Semantics of Hoare Logic

Aquinas Hobor and Martin Henz
What does a Hoare triple mean?

\{ \phi \} \ c \ \{ \psi \}

Informal meaning (already given):

“If the program c is run in a state that satisfies \phi and c terminates, then the state resulting from c’s execution will satisfy \psi.”
We would like to formalize

\{\phi\} \ c \ \{\psi\}

Informal meaning (already given):

“If the program \(c\) is run in a state that satisfies \(\phi\) and \(c\) terminates, then the state resulting from \(c\)’s execution will satisfy \(\psi\).”
We would like to formalize

\{\phi\} \text{c} \{\psi\}

Need to define:
1. Running a program c until it terminates
2. Initial state satisfies \( \phi \)
3. Resulting state satisfies \( \psi \).
We would like to formalize

\{\phi\} \ c \ \{\psi\}

For better clarity and “fun”, we will do it in Coq.
We would like to formalize

\\{
\phi
\}\ c\ \\{\psi\}

For better clarity and “fun”, we will do it in Coq.

(And by “we”, I mean I will do part of it in class and you will do the rest at home...)
We would like to formalize

\{\phi\} P \{\psi\}

Need to define:
1. Running a program P until it terminates
2. Initial state satisfies \(\phi\)
3. Resulting state satisfies \(\psi\).
Operational Semantics

• Numeric Expressions E:
  – z | x | (E + E) | (E – E) | (E * E)

• Boolean Expressions B:
  – (E < E) | (B | | B) | (!B)

• Commands C:
  – skip | x = E | C;C | if (B) {C} else {C} | while (B) {C}
We have to specify exactly how each evaluates

- Numeric Expressions E:
  - \( z \) | \( x \) | \( (E + E) \) | \( (E - E) \) | \( (E * E) \)

First problem: what are our variables “\( x \)”? 

We will use our usual trick of letting variables be natural numbers:

\[
\text{Definition } \text{var} : \text{Type} := \text{nat}.
\]
We have to specify exactly how each evaluates

- Numeric Expressions $E$:
  - $z$ | $x$ | $(E + E)$ | $(E - E)$ | $(E \times E)$

Next: how do we define our expressions?

Inductive $nExpr : Type :=$

| Num: forall $z : Z$, $nExpr$
| Var: forall $v : \text{var}$, $nExpr$
| Plus: forall $ne1 ne2 : nExpr$, $nExpr$
| Minus: forall $ne1 ne2 : nExpr$, $nExpr$
| Times: forall $ne1 ne2 : nExpr$, $nExpr$.
We have to specify exactly how each evaluates

- Numeric Expressions $E$:
  - $z \mid x \mid (E + E) \mid (E - E) \mid (E \times E)$

Now, what does evaluation of an $E$ mean?

We want to write $E \downarrow n$ to mean “the expression $E$ evaluates to the numeric $n$”

But what about $E = x$? By itself, we don’t know what to do...
We have to specify exactly how each evaluates

• Numeric Expressions $E$:
  
  $\ -z \ | \ x \ | \ (E + E) \ | \ (E - E) \ | \ (E \ast E)$

Define a context $\rho$ to be a function from variables to numbers.

Definition $\text{ctx} := \text{var} \rightarrow \text{num}$. 
We have to specify exactly how each evaluates

• Numeric Expressions E:
  – \( z \) | \( x \) | \((E + E)\) | \((E - E)\) | \((E \times E)\)

Now define \( \rho \vdash E \downarrow n \) to mean “in context \( \rho \), the expression E evaluates to the numeric n.”
Numeric Evaluation in Coq

Fixpoint neval (g : ctx) (ne : nExpr) : num :=
    match ne with
    | Num n => n
    | Var x => g x
    | Plus ne1 ne2 => neEval g ne1) + (neEval g ne2)
    | Minus ne1 ne2 => (neEval g ne1) - (neEval g ne2)
    | Times ne1 ne2 => (neEval g ne1) * (neEval g ne2)
    end.
Boolean Expressions

- Boolean Expressions B:
  - \((E \leq E) \mid (B \| B) \mid (!B)\)

\[
\text{Inductive } bExpr : \text{Type} := \\
| \text{LE} : \forall \text{ne1 ne2} : \text{nExpr}, bExpr \\
| \text{Or} : \forall \text{be1 be2} : bExpr, bExpr \\
| \text{bNeg} : \forall \text{be} : bExpr, bExpr.
\]
Boolean Evaluation

- Boolean Expressions $B$:
  - $(E \leq E) \mid (B \mid B) \mid (!B)$

Since $B$ includes $E$, we will need contexts to evaluate $B$s.

What do we evaluate to? How about $Prop$.

So define $\rho \vdash B \downarrow P$ to mean “in context $\rho$, the expression $B$ evaluates to the proposition $P$.”
Boolean Evaluation

\[
\text{Fixpoint beval}\ (g : \text{ctx})\ (be : \text{bExpr}) : \text{Prop} :=
\begin{array}{l}
\text{match be with} \\
\quad | \text{LE}\ ne1\ ne2 \Rightarrow (\text{neEval}\ g\ ne1) \leq (\text{neEval}\ g\ ne2) \\
\quad | \text{Or}\ be1\ be2 \Rightarrow (\text{beEval}\ g\ be1) \text{\ \text{\slash}\ \text{\slash}\ (\text{beEval}\ g\ be2)} \\
\quad | \text{bNeg}\ be \Rightarrow \sim (\text{beEval}\ g\ be) \\
\end{array}
\]
Commands

- Commands C:
  - skip | x = E | C;C | if B {C} else {C} | while B {C}

Inductive Coms : Type :=
  | Skip : Coms
  | Assign : forall (x : var) (e: nExpr), Coms
  | Seq : forall c1 c2 : Coms, Coms
  | If : forall (b : bExpr) (c1 c2 : Coms), Coms
  | While : forall (b : bExpr) (c : Coms), Coms.
Command Evaluation

• Idea: executing command $c$ moves the machine from a starting context $\rho_\alpha$ to an ending context $\rho_\omega$

• We define a step relation that looks like this:

$$c \vdash \rho_\alpha \rightsquigarrow \rho_\omega$$

• This will be defined as the least relation (i.e., inductively) satisfying a set of rules

Inductive BStep : Coms -> ctx -> ctx -> Prop :=
Step relation, skip

\[ \text{skip} \vdash \rho \rightsquigarrow \rho \]

| bSkip : forall rho,  
  BStep Skip rho rho  

Step relation, assign

\[ \rho \vdash E \downarrow n \]

\[ (x = E) \vdash \rho \leadsto [x \mapsto n] \rho \]

| bAssign : forall x ne rho, BStep (Assign x ne) rho (upd_ctx rho x (neval rho ne)) |
Step relation, seq

\[
\begin{align*}
C_1 & \vdash \rho_1 \leadsto \rho_2 \\
C_2 & \vdash \rho_2 \leadsto \rho_3 \\
(C_1 ; C_2) & \vdash \rho_1 \leadsto \rho_3
\end{align*}
\]

| bSeq : forall rho rho' rho'' c1 c2, 
  BStep c1 rho rho' -> 
  BStep c2 rho' rho'' -> 
  BStep (Seq c1 c2) rho rho'' |
Step relation, if (a)

\[ \rho \vdash B \downarrow \text{True} \quad C_1 \vdash \rho_1 \leadsto \rho_2 \]

if \((B)\) then \(\{C_1\}\) else \(\{C_2\}\) \(\vdash \rho_1 \leadsto \rho_2\)

\[
| \text{bIf1 : forall rho rho' b c1 c2,} \\
| \text{beval rho b -> } \\
| \text{BStep c1 rho rho' -> } \\
| \text{BStep (If b c1 c2) rho rho'}
\]
Step relation, if (b)

\[
\rho \models B \Downarrow \text{False} \quad C_2 \models \rho_1 \rightsquigarrow \rho_2
\]

if (B) then \{C_1\} else \{C_2\} \models \rho_1 \rightsquigarrow \rho_2

\text{bIf2} : \text{forall } \rho, \rho', b, c_1, c_2, \text{beval } \rho, b \rightarrow \text{BStep } c_2 \rho, \rho' \rightarrow \text{BStep (If } b, c_1, c_2) \rho, \rho'
Step relation, while (a)

\[ \gamma \vdash B \downarrow \text{False} \]

while (B) \{C\} \vdash \rho \rightsquigarrow \rho

| bWhile1 : forall rho b c,  
| ~beval rho b ->  
| BStep (While b c) rho rho. |
Step relation, while (b)

$$\gamma \vdash B \downarrow \text{True} \quad C \vdash \rho \leadsto \rho' \quad \text{while (B) \{C\} \vdash \rho' \leadsto \rho''}$$

while B \{C\} \vdash \rho \leadsto \rho''

| bWhile1 : forall rho rho' rho'' b c,  
  beval rho b ->  
  BStep c rho rho' ->  
  BStep (While b c) rho' rho'' ->  
  BStep (While b c) rho rho''
Inductive BStep : Command -> context -> context -> Prop :=
| bSkip : forall rho,
  BStep Skip rho rho |
| bAssign : forall x ne rho,
  BStep (Assign x ne) rho (upd_ctx rho x (neval rho ne)) |
| bSeq : forall rho rho' rho'' c1 c2,
  BStep c1 rho rho' ->
  BStep c2 rho' rho'' ->
  BStep (Seq c1 c2) rho rho'' |
| bIf1 : forall rho rho' b c1 c2,
  beval rho b ->
  BStep c1 rho rho' ->
  BStep (If b c1 c2) rho rho' |
| bIf2 : forall rho rho' b c1 c2,
  ~beval rho b ->
  BStep c2 rho rho' ->
  BStep (If b c1 c2) rho rho' |
| bWhile2 : forall rho rho' rho'' b c,
  beval rho b ->
  BStep c rho rho' ->
  BStep (While b c) rho' rho'' ->
  BStep (While b c) rho rho'' |
| bWhile1 : forall rho b c,
  ~beval rho b ->
  BStep (While b c) rho rho.
We would like to formalize

\( \{\phi\} P \{\psi\} \)

Need to define:

1. Running a program \( P \) until it terminates
2. Initial state satisfies \( \phi \)
3. Resulting state satisfies \( \psi \).
What is an assertion?

We can do what we did for modal logic:

Definition assertion : Type :=
    ctx -> Prop.

We can even write $\rho \vdash \psi$ as shorthand for $\psi(\rho)$

Thus, we can use the rules of our Coq metalogic to easily reason about our Hoare assertions.
Lifting Assertions to Metalogic 1

\[ \rho \models \phi \land \psi \equiv (\rho \models \psi) \land (\rho \models \phi) \]

\begin{definition}
assertAnd (P Q : assertion) : assertion :=
    fun g => P g \land Q g.
\end{definition}

\textbf{Notation} "P \&\& Q" := (assertAnd P Q).

\[ \rho \models B \equiv \rho \vdash B \Downarrow \text{True} \]

\begin{definition}
assertbEval (b : bExpr) : assertion :=
    fun g => beEval g b.
\end{definition}

\textbf{Notation} "[ b ]" := (assertbEval b).
Defining Multimodal Operators

• Recall from modal logic the definitions of $\square$ and $\Diamond$ over some relation R:

  • $\rho \models \square P \equiv \forall \rho' (\rho R \rho' \rightarrow \rho' \models P)$
  • $\rho \models \Diamond P \equiv \exists \rho' (\rho R \rho' \land \rho' \models P)$
Defining Multimodal Operators

• We are going to generalize this idea: instead of “baking in” $R$, $\square$ and $\Diamond$ will take $R$ as a parameter:

• $\rho \models \square_R P \equiv \forall \rho' (\rho R \rho' \rightarrow \rho' \models P)$

• $\rho \models \Diamond_R P \equiv \exists \rho' (\rho R \rho' \land \rho' \models P)$
Defining Multimodal Operators

• Now all we have to do is define a relation between worlds and we automatically get a "reasonable" pair of □/◇ modal operators

• What kinds of relations might be of interest?

• What about the step relation c ⊩ ρ ↛ ρ′?

• Given a command c, this relates two contexts
Defining Multimodal Operators

- Here is what this idea looks like:

- \( \rho \models \Box_c P \equiv \forall \rho' (c \models \rho \sim \rho' \rightarrow \rho' \models P) \)
- \( \rho \models \Diamond_c P \equiv \exists \rho' (c \models \rho \sim \rho' \land \rho' \models P) \)

- What do these mean? How are they similar/different?
We would like to **formalize**

\[ \{ \phi \} P \{ \psi \} \]

Need to define:
1. Running a program P until termination
2. Initial state satisfies \( \phi \)
3. Resulting state satisfies \( \psi \).
Putting it all together

\{ \psi \} C \{ \phi \} \equiv \forall \rho \ (\rho \models (\psi \rightarrow c \phi))

[\psi] C [\phi] \equiv \forall \rho \ (\rho \models (\psi \rightarrow c \phi))

Definition HTriple (P) (c) (Q) :=
forall rho, (Impl P (SBox c Q)) rho.

Definition THTriple (P) (c) (Q) :=
forall rho, (Impl P (SDiam c Q)) rho.
Now what?

• Prove the Hoare rules as lemmas from definitions!

\[
\{\psi\} \; c_1 ; \; c_2 \; \{\psi\}
\]

Lemma HT_Skip: \(\text{forall } P,\)
\(\text{Htriple } P \; \text{Skip} \; P.\)
Assignment Rule

\[
\begin{array}{c}
\{[x \rightarrow E] \psi\} \quad x = E \quad \{\psi\}
\end{array}
\]

Lemma HT_Asgn: forall x e psi, HTriple [x => e @ psi] (Assign x e) psi.
Lifting Assertions to Metalogic 2

\[ \gamma \vdash [x \rightarrow e] \psi \equiv [x \rightarrow n] \gamma \vdash \psi \quad (\text{where } \gamma \vdash e \Downarrow n) \]

Definition assertReplace (x : var) (e : nExpr) (psi : assertion) : assertion :=
fun g => psi (upd_ctx g x (neEval g e)).

Notation "\[ x \Rightarrow e @ psi \]" :=
(assertReplace x e psi).

\[ \vdash_{AR} \phi \rightarrow \psi \equiv \forall \gamma, (\gamma \vdash \phi) \Rightarrow (\gamma \vdash \psi) \]

Definition Implies (P Q : assertion) : Prop :=
forall g, P g -> Q g.

Notation "P |-- Q" :=
(Implies P Q) (at level 30).
Sequence Rule

\[
\frac{\{\psi\} c_1 \{\chi\} \quad \{\chi\} c_2 \{\phi\}}{\{\psi\} c_1 ; c_2 \{\phi\}}
\]

Lemma HT_Seq: forall a1 c1 a2 c2 a3,

\[
\text{HTriple a1 c1 a2} \rightarrow \\
\text{HTriple a2 c2 a3} \rightarrow \\
\text{HTriple a1 (Seq c1 c2) a3}.
\]
Implied (Consequence) Rule

\[ \vdash_{AR} \phi' \rightarrow \phi \quad \{\phi\} \text{C} \quad \{\psi\} \quad \vdash_{AR} \psi \rightarrow \psi' \]

\[ \{\phi'\} \text{C} \quad \{\psi'\} \]

Lemma HT_Cons:

forall phi phi' psi psi' c,

phi' |-- phi ->
HTriple phi c psi ->
psi |-- psi' ->
HTriple phi' c psi'.
If Rule

\[
\begin{align*}
\{\phi \land B\} & \quad C_1 \quad \{\psi\} \quad \{\phi \land \neg B\} \quad C_2 \quad \{\psi\} \\
\{\phi\} & \quad \text{if } B \{C_1\} \quad \text{else } \{C_2\} \quad \{\psi\}
\end{align*}
\]

Lemma HT_If: forall \(\phi\) \(b\) \(c_1\) \(psi\) \(c_2\),
HTriple \((\phi \land \neg B)\) \(b\) \(c_2\) \(psi\) \(->\)
HTriple \((\phi \land bNeg \ b)\) \(c_2\) \(psi\) \(->\)
HTriple \(\phi\) (If \(b\) \(c_1\) \(c_2\)) \(psi\).
While Rule

\[ \{ \psi \land B \} \ C \ \{ \psi \} \]
\[ \{ \psi \} \text{ while } B \ \{ C \} \ \{ \psi \land \neg B \} \]

Lemma HT_While: forall psi b c,
HTriple (psi && [b]) c psi ->
HTriple psi (While b c) (psi && [bNeg b]).
Your task: Prove these lemmas

HT_Skip : 10 points
HT_Asgn : 10 points
HT_Seq : 10 points
HT_Implied : 10 points
HT_If : 10 points
HT_While : 20 points extra credit
Your task: Prove these lemmas

- THT_Skip : 10 points
- THT_Asgn : 10 points
- THT_Seq : 10 points
- THT_Implied : 10 points
- THT_If : 10 points
- THT_While : 20 points extra credit

(These are the total correctness versions)
Finally

Definition x : var := 0.
Definition y : var := 1.
Definition z : var := 2.
Open Local Scope Z_scope.

Definition neq (ne1 ne2 : nExpr) : bExpr :=
  Or (LT ne1 ne2) (LT ne2 ne1).

Definition factorial_prog : Coms :=
  Seq (Assign y (Num 1)) (* y := 1 *)
  (Seq (Assign z (Num 0)) (* z := 0 *)
    (While (neq (Var z) (Var x)) (* while z <> x { *)
      (Seq (Assign z (Plus (Var z) (Num 1))))
        (* z := z + 1 *)
      (Assign y (Times (Var y) (Var z)))) (* y := y * z *)
    ) (* } *)
  ).
Statement of Theorem

Definition Top : assertion := fun _ => True.

Open Local Scope nat_scope.

Fixpoint factorial (n : nat) :=
  match n with
  | O => 1
  | S n' => n * (factorial n')
  end.

Open Local Scope Z_scope.

Lemma factorial_good:
  HTuple Top factorial_prog (fun g => g y = Z_of_nat (factorial (Zabs_nat (g x)))).
Casts

Definition Top : assertion := fun _ => True.

Open Local Scope nat_scope.

Fixpoint factorial (n : nat) :=
  match n with
  | O => 1
  | S n' => n * (factorial n')
  end.

Open Local Scope Z_scope.

Lemma factorial_good:
  HTuple Top factorial_prog
  (fun g => g y = Z_of_nat (factorial (Zabs_nat (g x)))).
Proof of Theorem

Lemma factorial_good:

\[
\text{HTuple Top factorial_prog (fun g => g y = 0) with (fun g : ctx => g y = 1)).}
\]

Proof.

apply HT_Seq with (fun g => g y = 1).
replace Top with ((y => (Num 1) @ (fun g : ctx => g y = 1))).
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Seq with (fun g : ctx => g z = 0 /\ g y = 1).
replace (fun g : var => Z => g y = 1) with

\[
\text{((z => (Num 0)) @ (fun g : ctx => g z = 0 /\ g y = 1))}.
\]

apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Implicated with

\[
\text{(fun g => g z >= 0 /\ (g y) * ((g z) + 1) = Z_of_nat (factorial (Zabs_nat (g z)) + 1))}
\]

apply prop_ext.
firstorder.
unfold upd_ctx in H.
simpl in H.
auto with zarith.
simpl.
unfold upd_ctx.
simpl.
auto with zarith.
replace (fun g : var => Z => g z - 1 >= 0 /\ g y * (g z + 1) = Z_of_nat (factorial (Zabs_nat (g z)))) with

\[
\text{[(y => (Times (Var y) (Var z)) @ (fun g : var => Z => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))]}.
\]

apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
apply HT_While.
apply HT_Implicated with

\[
\text{(fun g => g z >= 0 /\ (g y)*((g z)+1) = Z_of_nat (factorial (Zabs_nat (g z)) + 1))}
\]

apply prop_ext.
firstorder.
unfold upd_ctx in H.
simpl in H.
auto with zarith.
simpl.
unfold upd_ctx.
simpl.
auto with zarith.
replace (fun g : var => Z => g z - 1 >= 0 /\ g y * (g z + 1) = Z_of_nat (factorial (Zabs_nat (g z)))) with

\[
\text{[(y => (Times (Var y) (Var z)) @ (fun g : var => Z => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))]}.
\]

apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
apply HT_While.
apply prop_ext.
firstorder.
apply HT_Implied with

\[
\text{(fun g => g z >= 0 /\ (g y)*((g z)+1) = Z_of_nat (factorial (Zabs_nat (g z)) + 1))}
\]

apply prop_ext.
firstorder.
unfold upd_ctx in H.
simpl in H.
auto with zarith.
simpl.
unfold upd_ctx.
simpl.
auto with zarith.
replace (fun g : var => Z => g z - 1 >= 0 /\ g y * (g z + 1) = Z_of_nat (factorial (Zabs_nat (g z)))) with

\[
\text{[(y => (Times (Var y) (Var z)) @ (fun g : var => Z => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))]}.
\]

apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
apply HT_While.
apply HT_Implicated with

\[
\text{(fun g => g z >= 0 /\ (g y)*((g z)+1) = Z_of_nat (factorial (Zabs_nat (g z)) + 1))}
\]

apply prop_ext.
firstorder.
unfold upd_ctx in H.
simpl in H.
auto with zarith.
simpl.
unfold upd_ctx.
simpl.
auto with zarith.
replace (fun g : var => Z => g z - 1 >= 0 /\ g y * (g z + 1) = Z_of_nat (factorial (Zabs_nat (g z)))) with

\[
\text{[(y => (Times (Var y) (Var z)) @ (fun g : var => Z => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))]}.
\]

apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
apply HT_While.
apply prop_ext.
The good news...

Your HW does **not** require you to do one of these yourself (we are not without mercy...)

Still... why did I show it to you?
Seems like a lot of work... why bother?

Lemma factorial_good:
HTuple Top factorial_prog (fun g => g y =
  Z_of_nat (factorial (Zabs_nat (g
x)))).
Proof.
  apply HT_SEQ with (fun g => g y = 1).
  replace Top with (\{y => (Num 1) \} \$ (fun g :
  ctx => g y = 1))).
  apply HT_Asgn.
  extensionality g.
  unfold assertReplace, Top, upd_ctx.
  simpl.
  apply prop_ext.
  firstorder.
  apply HT_SEQ with (fun g :ctx => g z = 0
  \$ g y = 1).
  replace (fun g : var => Z => g y = 1)
  with
    (\{z => \{Num 0\} \} \$ (fun g :ctx
    => g z = 0 \$ g y = 1))).
  apply HT_Asgn.
  extensionality g.
  unfold assertReplace, Top, upd_ctx.
  simpl.
  apply prop_ext.
  firstorder.
  apply HT_Implied with
    (fun g => g z => 0 \$ g y = Z_of_nat
    (factorial (Zabs_nat (g z)))\$)
    -
    (fun g => g z => 0 \$ g y = Z_of_nat
    (factorial (Zabs_nat (g z)))\$ $&
    [bNeg (neg (Var z) (Var x))).
  repeat intro.
  destruct H.
  rewrite H, H0.
  firstorder.
  apply HT_While.
  apply HT_Implied with
    (fun g => g z => 0 \$ g y = Z_of_nat
    (factorial (Zabs_nat (g z)))\$)
  -
    (fun g => g z => 0 \$ g y * (g z + 1) = Z_of_nat
    (factorial (Zabs_nat (g z + 1))))\$)
    &x
    [bNeg (neg (Var z) (Var x))).
  repeat intro.
  destruct H.
  rewrite H, H0.
  simp.
  firstorder.
  apply HT_While.
  apply HT_Implied with
    (fun g => g z => 0 \$ g y = Z_of_nat
    (factorial (Zabs_nat (g z)))\$)
  -
    (fun g => g z => 0 \$ g y * (g z + 1) = Z_of_nat
    (factorial (Zabs_nat (g z + 1))))\$)
    &x
    [bNeg (neg (Var z) (Var x))).
  apply prop_ext.
  firstorder.
  unfold upd_ctx in H.
  simpl in H.
  auto with zarith.
  simpl.
  unfold upd_ctx.
  simpl.
  auto with zarith.
  replace (fun g : var => Z => g z - 1 => 0
  \$ g y = Z_of_nat (factorial (Zabs_nat (g
  z))))\$)
    with
    [y => (Times (Var y) (Var z)) \$ (fun g :
      var => Z => g z - 1 => 0 \$ g y =
      Z_of_nat (factorial (Zabs_nat (g
      z))))\$)].
  apply HT_Asgn.
  extensionality g.
  apply prop_ext.
  firstorder.
  unfold prop_ext.
  simp.
  auto with zarith.
  simp.
  unfold prop_ext.
  simp.
  auto with zarith.
  replace (fun g : var => Z => g z - 1 => 0
  \$ g y = Z_of_nat (factorial (Zabs_nat (g
  z))))\$)
    with
    [y => (Times (Var y) (Var z)) \$ (fun g :
      var => Z => g z - 1 => 0 \$ g y =
      Z_of_nat (factorial (Zabs_nat (g
      z))))\$)].
  apply HT_Asgn.
  extensionality g.
  apply prop_ext.
  firstorder.
  unfold prop_ext.
  simp.
  auto with zarith.
  simp.
  unfold prop_ext.
  simp.
  auto with zarith.
  replace (fun g : var => Z => g z - 1 => 0
  \$ g y = Z_of_nat (factorial (Zabs_nat (g
  z))))\$)
    with
    [y => (Times (Var y) (Var z)) \$ (fun g :
      var => Z => g z - 1 => 0 \$ g y =
      Z_of_nat (factorial (Zabs_nat (g
      z))))\$)].
  apply HT_Asgn.
  extensionality g.
  apply prop_ext.
  firstorder.
Lemma factorial_good:
HTuple Top factorial_prog (fun g => g y = Z_of_nat (factorial (Zabs_nat (g x))))
Proof.
apply HT_Seq with (fun g => g y = 1).
replace Top with ((y => (Num 1) \ (fun g : ctx => g y = 1))).
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Seq with (fun g : ctx => g z = 0 \ g y = 1).
replace (fun g : var => Z => g y = 1) with
  \(\{z => \text{Num}(0)\} \ \& \ (\text{fun g : ctx => g z = 0 \ g y = 1})\).
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Implied with
\(\text{fun g => g z >= 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z)))}\) -
\(\text{fun g => g z >= 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z)))}\) \& &\n\[\text{bNeg (neq (Var z) (Var x))}\].
repeat intro.
destruct H.
rewrite H1.
split; auto.
remember (g z) as n.
clear H.
destruct n; auto.
simpl.
rewrite <- Pplus_one_succ_r.
rewrite inj_plus.
rewrite inj_mult.
auto with zarith.
replace (fun g : var => Z => g z >= 0 \ g y * g z = Z_of_nat (factorial (Zabs_nat (g z)))) with
\(\text{fun g : var => Z => g z >= 0 \ g y * g z = Z_of_nat (factorial (Zabs_nat (g z)))}\).
apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
unwrap upd_ctx in H.
simpl in H.
auto with zarith.
simpl.
unfold upd_ctx.
simpl.
auto with zarith.
rewrite (fun g : var => Z => g z - 1 >= 0 \ g y * g z = Z_of_nat (factorial (Zabs_nat (g z))) with
\(\text{fun g : var => Z => g z - 1 >= 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z)))}\)).
apply HT_Asgn.
extensionality g.
apply prop_ext.
firstorder.
destruct H.
rewrite H1.
simpl.
destruct intro; firstorder.
repeat intro.
destruct H.
destruct H.
rewrite H1.
simpl in H0.
destruct (Ztrichotomy (g z) (g x)).
contradiction H0; auto.destruct H2.
rewrite H2.
simpl.
rewrite <- H2.
trivial.
contradiction H0; right.
apply Zgt_lt.
trivial.
Qed.
Forgot to track boundary condition

\( z \geq 0 \) at all times in the loop

Lemma factorial_good:

\[ \text{HTuple Top factorial prog (fun g => g y = Z_of_nat (factorial (Zabs_nat (g x))) \} \]  

Proof.
apply HT_Seq with (fun g => g y = 1).
replace Top with \( (y \mapsto (\text{Num 1}) \ O \ (\text{fun } g:\ ctx \mapsto g y = 1)) \) .
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Seq with (fun g : ctx => g z = 0 \ g y = 1).
replace (fun g : var => Z => g y = 1) with
\( (\{z \mapsto \{\text{Num 0}\} \ O \ (\text{fun } g:\ ctx \mapsto g z = 0 \ / \ g y = 1)) \) .
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
apply HT_Implied with (fun g => g z = 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z)))) .
apply HT_Implied with
replace (fun g : var => Z => g y = Z_of_nat (factorial (Zabs_nat (g z))) \) .
repeat intro.
destruct H. 
destruct n; auto.
simplify.
rewrite H1. 
rewrite inj_plus.
rewrite inj_mult.
rewrite \( \text{Zpos eq} \ Z_{of\_nat} \_o\_nat_{of\_P} \_\_\_\text{ring} \).
unify_type False.
auto with zarith.
apply HT_Seq with (fun g => g z = 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z))) \) .
apply HT_Seq with (fun g : ctx => g z = 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z))) \) .
replace (fun g : var => Z => g z = 0 \ g y = Z_of_nat (factorial (Zabs_nat (g z))) \) .
apply HT_Asgn.
extensionality g.
Coercions (easily forgotten about...)

Fixpoint factorial (n : nat) := 
  match n with 
  | 0 => 1 
  | S n' => n * (factorial n') 
end.

fun g =>
g y = Z_of_nat (factorial (Zabs_nat (g x))).

We define factorial on nats because that way we have the best chance of not making a mistake in our specification.

But there is a cost: we must coerce from Z to N and back to Z...
Where you need this fact in the proof

Our “x!” has an implicit coercion in it: first we take the integer x, get the absolute value of it, and then calculate factorial on nats (and then coerce back to Z)...

while (z <> x) {
    {y = z! ∧ z <> x}  Now use Implied
    {y * (z + 1) = (z + 1)!}
Where you need this fact in the proof

Our “x!” has an implicit coercion in it: first we take the integer x, get the absolute value of it, and then calculate factorial on nats (and then coerce back to Z)...

```plaintext
while (z <> x) {
    {y = z! ∧ z <> x}       Now use Implied
    {y * (z + 1) = (z + 1)!}  ← But wait! What if z < 0?
}
```

Try y = 3, z = -4:

- $3 \times (-4 + 1) = -9$
- $(-4 + 1)! = (-3)! = 3! = 6$
The Explosion of the Ariane 5

- On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana.

- The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.

- A board of inquiry investigated the causes of the explosion and in two weeks issued a report.

- It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed.