-def)
  apply (rule WhileFalseEvalAlt)
  apply (simp add: rev-app-def k-def less-b-def)
 apply (rule ext)
 by simp
```

```
lemma ex8:
⟦(WHILE $ ''i'' [<] k 2 DO
   (''i'' ::= $ ''i'' [+] k 2;;
    ''i'' ::= $ ''i'' [−] k 1)
   OD, m1) ⇓ m2;
 m1 ''i'' = 0 ⟧ ⟹ ($ ''i'' [=] k 2) m2
 — We must find and prove an invariant and prove the result from the
 — invariant and the negation of the boolean guard. I choose i ≤ 2.
 apply (drule-tac P=($ ''i'' [≤] k 2) in while)
   apply clarsimp
   apply (drule sequence)
   apply clarsimp
   apply (drule assign)
   apply (drule assign)
   apply (simp add: rev-app-def k-def less-b-def less-eq-b-def plus-e-def minus-e-def)
  apply (simp add: rev-app-def k-def less-eq-b-def)
 by (simp add: rev-app-def k-def less-b-def less-eq-b-def eq-b-def and-b-def not-b-def)
```

7 Problems

The problems below are designed to step you through some of the pieces of reasoning about Natural Semantics evaluations in Isabelle.

7.1 Lifted Propositional Logic

In the first three problems below you will prove the "lifted" versions of three problems from MP1. You are free to use any and all theorem proving methods in Isabelle to prove them. You may wish to refer to the definitions and theorems in `lifted_basic` and `lifted_predicate_logic`. For an example, here is the "lifted" version of the first problem from MP1:

Remove the `oops` from each problem and put in your own proof.

1. (5 pts)

lemma *problem1*:
$(''y'' ::= \$\ ''y''\ [+]\ k\ 1,\ (\lambda\ s.\ 1)) \Downarrow (\lambda s.\ if\ s = ''y''\ then\ 2\ else\ 1)$
oops

2. (7 pts)

lemma *problem2*:
$\langle ''y'' ::= \$\ ''y''\ [+]\ k\ 1,\ (\lambda\ s.\ 1)\rangle \rightarrow \langle\langle(\lambda s.\ if\ s = ''y''\ then\ 2\ else\ 1)\rangle\rangle$
oops

3. (5 pts)

lemma *problem3*:
$[\![(''y'' ::= \$\ ''y''\ [+]\ k\ 1,\ m1) \Downarrow m2\ ]\!] \Longrightarrow m2\ ''y'' > m1\ ''y''$
oops

4. (10 pts)

lemma *problem4*:
$((''y'' ::= \$\ ''y''\ [+]\ k\ 1;;\ ''x'' ::= \$\ ''x''\ [-]\ k\ 1),$
$(\lambda s.\ if\ s = ''y''\ then\ a\ else\ if\ s = ''x''\ then\ b\ else\ c)) \Downarrow$
$(\lambda s.\ if\ s = ''y''\ then\ a + 1\ else\ if\ s = ''x''\ then\ b - 1\ else\ c)$
oops

5. (8 pts)

lemma *problem5*:
$\langle(''y'' ::= \$\ ''y''\ [+]\ k\ 1;;\ ''x'' ::= \$\ ''x''\ [-]\ k\ 1),$
$(\lambda s.\ if\ s = ''y''\ then\ a\ else\ if\ s = ''x''\ then\ b\ else\ c)\rangle$
$\rightarrow$
$\langle(''x'' ::= \$\ ''x''\ [-]\ k\ 1),$
$(\lambda s.\ if\ s = ''y''\ then\ a\ +1\ else\ if\ s = ''x''\ then\ b\ else\ c)\rangle$
oops

6. (9 pts)

lemma *problem6*:
$[\![((''y'' ::= \$\ ''y''\ [+]\ k\ 1;;\ ''x'' ::= \$\ ''x''\ [-]\ k\ 1),\ m1) \Downarrow m2\ ]\!] \Longrightarrow$
$m2\ ''x'' + m2\ ''y'' = m1\ ''x'' + m1\ ''y''$
oops

7. (12 pts)

lemma *problem7*:
$[\![(\$''y''\ [=]\ \$''a'')m1;$
$((IF\ \$''y''\ [mod]\ (k\ 2)\ [=]\ (k\ 0)\ THEN\ (''y'' ::= \$''y'')$
$ELSE\ (''y'' ::= \$''y''\ [+]\ (k\ 1))\ FI),\ m1)\ \Downarrow m2]\!] \Longrightarrow$
$((\$''y''\ [\geq]\ \$''a'')\ [\wedge]$
$(\$''y''\ [\leq]\ (\$''a''\ [+]\ (k\ 1)))\ [\wedge]$
$(\$''y''\ [mod]\ (k\ 2)\ [=]\ k\ 0))\ m2$
oops

7.2 Extra Credit

8. (8 pts)

lemma *problem8*:
⟦($ ''y'' [=] k a [∧] $ ''x'' [=] k b [∧] k b [>] k 0) m1;
(WHILE $ ''x'' [>] k 0 DO
(''y'' ::= $ ''y'' [+] k 1;;
''x'' ::= $ ''x'' [−] k 1)
OD, m1) ⇓ m2⟧ ⟹ ($ ''y'' [=] k(a + b)) m2
oops