Semantics of Hoare Logic

Aquinas Hobor and Martin Henz

What does a Hoare triple mean?

$$\{\phi\}\ \mathrm{c}\ \{\psi\}$$

Informal meaning (already given):

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

We would like to **formalize**

$$\{\phi\}\ \mathrm{c}\ \{\psi\}$$

Informal meaning (already given):

“If the program c is run in a state that satisfies ϕ and c terminates, then the state resulting from c’s execution will satisfy ψ.”

We would like to **formalize**

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

Need to define:

1. Running a program c until it terminates
2. Initial state satisfies ϕ
3. Resulting state satisfies ψ.

We would like to **formalize**

$$\{\phi\}\ \mathsf{c}\ \{\psi\}$$

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

We would like to **formalize**

$$\{\phi\}\ \mathsf{c}\ \{\psi\}$$

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**

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

Need to define:

1. **Running a program P until it terminates**
2. Initial state satisfies ϕ
3. Resulting state satisfies ψ.

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:

```
Definition var : Type := nat.
```

We have to specify exactly how each evaluates

- Numeric Expressions E:
  - z | x | (E + E) | (E − E) | (E ∗ E)

Next: how do we define our expressions?

```
Inductive nExpr : Type :=
  | Num: forall z : Z, nExpr
  | Var: forall v : 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 | x | (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:
  - z | x | (E + E) | (E − E) | (E ∗ E)

Define a context ρ to be a function from variables to numbers.

```
Definition ctx := var -> num.
```

We have to specify exactly how each evaluates

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

Now define $\rho \vdash E \Downarrow n$ to mean “in context ρ, 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 <= E) | (B||B) | (!B)

```
Inductive bExpr : Type :=
 | LE : forall ne1 ne2 : nExpr, bExpr
 | Or : forall be1 be2 : bExpr, bExpr
 | bNeg : forall be : bExpr, bExpr.
```

Boolean Evaluation

- Boolean Expressions B:
  - (E <= E) | (B||B) | (!B)

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

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

So define $\rho \vdash B \Downarrow P$ to mean "in context ρ, the expression B evaluates to the proposition P."

Boolean Evaluation

```
Fixpoint beval (g : ctx) (be : bExpr) : Prop :=
 match be with
  | LE ne1 ne2 => (neEval g ne1) <= (neEval g ne2)
  | Or be1 be2 => (beEval g be1) \/ (beEval g be2)
  | bNeg be => ~(beEval g be)
end.
```

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 ρ_α to an ending context ρ_ω
- We define a step relation that looks like this:

$$\text{c} \vdash \rho_\alpha \rightsquigarrow \rho_\omega$$

- 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

$$\frac{}{\mathsf{skip} \vdash \rho \rightsquigarrow \rho}$$

```
| bSkip : forall rho,
    BStep Skip rho rho
```

Step relation, assign

$$\frac{\rho \vdash \mathsf{E} \Downarrow \mathsf{n}}{(\mathsf{x} = \mathsf{E}) \vdash \rho \rightsquigarrow [\mathsf{x} \mapsto \mathsf{n}]\ \rho}$$

```
| bAssign : forall x ne rho,
  BStep (Assign x ne)
        rho
        (upd_ctx rho x (neval rho ne))
```

Step relation, seq

$$\frac{C_1 \vdash \rho_1 \rightsquigarrow \rho_2 \qquad C_2 \vdash \rho_2 \rightsquigarrow \rho_3}{(C_1 \,;\, C_2) \vdash \rho_1 \rightsquigarrow \rho_3}$$

```
| 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)

$$\frac{\rho \vdash B \Downarrow \text{True} \qquad C_1 \vdash \rho_1 \rightsquigarrow \rho_2}{\text{if (B) then } \{C_1\} \text{ else } \{C_2\} \vdash \rho_1 \rightsquigarrow \rho_2}$$

```
| bIf1 : forall rho rho' b c1 c2,
    beval rho b ->
    BStep c1 rho rho' ->
    BStep (If b c1 c2) rho rho'
```

Step relation, if (b)

$$\frac{\rho \vdash B \Downarrow \text{False} \qquad C_2 \vdash \rho_1 \rightsquigarrow \rho_2}{\text{if (B) then } \{C_1\} \text{ else } \{C_2\} \vdash \rho_1 \rightsquigarrow \rho_2}$$

```
| bIf2 : forall rho rho' b c1 c2,
    ~beval rho b ->
    BStep c2 rho rho' ->
    BStep (If b c1 c2) rho rho'
```

Step relation, while (a)

$$\frac{\gamma \vdash \text{B} \Downarrow \text{False}}{\text{while (B) \{C\}} \vdash \rho \rightsquigarrow \rho}$$

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

Step relation, while (b)

$$\frac{\gamma \vdash \mathsf{B} \Downarrow \mathsf{True} \quad \mathsf{C} \vdash \rho \rightsquigarrow \rho' \quad \mathsf{while}\ (\mathsf{B})\ \{\mathsf{C}\} \vdash \rho' \rightsquigarrow \rho''}{\mathsf{while}\ \mathsf{B}\ \{\mathsf{C}\} \vdash \rho \rightsquigarrow \rho''}$$

```
| 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**

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

Need to define:

1. Running a program P until it terminates
2. **Initial state satisfies** ϕ
3. Resulting state satisfies ψ.

What is an assertion?

We can do what we did for modal logic:

```
Definition assertion : Type :=
   ctx -> Prop.
```

We can even write $\rho \models \psi$ 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

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

```
Definition assertAnd (P Q : assertion) :
  assertion :=
  fun g => P g /\ Q g.
Notation "P && Q" := (assertAnd P Q).
```

$$\rho \vDash \mathrm{B} \quad \equiv \quad \rho \vdash \mathrm{B} \Downarrow \mathrm{True}$$

```
Definition assertbEval (b : bExpr) :
  assertion :=
  fun g => beEval g b.
Notation "[ b ]" := (assertbEval b).
```

Defining Multimodal Operators

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

- $\rho \models \Box P \quad \equiv \quad \forall \rho' (\rho \, R \, \rho' \rightarrow \rho' \models P)$
- $\rho \models \Diamond P \quad \equiv \quad \exists \rho' (\rho \, R \, \rho' \wedge \rho' \models P)$

Defining Multimodal Operators

- We are going to generalize this idea: instead of "baking in" R, $\Box$ and $\Diamond$ will take R as a parameter:

- $\rho \models \Box_R P \quad \equiv \quad \forall \rho' (\rho R \rho' \rightarrow \rho' \models P)$
- $\rho \models \Diamond_R P \quad \equiv \quad \exists \rho' (\rho R \rho' \land \rho' \models P)$

Defining Multimodal Operators

- Now all we have to do is define a relation between worlds and we automatically get a "reasonable" pair of $\Box$/$\Diamond$ modal operators

- What kinds of relations might be of interest?

- What about the step relation $c \vdash \rho \rightsquigarrow \rho'$!
- Given a command c, this relates two contexts

Defining Multimodal Operators

- Here is what this idea looks like:

- $\rho \models \Box_c \; P \quad \equiv \quad \forall \rho' \, (c \vdash \rho \rightsquigarrow \rho' \rightarrow \rho' \models P)$
- $\rho \models \Diamond_c \, P \quad \equiv \quad \exists \rho' \, (c \vdash \rho \rightsquigarrow \rho' \wedge \rho' \models P)$

- What do these mean? How are they similar/different?

We would like to **formalize**

$$\{\phi\} \text{ P } \{\psi\}$$

Need to define:

1. Running a program P until termination
2. Initial state satisfies ϕ
3. **Resulting state satisfies ψ.**

Putting it all together

$$\{\psi\}\ \mathrm{C}\ \{\phi\} \quad \equiv \quad \forall\, \rho\ \ (\rho \models (\psi \rightarrow \Box_{\mathrm{c}}\, \phi))$$

$$[\psi]\ \mathrm{C}\ [\phi] \quad \equiv \quad \forall\, \rho\ \ (\rho \models (\psi \rightarrow \Diamond_{\mathrm{c}}\, \phi))$$

```
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!

$$\frac{}{\{\psi\}\ c_1\ ;\ c_2\ \{\psi\}}$$

```
Lemma HT_Skip: forall P,
  Htriple P Skip P.
```

Assignment Rule

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

```
Lemma HT_Asgn: forall x e psi,
HTriple [x => e @ psi] (Assign x e)
  psi.
```

Lifting Assertions to Metalogic 2

$$\gamma \models [x \to e]\,\psi \quad \equiv \quad [x \to n]\,\gamma \models \psi \quad \text{(where } \gamma \vdash e \Downarrow n\text{)}$$

```
Definition assertReplace (x : var)
 (e : nExpr) (psi : assertion) : assertion :=
fun g => psi (upd_ctx g x (neEval g e)).
Notation "[ x => e @ psi ]" :=
  (assertReplace x e psi).
```

$$\vdash_{\mathbf{AR}} \phi \to \psi \quad \equiv \quad \forall\, \gamma, (\gamma \models \phi) \Rightarrow (\gamma \models \psi)$$

```
Definition Implies (P Q: assertion) : Prop :=
  forall g, P g -> Q g.
Notation "P |-- Q" :=
  (Implies P Q) (at level 30).
```

Sequence Rule

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

```
Lemma HT_Seq: forall a1 c1 a2 c2
  a3,
   HTriple a1 c1 a2 ->
   HTriple a2 c2 a3 ->
   HTriple a1 (Seq c1 c2) a3.
```

Implied (Consequence) Rule

$$\frac{\vdash_{\mathbf{AR}} \phi' \rightarrow \phi \qquad \{\phi\}\ \mathsf{C}\ \{\psi\} \qquad \vdash_{\mathbf{AR}} \psi \rightarrow \psi'}{\{\phi'\}\ \mathsf{C}\ \{\psi'\}}$$

```
Lemma HT_Cons:
 forall phi phi' psi psi' c,
  phi' |-- phi ->
  HTriple phi c psi ->
  psi |-- psi' ->
  HTriple phi' c psi'.
```

If Rule

$$\frac{\{\phi \wedge B\}\ C_1\ \{\psi\} \qquad \{\phi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\phi\}\ \text{if } B\ \{C_1\}\ \text{else}\ \{C_2\}\ \{\psi\}}$$

```
Lemma HT_If: forall phi b c1 psi c2,
  HTriple (phi && [b]) c1 psi ->
  HTriple (phi && ⌈bNeg b⌉) c2 psi->
  HTriple phi (If b c1 c2) psi.
```

While Rule

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

```
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:
  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.