Semantics of Hoare Logic

Aquinas Hobor and Martin Henz

What does a Hoare tuple mean?

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Informal meaning (already given):

> "If the program P is run in a state that satisfies ϕ and P terminates, then the state resulting from P's execution will satisfy ψ."

We would like to **formalize**

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Informal meaning (already given):

> "If the program P is run in a state that satisfies ϕ and P terminates, then the state resulting from P's execution will satisfy ψ."

We would like to **formalize**

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Need to define:

1. **Running a program P**
2. P terminates
3. State satisfies ϕ
4. Resulting state satisfies ψ.

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 | x | (−E) | (E + E) | (E − E) | (E ∗ E)

Define a context γ 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 γ, 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 γ, 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 σ?
- Pair of context γ (data) and control k (code)
- Control k is either kStop (we are done) or kSeq C k
  - We can write C ● k for kSeq if that is easier
  - We can also write ■ for kHalt

Step relation, assign

We now define the step relation, written

$$\sigma_1 \mapsto \sigma_2$$

that is, "state σ_1 steps to state σ_2", in parts:

$$\frac{\gamma \vdash E \Downarrow n \qquad \gamma' = [x \to n]\,\gamma}{(\gamma, (x = E) \bullet k) \mapsto (\gamma', k)}$$

Step relation, seq

$$\frac{}{(\gamma, (C_1 ; C_2) \bullet k) \mapsto (\gamma, C_1 \bullet (C_2 \bullet k))}$$

Step relation, if (1 and 2)

$$\frac{\gamma \vdash B \Downarrow \text{True}}{(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_1 \bullet k)}$$

$$\frac{\gamma \vdash B \Downarrow \text{False}}{(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_2 \bullet k)}$$

Step relation, while (1 and 2)

$$\frac{\gamma \vdash B \Downarrow \text{True}}{(\gamma, (\text{while } B \{C\}) \bullet k) \mapsto (\gamma, C \bullet (\text{while } B \{C\} \bullet k))}$$

$$\frac{\gamma \vdash B \Downarrow \text{False}}{(\gamma, (\text{while } B \{C\}) \bullet k) \mapsto (\gamma, k))}$$

Entire step relation

$$\frac{\gamma \vdash E \Downarrow n \qquad \gamma' = [x \to n]\,\gamma}{(\gamma, (x = E) \bullet k) \mapsto (\gamma', k)}$$

$$\frac{}{(\gamma, (C_1 ; C_2) \bullet k) \mapsto (\gamma, C_1 \bullet (C_2 \bullet k))}$$

$$\frac{\gamma \vdash B \Downarrow \text{True}}{(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_1 \bullet k)}$$

$$\frac{\gamma \vdash B \Downarrow \text{False}}{(\gamma, (\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}) \bullet k) \mapsto (\gamma, C_2 \bullet k)}$$

$$\frac{\gamma \vdash B \Downarrow \text{True}}{(\gamma, (\text{while } B \{C\}) \bullet k) \mapsto (\gamma, C \bullet (\text{while } B \{C\} \bullet k))}$$

$$\frac{\gamma \vdash B \Downarrow \text{False}}{(\gamma, (\text{while } B \{C\}) \bullet k) \mapsto (\gamma, k))}$$

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 consider 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 σ steps to the state σ' in some number of steps.

From step to step*

$$\frac{}{\sigma \mapsto^* \sigma}$$

$$\frac{\sigma \mapsto \sigma' \qquad \sigma' \mapsto^* \sigma''}{\sigma \mapsto^* \sigma''}$$

We would like to **formalize**

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Need to define:

1. Running a program P
2. **P terminates**
3. State satisfies ϕ
4. Resulting state satisfies ψ.

First Attempt: Terminates means eventually halted

- We say a state (γ, k) is halted when k = ■

(First Attempt:)

- σ terminates if $\exists\, \sigma'$ such that $\sigma \mapsto^* \sigma'$ and σ' 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

$$\frac{\{\psi\}\ c_1\ \{\chi\} \qquad \{\chi\}\ c_2\ \{\phi\}}{\{\psi\}\ c_1\ ;\ c_2\ \{\phi\}}$$

Premise 1: if … c1 terminates … then …

Premise 2: if … c2 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**

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Need to define:

1. Running a program P
2. P terminates (**Deferred until step 4**)
3. **State satisfies ϕ**
4. Resulting state satisfies ψ.

What is an assertion?

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

$$\psi, \phi \quad : \quad \gamma \rightarrow \{\mathsf{T}, \mathsf{F}\}$$

We can even write $\gamma \vDash \psi$ 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:

$$\gamma \vDash \phi \wedge \psi \quad \equiv \quad (\gamma \vDash \psi) \wedge (\gamma \vDash \phi)$$

$$\gamma \vDash \mathsf{B} \quad \equiv \quad \gamma \vdash \mathsf{B} \Downarrow \mathsf{True}$$

$$\gamma \vDash [\mathsf{x} \to \mathsf{e}]\, \psi \quad \equiv \quad [\mathsf{x} \to \mathsf{n}]\, \gamma \vDash \psi$$

(where $\gamma \vdash \mathsf{e} \Downarrow \mathsf{n}$)

etc.

Implication of Assertions

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

$$\phi \vdash \psi \quad \equiv \quad \forall\, \gamma, (\gamma \models \phi) \Rightarrow (\gamma \models \psi)$$

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

$$\gamma \models \psi \Rightarrow \phi \quad \equiv \quad (\gamma \models \psi) \Rightarrow (\gamma \models \phi)$$

We would like to **formalize**

$$\{\phi\}\ \mathrm{P}\ \{\psi\}$$

Need to define:

1. Running a program P
2. P terminates
3. State satisfies ϕ
4. **Resulting state satisfies ψ.**

Better Approach

- Define safe(σ) as,
  - $\forall\, \sigma'.\ \ \boldsymbol{\sigma} \mapsto^{*} \boldsymbol{\sigma'} \ \ \Rightarrow$
    ($\exists\, \sigma''.\ \boldsymbol{\sigma'} \mapsto \boldsymbol{\sigma''}$) $\ \vee\ $ (σ' is halted)

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

- Define guards(P, k) as,
  - $\forall\, \gamma.\ \ \gamma \vDash \mathsf{P} \ \ \Rightarrow\ \ \mathsf{safe}(\gamma, \mathsf{k})$

- 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. guards(ϕ, k) $\Rightarrow$
guards (ψ, C $\bullet$ k)

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

Question: does ϕ hold after executing C?

Testers

- Answer: yes! We pick a k that "tests ϕ".
- 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 $\gamma \vDash \phi$, then this k is safe (since x=x does no harm).
- But if ϕ does not hold, then this program will not be safe.

Putting it all together

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

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

Now what?

- Prove the Hoare rules as lemmas from definitions!

$$\frac{\{\psi\}\ c_1\ \{\chi\} \qquad \{\chi\}\ c_2\ \{\phi\}}{\{\psi\}\ c_1\ ;\ c_2\ \{\phi\}}$$

$$\frac{}{\{[x \to E]\ \psi\} \quad x = E \quad \{\psi\}}$$

If, While Rules

$$\frac{\{\phi \wedge \mathrm{B}\}\ \mathrm{C}_1\ \{\psi\} \qquad \{\phi \wedge \neg\, \mathrm{B}\}\ \mathrm{C}_2\ \{\psi\}}{\{\phi\}\ \text{if B } \{\mathrm{C}_1\} \text{ else } \{\mathrm{C}_2\}\ \{\psi\}}$$

$$\frac{\{\psi \wedge \mathrm{B}\}\ \mathrm{C}\ \{\psi\}}{\{\psi\} \text{ while B } \{\mathrm{C}\}\ \{\psi \wedge \neg\, \mathrm{B}\}}$$

Implied Rule

$$\frac{\phi' \vdash \phi \qquad \{\phi\}\ C\ \{\psi\} \qquad \psi \vdash \psi'}{\{\phi'\}\ C\ \{\psi'\}}$$

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 : 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:
  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 (neq (Var z) (Var x))]).
    repeat intro.
    destruct H.
    rewrite H, H0.
    simpl.
    firstorder.
  apply HT_While.
  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 - 1 >= 0
    /\ g y * g z = Z_of_nat (factorial
    (Zabs_nat (g z)))).
replace (fun g : var -> Z => g z >= 0 /\ g
    y * (g z + 1) = Z_of_nat (factorial
    (Zabs_nat (g z + 1)))) with
  [z => (Plus (Var z) (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.
    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.
    rewrite H1.
    simpl in H0.
    destruct (Ztrichotomy (g z) (g x)).
    contradiction H0; auto.
    destruct H2.
    rewrite <- H2.
    trivial.
    contradiction H0.
    right.
    apply Zgt_lt .
    trivial.
Qed.
```

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 (neq (Var z) (Var x))]).
    repeat intro.
    destruct H.
    rewrite H, H0.
    simpl.
    firstorder.
  apply HT_While.
  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 - 1 >= 0
    /\ g y * g z = Z_of_nat (factorial
    (Zabs_nat (g z)))).
replace (fun g : var -> Z => g z >= 0 /\ g
    y * (g z + 1) = Z_of_nat (factorial
    (Zabs_nat (g z + 1)))) with
  [z => (Plus (Var z) (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.
    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.
    rewrite H1.
    simpl in H0.
    destruct (Ztrichotomy (g z) (g x)).
    contradiction H0; auto.
    destruct H2.
    rewrite <- H2.
    trivial.
    contradiction H0.
    right.
    apply Zgt_lt .
    trivial.
Qed.
```

Bug in Paper Proof

```
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 (neq (Var z) (Var x))]).
    repeat intro.
    destruct H.
    rewrite H, H0.
    simpl.
    firstorder.
  apply HT_While.
  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 - 1 >= 0
    /\ g y * g z = Z_of_nat (factorial
    (Zabs_nat (g z)))).
replace (fun g : var -> Z => g z >= 0 /\ g
    y * (g z + 1) = Z_of_nat (factorial
    (Zabs_nat (g z + 1)))) with
  [z => (Plus (Var z) (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.
    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.
    rewrite H1.
    simpl in H0.
    destruct (Ztrichotomy (g z) (g x)).
    contradiction H0; auto.
    destruct H2.
    rewrite <- H2.
    trivial.
    contradiction H0.
    right.
    apply Zgt_lt .
    trivial.
Qed.
```

Forgot to track boundary condition (z >= 0 at all times in the loop)

```
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 (neq (Var z) (Var x))]).
    repeat intro.
    destruct H.
    rewrite H, H0.
    simpl.
    firstorder.
  apply HT_While.
  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 - 1 >= 0
    /\ g y * g z = Z_of_nat (factorial
    (Zabs_nat (g z)))).
replace (fun g : var -> Z => g z >= 0 /\ g
    y * (g z + 1) = Z_of_nat (factorial
    (Zabs_nat (g z + 1)))) with
  [z => (Plus (Var z) (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.
    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.
    rewrite H1.
    simpl in H0.
    destruct (Ztrichotomy (g z) (g x)).
    contradiction H0; auto.
    destruct H2.
    rewrite <- H2.
    trivial.
    contradiction H0.
    right.
    apply Zgt_lt .
    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)…

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)…

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 * (-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.