Semantics of Hoare Logic

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

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

Informal meaning (already given):

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

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

Informal meaning (already given):

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

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

Need to define:

1. Running a program P
2. P terminates
3. State satisfies $\phi$
4. Resulting state satisfies $\psi$. 
Operational Semantics

• Numeric Expressions E:
  - n | x | (−E) | (E + E) | (E − E) | (E * E)

• Boolean Expressions B:
  - true | false | (!B) | (B&B) | (B||B) | (E < E)

• Commands C:
  - x = E | C;C | if B {C} else {C} | while B {C}
Expressions: syntax and semantics

• Numeric Expressions E:
  - n | x | (−E) | (E + E) | (E − E) | (E * 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 $E$:
  $-n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E \times E)$

Define a context $\gamma$ to be a function from variables to numbers.
We have to specify exactly how each evaluates

• Numeric Expressions E:
  - n | x | (−E) | (E + E) | (E − E) | (E * E)

Now define $\gamma \vdash E \downarrow n$ to mean “in context $\gamma$, the expression E evaluates to the numeric n.”
Boolean Evaluation

- Boolean Expressions B:
  - true | false | (!B) | (B&B) | (B||B) | (E < E)

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

What do we evaluate to? How about propositions?

So define $\gamma \vdash B \Downarrow P$ to mean “in context $\gamma$, the expression B evaluates to the proposition P.”
Commands

• Commands C:
  - $x = E$ | $C;C$ | if $B$ {C} else {C} | while $B$ {C} | crash

All of these look normal except for “crash” – which you can think of as dividing by zero. We add it to make the language a bit more interesting.
Command Evaluation

• Idea: executing command $C$ for one step moves the machine from one state to the next

• What is a state $\sigma$?

• Pair of context $\gamma$ (data) and control $k$ (code)

• Control $k$ is either $k_{\text{Stop}}$ (we are done) or $k_{\text{Seq} \ C \ k}$
  – We can write $C \bullet k$ for $k_{\text{Seq}}$ if that is easier
  – We can also write $\square$ for $k_{\text{Halt}}$
We now define the step relation, written

\[ \sigma_1 \rightarrow \sigma_2 \]

that is, “state \( \sigma_1 \) steps to state \( \sigma_2 \)”, in parts:

\[
\gamma \vdash E \downarrow n \quad \gamma' = [x \rightarrow n] \gamma \\
(\gamma, (x = E) \bullet k) \leftrightarrow (\gamma', k)
\]
Step relation, seq

\[(\gamma, (C_1 \; ; \; C_2) \bullet k) \iff (\gamma, C_1 \bullet (C_2 \bullet k))\]
Step relation, if (1 and 2)

\[ \gamma \vdash B \Downarrow \text{True} \]

\[ (\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_1 \bullet k) \]

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

\[ (\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_2 \bullet k) \]
Step relation, while (1 and 2)

\[
\begin{align*}
\gamma \vdash B \Downarrow \text{True} & \quad \Downarrow \quad (\gamma, (\text{while } B \{ C \} \bullet k) \leftrightarrow (\gamma, C \bullet (\text{while } B \{ C \} \bullet k))) \\
\gamma \vdash B \Downarrow \text{False} & \quad \Downarrow \quad (\gamma, (\text{while } B \{ C \} \bullet k) \leftrightarrow (\gamma, k)))
\end{align*}
\]
Entire step relation

\[
\begin{align*}
\gamma \vdash E \downarrow n & \quad \gamma' = [x \rightarrow n] \gamma \\
& \quad (\gamma, (x = E) \cdot k) \mapsto (\gamma', k)
\end{align*}
\]

\[
(\gamma, (C_1 ; C_2) \cdot k) \mapsto (\gamma, C_1 \cdot (C_2 \cdot k))
\]

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

(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \cdot k) \mapsto (\gamma, C_1 \cdot k)

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

(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \cdot k) \mapsto (\gamma, C_2 \cdot k)

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

(\gamma, (\text{while } B \{C\}) \cdot k) \mapsto (\gamma, C \cdot (\text{while } B \{C\}) \cdot k))

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

(\gamma, (\text{while } B \{C\}) \cdot k) \mapsto (\gamma, k))
What about crash??

• The point is that crash does not step anywhere – it just stops the machine in some kind of invalid state.

• This is different from ■, which also does not step anywhere but which is considered to be a “proper” way to stop the program.
From step to step*

• Usually we want to run our program for more than one step.

• We write $\sigma \mapsto^* \sigma'$ to mean that the state $\sigma$ steps to the state $\sigma'$ in some number of steps.
From step to step*

\[
\sigma \quad \vdash \quad \sigma^* \quad \sigma \\
\sigma \quad \vdash \quad \sigma' \\
\sigma' \quad \vdash \quad \sigma'' \\
\sigma \quad \vdash \quad \sigma''^* 
\]
We would like to formalize

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

Need to define:

1. Running a program $P$
2. $P$ terminates
3. State satisfies $\phi$
4. Resulting state satisfies $\psi$. 
First Attempt:
Terminates means eventually halted

• We say a state $(\gamma, k)$ is halted when $k = \blacksquare$

(First Attempt:)

• $\sigma$ terminates if $\exists \sigma'$ such that $\sigma \xrightarrow{*} \sigma'$ and $\sigma'$ is halted.

• This works well... except that it is terrible when we want to use it as a hypothesis.
Example: sequence rule

• Consider trying to prove the following rule

\[
\begin{array}{c}
\{\psi\} \ c_1 \ \{\chi\} \quad \{\chi\} \ c_2 \ \{\phi\} \\
\hline
\{\psi\} \ c_1 \ ; \ c_2 \ \{\phi\}
\end{array}
\]

Premise 1: if ... \(c_1\) terminates ... then ...
Premise 2: if ... \(c_2\) terminates ... then ...

c_1 ; c_2 *does not terminate* after running \(c_1\) – it then starts on \(c_2\). But that means that we can’t use premise 1 in our proof (or at least not very easily).
We would like to formalize

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

Need to define:

1. Running a program P
2. P terminates  \quad (Deferred until step 4)
3. State satisfies \phi
4. Resulting state satisfies \psi.
What is an assertion?

The idea is that an assertion is a formula whose truth depends on the context:

\[ \psi, \phi : \gamma \rightarrow \{T, F\} \]

We can even write \( \gamma \models \psi \) as shorthand for \( \psi(\gamma) \)

We will see that this approach is very similar to modal logic (but not for a few more weeks)
Lifting Assertions to Metalogic

Now we want to define how the logical operators:

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

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

\[ \gamma \models [x \rightarrow e] \psi \equiv [x \rightarrow n] \gamma \models \psi \]
   
   (where \( \gamma \vdash e \Downarrow n \))

etc.
Implication of Assertions

It is also useful to have a notion that one formula implies another for any context.

\[ \phi \vdash \psi \equiv \forall \gamma, (\gamma \vdash \phi) \Rightarrow (\gamma \vdash \psi) \]

Note that this is very different from implication at the object level:

\[ \gamma \vdash \psi \Rightarrow \phi \equiv (\gamma \vdash \psi) \Rightarrow (\gamma \vdash \phi) \]
We would like to formalize \( \{\phi\} P \{\psi\} \)

Need to define:
1. Running a program P
2. P terminates
3. State satisfies \( \phi \)
4. Resulting state satisfies \( \psi \).
Better Approach

• Define safe(σ) as,
  - \( \forall \sigma'. \sigma \mapsto^* \sigma' \Rightarrow \) \( (\exists \sigma''. \sigma' \mapsto \sigma'') \lor (\sigma' \text{ is halted}) \)

• Among other things, if σ is safe then it never reaches crash.

• Define guards(P, k) as,
  - \( \forall \gamma. \gamma \models P \Rightarrow \text{safe}(\gamma, k) \)

• The idea is that if P guards the control k, then as long as P is true then k is safe to run.
Putting it all together

\{\psi\} C \{\phi\} \equiv \forall k. \text{guards}(\phi, k) \Rightarrow \text{guards}(\psi, C \circ k)

That is, for any continuation (rest of program) k, if \(\phi\) is enough to make k safe, then \(\psi\) is enough to make C followed by k safe.

Question: does \(\phi\) hold after executing C?
Testers

• Answer: yes! We pick a k that “tests $\phi$”.

• For example, if $\phi \equiv x = 3$, then we pick
  – k $\equiv$ if x = 3 then x = x else crash
  – (this is why crash is useful to add to the language!)

• Obviously, if $\gamma \models \phi$, then this k is safe (since x=x does no harm).

• But if $\phi$ does not hold, then this program will not be safe.
Putting it all together

\{\psi\} C \{\phi\} \equiv \forall k. \text{guards}(\phi, k) \Rightarrow
\text{guards}(\psi, C \bullet k)

Thus in fact, if we know \{\psi\} C \{\phi\}, we know that C must make \phi true after it executes (assuming that \psi was true before running C)
Now what?

- Prove the Hoare rules as lemmas from definitions!

\[
\begin{array}{c}
\{\psi\} \quad c_1 \quad \{\chi\} \\
\{\chi\} \quad c_2 \quad \{\phi\} \\
\{\psi\} \quad c_1 \; ; \; c_2 \quad \{\phi\} \\
\hline
\{[x \rightarrow E] \quad \psi\} \quad x = E \quad \{\psi\}
\end{array}
\]
If, While Rules

{ϕ∧B} C₁ {ψ} {ϕ∧¬B} C₂ {ψ}

{ϕ} if B {C₁} else {C₂} {ψ}

{ψ∧B} C {ψ}

{ψ} while B {C} {ψ∧¬B}
Implied Rule

\[
\phi' \vdash \phi \quad \{\phi\} \mathcal{C} \{\psi\} \quad \psi \vdash \psi'
\]

\[
\{\phi'\} \mathcal{C} \{\psi'\}
\]
Your task on the next homework:
Prove these lemmas

HT_Seq : 10 points
HT_Asgn : 10 points
HT_If : 10 points
HT_Implied : 5 points
HT_While : 20 points extra credit
(good luck!)
Finally

Definition \( x : \text{var} := 0 \).
Definition \( y : \text{var} := 1 \).
Definition \( z : \text{var} := 2 \).
Open Local Scope \( Z\_scope \).

Definition \( \text{neq} (\text{ne1 ne2} : \text{nExpr}) : \text{bExpr} := \)
\hspace{1em} \text{Or} (\text{LT ne1 ne2}) (\text{LT ne2 ne1}).

Definition \( \text{factorial\_prog} : \text{Coms} := \)
\hspace{1em} \text{Seq} (\text{Assign} \ y \ (\text{Num} 1)) (* \ y := 1 *)
\hspace{1em} (\text{Seq} (\text{Assign} \ z \ (\text{Num} 0)) (* \ z := 0 *)
\hspace{1em} (\text{While} (\text{neq} (\text{Var} z) (\text{Var} x)) (* \ \text{while} \ z <> x \ { * )
\hspace{1em} \text{Seq} (\text{Assign} \ z \ (\text{Plus} (\text{Var} z) (\text{Num} 1))))
\hspace{1em} (* \ z := z + 1 *)
\hspace{1em} (\text{Assign} \ y \ (\text{Times} (\text{Var} y) (\text{Var} z))))(* \ y := y * z *)
\hspace{1em} (* \ } *)
).
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
  | 0 => 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:
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_Imp with
(fun g => g z >=0 /\ (g y) * ((g z) + 1) = Z_of_nat (factorial (Zabs_nat (g z + 1))))
(fun : ctx => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))).
repeat intro.
destruct H.
destruct H.
clear H0.
rewrite H1.
split; auto.
remember (g z) as n.
clear -H.
destruct n; auto.
simpl.
rewrite <- Pplus_one_succ_r.
rewrite nat_of_P_succ_morphism.
simpl.
remember (factorial (nat_of_P n)).
clear.
rewrite Zpos_succ_morphism.
rewrite inj_plus.
rewrite inj_mult.
rewrite <- Zpos_eq_Z_of_nat_o_nat_of_P.
ring.
elemtypen True.
auto with zarith.
apply HT_Seq with (fun g => g z - 1 >= 0 /\ g y * g z = Z_of_nat (factorial (Zabs_nat (g z)))).
repeat intro.
destruct H.
rewrite H0.
simpl.
firstorder.
apply HT_While.
apply HT_Imp with
(fun g => g z >=0 /\ (g y) * ((g z) + 1) = Z_of_nat (factorial (Zabs_nat (g z + 1))))
(fun : ctx => g z - 1 >= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))).
repeat intro.
destruct H.
destruct H.
clear H0.
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.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
firstorder.
destructor.
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 => (Plus (Var y) (Num 1)) @ (fun g : var -> Z => g z - 1 >= 0 /\ g y * g z = Z_of_nat (factorial (Zabs_nat (g z))))).
apply HT_Asgn.
extensionality g.
repeat intro; firstorder.
destructor.
destructor.
destructor.
destructor.
rewrite H1.
simpl in H0.
destructor (Ztrichotomy (g z) (g x)).
simpl in H0.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2.
rewrite H1.
rewrite H1.
rewrite H1.
destructor H2.
rewrite H2. 
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.
fistorder.
apply HT_Seq with (fun g :ctx => g z = 0
\/ \ y y = 1).
replace (fun g : var -> Z = g y = 1)
with
((z => (Num 0) @ (fun g :ctx
=> g z = 0 \ /\ y y = 1)).
apply HT_Asgn.
extensionality g.
unfold assertReplace, Top, upd_ctx.
simpl.
apply prop_ext.
fistorder.
apply HT_Seq with (fun g :ctx => g z = 0
\ /\ y y = 1).
apply HT_Implied with
(fun g => g z >=0 \ /\ (g y) * ((g z) + 1)
= Z_of_nat (factorial (Zabs_nat (g z)
+ 1)))]
(fun g : ctx => g z = 1 >= 0 \ /\ g y =
Z_of_nat (factorial (Zabs_nat (g
z)))).
repeat intro.
destruct H.
destruct H.
clear H0.
rewrite H1.
split; auto.
remember (g z) as n.
clear -H.
destruct n; auto.
simpl.
rewrite Pplus_one_succ_r.
rewrite nat_of_P_succ_morphism.
simpl.
remember (factorial (nat_of_P p)).
clear.
rewrite Zpos_succ_morphism.
rewrite inj_plus.
rewrite inj_mult.
rewrite -Zpos_eq_Z_of_nat_o_nat_of_P.
ring.
elimtype False.
auto with zarith.
apply HT_Seq with (fun g => g z = Z_of_nat
(factorial (Zabs_nat (g z)))]
(fun g => g z = 0 \ /\ g y =
Z_of_nat (factorial (Zabs_nat (g
z))))) & &

[\neg (eq (Var z) (Var x))].
repeat intro.
destruct H.
rewrite H, H0.
simpl.
fistorder.
apply HT_Asgn.
extensionality g.
apply prop_ext.
fistorder.
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))))]
apply HT_Asgn.
extensionality g.
apply prop_ext.
fistorder.
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))))]
apply HT_Asgn.
extensionality g.
apply prop_ext.
fistorder.
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))))]
apply HT_Asgn.
extensionality g.
apply prop_ext.
fistorder.
Lemma factorial_good:

\text{HTuple \text{Top \ factorial\_prog \ (fun \ g \to \ g \ y = \ Z\_of\_nat \ (\text{factorial} \ (\text{Zabs\_nat} \ (g \ x)))\)).}

Proof.

- apply \text{HT\_Seq} with (fun \ g \to \ g \ y = 1).
- replace \text{Top with ([y \to \ (\text{Num} 1) \@ (fun \ g : ctx \Rightarrow g \ y = 1)]).}
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simpl.
- apply \text{prop\_ext}.
- firstorder.
- apply \text{HT\_Seq with (fun \ g : ctx \to g \ z = 0 \&\& g \ y = 1)}.
- replace (fun \ g : var \to Z \Rightarrow g \ y = 1) with
  
  \((z \Rightarrow (\text{Num} 0) \@ (fun \ g : ctx \Rightarrow g \ z = 0 \&\& g \ y = 1))).
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simpl.
- apply \text{prop\_ext}.
- firstorder.
- apply \text{HT\_Implied with}

  \((\text{fun g \Rightarrow g z \Rightarrow 0} \&\& g y = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z))))\))

  &
  
  \((B\text{Neg} \ (\text{neq} \ (\text{Var} z) \ (\text{Var} x))))\).
- repeat intro.
- destruct \text{H}.
- rewrite \text{H}.
- simpl.
- unfold assertReplace in \text{H}.
- simpl.
- use \text{zarith}.
- simpl.
- unfold assertReplace.
- simpl.
- use \text{zarith}.
- apply \text{HT\_Seq with (fun \ g : ctx \to g \ z = 0 \&\& g \ y = 1)}.
- replace (fun \ g : var \to Z \Rightarrow g \ y * g z = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z)))) with
  
  \((y \Rightarrow (\text{Times} \ (\text{Var} y) \ (\text{Var} z)) \@ (fun \ g : var \to Z \Rightarrow g z = \text{Z\_of\_nat} \ (\text{factorial} \ (Zabs\_nat \ (g z))))))\).
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simp.
- apply \text{prop\_ext}.
- firstorder.
- unfold upd\_ctx in \text{H}.
- simp.
- auto with \text{zarith}.
- simplify in \text{H}.
- auto with \text{zarith}.
- apply \text{HT\_Seq with (fun \ g : ctx \Rightarrow g \ z = 0 \&\& g \ y = 1)}.
- replace (fun \ g : var \to Z \Rightarrow g \ y * (g z + 1) = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z)))) with
  
  \((z \Rightarrow (\text{Plus} \ (\text{Var} z) \ (\text{Num} 1)) \@ (fun \ g : var \to Z \Rightarrow g \ z = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z))))))\).
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simp.
- apply \text{prop\_ext}.
- firstorder.
- unfold \text{HT\_Seq in H}.
- simp in H.
- auto with \text{zarith}.
- simp.
- unfold \text{HT\_Seq in H}.
- simp.
- auto with \text{zarith}.
- replace (fun \ g : var \to Z \Rightarrow g \ z = 0 \&\& g \ y = 1)
  
  &
  
  \((\text{fun g \Rightarrow g z \Rightarrow 0} \&\& g y = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z))))\))

  &
  
  \((B\text{Neg} \ (\text{neq} \ (\text{Var} z) \ (\text{Var} x))))\).
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simp.
- auto with \text{zarith}.
- simpl.
- unfold assertReplace in \text{H}.
- simpl.
- use \text{zarith}.
- simpl.
- unfold assertReplace.
- simpl.
- use \text{zarith}.
- apply \text{HT\_Seq with (fun \ g : ctx \Rightarrow g \ z = 0 \&\& g \ y = 1)}.
- replace (fun \ g : var \to Z \Rightarrow g \ y * (g z + 1) = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z)))) with
  
  \((z \Rightarrow (\text{Plus} \ (\text{Var} z) \ (\text{Num} 1)) \@ (fun \ g : var \to Z \Rightarrow g \ z = Z\_of\_nat \ (\text{factorial} \ (Zabs\_nat \ (g z))))))\).
- apply \text{HT\_Asgn}.
- unfold assertReplace, \text{Top, upd\_ctx}.
- simp.
- auto with \text{zarith}.
- simp.
- unfold assertReplace in \text{H}.
- simpl.
- apply \text{prop\_ext}.
- firstorder.
- repeat intro; firstorder.
- destruct \text{H}.
- destruct \text{H}.
- rewrite \text{H}.
- simp in \text{H}.
- destruct \text{Z\_trichotomy} \text{g z} \text{g x}.
- contradiction \text{H0}.
- auto with \text{zarith}.
- contradiction \text{H0}.
- destruct \text{H2}.
- rewrite \text{H}.
- trivial.
- contradiction \text{H0}.
- right.
- apply \text{Z\_gt\_lt}.
- trivial.
- Qed.

42
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
((x => (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 y = 0 /\ (g y) * ((g y) + 1) = Z_of_nat (factorial (Zabs_nat ((g y) + 1))))
(fun g : ctx => g z <= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z))))).
repeat intro.
destruct H.
destruct H.
clear H0.
rewrite H1.
split; auto.
remember (g z) as n.
clear -H.
destruct n; auto.
simpl.
rewrite <- Pplus_one_succ_r.
rewrite nat_of_P_succ_morphism.
simpl.
remember (factorial (nat_of_P n)).
clear.
rewrite Zpos_succ_morphism.
rewrite inj_plus.
rewrite inj_mult.
rewrite <- Zpos_eq_Z_of_nat_o_nat_of_P.
ring.
eilitype False.
auto with zarith.
apply HT_Seq with (fun g => g z <= 0 /\ (g y) * ((g z) + 1) = Z_of_nat (factorial (Zabs_nat (g z + 1))))
(fun g : ctx => g z <= 0 /\ g y = Z_of_nat (factorial (Zabs_nat (g z)))).
repeat intro.
destruct H.
destruct H.
simpl.
firstorder.
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 = 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.
repeat intro; firstorder.
repeat intro.
destruct H.
destruct H.
destruct H.
destruct H.
rewrite H1.
simpl in H0.
destruct (Ztrichotomy (g z) (g x)).
contrastiction H0; auto.
destruct H2.
rewrite <- H2.
trivial.
contrastiction H0.
right.
apply 2gt_1t .
trivial.
Qed.
Coercions (easily forgotten about...)

Fixpoint factorial (n : nat) :=
  match n with
  | O => 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)...

\[
\text{while } (z <> x) \{
\begin{align*}
\{ & y = z! \land z <> x \} \quad \text{Now use Implied} \\
& \{ y \times (z + 1) = (z + 1)! \}
\end{align*}
\]
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} \quad \text{Now use Implied}
    \{y * (z + 1) = (z + 1)!\} \quad \text{← But wait! What if z < 0?}

Try y = 3, z = -4:

\begin{align*}
3 * (-4 + 1) &= -9 \\
(-4 + 1)! &= (-3)! = 3! &= 6
\end{align*}
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.