Chapter 1

Library ddifc-coq

```
Require Import Omega.
Require Import Arith.
Require Import ZArith.
Require Import List.
Require Import Classical.
Require Import ProofIrrelevance.
Require Import FunctionalExtensionality.
Require Import Coq.Bool.Bool.

Ltac inv H := inversion H; try subst; try clear H.
Ltac dup H := generalize H; intro.
Ltac intuit := try solve [intuition].
Ltac decomp H := decompose [and or] H; try clear H.

Notation "[ ]" := nil (at level 1).
Notation "[ a ; .. ; b ]" := (a :: .. (b :: []) ..) (at level 1).

Proposition app_assoc {A} : ∀ l1 l2 l3 : list A, (l1 ++ l2) ++ l3 = l1 ++ l2 ++ l3.
Proof.
induction l1; simpl; intros; auto.
rewrite IHl1; auto.
Qed.

Proposition in_app_iff {A} : ∀ (l1 l2 : list A) x, In x (l1++l2) ↔ In x l1 ∨ In x l2.
Proof.
intros; split; intros.
apply in_app_or; auto.
apply in_or_app; auto.
Qed.

Proposition app_nil_r {A} : ∀ l : list A, l ++ [] = l.
Proof.
induction l; auto.
simpl; rewrite IHl; auto.
```

```
Qed.

Proposition list_finite {A} : ∀ (l : list A) x, l ≠ x :: l.
Proof.
induction l; intros; intro.
inv H.
inv H.
specialize (IHl x); contradiction.
Qed.

Proposition list_finite' {A} : ∀ l l' : list A, l' ≠ [] → l ≠ l' ++ l.
Proof.
induction l; intros; intro.
rewrite app_nil_r in H0; subst.
contradict H; auto.
destruct l'.
contradict H; auto.
inv H0.
subst a.
contradiction (IHl (l'++[a0])).
intro.
destruct l'; inv H0.
rewrite app_assoc; auto.
Qed.

Proposition app_cancel_l {A} : ∀ l l1 l2 : list A, l ++ l1 = l ++ l2 → l1 = l2.
Proof.
induction l; intros; auto.
inv H; intuit.
Qed.

Proposition app_cancel_r_help {A} : ∀ (l1 l2 : list A) x, l1 ++ [x] = l2 ++ [x] → l1 =
l2.
Proof.
induction l1; intros.
destruct l2; auto; inv H.
destruct l2; inv H2.
destruct l2; inv H.
destruct l1; inv H2.
apply IHl1 in H2; subst; auto.
Qed.

Proposition app_cancel_r {A} : ∀ l l1 l2 : list A, l1 ++ l = l2 ++ l → l1 = l2.
Proof.
induction l; intros.
repeat rewrite app_nil_r in H; auto.
```

```
change (l1++([a]++l) = l2++([a]++l)) in H.
repeat rewrite ← app_assoc in H; apply IHl in H.
apply app_cancel_r_help in H; auto.
Qed.

Definition var := nat.
Definition lvar1 := nat.
Definition lvar2 := nat.
Definition addr := nat.
Definition fname := nat.

Open Scope Z_scope.

Definition nat_of_Z (v : Z) (pf : v ≥ 0) : nat.
intros.
destruct v.
apply O.
apply (nat_of_P p).
assert (~ Zneg p ≥ 0).
clear pf; induction p.
intro; contradiction.
intro; contradiction.
intro H; contradiction H; simpl; auto.
contradiction.
Defined.

Proposition Zneg_dec : ∀ v : Z, {v ≥ 0} + {v < 0}.
Proof.
intros.
destruct v.
left; omega.
left.
induction p; auto.
omega.
right.
induction p; auto.
omega.
Qed.

Record poset {A : Set} : Type :=
{leq : A → A → bool;
 leq_refl : ∀ x : A, leq x x = true;
 leq_antisym : ∀ x y : A, leq x y = true → leq y x = true → x = y;
 leq_trans : ∀ x y z : A, leq x y = true → leq y z = true → leq x z = true}.

Record join_semi {A : Set} : Type :=
```

```
{po : poset (A:=A);
 lub : A → A → A;
 lub_l : ∀ x y : A, leq po x (lub x y) = true;
 lub_r : ∀ x y : A, leq po y (lub x y) = true;
 lub_least : ∀ x y z : A, leq po x z = true → leq po y z = true → leq po (lub x y) z =
true}.

Record join_semi' {A : Set} (js : join_semi (A:=A)) : Type :=
{lub_idem : ∀ x : A, lub js x x = x;
 lub_comm : ∀ x y : A, lub js x y = lub js y x;
 lub_assoc : ∀ x y z : A, lub js (lub js x y) z = lub js x (lub js y z);
 lub_leq : ∀ x y z : A, leq (po js) (lub js x y) z = true ↔ leq (po js) x z = true ∧ leq (po
js) y z = true}.

Definition join_semi_extend {A : Set} (js : join_semi (A:=A)) : join_semi' (A:=A) js.
intros; split; intros.
apply (leq_antisym (po js)).
apply lub_least; apply leq_refl.
apply lub_l.
apply (leq_antisym (po js)); solve [apply lub_least; [apply lub_r | apply lub_l]].
apply (leq_antisym (po js)).
apply lub_least.
apply lub_least.
apply lub_l.
apply (leq_trans _ _ (lub js y z) _); [apply lub_l | apply lub_r].
apply (leq_trans _ _ (lub js y z) _); [apply lub_r | apply lub_r].
apply lub_least.
apply (leq_trans _ _ (lub js x y) _); [apply lub_l | apply lub_l].
apply lub_least.
apply (leq_trans _ _ (lub js x y) _); [apply lub_r | apply lub_l].
apply lub_r.
split; intros; try split.
apply (leq_trans _ _ (lub js x y) _); [apply lub_l | auto].
apply (leq_trans _ _ (lub js x y) _); [apply lub_r | auto].
apply lub_least; intuit.
Qed.

Record bounded_join_semi {A : Set} : Type :=
{js : join_semi (A:=A);
 bot : A;
 leq_bot : ∀ x : A, leq (po js) bot x = true}.

Record bounded_join_semi' {A : Set} (bjs : bounded_join_semi (A:=A)) : Type :=
{bot_unit : ∀ x : A, lub (js bjs) x (bot bjs) = x}.

Definition bounded_join_semi_extend {A : Set} (bjs : bounded_join_semi (A:=A)) : bounded_join_semi'
```

```
(A:=A) bjs.
intros; split; intros.
apply (leq_antisym (po (js bjs))).
apply lub_least; [apply leq_refl | apply leq_bot].
apply lub_l.
Qed.

Coercion po : join_semi >-> poset.
Coercion js : bounded_join_semi >-> join_semi.

Parameter lbl : Set.
Parameter lbl_lattice : bounded_join_semi (A:=lbl).
Definition lbl_lattice' := join_semi_extend lbl_lattice.
Definition lbl_lattice'' := bounded_join_semi_extend lbl_lattice.
Definition bottom := bot lbl_lattice.
Definition llub := lub lbl_lattice.
Definition lleq := leq lbl_lattice.

Ltac llub_simpl H := apply (lub_leq lbl_lattice lbl_lattice') in H; destruct H.

Inductive glbl := Lo | Hi.
Definition grp (L l : lbl) := if lleq l L then Lo else Hi.

Definition glbl_poset : poset (A:=glbl).
apply Build_poset with (leq := fun l1 l2 : glbl ⇒ if l1 then true else (if l2 then false
else true)); intros.
destruct x; auto.
destruct x; destruct y; simpl in *; auto; inv H; inv H0.
destruct x; destruct y; destruct z; simpl in *; auto.
Defined.

Definition glbl_join_semi : join_semi (A:=glbl).
apply Build_join_semi with (po := glbl_poset) (lub := fun l1 l2 : glbl ⇒ if l1 then l2
else Hi); intros.
destruct x; destruct y; auto.
destruct x; destruct y; auto.
destruct x; destruct y; auto.
Defined.

Definition glbl_lattice : bounded_join_semi (A:=glbl).
apply Build_bounded_join_semi with (js := glbl_join_semi) (bot := Lo); auto.
Defined.

Definition glbl_lattice' := join_semi_extend glbl_lattice.
Definition glbl_lattice'' := bounded_join_semi_extend glbl_lattice.
Definition gleq := leq glbl_lattice.
Definition glub := lub glbl_lattice.

Delimit Scope glbl_scope with glbl.
```

```
Bind Scope glbl_scope with glbl.
Delimit Scope lbl_scope with lbl.
Bind Scope lbl_scope with lbl.
Notation "x «= y" := (gleq x y = true) (at level 70) : glbl_scope.
Notation "x \_/ y" := (glub x y) (at level 50) : glbl_scope.
Notation "x «= y" := (lleq x y = true) (at level 70) : lbl_scope.
Notation "x \_/ y" := (llub x y) (at level 50) : lbl_scope.

Open Scope lbl_scope.

Proposition glub_homo : ∀ l l1 l2, grp l (llub l1 l2) = glub (grp l l1) (grp l l2).
Proof.
intros; case_eq (lleq l1 l); intros.
case_eq (lleq l2 l); intros; unfold grp; rewrite H; rewrite H0; simpl.
assert (l1 \_/ l2 «= l).
rewrite (lub_leq lbl_lattice lbl_lattice'); split; auto.
rewrite H1; auto.
assert (~ l1 \_/ l2 «= l).
rewrite (lub_leq lbl_lattice lbl_lattice'); intro.
destruct H1.
unfold lleq in H0; rewrite H2 in H0; inv H0.
destruct (lleq (l1 \_/ l2)%lbl l); auto.
contradiction H1; auto.
unfold grp; rewrite H; simpl.
assert (~ l1 \_/ l2 «= l).
rewrite (lub_leq lbl_lattice lbl_lattice'); intro.
destruct H0.
unfold lleq in H; rewrite H0 in H; inv H.
destruct (lleq (l1 \_/ l2) l); auto.
contradiction H0; auto.
Qed.

Close Scope lbl_scope.

Proposition glub_leq : ∀ l l1 l2, glub (grp l l1) (grp l l2) = Lo ↔ grp l l1 = Lo ∧ grp l l2
= Lo.
Proof.
intros; unfold grp; destruct (lleq l1 l); destruct (lleq l2 l); simpl; intuit.
Qed.

Proposition glub_lo : ∀ l1 l2, glub l1 l2 = Lo ↔ l1 = Lo ∧ l2 = Lo.
Proof.
destruct l1; destruct l2; intuit.
Qed.

Ltac glub_simpl H := apply glub_lo in H; destruct H.
Ltac glub_simpl_grp H := try (rewrite glub_homo in H); apply glub_leq in H; destruct
```

H.

```
Inductive binop := Plus | Minus | Mult | Div | Mod.
Inductive bbinop := And | Or | Impl.

Inductive exp :=
| Var : var → exp
| LVar : lvar1 → exp
| Num : Z → exp
| BinOp : binop → exp → exp → exp.

Fixpoint expvars (e : exp) (x : var) : bool :=
  match e with
  | Var y ⇒ if eq_nat_dec y x then true else false
  | BinOp _ e1 e2 ⇒ if expvars e1 x then true else expvars e2 x
  | _ ⇒ false
  end.

Fixpoint no_lvars_exp (e : exp) :=
  match e with
  | LVar _ ⇒ False
  | BinOp _ e1 e2 ⇒ no_lvars_exp e1 ∧ no_lvars_exp e2
  | _ ⇒ True
  end.

Proposition exp_eq_dec : ∀ e1 e2 : exp, {e1 = e2} + {e1 ≠ e2}.
Proof.
induction e1; destruct e2; try solve [right; discriminate].
destruct (eq_nat_dec v v0); subst.
left; auto.
right; intro H; inv H; contradiction n; auto.
destruct (eq_nat_dec l l0); subst.
left; auto.
right; intro H; inv H; contradiction n; auto.
destruct (Z_eq_dec z z0); subst.
left; auto.
right; intro H; inv H; contradiction n; auto.
assert ({b = b0} + {BinOp b e1_1 e1_2 ≠ BinOp b0 e2_1 e2_2}).
destruct b; destruct b0; auto; try solve [right; intro H; inv H].
destruct H; auto; subst.
destruct (IHe1_1 e2_1); subst.
destruct (IHe1_2 e2_2); subst; auto.
right; intro H; inv H; contradiction n; auto.
right; intro H; inv H; contradiction n; auto.
Qed.

Inductive bexp :=
```

```
| FF : bexp
| TT : bexp
| Eq : exp → exp → bexp
| Not : bexp → bexp
| BBinOp : bbinop → bexp → bexp → bexp.

Fixpoint bexpvars (b : bexp) (x : var) : bool :=
  match b with
  | Eq e1 e2 ⇒ if expvars e1 x then true else expvars e2 x
  | Not b ⇒ bexpvars b x
  | BBinOp _ b1 b2 ⇒ if bexpvars b1 x then true else bexpvars b2 x
  | _ ⇒ false
  end.

Fixpoint no_lvars_bexp (b : bexp) :=
  match b with
  | Eq e1 e2 ⇒ no_lvars_exp e1 ∧ no_lvars_exp e2
  | Not b ⇒ no_lvars_bexp b
  | BBinOp _ b1 b2 ⇒ no_lvars_bexp b1 ∧ no_lvars_bexp b2
  | _ ⇒ True
  end.

Inductive cmd :=
| Skip : cmd
| Output : exp → cmd
| Assign : var → exp → cmd
| Read : var → exp → cmd
| Write : exp → exp → cmd
| Seq : cmd → cmd → cmd
| If : bexp → cmd → cmd → cmd
| While : bexp → cmd → cmd.

Fixpoint mods (C : cmd) : list var :=
  match C with
  | Assign x _ ⇒ [x]
  | Read x _ ⇒ [x]
  | Seq C1 C2 ⇒ mods C1 ++ mods C2
  | If _ C1 C2 ⇒ mods C1 ++ mods C2
  | While _ C ⇒ mods C
  | _ ⇒ []
  end.

Fixpoint modifies (K : list cmd) : list var :=
  match K with
  | [] ⇒ []
  | C::K ⇒ mods C ++ modifies K
```

```
  end.

Fixpoint no_lvars_cmd (C : cmd) :=
  match C with
  | Skip ⇒ True
  | Output e ⇒ no_lvars_exp e
  | Assign _ e ⇒ no_lvars_exp e
  | Read _ e ⇒ no_lvars_exp e
  | Write e1 e2 ⇒ no_lvars_exp e1 ∧ no_lvars_exp e2
  | Seq C1 C2 ⇒ no_lvars_cmd C1 ∧ no_lvars_cmd C2
  | If b C1 C2 ⇒ no_lvars_bexp b ∧ no_lvars_cmd C1 ∧ no_lvars_cmd C2
  | While b C ⇒ no_lvars_bexp b ∧ no_lvars_cmd C
  end.

Fixpoint no_lvars (K : list cmd) :=
  match K with
  | [] ⇒ True
  | C::K ⇒ no_lvars_cmd C ∧ no_lvars K
  end.

Definition val := prod Z glbl.
Definition lmap := prod (lvar1 → Z) (lvar2 → glbl).
Definition store := var → option val.
Definition heap := addr → option val.
Inductive state := St : lmap → store → heap → state.
Definition getLmap (st : state) := let (i,_,_) := st in i.
Coercion getLmap : state >-> lmap.
Definition getStore (st : state) := let (_,s,_) := st in s.
Coercion getStore : state >-> store.
Definition getHeap (st : state) := let (_,_,h) := st in h.
Coercion getHeap : state >-> heap.

Proposition val_eq_dec : ∀ v1 v2 : val, {v1 = v2} + {v1 ≠ v2}.
Proof.
destruct v1; destruct v2.
destruct g; destruct g0; try solve [right; intro H; inv H].
destruct (Z_eq_dec z z0); subst.
left; auto.
right; intro H; inv H; contradiction n; auto.
destruct (Z_eq_dec z z0); subst.
left; auto.
right; intro H; inv H; contradiction n; auto.
Qed.

Proposition opt_eq_dec {A} : (∀ a1 a2 : A, {a1 = a2} + {a1 ≠ a2}) → ∀ o1 o2 : option
A, {o1 = o2} + {o1 ≠ o2}.
```

```
Proof.
intros.
destruct o1; destruct o2.
destruct (X a a0); subst; auto.
right; intro H; inv H; contradiction n; auto.
right; discriminate.
right; discriminate.
left; auto.
Qed.
```

```
Definition upd {A} (x : nat → option A) y z : nat → option A := fun w ⇒ if eq_nat_dec
w y then Some z else x w.
```

```
Record SepAlg : Type := mkSepAlg {
  sepstate : Set;
  unit : sepstate → Prop;
  dot : sepstate → sepstate → sepstate → Prop;
  dot_func : ∀ x y z1 z2, dot x y z1 → dot x y z2 → z1 = z2;
  dot_comm : ∀ x y z, dot x y z → dot y x z;
  dot_assoc : ∀ x y z a b, dot x y a → dot a z b → ∃ c, dot y z c ∧ dot x c b;
  dot_unit : ∀ x, ∃ u, unit u ∧ dot u x x;
  dot_unit_min : ∀ u x y, unit u → dot u x y → x = y}.
```

```
Definition mycombine {A} (s1 s2 : nat → option A) (n : nat) : option A :=
  match s1 n, s2 n with
  | Some a, _ ⇒ Some a
  | None, Some a ⇒ Some a
  | None, None ⇒ None
  end.
```

```
Definition mydot {A} (s1 s2 s : nat → option A) : Prop := ∀ n,
  match s n with
  | None ⇒ s1 n = None ∧ s2 n = None
  | Some a ⇒ (s1 n = Some a ∧ s2 n = None) ∨ (s1 n = None ∧ s2 n = Some a)
  end.
```

```
Definition mysep : SepAlg.
apply (mkSepAlg state (fun st ⇒ match st with St _ _ h ⇒ h = (fun _ ⇒ None) end)
          (fun st1 st2 st3 ⇒
              match st1, st2, st3 with St i1 s1 h1, St i2 s2 h2, St i3 s3 h3 ⇒
                 i1 = i2 ∧ i1 = i3 ∧ s1 = s2 ∧ s1 = s3 ∧ mydot h1 h2 h3
              end)); intros.
destruct x as [i1 s1 h1]; destruct y as [i2 s2 h2]; destruct z1 as [i3 s3 h3]; destruct
z2 as [i4 s4 h4].
decomp H; decomp H0; repeat subst.
apply f_equal; apply functional_extensionality; intro n.
```

```
specialize (H6 n); specialize (H10 n).
destruct (h3 n); destruct (h4 n); auto.
decomp H6; decomp H10.
rewrite H1 in H3; auto.
rewrite H1 in H3; inv H3.
rewrite H1 in H3; inv H3.
rewrite H2 in H4; auto.
decomp H6; decomp H10.
rewrite H1 in H0; inv H0.
rewrite H2 in H3; inv H3.
decomp H6; decomp H10.
rewrite H0 in H3; inv H3.
rewrite H1 in H4; inv H4.
destruct x as [i1 s1 h1]; destruct y as [i2 s2 h2]; destruct z as [i3 s3 h3].
decomp H; repeat split; repeat subst; auto.
intro n; specialize (H5 n).
destruct (h3 n); intuit.
destruct x as [i1 s1 h1]; destruct y as [i2 s2 h2]; destruct z as [i3 s3 h3]; destruct
a as [i4 s4 h4]; destruct b as [i5 s5 h5].
decomp H; decomp H0; repeat subst.
∃ (St i5 s5 (mycombine h2 h3)).
repeat split; auto.
intro n; unfold mycombine; specialize (H6 n); specialize (H10 n).
destruct (h2 n); destruct (h3 n); auto.
destruct (h4 n); destruct (h5 n); intuit.
decomp H6.
inv H1.
decomp H10.
inv H3.
inv H2.
destruct H6.
inv H0.
intro n; unfold mycombine; specialize (H6 n); specialize (H10 n).
destruct (h4 n); destruct (h5 n).
decomp H6.
decomp H10.
inv H2; rewrite H0; left; split; auto.
rewrite H1; rewrite H3; auto.
inv H2.
right; split; auto.
rewrite H1.
decomp H10; auto.
```

```
inv H2.
destruct H10.
inv H.
decomp H10.
inv H0.
destruct H6; right; split; auto.
rewrite H2; rewrite H1; auto.
destruct H6; destruct H10.
rewrite H0; rewrite H2; auto.
destruct x as [i s h].
∃ (St i s (fun _ ⇒ None)); repeat split.
intro n.
destruct (h n); auto.
destruct u as [i s h]; subst.
destruct x as [i1 s1 h1]; destruct y as [i2 s2 h2].
decomp H0; repeat subst.
apply f_equal; apply functional_extensionality; intro n; specialize (H5 n).
destruct (h1 n); destruct (h2 n); intuit.
decomp H5; auto.
inv H1.
Defined.

Proposition mydot_upd {A} : ∀ (x y z : nat → option A) n v,
  mydot x y z → y n = None → mydot (upd x n v) y (upd z n v).
Proof.
unfold mydot; unfold upd; intros.
destruct (eq_nat_dec n0 n); subst; intuit.
apply (H n0).
Qed.

Definition option_map2 {A B C} (op : A → B → C) x y : option C :=
  match x, y with
  | Some x, Some y ⇒ Some (op x y)
  | _, _ ⇒ None
  end.

Open Scope Z_scope.
Open Scope glbl_scope.

Definition opden (bop : binop) : Z → Z → Z :=
  match bop with
  | Plus ⇒ Zplus
  | Minus ⇒ Zminus
  | Mult ⇒ Zmult
  | Div ⇒ Zdiv
```

```
  | Mod ⇒ Zmod
  end.

Fixpoint eden (e : exp) (i : lmap) (s : store) : option val :=
  match e with
  | Var x ⇒ s x
  | LVar X ⇒ Some (fst i X, Lo)
  | Num c ⇒ Some (c,Lo)
  | BinOp bop e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ (opden bop (fst v1) (fst v2), snd v1
\_/ snd v2)) (eden e1 i s) (eden e2 i s)
  end.

Fixpoint edenZ (e : exp) (i : lmap) (s : store) : option Z :=
  match e with
  | Var x ⇒ option_map (fun v ⇒ fst v) (s x)
  | LVar X ⇒ Some (fst i X)
  | Num c ⇒ Some c
  | BinOp bop e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ opden bop v1 v2) (edenZ e1 i s) (edenZ
e2 i s)
  end.

Proposition edenZ_some : ∀ e i s v, edenZ e i s = Some v ↔ ∃ l, eden e i s = Some
(v,l).
Proof.
induction e; simpl; intros; split; intros.
destruct (s v) as [[v1 l1]||]; inv H.
∃ l1; auto.
destruct H as [l]; rewrite H; auto.
inv H; ∃ Lo; auto.
destruct H as [l0]; inv H; auto.
inv H; ∃ Lo; auto.
destruct H as [l]; inv H; auto.
case_eq (edenZ e1 i s); intros.
case_eq (edenZ e2 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite IHe1 in H0; rewrite IHe2 in H1.
destruct H0 as [l1]; destruct H1 as [l2].
rewrite H; rewrite H0; ∃ (l1 \_/ l2); auto.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite H0 in H; inv H.
destruct H as [l].
case_eq (eden e1 i s); intros.
case_eq (eden e2 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
destruct v0 as [v0 l0]; destruct v1 as [v1 l1].
```

```
assert (∃ l, eden e1 i s = Some (v0,l)).
∃ l0; auto.
assert (∃ l, eden e2 i s = Some (v1,l)).
∃ l1; auto.
rewrite ← IHe1 in H; rewrite ← IHe2 in H2.
rewrite H; rewrite H2; auto.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite H0 in H; inv H.
Qed.

Proposition edenZ_none : ∀ e i s, edenZ e i s = None ↔ eden e i s = None.
Proof.
induction e; simpl; intros; split; intros.
destruct (s v); inv H; auto.
rewrite H; auto.
inv H.
inv H.
inv H.
inv H.
case_eq (edenZ e1 i s); intros.
case_eq (edenZ e2 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite IHe2 in H1; rewrite H1.
destruct (eden e1 i s); auto.
rewrite IHe1 in H0; rewrite H0; auto.
case_eq (eden e1 i s); intros.
case_eq (eden e2 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite ← IHe2 in H1; rewrite H1.
destruct (edenZ e1 i s); auto.
rewrite ← IHe1 in H0; rewrite H0; auto.
Qed.

Definition bopden (bop : bbinop) : bool → bool → bool :=
  match bop with
  | And ⇒ andb
  | Or ⇒ orb
  | Impl ⇒ fun v1 v2 ⇒ if v1 then v2 else true
  end.

Fixpoint bden (b : bexp) (i : lmap) (s : store) : option (bool × glbl) :=
  match b with
  | FF ⇒ Some (false,Lo)
  | TT ⇒ Some (true,Lo)
  | Eq e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ (if Z_eq_dec (fst v1) (fst v2) then true else
```

```
false, snd v1 \_/ snd v2)) (eden e1 i s) (eden e2 i s)
  | Not b ⇒ option_map (fun v ⇒ (negb (fst v), snd v)) (bden b i s)
  | BBinOp bop b1 b2 ⇒ option_map2 (fun v1 v2 ⇒ (bopden bop (fst v1) (fst v2), snd v1
\_/ snd v2)) (bden b1 i s) (bden b2 i s)
  end.
Fixpoint bdenZ (b : bexp) (i : lmap) (s : store) : option bool :=
  match b with
  | FF ⇒ Some false
  | TT ⇒ Some true
  | Eq e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ if Z_eq_dec v1 v2 then true else false) (edenZ
e1 i s) (edenZ e2 i s)
  | Not b ⇒ option_map (fun v ⇒ negb v) (bdenZ b i s)
  | BBinOp bop b1 b2 ⇒ option_map2 (fun v1 v2 ⇒ bopden bop v1 v2) (bdenZ b1 i s)
(bdenZ b2 i s)
  end.
Proposition bdenZ_some : ∀ b i s v, bdenZ b i s = Some v ↔ ∃ l, bden b i s = Some (v,l).
Proof.
induction b; simpl; intros; split; intros.
inv H; ∃ Lo; auto.
destruct H as [l]; inv H; auto.
inv H; ∃ Lo; auto.
destruct H as [l]; inv H; auto.
case_eq (edenZ e i s); intros.
case_eq (edenZ e0 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite edenZ_some in H0; rewrite edenZ_some in H1.
destruct H0 as [l]; destruct H1 as [l0]; rewrite H; rewrite H0.
∃ (l \_/ l0); auto.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite H0 in H; inv H.
destruct H as [l].
case_eq (eden e i s); intros.
case_eq (eden e0 i s); intros.
destruct v0 as [v0 l0]; destruct v1 as [v1 l1].
rewrite H0 in H; rewrite H1 in H; inv H.
assert (∃ l, eden e i s = Some (v0,l)).
∃ l0; auto.
assert (∃ l, eden e0 i s = Some (v1,l)).
∃ l1; auto.
rewrite ← edenZ_some in H; rewrite ← edenZ_some in H2.
rewrite H; rewrite H2; auto.
rewrite H0 in H; rewrite H1 in H; inv H.
```

```
rewrite H0 in H; inv H.
case_eq (bdenZ b i s); intros.
rewrite H0 in H; inv H.
rewrite IHb in H0; destruct H0 as [l]; ∃ l.
rewrite H; auto.
rewrite H0 in H; inv H.
destruct H as [l].
case_eq (bden b i s); intros.
destruct p as [v1 l1].
assert (∃ l, bden b i s = Some (v1,l)).
∃ l1; auto.
rewrite H0 in H; inv H.
rewrite ← IHb in H1; rewrite H1; auto.
rewrite H0 in H; inv H.
case_eq (bdenZ b2 i s); intros.
case_eq (bdenZ b3 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite IHb1 in H0; rewrite IHb2 in H1.
destruct H0 as [l1]; destruct H1 as [l2].
rewrite H; rewrite H0; ∃ (l1 \_/ l2); auto.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite H0 in H; inv H.
destruct H as [l].
case_eq (bden b2 i s); intros.
case_eq (bden b3 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
destruct p as [v0 l0]; destruct p0 as [v1 l1].
assert (∃ l, bden b2 i s = Some (v0,l)).
∃ l0; auto.
assert (∃ l, bden b3 i s = Some (v1,l)).
∃ l1; auto.
rewrite ← IHb1 in H; rewrite ← IHb2 in H2.
rewrite H; rewrite H2; auto.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite H0 in H; inv H.
Qed.

Proposition bdenZ_none : ∀ b i s, bdenZ b i s = None ↔ bden b i s = None.
Proof.
induction b; simpl; intros; split; intros.
inv H.
inv H.
inv H.
```

```
inv H.
case_eq (edenZ e i s); intros.
case_eq (edenZ e0 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite edenZ_none in H1; rewrite H1.
destruct (eden e i s); auto.
rewrite edenZ_none in H0; rewrite H0; auto.
case_eq (eden e i s); intros.
case_eq (eden e0 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite ← edenZ_none in H1; rewrite H1.
destruct (edenZ e i s); auto.
rewrite ← edenZ_none in H0; rewrite H0; auto.
case_eq (bdenZ b i s); intros.
rewrite H0 in H; inv H.
rewrite IHb in H0; rewrite H0; auto.
case_eq (bden b i s); intros.
rewrite H0 in H; inv H.
rewrite ← IHb in H0; rewrite H0; auto.
case_eq (bdenZ b2 i s); intros.
case_eq (bdenZ b3 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite IHb2 in H1; rewrite H1.
destruct (bden b2 i s); auto.
rewrite IHb1 in H0; rewrite H0; auto.
case_eq (bden b2 i s); intros.
case_eq (bden b3 i s); intros.
rewrite H0 in H; rewrite H1 in H; inv H.
rewrite ← IHb2 in H1; rewrite H1.
destruct (bdenZ b2 i s); auto.
rewrite ← IHb1 in H0; rewrite H0; auto.
Qed.
```

```
Proposition eden_local : ∀ e i1 s1 h1 i2 s2 h2 i3 s3 h3 v,
  dot mysep (St i1 s1 h1) (St i2 s2 h2) (St i3 s3 h3) → eden e i1 s1 = Some v → eden
e i3 s3 = Some v.
Proof.
intros.
simpl in H; decomp H; repeat subst; auto.
Qed.
```

```
Proposition bden_local : ∀ b i1 s1 h1 i2 s2 h2 i3 s3 h3 v,
  dot mysep (St i1 s1 h1) (St i2 s2 h2) (St i3 s3 h3) → bden b i1 s1 = Some v → bden
b i3 s3 = Some v.
```

```
Proof.
intros.
simpl in H; decomp H; repeat subst; auto.
Qed.

Proposition eden_no_lvars : ∀ e i i' s, no_lvars_exp e → eden e i s = eden e i' s.
Proof.
induction e; simpl; intros; intuit.
rewrite (IHe1 _ i'); intuit; rewrite (IHe2 _ i'); intuit.
Qed.

Proposition bden_no_lvars : ∀ b i i' s, no_lvars_bexp b → bden b i s = bden b i' s.
Proof.
induction b; simpl; intros; intuit.
rewrite (eden_no_lvars e _ i'); intuit; rewrite (eden_no_lvars e0 _ i'); intuit.
rewrite (IHb _ i'); intuit.
rewrite (IHb1 _ i'); intuit; rewrite (IHb2 _ i'); intuit.
Qed.

Definition context := glbl.
Inductive config := Cf : state → cmd → list cmd → config.
Definition getStoreFromConfig (cf : config) := match cf with Cf (St _ s _) _ _ ⇒ s end.
Coercion getStoreFromConfig : config >-> store.

Definition taint_vars (K : list cmd) (s : store) : store :=
  fun x ⇒ if In_dec eq_nat_dec x (modifies K) then
             match s x with Some (v,_) ⇒ Some (v,Hi) | None ⇒ Some (0,Hi) end
           else s x.

Definition taint_vars_cf (cf : config) : config :=
  match cf with Cf (St i s h) C K ⇒ Cf (St i (taint_vars (C::K) s) h) C K end.

Inductive hstep : config → config → Prop :=
| HStep_skip : ∀ st C K, hstep (Cf st Skip (C::K)) (Cf st C K)
| HStep_assign : ∀ i s h K x e v l,
    eden e i s = Some (v,l) →
    hstep (Cf (St i s h) (Assign x e) K) (Cf (St i (upd s x (v, Hi)) h) Skip K)
| HStep_read : ∀ i s h K x e v1 l1 v2 l2 (pf : v1 ≥ 0),
    eden e i s = Some (v1,l1) → h (nat_of_Z v1 pf) = Some (v2,l2) →
    hstep (Cf (St i s h) (Read x e) K) (Cf (St i (upd s x (v2, Hi)) h) Skip K)
| HStep_write : ∀ i s h K e1 e2 v1 l1 v2 l2 (pf : v1 ≥ 0),
    eden e1 i s = Some (v1,l1) → eden e2 i s = Some (v2,l2) → h (nat_of_Z v1 pf) ≠
None →
    hstep (Cf (St i s h) (Write e1 e2) K) (Cf (St i s (upd h (nat_of_Z v1 pf) (v2, Hi)))
Skip K)
| HStep_seq : ∀ st C1 C2 K, hstep (Cf st (Seq C1 C2) K) (Cf st C1 (C2::K))
| HStep_if_true : ∀ i s h C1 C2 K b l,
```

bden b i s = Some (true,l) → hstep (Cf (St i s h) (If b C1 C2) K) (Cf (St i s h) C1 K)
| *HStep_if_false* : ∀ *i s h C1 C2 K b l*,
bden b i s = Some (false,l) → hstep (Cf (St i s h) (If b C1 C2) K) (Cf (St i s h) C2 K)
| *HStep_while_true* : ∀ *i s h C K b l*,
bden b i s = Some (true,l) → hstep (Cf (St i s h) (While b C) K) (Cf (St i s h) C (While b C :: K))
| *HStep_while_false* : ∀ *i s h C K b l*,
bden b i s = Some (false,l) → hstep (Cf (St i s h) (While b C) K) (Cf (St i s h) Skip K).

`Inductive` *hstepn* : *nat → config → config →* `Prop` :=
| *HStep_zero* : ∀ *cf*, *hstepn* 0 *cf cf*
| *HStep_succ* : ∀ *n cf cf' cf''*, *hstep cf cf' → hstepn n cf' cf'' → hstepn (S n) cf cf''*.

`Definition` *halt_config cf* := `match` *cf* `with` *Cf _ Skip* [] ⇒ *true* | _ ⇒ *false* `end`.
`Inductive` *can_hstep* : *config →* `Prop` := *Can_hstep* : ∀ *cf cf'*, *hstep cf cf' → can_hstep cf*.
`Definition` *hsafe cf* := ∀ *n cf'*, *hstepn n cf cf' → halt_config cf' = false → can_hstep cf'*.

`Inductive` *lstep* : *config → config → list Z →* `Prop` :=
| *LStep_skip* : ∀ *st C K*, *lstep (Cf st Skip (C::K)) (Cf st C K)* []
| *LStep_output* : ∀ *i s h K e v*,
eden e i s = Some (v,Lo) →
lstep (Cf (St i s h) (Output e) K) (Cf (St i s h) Skip K) [*v*]
| *LStep_assign* : ∀ *i s h K x e v l*,
eden e i s = Some (v,l) →
lstep (Cf (St i s h) (Assign x e) K) (Cf (St i (upd s x (v, l)) h) Skip K) []
| *LStep_read* : ∀ *i s h K x e v1 l1 v2 l2 (pf : v1* ≥ 0),
eden e i s = Some (v1,l1) → h (nat_of_Z v1 pf) = Some (v2,l2) →
lstep (Cf (St i s h) (Read x e) K) (Cf (St i (upd s x (v2, l1 _/ l2)) h) Skip K) []
| *LStep_write* : ∀ *i s h K e1 e2 v1 l1 v2 l2 (pf : v1* ≥ 0),
eden e1 i s = Some (v1,l1) → eden e2 i s = Some (v2,l2) → h (nat_of_Z v1 pf) ≠ None →
lstep (Cf (St i s h) (Write e1 e2) K) (Cf (St i s (upd h (nat_of_Z v1 pf) (v2, l1 _/ l2))) Skip K) []
| *LStep_seq* : ∀ *st C1 C2 K*, *lstep (Cf st (Seq C1 C2) K) (Cf st C1 (C2::K))* []
| *LStep_if_true* : ∀ *i s h C1 C2 K b*,
bden b i s = Some (true,Lo) → lstep (Cf (St i s h) (If b C1 C2) K) (Cf (St i s h) C1 K) []
| *LStep_if_false* : ∀ *i s h C1 C2 K b*,
bden b i s = Some (false,Lo) → lstep (Cf (St i s h) (If b C1 C2) K) (Cf (St i s h) C2 K) []
| *LStep_while_true* : ∀ *i s h C K b*,

```
    bden b i s = Some (true,Lo) → lstep (Cf (St i s h) (While b C) K) (Cf (St i s h) C (While b C :: K)) []
| LStep_while_false : ∀ i s h C K b,
    bden b i s = Some (false,Lo) → lstep (Cf (St i s h) (While b C) K) (Cf (St i s h) Skip K) []
| LStep_if_hi : ∀ i s h st' C1 C2 K b v n,
    bden b i s = Some (v,Hi) → hsafe (taint_vars_cf (Cf (St i s h) (If b C1 C2) [])) →
    hstepn n (taint_vars_cf (Cf (St i s h) (If b C1 C2) [])) (Cf st' Skip []) →
    lstep (Cf (St i s h) (If b C1 C2) K) (Cf st' Skip K) []
| LStep_if_hi_dvg : ∀ i s h C1 C2 K b v,
    bden b i s = Some (v,Hi) → hsafe (taint_vars_cf (Cf (St i s h) (If b C1 C2) [])) →
    (∀ n st', ¬ hstepn n (taint_vars_cf (Cf (St i s h) (If b C1 C2) [])) (Cf st' Skip [])) →
    lstep (Cf (St i s h) (If b C1 C2) K) (Cf (St i s h) (If b C1 C2) K) []
| LStep_while_hi : ∀ i s h st' C K b v n,
    bden b i s = Some (v,Hi) → hsafe (taint_vars_cf (Cf (St i s h) (While b C) [])) →
    hstepn n (taint_vars_cf (Cf (St i s h) (While b C) [])) (Cf st' Skip []) →
    lstep (Cf (St i s h) (While b C) K) (Cf st' Skip K) []
| LStep_while_hi_dvg : ∀ i s h C K b v,
    bden b i s = Some (v,Hi) → hsafe (taint_vars_cf (Cf (St i s h) (While b C) [])) →
    (∀ n st', ¬ hstepn n (taint_vars_cf (Cf (St i s h) (While b C) [])) (Cf st' Skip [])) →
    lstep (Cf (St i s h) (While b C) K) (Cf (St i s h) (While b C) K) [].

Inductive lstepn : nat → config → config → list Z → Prop :=
| LStep_zero : ∀ cf, lstepn 0 cf cf []
| LStep_succ : ∀ n cf cf' cf'' o o', lstep cf cf' o → lstepn n cf' cf'' o' → lstepn (S n) cf cf'' (o++o').

Inductive can_lstep : config → Prop := Can_lstep : ∀ cf cf' o, lstep cf cf' o → can_lstep cf.
Definition lsafe cf := ∀ n cf' o, lstepn n cf cf' o → halt_config cf' = false → can_lstep cf'.

Definition side_condition C (st1 st2 : state) :=
  match C, st1, st2 with
  | Read _ e, St i1 s1 h1, St i2 s2 h2 ⇒
      match (eden e i1 s1), (eden e i2 s2) with
      | Some (v1,_), Some (v2,_) ⇒
          match Zneg_dec v1, Zneg_dec v2 with
          | left pf1, left pf2 ⇒
              match h1 (nat_of_Z v1 pf1), h2 (nat_of_Z v2 pf2) with
              | Some (_,l1), Some (_,l2) ⇒ l1 = l2
              | _, _ ⇒ False
              end
          | _, _ ⇒ False
          end
```

```
      | _, _ ⇒ False
      end
  | _, _, _ ⇒ True
  end.

Close Scope Z_scope.

Proposition dvg_ex_mid : ∀ cf,
  (∀ n st, ¬ hstepn n cf (Cf st Skip [])) ∨ ∃ n, ∃ st, hstepn n cf (Cf st Skip []).
Proof.
intros.
dup (classic (∃ n, ∃ st, hstepn n cf (Cf st Skip []))).
destruct H; [right | left]; auto.
intros; intro; contradiction H.
∃ n; ∃ st; auto.
Qed.

Lemma hstep_trans : ∀ n1 n2 cf1 cf2 cf3, hstepn n1 cf1 cf2 → hstepn n2 cf2 cf3 → hstepn
(n1+n2) cf1 cf3.
Proof.
induction n1 using (well_founded_induction lt_wf); intros.
inv H0; simpl; auto.
apply HStep_succ with (cf' := cf'); auto.
apply H with (cf2 := cf2); auto.
Qed.

Lemma lstep_trans : ∀ n1 n2 cf1 cf2 cf3 o1 o2, lstepn n1 cf1 cf2 o1 → lstepn n2 cf2 cf3
o2 → lstepn (n1+n2) cf1 cf3 (o1++o2).
Proof.
induction n1 using (well_founded_induction lt_wf); intros.
inv H0; simpl; auto.
rewrite app_assoc; apply LStep_succ with (cf' := cf'); auto.
apply H with (cf2 := cf2); auto.
Qed.

Lemma hstep_extend : ∀ st C K st' C' K' K0,
  hstep (Cf st C K) (Cf st' C' K') → hstep (Cf st C (K++K0)) (Cf st' C' (K'++K0)).
Proof.
intros.
inv H.
apply HStep_skip.
apply HStep_assign with (l := l); auto.
apply HStep_read with (v1 := v1) (pf := pf) (l1 := l1) (l2 := l2); auto.
apply HStep_write with (l1 := l1) (l2 := l2); auto.
apply HStep_seq.
apply HStep_if_true with (l := l); auto.
```

```
apply HStep_if_false with (l := l); auto.
apply HStep_while_true with (l := l); auto.
apply HStep_while_false with (l := l); auto.
Qed.
```

```
Lemma hstepn_extend : ∀ n st C K st' C' K' K0,
  hstepn n (Cf st C K) (Cf st' C' K') → hstepn n (Cf st C (K++K0)) (Cf st' C'
(K'++K0)).
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
apply HStep_zero.
destruct cf' as [st'' C'' K''].
apply HStep_succ with (cf' := Cf st'' C'' (K''++K0)).
apply hstep_extend; auto.
apply H; auto.
Qed.
```

```
Lemma lstep_extend : ∀ st C K st' C' K' K0 o,
  lstep (Cf st C K) (Cf st' C' K') o → lstep (Cf st C (K++K0)) (Cf st' C' (K'++K0))
o.
Proof.
intros.
inv H.
apply LStep_skip.
apply LStep_output; auto.
apply LStep_assign with (l := l); auto.
apply LStep_read with (v1 := v1) (pf := pf) (l1 := l1) (l2 := l2); auto.
apply LStep_write with (l1 := l1) (l2 := l2); auto.
apply LStep_seq.
apply LStep_if_true; auto.
apply LStep_if_false; auto.
apply LStep_while_true; auto.
apply LStep_while_false; auto.
apply LStep_if_hi with (b := b) (v := v) (n := n); auto.
apply LStep_if_hi_dvg with (b := b) (v := v); auto.
apply LStep_while_hi with (b := b) (v := v) (n := n); auto.
apply LStep_while_hi_dvg with (b := b) (v := v); auto.
Qed.
```

```
Lemma lstepn_extend : ∀ n st C K st' C' K' K0 o,
  lstepn n (Cf st C K) (Cf st' C' K') o → lstepn n (Cf st C (K++K0)) (Cf st' C'
(K'++K0)) o.
Proof.
induction n using (well_founded_induction lt_wf); intros.
```

```
inv H0.
apply LStep_zero.
destruct cf' as [st'' C'' K''].
apply LStep_succ with (cf' := Cf st'' C'' (K''++K0)).
apply lstep_extend; auto.
apply H; auto.
Qed.

Lemma hstep_trans_inv : ∀ n st st' C C' K0 K K',
  hstepn n (Cf st C (K0++K)) (Cf st' C' K') →
  (∃ K'', hstepn n (Cf st C K0) (Cf st' C' K'') ∧ K' = K''++K) ∨
  ∃ st'', ∃ n1, ∃ n2,
    hstepn n1 (Cf st C K0) (Cf st'' Skip []) ∧ hstepn n2 (Cf st'' Skip K) (Cf st' C' K')
∧
    n = n1 + n2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
left; ∃ K0.
split; auto; apply HStep_zero.
inv H1.

destruct K0.
simpl in H5; subst.
right; ∃ st; ∃ 0; ∃ (S n0); repeat (split; auto).
apply HStep_zero.
apply HStep_succ with (cf' := Cf st C0 K1); auto.
apply HStep_skip.
inv H5.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf st c K0); auto.
apply HStep_skip.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf st c K0); auto.
apply HStep_skip.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i (upd s x (v,Hi)) h) Skip K0); auto.
```

```
apply HStep_assign with (l := l); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i (upd s x (v,Hi)) h) Skip K0); auto.
apply HStep_assign with (l := l); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i (upd s x (v2,Hi)) h) Skip K0); auto.
apply HStep_read with (v1 := v1) (l1 := l1) (l2 := l2) (pf := pf); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i (upd s x (v2, Hi)) h) Skip K0); auto.
apply HStep_read with (v1 := v1) (l1 := l1) (l2 := l2) (pf := pf); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i s (upd h (nat_of_Z v1 pf) (v2,Hi))) Skip K0);
auto.
apply HStep_write with (l1 := l1) (l2 := l2); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i s (upd h (nat_of_Z v1 pf) (v2,Hi))) Skip K0);
auto.
apply HStep_write with (l1 := l1) (l2 := l2); auto.

change (hstepn n0 (Cf st C1 ((C2 :: K0) ++ K)) (Cf st' C' K')) in H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf st C1 (C2::K0)); auto.
apply HStep_seq.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf st C1 (C2::K0)); auto.
apply HStep_seq.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
```

```
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C1 K0); auto.
apply HStep_if_true with (l := l); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i s h) C1 K0); auto.
apply HStep_if_true with (l := l); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C2 K0); auto.
apply HStep_if_false with (l := l); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i s h) C2 K0); auto.
apply HStep_if_false with (l := l); auto.

change (hstepn n0 (Cf (St i s h) C0 ((While b C0 :: K0) ++ K)) (Cf st' C' K')) in H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C0 (While b C0 :: K0)); auto.
apply HStep_while_true with (l := l); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i s h) C0 (While b C0 :: K0)); auto.
apply HStep_while_true with (l := l); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply HStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply HStep_while_false with (l := l); auto.
destruct H0 as [st'' [n1 [n2 [H0 [H1]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; repeat (split; auto).
apply HStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply HStep_while_false with (l := l); auto.
Qed.

Lemma lstep_trans_inv : ∀ n st st' C C' K0 K K' o,
  lstepn n (Cf st C (K0++K)) (Cf st' C' K') o →
```

```
  (∃ K'', lstepn n (Cf st C K0) (Cf st' C' K'') o ∧ K' = K''++K) ∨
  ∃ st'', ∃ n1, ∃ n2, ∃ o1, ∃ o2,
    lstepn n1 (Cf st C K0) (Cf st'' Skip []) o1 ∧ lstepn n2 (Cf st'' Skip K) (Cf st' C' K') o2 ∧
    n = n1 + n2 ∧ o = o1 ++ o2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
left; ∃ K0.
split; auto; apply LStep_zero.
inv H1.

destruct K0.
simpl in H5; subst.
right; ∃ st; ∃ 0; ∃ (S n0); ∃ []; ∃ ([]++o'); repeat (split; auto).
apply LStep_zero.
apply LStep_succ with (cf' := Cf st C0 K1); auto.
apply LStep_skip.
inv H5.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf st c K0); auto.
apply LStep_skip.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf st c K0); auto.
apply LStep_skip.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply LStep_output; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([v]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply LStep_output; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
```

```
apply LStep_succ with (cf' := Cf (St i (upd s x (v,l)) h) Skip K0); auto.
apply LStep_assign; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i (upd s x (v, l)) h) Skip K0); auto.
apply LStep_assign; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i (upd s x (v2, l1 \_/ l2)) h) Skip K0); auto.
apply LStep_read with (v1 := v1) (pf := pf); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i (upd s x (v2, l1 \_/ l2)) h) Skip K0); auto.
apply LStep_read with (v1 := v1) (pf := pf); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s (upd h (nat_of_Z v1 pf) (v2, l1 \_/ l2))) Skip
K0); auto.
apply LStep_write; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s (upd h (nat_of_Z v1 pf) (v2, l1 \_/ l2))) Skip
K0); auto.
apply LStep_write; auto.

change (lstepn n0 (Cf st C1 ((C2 :: K0) ++ K)) (Cf st' C' K') o') in H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf st C1 (C2::K0)); auto.
apply LStep_seq.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf st C1 (C2::K0)); auto.
apply LStep_seq.

apply H in H2; auto.
destruct H2.
```

```
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) C1 K0); auto.
apply LStep_if_true; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) C1 K0); auto.
apply LStep_if_true; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) C2 K0); auto.
apply LStep_if_false; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) C2 K0); auto.
apply LStep_if_false; auto.

change (lstepn n0 (Cf (St i s h) C0 ((While b C0 :: K0) ++ K)) (Cf st' C' K') o') in
H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) C0 (While b C0 :: K0)); auto.
apply LStep_while_true; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) C0 (While b C0 :: K0)); auto.
apply LStep_while_true; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply LStep_while_false; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) Skip K0); auto.
apply LStep_while_false; auto.

apply H in H2; auto.
```

```
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf st'0 Skip K0); auto.
apply LStep_if_hi with (b := b) (v := v) (n := n); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf st'0 Skip K0); auto.
apply LStep_if_hi with (b := b) (v := v) (n := n); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) (If b C1 C2) K0); auto.
apply LStep_if_hi_dvg with (b := b) (v := v); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) (If b C1 C2) K0); auto.
apply LStep_if_hi_dvg with (b := b) (v := v); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf st'0 Skip K0); auto.
apply LStep_while_hi with (b := b) (v := v) (n := n); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf st'0 Skip K0); auto.
apply LStep_while_hi with (b := b) (v := v) (n := n); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply LStep_succ with (cf' := Cf (St i s h) (While b C0) K0); auto.
apply LStep_while_hi_dvg with (b := b) (v := v); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := Cf (St i s h) (While b C0) K0); auto.
apply LStep_while_hi_dvg with (b := b) (v := v); auto.
Qed.

Lemma hstep_trans_inv' : ∀ a b cf cf',
```

```
  hstepn (a+b) cf cf' → ∃ cf'', hstepn a cf cf'' ∧ hstepn b cf'' cf'.
Proof.
induction a using (well_founded_induction lt_wf); intros.
inv H0.
assert (a = 0); try omega.
assert (b = 0); try omega; subst.
∃ cf'; split; apply HStep_zero.
destruct a; simpl in H1; subst.
∃ cf; split.
apply HStep_zero.
apply HStep_succ with (cf' := cf'0); auto.
inv H1.
apply H in H3; auto.
destruct H3 as [cf'' [H3]]; ∃ cf''; split; auto.
apply HStep_succ with (cf' := cf'0); auto.
Qed.

Lemma lstep_trans_inv' : ∀ a b cf cf' o,
  lstepn (a+b) cf cf' o → ∃ cf'', ∃ o1, ∃ o2,
    lstepn a cf cf'' o1 ∧ lstepn b cf'' cf' o2 ∧ o = o1 ++ o2.
Proof.
induction a using (well_founded_induction lt_wf); intros.
inv H0.
assert (a = 0); try omega.
assert (b = 0); try omega; subst.
∃ cf'; ∃ []; ∃ []; repeat (split; auto); apply LStep_zero.
destruct a; simpl in H1; subst.
∃ cf; ∃ []; ∃ (o0++o'); repeat (split; auto).
apply LStep_zero.
apply LStep_succ with (cf' := cf'0); auto.
inv H1.
apply H in H3; auto.
destruct H3 as [cf'' [o1 [o2 [H3 [H4]]]]]; ∃ cf''; ∃ (o0++o1); ∃ o2; repeat (split; auto).
apply LStep_succ with (cf' := cf'0); auto.
subst; rewrite app_assoc; auto.
Qed.

Lemma hstep_det : ∀ cf cf1 cf2, hstep cf cf1 → hstep cf cf2 → cf1 = cf2.
Proof.
intros.
inv H; inv H0; auto.
rewrite H8 in H1; inv H1; auto.
rewrite H9 in H1; inv H1.
rewrite (proof_irrelevance _ pf0 pf) in H10; rewrite H10 in H2; inv H2; auto.
```

```
rewrite H10 in H1; inv H1; rewrite H11 in H2; inv H2.
rewrite (proof_irrelevance _ pf0 pf); auto.
rewrite H9 in H1; inv H1.
rewrite H9 in H1; inv H1.
rewrite H8 in H1; inv H1.
rewrite H8 in H1; inv H1.
Qed.

Lemma hstepn_det : ∀ n cf cf1 cf2, hstepn n cf cf1 → hstepn n cf cf2 → cf1 = cf2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; inv H1; auto.
dup (hstep_det _ _ _ H2 H4); subst.
apply H with (y := n0) (cf := cf'0); auto.
Qed.

Lemma hstepn_det_term : ∀ n1 n2 cf st1 st2, hstepn n1 cf (Cf st1 Skip []) → hstepn n2 cf
(Cf st2 Skip []) → n1 = n2.
Proof.
intros.
assert (n1 = n2 ∨ n1 < n2 ∨ n2 < n1); try omega.
decomp H1; auto.
assert (n1 + (n2-n1) = n2); try omega.
rewrite ← H1 in H0; apply hstep_trans_inv' in H0.
destruct H0 as [cf' [H0]].
dup (hstepn_det _ _ _ _ H H0); subst cf'.
inv H2; try omega.
inv H5.
assert (n2 + (n1-n2) = n1); try omega.
rewrite ← H1 in H; apply hstep_trans_inv' in H.
destruct H as [cf' [H]].
dup (hstepn_det _ _ _ _ H H0); subst cf'.
inv H2; try omega.
inv H5.
Qed.

Lemma lstep_det : ∀ cf cf1 cf2 o1 o2, lstep cf cf1 o1 → lstep cf cf2 o2 → cf1 = cf2 ∧ o1
= o2.
Proof.
intros.
inv H.
inv H0; auto.
inv H0.
rewrite H8 in H1; inv H1; auto.
inv H0.
```

```
rewrite H9 in H1; inv H1; auto.
inv H0.
rewrite H10 in H1; inv H1.
rewrite (proof_irrelevance _ pf0 pf) in H11; rewrite H11 in H2; inv H2; auto.
inv H0.
rewrite H11 in H1; inv H1; rewrite H12 in H2; inv H2.
rewrite (proof_irrelevance _ pf0 pf); auto.
inv H0; auto.
inv H0; auto.
rewrite H10 in H1; inv H1.
rewrite H10 in H1; inv H1.
rewrite H10 in H1; inv H1.
inv H0; auto.
rewrite H10 in H1; inv H1.
rewrite H10 in H1; inv H1.
rewrite H10 in H1; inv H1.
inv H0; auto.
rewrite H9 in H1; inv H1.
rewrite H9 in H1; inv H1.
rewrite H9 in H1; inv H1.
inv H0; auto.
rewrite H9 in H1; inv H1.
rewrite H9 in H1; inv H1.
rewrite H9 in H1; inv H1.
inv H0; auto.
rewrite H12 in H1; inv H1.
rewrite H12 in H1; inv H1.
dup (hstepn_det_term _ _ _ _ _ H3 H14); subst.
dup (hstepn_det _ _ _ _ H3 H14).
inv H; auto.
contradiction (H14 n st').
inv H0; auto.
rewrite H12 in H1; inv H1.
rewrite H12 in H1; inv H1.
contradiction (H3 n st').
inv H0; auto.
rewrite H11 in H1; inv H1.
rewrite H11 in H1; inv H1.
dup (hstepn_det_term _ _ _ _ _ H3 H13); subst.
dup (hstepn_det _ _ _ _ H3 H13).
inv H; auto.
contradiction (H13 n st').
```

```
inv H0; auto.
rewrite H11 in H1; inv H1.
rewrite H11 in H1; inv H1.
contradiction (H3 n st').
Qed.

Lemma lstepn_det : ∀ n cf cf1 cf2 o1 o2, lstepn n cf cf1 o1 → lstepn n cf cf2 o2 → cf1 =
cf2 ∧ o1 = o2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; inv H1; auto.
destruct (lstep_det _ _ _ _ _ H2 H4); subst.
assert (n0 < S n0); try omega.
destruct (H _ H0 _ _ _ _ _ H3 H5); subst; auto.
Qed.

Lemma lstepn_det_term : ∀ n1 n2 cf st1 st2 o1 o2, lstepn n1 cf (Cf st1 Skip []) o1 →
lstepn n2 cf (Cf st2 Skip []) o2 → n1 = n2.
Proof.
intros.
assert (n1 = n2 ∨ n1 < n2 ∨ n2 < n1); try omega.
decomp H1; auto.
assert (n1 + (n2-n1) = n2); try omega.
rewrite ← H1 in H0; clear H1; apply lstep_trans_inv' in H0.
destruct H0 as [cf' [o3 [o4 [H0 [H2]]]]]; subst.
destruct (lstepn_det _ _ _ _ _ _ H H0); subst.
inv H2; try omega.
inv H4.
assert (n2 + (n1-n2) = n1); try omega.
rewrite ← H1 in H; clear H1; apply lstep_trans_inv' in H.
destruct H as [cf' [o3 [o4 [H [H1]]]]]; subst.
destruct (lstepn_det _ _ _ _ _ _ H H0); subst.
inv H1; try omega.
inv H4.
Qed.

Definition diverge cf := ∀ n, ∃ cf', ∃ o, lstepn n cf cf' o.

Corollary diverge_halt : ∀ n cf st o, diverge cf → lstepn n cf (Cf st Skip []) o → False.
Proof.
intros.
destruct (H (n+1)) as [cf' [o']].
apply lstep_trans_inv' in H1.
destruct H1 as [cf'' [o1 [o2]]]; decomp H1; subst.
destruct (lstepn_det _ _ _ _ _ _ H0 H2); subst; inv H4.
```

inv H3.
Qed.

Proposition *diverge_same_cf* : ∀ *cf o*, *lstep cf cf o* → *diverge cf*.
Proof.
intros.
assert (∀ *n*, ∃ *o*, *lstepn n cf cf o*).
induction *n*; intros.
∃ []; apply *LStep_zero*.
destruct *IHn* as [*o'*]; ∃ (*o*++*o'*); apply *LStep_succ* with (*cf'* := *cf*); auto.
intro *n*; destruct (*H0 n*) as [*o'*].
∃ *cf*; ∃ *o'*; auto.
Qed.

Lemma *diverge_seq1* : ∀ *C1 C2 K st*, *diverge* (*Cf st C1* []) → *diverge* (*Cf st* (*Seq C1 C2*) *K*).
Proof.
intros; intro *n*.
destruct *n*.
∃ (*Cf st* (*Seq C1 C2*) *K*); ∃ []; apply *LStep_zero*.
destruct (*H n*) as [[*st' C' K'*] [*o*]].
∃ (*Cf st' C'* (*K'*++[*C2*]++*K*)); ∃ ([]++*o*).
apply *LStep_succ* with (*cf'* := *Cf st C1* ([]++[*C2*]++*K*)).
apply *LStep_seq*.
apply *lstepn_extend*; auto.
Qed.

Lemma *diverge_seq2* : ∀ *C1 C2 K st st' n o*,
lstepn n (*Cf st C1* []) (*Cf st' Skip* []) *o* → *diverge* (*Cf st' C2 K*) → *diverge* (*Cf st* (*Seq C1 C2*) *K*).
Proof.
intros; intro *n'*.
assert (*n'* ≤ *S n* ∨ *n'* > *S n*); try omega.
destruct *H1*.
destruct *n'*.
∃ (*Cf st* (*Seq C1 C2*) *K*); ∃ []; apply *LStep_zero*.
assert (*n* = *n'*+(*n*-*n'*)); try omega.
rewrite *H2* in *H*; apply *lstep_trans_inv'* in *H*.
destruct *H* as [[*st'' C'' K''*] [*o1''* [*o2''*]]]; *decomp H*.
∃ (*Cf st'' C''* (*K''*++[*C2*]++*K*)); ∃ ([]++*o1''*).
apply *LStep_succ* with (*cf'* := *Cf st C1* ([]++[*C2*]++*K*)).
apply *LStep_seq*.
apply *lstepn_extend*; auto.
destruct (*H0* (*n'* - *S* (*S n*))) as [*cf* [*o'*]].
∃ *cf*; ∃ ([]++*o*++[]++*o'*).

```
assert (n' = S (n + S (n' - S (S n)))); try omega.
rewrite H3; apply LStep_succ with (cf' := Cf st C1 ([]++[C2]++K)).
apply LStep_seq.
apply lstep_trans with (cf2 := Cf st' Skip ([]++[C2]++K)).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st' C2 K); auto.
apply LStep_skip.
Qed.

Lemma hstep_ff : ∀ C K C' K' i s h1 h2 h3 i' s' h1',
  mydot h1 h2 h3 → hstep (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K') →
  ∃ h3', mydot h1' h2 h3' ∧ hstep (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K').
Proof.
intros.
inv H0.

∃ h3; split; auto; apply HStep_skip.

∃ h3; split; auto; apply HStep_assign with (l := l); auto.

∃ h3; split; auto; apply HStep_read with (l1 := l1) (l2 := l2) (pf := pf); auto.
specialize (H (nat_of_Z v1 pf)); destruct (h3 (nat_of_Z v1 pf)); decomp H.
rewrite H1 in H12; inv H12; auto.
rewrite H1 in H12; inv H12.
rewrite H0 in H12; inv H12.

∃ (upd h3 (nat_of_Z v1 pf) (v2,Hi)); split.
apply mydot_upd; auto.
specialize (H (nat_of_Z v1 pf)); destruct (h3 (nat_of_Z v1 pf)); decomp H; auto; try
contradiction.
apply HStep_write with (l1 := l1) (l2 := l2); auto.
contradict H13; specialize (H (nat_of_Z v1 pf)).
rewrite H13 in H; intuit.

∃ h3; split; auto; apply HStep_seq.

∃ h3; split; auto; apply HStep_if_true with (l := l); auto.

∃ h3; split; auto; apply HStep_if_false with (l := l); auto.

∃ h3; split; auto; apply HStep_while_true with (l := l); auto.

∃ h3; split; auto; apply HStep_while_false with (l := l); auto.
Qed.

Lemma hstepn_ff : ∀ n C K C' K' i s h1 h2 h3 i' s' h1',
  mydot h1 h2 h3 → hstepn n (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K') →
  ∃ h3', mydot h1' h2 h3' ∧ hstepn n (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K').
Proof.
induction n using (well_founded_induction lt_wf); intros.
```

```
inv H1.
∃ h3; split; auto; apply HStep_zero.
destruct cf' as [[i'' s'' h''] C'' K'']; apply hstep_ff with (h2 := h2) (h3 := h3) in H2;
auto.
destruct H2 as [h3' [H2]].
assert (n0 < S n0); try omega.
destruct (H _ H4 _ _ _ _ _ _ _ _ _ _ _ _ H2 H3) as [h3'' [H5]]; ∃ h3''; split; auto.
apply HStep_succ with (cf' := Cf (St i'' s'' h3') C'' K''); auto.
Qed.

Lemma hstep_bf : ∀ C K C' K' i s h1 h2 h3 i' s' h3',
  mydot h1 h2 h3 → hstep (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K') → hsafe (Cf
(St i s h1) C K) →
  ∃ h1', mydot h1' h2 h3' ∧ hstep (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K').
Proof.
intros.
inv H0.
∃ h1; split; auto; apply HStep_skip.
∃ h1; split; auto; apply HStep_assign with (l := l); auto.
∃ h1; split; auto; apply HStep_read with (l1 := l1) (l2 := l2) (pf := pf); auto.
specialize (H (nat_of_Z v1 pf)); rewrite H13 in H; decomp H; auto.
specialize (H1 0 (Cf (St i' s h1) (Read x e) K') (HStep_zero _) (refl_equal _)).
inv H1.
inv H.
rewrite H10 in H4; inv H4.
rewrite (proof_irrelevance _ pf0 pf) in H11; rewrite H11 in H2; inv H2.
specialize (H1 0 (Cf (St i' s' h1) (Write e1 e2) K') (HStep_zero _) (refl_equal _)).
inv H1.
inv H0.
rewrite H9 in H5; inv H5.
rewrite (proof_irrelevance _ pf0 pf) in H11.
∃ (upd h1 (nat_of_Z v1 pf) (v2,Hi)); split.
apply mydot_upd; auto.
specialize (H (nat_of_Z v1 pf)); destruct (h3 (nat_of_Z v1 pf)); decomp H; auto; try
contradiction.
apply HStep_write with (l1 := l1) (l2 := l2); auto.
∃ h1; split; auto; apply HStep_seq.
∃ h1; split; auto; apply HStep_if_true with (l := l); auto.
∃ h1; split; auto; apply HStep_if_false with (l := l); auto.
∃ h1; split; auto; apply HStep_while_true with (l := l); auto.
∃ h1; split; auto; apply HStep_while_false with (l := l); auto.
```

```
Qed.

Lemma hstepn_bf : ∀ n C K C' K' i s h1 h2 h3 i' s' h3',
  mydot h1 h2 h3 → hstepn n (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K') → hsafe
(Cf (St i s h1) C K) →
  ∃ h1', mydot h1' h2 h3' ∧ hstepn n (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K').
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H1.
∃ h1; split; auto; apply HStep_zero.
destruct cf' as [[i'' s'' h''] C'' K'']; apply hstep_bf with (h1 := h1) (h2 := h2) in H3;
auto.
destruct H3 as [h1' [H3]].
assert (n0 < S n0); try omega.
assert (hsafe (Cf (St i'' s'' h1') C'' K'')).
unfold hsafe; intros.
apply (H2 (S n)); auto.
apply HStep_succ with (cf' := Cf (St i'' s'' h1') C'' K''); auto.
destruct (H _ H5 _ _ _ _ _ _ _ _ _ _ _ _ H3 H4 H6) as [h1'' [H7]]; ∃ h1''; split; auto.
apply HStep_succ with (cf' := Cf (St i'' s'' h1') C'' K''); auto.
Qed.

Lemma lstep_ff : ∀ C K C' K' i s h1 h2 h3 i' s' h1' o,
  mydot h1 h2 h3 → lstep (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K') o →
  ∃ h3', mydot h1' h2 h3' ∧ lstep (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K') o.
Proof.
intros.
inv H0.
∃ h3; split; auto; apply LStep_skip.
∃ h3; split; auto; apply LStep_output; auto.
∃ h3; split; auto; apply LStep_assign; auto.
∃ h3; split; auto; apply LStep_read with (v1 := v1) (pf := pf); auto.
specialize (H (nat_of_Z v1 pf)); rewrite H13 in H.
destruct (h3 (nat_of_Z v1 pf)); decomp H; auto.
inv H1.
∃ (upd h3 (nat_of_Z v1 pf) (v2, l1 \_/ l2)); split.
apply mydot_upd; auto.
specialize (H (nat_of_Z v1 pf)).
destruct (h3 (nat_of_Z v1 pf)); decomp H0; auto; try contradiction.
apply LStep_write; auto.
contradict H14; specialize (H (nat_of_Z v1 pf)).
rewrite H14 in H; intuit.
∃ h3; split; auto; apply LStep_seq.
```

∃ *h3*; split; auto; apply *LStep_if_true*; auto.

∃ *h3*; split; auto; apply *LStep_if_false*; auto.

∃ *h3*; split; auto; apply *LStep_while_true*; auto.

∃ *h3*; split; auto; apply *LStep_while_false*; auto.

apply *hstepn_ff* with (*h2* := *h2*) (*h3* := *h3*) in *H12*; auto.
destruct *H12* as [*h3'* [*H12*]]; ∃ *h3'*; split; auto.
apply *LStep_if_hi* with (*v* := *v*) (*n* := *n*); auto.
unfold *hsafe*; intros.
destruct *cf'* as [[*i'' s'' h''*] *C'' K''*]; apply *hstepn_bf* with (*h1* := *h1*) (*h2* := *h2*) in *H1*; auto.
destruct *H1* as [*h1''* [*H1*]].
apply *H11* in *H3*; apply *H3* in *H2*.
inv H2.
destruct *cf'* as [[*i''' s''' h'''*] *C''' K'''*]; apply *hstep_ff* with (*h2* := *h2*) (*h3* := *h''*) in *H4*; auto.
destruct *H4* as [*h3''* [*H4*]].
apply (*Can_hstep* _ _ *H2*).

∃ *h3*; split; auto.
apply *LStep_if_hi_dvg* with (*v* := *v*); auto.
unfold *hsafe*; intros.
destruct *cf'* as [[*i'' s'' h''*] *C'' K''*]; apply *hstepn_bf* with (*h1* := *h1'*) (*h2* := *h2*) in *H0*; auto.
destruct *H0* as [*h1''* [*H0*]].
apply *H13* in *H2*; apply *H2* in *H1*.
inv H1.
destruct *cf'* as [[*i''' s''' h'''*] *C''' K'''*]; apply *hstep_ff* with (*h2* := *h2*) (*h3* := *h''*) in *H3*; auto.
destruct *H3* as [*h3'* [*H3*]].
apply (*Can_hstep* _ _ *H1*).
intros; intro.
destruct *st'* as [*i'' s'' h3'*]; apply *hstepn_bf* with (*h1* := *h1'*) (*h2* := *h2*) in *H0*; auto.
destruct *H0* as [*h1''* [*H0*]].
contradiction (H14 n (St i'' s'' h1'')).

apply *hstepn_ff* with (*h2* := *h2*) (*h3* := *h3*) in *H12*; auto.
destruct *H12* as [*h3'* [*H12*]]; ∃ *h3'*; split; auto.
apply *LStep_while_hi* with (*v* := *v*) (*n* := *n*); auto.
unfold *hsafe*; intros.
destruct *cf'* as [[*i'' s'' h''*] *C'' K''*]; apply *hstepn_bf* with (*h1* := *h1*) (*h2* := *h2*) in *H1*; auto.
destruct *H1* as [*h1''* [*H1*]].
apply *H11* in *H3*; apply *H3* in *H2*.

```
inv H2.
destruct cf' as [[i''' s''' h'''] C''' K''']; apply hstep_ff with (h2 := h2) (h3 := h'') in
H4; auto.
destruct H4 as [h3'' [H4]].
apply (Can_hstep _ _ H2).
∃ h3; split; auto.
apply LStep_while_hi_dvg with (v := v); auto.
unfold hsafe; intros.
destruct cf' as [[i'' s'' h''] C'' K'']; apply hstepn_bf with (h1 := h1') (h2 := h2) in H0;
auto.
destruct H0 as [h1'' [H0]].
apply H13 in H2; apply H2 in H1.
inv H1.
destruct cf' as [[i''' s''' h'''] C''' K''']; apply hstep_ff with (h2 := h2) (h3 := h'') in
H3; auto.
destruct H3 as [h3' [H3]].
apply (Can_hstep _ _ H1).
intros; intro.
destruct st' as [i'' s'' h3']; apply hstepn_bf with (h1 := h1') (h2 := h2) in H0; auto.
destruct H0 as [h1'' [H0]].
contradiction (H14 n (St i'' s'' h1'')).
Qed.

Lemma lstepn_ff : ∀ n C K C' K' i s h1 h2 h3 i' s' h1' o,
  mydot h1 h2 h3 → lstepn n (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K') o →
  ∃ h3', mydot h1' h2 h3' ∧ lstepn n (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K') o.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H1.
∃ h3; split; auto; apply LStep_zero.
destruct cf' as [[i'' s'' h''] C'' K'']; apply lstep_ff with (h2 := h2) (h3 := h3) in H2;
auto.
destruct H2 as [h3' [H2]].
assert (n0 < S n0); try omega.
destruct (H _ H4 _ _ _ _ _ _ _ _ _ _ _ _ _ H2 H3) as [h3'' [H5]]; ∃ h3''; split; auto.
apply LStep_succ with (cf' := Cf (St i'' s'' h3') C'' K''); auto.
Qed.

Corollary lstepn_nonincreasing : ∀ n i s h i' s' h' C K C' K' o a,
  lstepn n (Cf (St i s h) C K) (Cf (St i' s' h') C' K') o → h a = None → h' a = None.
Proof.
intros.
apply lstepn_ff with (h2 := fun n ⇒ if eq_nat_dec n a then Some (0%Z,Lo) else None)
(h3 := upd h a (0%Z,Lo)) in H.
```

```
destruct H as [h3' [H]].
specialize (H a).
destruct (h3' a); decomp H; auto.
destruct (eq_nat_dec a a); inv H4.
contradiction n0; auto.
intro a'.
unfold upd; destruct (eq_nat_dec a' a); subst; auto.
destruct (h a'); auto.
Qed.

Lemma lstep_bf : ∀ C K C' K' i s h1 h2 h3 i' s' h3' o,
  mydot h1 h2 h3 → lstep (Cf (St i s h3) C K) (Cf (St i' s' h3') C' K') o → lsafe (Cf
(St i s h1) C K) →
  ∃ h1', mydot h1' h2 h3' ∧ lstep (Cf (St i s h1) C K) (Cf (St i' s' h1') C' K') o.
Proof.
intros.
inv H0.
∃ h1; split; auto; apply LStep_skip.
∃ h1; split; auto; apply LStep_output; auto.
∃ h1; split; auto; apply LStep_assign with (l := l); auto.
∃ h1; split; auto; apply LStep_read with (l1 := l1) (l2 := l2) (pf := pf); auto.
specialize (H (nat_of_Z v1 pf)); rewrite H14 in H; decomp H; auto.
specialize (H1 0 (Cf (St i' s h1) (Read x e) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H.
rewrite H10 in H13; inv H13.
rewrite (proof_irrelevance _ pf0 pf) in H11; rewrite H11 in H2; inv H2.
specialize (H1 0 (Cf (St i' s' h1) (Write e1 e2) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H0.
rewrite H9 in H13; inv H13.
rewrite (proof_irrelevance _ pf0 pf) in H11.
∃ (upd h1 (nat_of_Z v1 pf) (v2, l1 \_/ l2)); split.
apply mydot_upd; auto.
specialize (H (nat_of_Z v1 pf)); destruct (h3 (nat_of_Z v1 pf)); decomp H; auto; try
contradiction.
apply LStep_write with (l1 := l1) (l2 := l2); auto.
∃ h1; split; auto; apply LStep_seq.
∃ h1; split; auto; apply LStep_if_true; auto.
∃ h1; split; auto; apply LStep_if_false; auto.
∃ h1; split; auto; apply LStep_while_true; auto.
```

```
∃ h1; split; auto; apply LStep_while_false; auto.

assert (hsafe (taint_vars_cf (Cf (St i s h1) (If b C1 C2) []))).
specialize (H1 0 (Cf (St i s h1) (If b C1 C2) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H0; auto.
rewrite H10 in H11; inv H11.
rewrite H10 in H11; inv H11.
apply hstepn_bf with (h1 := h1) (h2 := h2) in H13; auto.
destruct H13 as [h1' [H13]]; ∃ h1'; split; auto.
apply LStep_if_hi with (v := v) (n := n); auto.

∃ h1; split; auto.
apply LStep_if_hi_dvg with (v := v); auto.
specialize (H1 0 (Cf (St i' s' h1) (If b C1 C2) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H0; auto.
rewrite H10 in H13; inv H13.
rewrite H10 in H13; inv H13.
intros; intro.
destruct st' as [i'' s'' h'']; apply hstepn_ff with (h2 := h2) (h3 := h3') in H0; auto.
destruct H0 as [h3'' [H0]].
contradiction (H15 n (St i'' s'' h3'')).

assert (hsafe (taint_vars_cf (Cf (St i s h1) (While b C0) []))).
specialize (H1 0 (Cf (St i s h1) (While b C0) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H0; auto.
rewrite H9 in H11; inv H11.
rewrite H9 in H11; inv H11.
apply hstepn_bf with (h1 := h1) (h2 := h2) in H13; auto.
destruct H13 as [h1' [H13]]; ∃ h1'; split; auto.
apply LStep_while_hi with (v := v) (n := n); auto.

∃ h1; split; auto.
apply LStep_while_hi_dvg with (v := v); auto.
specialize (H1 0 (Cf (St i' s' h1) (While b C0) K') [] (LStep_zero _) (refl_equal _)).
inv H1.
inv H0; auto.
rewrite H9 in H13; inv H13.
rewrite H9 in H13; inv H13.
intros; intro.
destruct st' as [i'' s'' h'']; apply hstepn_ff with (h2 := h2) (h3 := h3') in H0; auto.
destruct H0 as [h3'' [H0]].
contradiction (H15 n (St i'' s'' h3'')).
```

Qed.

Lemma *lstepn_bf* : ∀ *n C K C' K' i s h1 h2 h3 i' s' h3' o*,
  mydot h1 h2 h3 → *lstepn n* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) *o* → *lsafe* (*Cf* (*St i s h1*) *C K*) →
  ∃ *h1'*, *mydot h1' h2 h3'* ∧ *lstepn n* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) *o*.
Proof.
induction *n* using (*well_founded_induction lt_wf*); intros.
inv H1.
∃ *h1*; split; auto; apply *LStep_zero*.
destruct *cf'* as [[*i'' s'' h''*] *C'' K''*]; apply *lstep_bf* with (*h1* := *h1*) (*h2* := *h2*) in *H3*; auto.
destruct *H3* as [*h1'* [*H3*]].
assert (*n0* < *S n0*); try omega.
assert (*lsafe* (*Cf* (*St i'' s'' h1'*) *C'' K''*)).
unfold *lsafe*; intros.
apply (*H2* (*S n*) _ (*o0*++*o*)); auto.
apply *LStep_succ* with (*cf'* := *Cf* (*St i'' s'' h1'*) *C'' K''*); auto.
destruct (*H* _ *H5* _ _ _ _ _ _ _ _ _ _ _ _ _ *H3 H4 H6*) as [*h1''* [*H7*]]; ∃ *h1''*; split; auto.
apply *LStep_succ* with (*cf'* := *Cf* (*St i'' s'' h1'*) *C'' K''*); auto.
Qed.

Lemma *hstep_modifies_monotonic* : ∀ *st st' C C' K K' x*,
  hstep (*Cf st C K*) (*Cf st' C' K'*) → *In x* (*modifies* (*C'*::*K'*)) → *In x* (*modifies* (*C*::*K*)).
Proof.
intros.
inv H; simpl in *; auto.
rewrite *app_assoc*; auto.
repeat rewrite *in_app_iff* in *H0* ⊢ *; *intuit*.
repeat rewrite *in_app_iff* in *H0* ⊢ *; *intuit*.
repeat rewrite *in_app_iff* in *H0* ⊢ *; *intuit*.
rewrite *in_app_iff*; *intuit*.
Qed.

Lemma *hstepn_modifies_monotonic* : ∀ *n st st' C C' K K' x*,
  hstepn n (*Cf st C K*) (*Cf st' C' K'*) → *In x* (*modifies* (*C'*::*K'*)) → *In x* (*modifies* (*C*::*K*)).
Proof.
induction *n* using (*well_founded_induction lt_wf*); intros.
inv H0; auto.
destruct *cf'* as [*st'' C'' K''*]; apply *H* with (*x* := *x*) in *H3*; auto.
apply *hstep_modifies_monotonic* with (*x* := *x*) in *H2*; auto.
Qed.

Lemma *lstep_modifies_monotonic* : ∀ *st st' C C' K K' x o*,

```
  lstep (Cf st C K) (Cf st' C' K') o → In x (modifies (C'::K')) → In x (modifies (C::K)).
Proof.
intros.
inv H; simpl in *; auto.
rewrite app_assoc; auto.
repeat rewrite in_app_iff in H0 ⊢ *; intuit.
repeat rewrite in_app_iff in H0 ⊢ *; intuit.
repeat rewrite in_app_iff in H0 ⊢ *; intuit.
rewrite in_app_iff; intuit.
repeat rewrite in_app_iff; intuit.
rewrite in_app_iff; intuit.
Qed.

Lemma lstepn_modifies_monotonic : ∀ n st st' C C' K K' x o,
  lstepn n (Cf st C K) (Cf st' C' K') o → In x (modifies (C'::K')) → In x (modifies
(C::K)).
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; auto.
destruct cf' as [st'' C'' K'']; apply H with (x := x) in H3; auto.
apply lstep_modifies_monotonic with (x := x) in H2; auto.
Qed.

Lemma hstep_modifies_const : ∀ st st' C C' K K' x,
  hstep (Cf st C K) (Cf st' C' K') → ¬ In x (modifies (C::K)) → (st:store) x = (st':store)
x.
Proof.
intros.
inv H; simpl; auto.
unfold upd; destruct (eq_nat_dec x x0); auto.
contradiction H0; simpl; auto.
unfold upd; destruct (eq_nat_dec x x0); auto.
contradiction H0; simpl; auto.
Qed.

Lemma hstepn_modifies_const : ∀ n st st' C C' K K' x,
  hstepn n (Cf st C K) (Cf st' C' K') → ¬ In x (modifies (C::K)) → (st:store) x =
(st':store) x.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; auto.
destruct cf' as [st'' C'' K''].
apply H with (x := x) in H3; auto.
apply hstep_modifies_const with (x := x) in H2; auto.
rewrite H2; rewrite H3; auto.
```

intro; *contradiction H1*.
apply *hstep_modifies_monotonic* with ($x := x$) in *H2*; auto.
Qed.

Lemma *lstep_modifies_const* : ∀ *st st' C C' K K' x o*,
lstep (*Cf st C K*) (*Cf st' C' K'*) *o* → ¬ *In x* (*modifies* (*C*::*K*)) → (*st*:*store*) *x* = (*st'*:*store*) *x*.
Proof.
intros.
inv H; simpl; auto.
unfold *upd*; destruct (*eq_nat_dec x x0*); auto.
contradiction H0; simpl; auto.
unfold *upd*; destruct (*eq_nat_dec x x0*); auto.
contradiction H0; simpl; auto.
apply *hstepn_modifies_const* with ($x := x$) in *H10*; simpl in *.
rewrite ← *H10*; unfold *taint_vars*.
destruct (*In_dec eq_nat_dec x* (*modifies* [*If b C1 C2*])); auto.
contradiction H0; simpl in *i0*; rewrite *in_app_iff* in *i0* ⊢ ×.
destruct *i0*; auto.
inv H.
intro; *contradiction H0*.
rewrite *in_app_iff* in *H* ⊢ ×.
destruct *H*; auto.
inv H.
apply *hstepn_modifies_const* with ($x := x$) in *H10*; simpl in *.
rewrite ← *H10*; unfold *taint_vars*.
destruct (*In_dec eq_nat_dec x* (*modifies* [*While b C0*])); auto.
contradiction H0; simpl in *i0*; rewrite *in_app_iff* in *i0* ⊢ ×.
destruct *i0*; auto.
inv H.
intro; *contradiction H0*.
rewrite *in_app_iff* in *H* ⊢ ×.
destruct *H*; auto.
inv H.
Qed.

Lemma *lstepn_modifies_const* : ∀ *n st st' C C' K K' x o*,
lstepn n (*Cf st C K*) (*Cf st' C' K'*) *o* → ¬ *In x* (*modifies* (*C*::*K*)) → (*st*:*store*) *x* = (*st'*:*store*) *x*.
Proof.
induction *n* using (*well_founded_induction lt_wf*); intros.
inv H0; auto.
destruct *cf'* as [*st'' C'' K''*].
apply *H* with ($x := x$) in *H3*; auto.

```
apply lstep_modifies_const with (x := x) in H2; auto.
rewrite H2; rewrite H3; auto.
intro; contradiction H1.
apply lstep_modifies_monotonic with (x := x) in H2; auto.
Qed.

Lemma hstep_taints_s : ∀ i s h i' s' h' C K C' K' x,
  hstep (Cf (St i s h) C K) (Cf (St i' s' h') C' K') →
  s x ≠ s' x → ∃ v, s' x = Some (v,Hi).
Proof.
intros.
inv H; try solve [contradiction H0; auto].
destruct (eq_nat_dec x x0); subst.
∃ v; unfold upd; destruct (eq_nat_dec x0 x0); auto.
contradiction n; auto.
contradiction H0; unfold upd.
destruct (eq_nat_dec x x0); auto; contradiction.
destruct (eq_nat_dec x x0); subst.
∃ v2; unfold upd; destruct (eq_nat_dec x0 x0); auto.
contradiction n; auto.
contradiction H0; unfold upd.
destruct (eq_nat_dec x x0); auto; contradiction.
Qed.

Lemma hstepn_taints_s : ∀ n i s h i' s' h' C K C' K' x,
  hstepn n (Cf (St i s h) C K) (Cf (St i' s' h') C' K') →
  s x ≠ s' x → ∃ v, s' x = Some (v,Hi).
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
contradiction H1; auto.
destruct cf' as [[i'' s'' h''] C'' K''].
destruct (opt_eq_dec val_eq_dec (s'' x) (s' x)).
rewrite ← e in H1 ⊢ ×.
apply hstep_taints_s with (x := x) in H2; auto.
assert (n0 < S n0); try omega.
apply (H _ H0 _ _ _ _ _ _ _ _ _ _ _ H3 n).
Qed.

Lemma hstep_taints_h : ∀ i s h i' s' h' C K C' K' a,
  hstep (Cf (St i s h) C K) (Cf (St i' s' h') C' K') →
  h a ≠ h' a → ∃ v, h' a = Some (v,Hi).
Proof.
intros.
inv H; try solve [contradiction H0; auto].
```

```
destruct (eq_nat_dec (nat_of_Z v1 pf) a); subst.
∃ v2; unfold upd; destruct (eq_nat_dec (nat_of_Z v1 pf) (nat_of_Z v1 pf)); auto.
contradiction n; auto.
contradiction H0; unfold upd.
destruct (eq_nat_dec a (nat_of_Z v1 pf)); auto; subst.
contradiction n; auto.
Qed.

Lemma hstepn_taints_h : ∀ n i s h i' s' h' C K C' K' a,
  hstepn n (Cf (St i s h) C K) (Cf (St i' s' h') C' K') →
  h a ≠ h' a → ∃ v, h' a = Some (v,Hi).
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
contradiction H1; auto.
destruct cf' as [[i'' s'' h''] C'' K''].
destruct (opt_eq_dec val_eq_dec (h'' a) (h' a)).
rewrite ← e in H1 ⊢ ×.
apply hstep_taints_h with (a := a) in H2; auto.
assert (n0 < S n0); try omega.
apply (H _ H0 _ _ _ _ _ _ _ _ _ _ _ H3 n).
Qed.

Proposition hstep_i_const : ∀ i s h i' s' h' C C' K K',
  hstep (Cf (St i s h) C K) (Cf (St i' s' h') C' K') → i' = i.
Proof.
intros.
inv H; auto.
Qed.

Proposition hstepn_i_const : ∀ n i s h i' s' h' C C' K K',
  hstepn n (Cf (St i s h) C K) (Cf (St i' s' h') C' K') → i' = i.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; auto.
destruct cf' as [[i'' s'' h''] C'' K'']; apply H in H2; subst; auto.
apply hstep_i_const in H1; auto.
Qed.

Proposition lstep_i_const : ∀ i s h i' s' h' C C' K K' o,
  lstep (Cf (St i s h) C K) (Cf (St i' s' h') C' K') o → i' = i.
Proof.
intros.
inv H; auto.
apply hstepn_i_const in H11; auto.
```

```
apply hstepn_i_const in H11; auto.
Qed.

Proposition lstepn_i_const : ∀ n i s h i' s' h' C C' K K' o,
  lstepn n (Cf (St i s h) C K) (Cf (St i' s' h') C' K') o → i' = i.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; auto.
destruct cf' as [[i'' s'' h''] C'' K'']; apply H in H2; subst; auto.
apply lstep_i_const in H1; auto.
Qed.

Close Scope Z_scope.

Definition obs_eq_s (s1 s2 : store) : Prop := ∀ x,
  match s1 x, s2 x with
  | None, None ⇒ True
  | Some (v1,l1), Some (v2,l2) ⇒ l1 = l2 ∧ (l1 = Lo → v1 = v2)
  | _, _ ⇒ False
  end.

Definition obs_eq_h (h1 h2 : heap) : Prop := ∀ n,
  match h1 n, h2 n with
  | Some (v1,l1), Some (v2,l2) ⇒ l1 = Lo → l2 = Lo → v1 = v2
  | _, _ ⇒ True
  end.

Definition obs_eq (st1 st2 : state) : Prop := (st1:lmap) = (st2:lmap) ∧ obs_eq_s st1 st2
∧ obs_eq_h st1 st2.

Proposition obs_eq_s_refl : ∀ s, obs_eq_s s s.
Proof.
unfold obs_eq_s; intros.
destruct (s x) as [[v l]|]; auto.
Qed.

Proposition obs_eq_h_refl : ∀ h, obs_eq_h h h.
Proof.
unfold obs_eq_h; intros.
destruct (h n) as [[v l]|]; auto.
Qed.

Proposition obs_eq_refl : ∀ st, obs_eq st st.
Proof.
unfold obs_eq; intuition.
apply obs_eq_s_refl.
apply obs_eq_h_refl.
Qed.
```

```
Proposition obs_eq_s_sym : ∀ s1 s2, obs_eq_s s1 s2 → obs_eq_s s2 s1.
Proof.
unfold obs_eq_s; intros.
specialize (H x); destruct (s1 x) as [[v1 l1]|]; destruct (s2 x) as [[v2 l2]|]; auto.
destruct H; split; auto; intros.
subst; intuit.
Qed.

Proposition obs_eq_h_sym : ∀ h1 h2, obs_eq_h h1 h2 → obs_eq_h h2 h1.
Proof.
unfold obs_eq_h; intros.
specialize (H n); destruct (h1 n) as [[v1 l1]|]; destruct (h2 n) as [[v2 l2]|]; intuit.
Qed.

Proposition obs_eq_sym : ∀ st1 st2, obs_eq st1 st2 → obs_eq st2 st1.
Proof.
unfold obs_eq; intuition.
apply obs_eq_s_sym; auto.
apply obs_eq_h_sym; auto.
Qed.

Lemma obs_eq_exp : ∀ e i1 s1 h1 i2 s2 h2, obs_eq (St i1 s1 h1) (St i2 s2 h2) →
  match eden e i1 s1, eden e i2 s2 with
  | None, None ⇒ True
  | Some (v1,l1), Some (v2,l2) ⇒ l1 = l2 ∧ (l1 = Lo → v1 = v2)
  | _, _ ⇒ False
  end.
Proof.
induction e; simpl; intros; auto.
unfold obs_eq in H; decomp H.
apply H2.
unfold obs_eq in H; decomp H; simpl in *; subst; auto.
specialize (IHe1 _ _ _ _ _ _ H); specialize (IHe2 _ _ _ _ _ _ H).
destruct (eden e1 i1 s1) as [[v1 l1]|]; destruct (eden e2 i2 s2) as [[v2 l2]|];
  destruct (eden e1 i2 s2) as [[v1' l1']|]; destruct (eden e2 i1 s1) as [[v2' l2']|]; simpl
in *; intuit.
destruct IHe1; destruct IHe2; destruct H; subst; split; auto; intros.
glub_simpl H0; rewrite H1; auto; rewrite H3; auto.
Qed.

Lemma obs_eq_bexp : ∀ b i1 s1 h1 i2 s2 h2, obs_eq (St i1 s1 h1) (St i2 s2 h2) →
  match bden b i1 s1, bden b i2 s2 with
  | None, None ⇒ True
  | Some (v1,l1), Some (v2,l2) ⇒ l1 = l2 ∧ (l1 = Lo → v1 = v2)
  | _, _ ⇒ False
```

```
  end.
Proof.
induction b; simpl; intros; auto.
dup H; apply (obs_eq_exp e) in H.
apply (obs_eq_exp e0) in H0.
destruct (eden e i1 s1) as [[v1 l1]||]; destruct (eden e0 i2 s2) as [[v2 l2]||];
  destruct (eden e i2 s2) as [[v1' l1']||]; destruct (eden e0 i1 s1) as [[v2' l2']||]; simpl
in *; intuit.
destruct H; destruct H0; subst; split; auto; intros.
glub_simpl H; rewrite H1; auto; rewrite H2; auto.
apply IHb in H.
destruct (bden b i1 s1) as [[v1 l1]||].
destruct (bden b i2 s2) as [[v2 l2]||]; auto; simpl.
destruct H; subst; split; auto; intros.
rewrite H0; auto.
destruct (bden b i2 s2); intuit.
dup H.
apply IHb1 in H; apply IHb2 in H0.
destruct (bden b2 i1 s1) as [[v1 l1]||]; destruct (bden b3 i2 s2) as [[v2 l2]||];
  destruct (bden b2 i2 s2) as [[v1' l1']||]; destruct (bden b3 i1 s1) as [[v2' l2']||]; simpl
in *; intuit.
destruct H; destruct H0; subst; split; auto; intros.
glub_simpl H; rewrite H1; auto; rewrite H2; auto.
Qed.

Inductive lexp :=
| Lbl : glbl → lexp
| Lblvar : nat → lexp
| Lub : lexp → lexp → lexp.

Definition toLexp (l : glbl) : lexp := Lbl l.
Coercion toLexp : glbl >-> lexp.

Fixpoint lden (L : lexp) (i : lmap) : glbl :=
  match L with
  | Lbl l ⇒ l
  | Lblvar X ⇒ snd i X
  | Lub L1 L2 ⇒ glub (lden L1 i) (lden L2 i)
  end.

Proposition lden_lblvars : ∀ L i1 i2 i, lden L (i1,i) = lden L (i2,i).
Proof.
induction L; simpl; auto; intros.
rewrite (IHL1 _ i2); rewrite (IHL2 _ i2); auto.
Qed.
```

```
Inductive assert :=
| TrueA : assert
| FalseA : assert
| Emp : assert
| Allocated : exp → assert
| Mapsto : exp → exp → lexp → assert
| BoolExp : bexp → assert
| EqLbl : lexp → lexp → assert
| LblEq : var → lexp → assert
| LblLeq : var → lexp → assert
| LblLeq' : lexp → var → assert
| LblExp : exp → lexp → assert
| LblBexp : bexp → lexp → assert
| Conj : assert → assert → assert
| Disj : assert → assert → assert
| Star : assert → assert → assert.

Fixpoint vars (P : assert) (x : var) : bool :=
  match P with
  | TrueA ⇒ false
  | FalseA ⇒ false
  | Emp ⇒ false
  | Allocated e ⇒ expvars e x
  | Mapsto e e' L ⇒ orb (expvars e x) (expvars e' x)
  | BoolExp b ⇒ bexpvars b x
  | EqLbl L1 L2 ⇒ false
  | LblEq y L ⇒ if eq_nat_dec y x then true else false
  | LblLeq y L ⇒ if eq_nat_dec y x then true else false
  | LblLeq' L y ⇒ if eq_nat_dec y x then true else false
  | LblExp e L ⇒ expvars e x
  | LblBexp b L ⇒ bexpvars b x
  | Conj P Q ⇒ orb (vars P x) (vars Q x)
  | Disj P Q ⇒ orb (vars P x) (vars Q x)
  | Star P Q ⇒ orb (vars P x) (vars Q x)
  end.
Notation " P `AND` Q " := (Conj P Q) (at level 91, left associativity).
Notation " P `OR` Q " := (Disj P Q) (at level 91, left associativity).
Notation " P ** Q " := (Star P Q) (at level 91, left associativity).

Fixpoint ereplace e x ex : exp :=
  match e with
  | Var y ⇒ if eq_nat_dec y x then ex else Var y
  | BinOp bop e1 e2 ⇒ BinOp bop (ereplace e1 x ex) (ereplace e2 x ex)
  | _ ⇒ e
```

```
  end.
Proposition ereplace_deletes : ∀ e x ex, expvars ex x = false → expvars (ereplace e x ex)
x = false.
Proof.
induction e; simpl; intros; auto.
destruct (eq_nat_dec v x); subst; simpl; auto.
destruct (eq_nat_dec v x); try contradiction; auto.
rewrite (IHe1 _ _ H); rewrite (IHe2 _ _ H); auto.
Qed.
Proposition eden_ereplace : ∀ e x ex i s, eden (Var x) i s = eden ex i s → eden (ereplace
e x ex) i s = eden e i s.
Proof.
induction e; simpl; intros; auto.
destruct (eq_nat_dec v x); subst; auto.
rewrite (IHe1 _ _ _ _ H); rewrite (IHe2 _ _ _ _ H); auto.
Qed.
Proposition edenZ_ereplace : ∀ e x ex i s, edenZ (Var x) i s = edenZ ex i s → edenZ
(ereplace e x ex) i s = edenZ e i s.
Proof.
induction e; simpl; intros; auto.
destruct (eq_nat_dec v x); subst; auto.
rewrite (IHe1 _ _ _ _ H); rewrite (IHe2 _ _ _ _ H); auto.
Qed.
Fixpoint aden (P : assert) (st : state) : Prop :=
match st with St i s h ⇒
  match P with
  | TrueA ⇒ True
  | FalseA ⇒ False
  | Emp ⇒ h = fun _ ⇒ None
  | Allocated e ⇒ ∃ v : Z, ∃ pf : (v>=0)%Z, edenZ e i s = Some v ∧
                              ∃ v', ∃ l', h = fun n ⇒ if eq_nat_dec n (nat_of_Z v pf) then Some
(v',l') else None
  | Mapsto e e' L ⇒ ∃ v : Z, ∃ pf : (v>=0)%Z, edenZ e i s = Some v ∧ ∃ v', edenZ e' i
s = Some v' ∧
                                   h = fun n ⇒ if eq_nat_dec n (nat_of_Z v pf) then Some (v',
lden L i) else None
  | BoolExp b ⇒ bdenZ b i s = Some true
  | EqLbl L1 L2 ⇒ lden L1 i = lden L2 i
  | LblEq x L ⇒ ∃ v, s x = Some (v, lden L i)
  | LblLeq x L ⇒ ∃ v, ∃ l, s x = Some (v,l) ∧ gleq l (lden L i) = true
  | LblLeq' L x ⇒ ∃ v, ∃ l, s x = Some (v,l) ∧ gleq (lden L i) l = true
```

```
  | LblExp e L ⇒ ∃ v, eden e i s = Some (v, lden L i)
  | LblBexp b L ⇒ ∃ v, bden b i s = Some (v, lden L i)
  | Conj P Q ⇒ aden P st ∧ aden Q st
  | Disj P Q ⇒ aden P st ∨ aden Q st
  | Star P Q ⇒ ∃ h1, ∃ h2, mydot h1 h2 h ∧ aden P (St i s h1) ∧ aden Q (St i s h2)
  end
end.
```

```
Definition aden2 (P : assert) (st1 st2 : state) : Prop := aden P st1 ∧ aden P st2 ∧
obs_eq st1 st2.
```

```
Definition implies (P Q : assert) := ∀ st, aden P st → aden Q st.
```

```
Fixpoint haslbl (P : assert) (x : var) : bool :=
  match P with
  | LblEq y L ⇒ if eq_nat_dec y x then true else false
  | LblLeq y L ⇒ if eq_nat_dec y x then true else false
  | LblLeq' L y ⇒ if eq_nat_dec y x then true else false
  | LblExp e L ⇒ expvars e x
  | LblBexp b L ⇒ bexpvars b x
  | Conj P Q ⇒ orb (haslbl P x) (haslbl Q x)
  | Disj P Q ⇒ orb (haslbl P x) (haslbl Q x)
  | Star P Q ⇒ orb (haslbl P x) (haslbl Q x)
  | _ ⇒ false
  end.
```

```
Proposition eden_upd : ∀ e x i s v l, expvars e x = false → eden e i (upd s x (v,l)) =
eden e i s.
Proof.
induction e; simpl; intros; auto.
unfold upd; destruct (eq_nat_dec v x); inv H; auto.
rewrite IHe1.
rewrite IHe2; auto.
destruct (expvars e1 x); destruct (expvars e2 x); inv H; auto.
destruct (expvars e1 x); inv H; auto.
Qed.
```

```
Proposition edenZ_upd : ∀ e x i s v l, expvars e x = false → edenZ e i (upd s x (v,l)) =
edenZ e i s.
Proof.
induction e; simpl; intros; auto.
unfold upd; destruct (eq_nat_dec v x); inv H; auto.
rewrite IHe1.
rewrite IHe2; auto.
destruct (expvars e1 x); destruct (expvars e2 x); inv H; auto.
destruct (expvars e1 x); inv H; auto.
```

Qed.

Proposition *bden_upd* : ∀ *b x i s v l*, *bexpvars b x* = *false* → *bden b i* (*upd s x* (*v*,*l*)) = *bden b i s*.
Proof.
induction *b*; simpl; intros; auto.
repeat rewrite *eden_upd*; auto.
destruct (*expvars e x*); destruct (*expvars e0 x*); *inv H*; auto.
destruct (*expvars e x*); *inv H*; auto.
rewrite *IHb*; auto.
rewrite *IHb1*.
rewrite *IHb2*; auto.
destruct (*bexpvars b2 x*); destruct (*bexpvars b3 x*); *inv H*; auto.
destruct (*bexpvars b2 x*); *inv H*; auto.
Qed.

Proposition *bdenZ_upd* : ∀ *b x i s v l*, *bexpvars b x* = *false* → *bdenZ b i* (*upd s x* (*v*,*l*)) = *bdenZ b i s*.
Proof.
induction *b*; simpl; intros; auto.
repeat rewrite *edenZ_upd*; auto.
destruct (*expvars e x*); destruct (*expvars e0 x*); *inv H*; auto.
destruct (*expvars e x*); *inv H*; auto.
rewrite *IHb*; auto.
rewrite *IHb1*.
rewrite *IHb2*; auto.
destruct (*bexpvars b2 x*); destruct (*bexpvars b3 x*); *inv H*; auto.
destruct (*bexpvars b2 x*); *inv H*; auto.
Qed.

Proposition *aden_upd* : ∀ *P x i s h v l*, *vars P x* = *false* → *aden P* (*St i s h*) → *aden P* (*St i* (*upd s x* (*v*,*l*)) *h*).
Proof.
induction *P*; simpl; intros; auto.
rewrite *edenZ_upd*; auto.
apply *orb_false_elim* in *H*.
repeat rewrite *edenZ_upd*; *intuit*.
rewrite *bdenZ_upd*; auto.
unfold *upd*; destruct (*eq_nat_dec v x*); *inv H*; auto.
unfold *upd*; destruct (*eq_nat_dec v x*); *inv H*; auto.
unfold *upd*; destruct (*eq_nat_dec v x*); *inv H*; auto.
rewrite *eden_upd*; auto.
rewrite *bden_upd*; auto.
apply *orb_false_elim* in *H*; *intuit*.
apply *orb_false_elim* in *H*; *intuit*.

```
apply orb_false_elim in H; destruct H0 as [h1 [h2]]; ∃ h1; ∃ h2; intuit.
Qed.

Proposition eden_vars_same : ∀ e i s s',
  (∀ x, expvars e x = true → s x = s' x) → eden e i s = eden e i s'.
Proof.
induction e; simpl; intros; auto.
apply H; destruct (eq_nat_dec v v); auto.
rewrite IHe1 with (s' := s'); intros.
rewrite IHe2 with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (expvars e1 x); auto.
apply H; rewrite H0; auto.
Qed.

Proposition edenZ_vars_same : ∀ e i s s',
  (∀ x, expvars e x = true → s x = s' x) → edenZ e i s = edenZ e i s'.
Proof.
induction e; simpl; intros; auto.
rewrite H; destruct (eq_nat_dec v v); auto.
rewrite IHe1 with (s' := s'); intros.
rewrite IHe2 with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (expvars e1 x); auto.
apply H; rewrite H0; auto.
Qed.

Proposition bden_vars_same : ∀ b i s s',
  (∀ x, bexpvars b x = true → s x = s' x) → bden b i s = bden b i s'.
Proof.
induction b; simpl; intros; auto.
rewrite eden_vars_same with (s' := s'); intros.
rewrite (eden_vars_same e0) with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (expvars e x); auto.
apply H; rewrite H0; auto.
rewrite IHb with (s' := s'); auto.
rewrite IHb1 with (s' := s'); intros.
rewrite IHb2 with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (bexpvars b2 x); auto.
apply H; rewrite H0; auto.
Qed.

Proposition bdenZ_vars_same : ∀ b i s s',
  (∀ x, bexpvars b x = true → s x = s' x) → bdenZ b i s = bdenZ b i s'.
Proof.
induction b; simpl; intros; auto.
rewrite edenZ_vars_same with (s' := s'); intros.
```

```
rewrite (edenZ_vars_same e0) with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (expvars e x); auto.
apply H; rewrite H0; auto.
rewrite IHb with (s' := s'); auto.
rewrite IHb1 with (s' := s'); intros.
rewrite IHb2 with (s' := s'); auto; intros.
apply H; rewrite H0; destruct (bexpvars b2 x); auto.
apply H; rewrite H0; auto.
Qed.

Proposition aden_vars_same : ∀ P i s s' h,
  (∀ x, vars P x = true → s x = s' x) → aden P (St i s h) → aden P (St i s' h).
Proof.
induction P; simpl; intros; auto.
rewrite edenZ_vars_same with (s' := s') in H0; auto.
rewrite edenZ_vars_same with (s' := s') in H0; intuit.
rewrite (edenZ_vars_same e0) with (s' := s') in H0; intuit.
rewrite bdenZ_vars_same with (s' := s') in H0; auto.
rewrite ← H; auto.
destruct (eq_nat_dec v v); auto.
rewrite ← H; auto.
destruct (eq_nat_dec v v); auto.
rewrite ← H; auto.
destruct (eq_nat_dec v v); auto.
rewrite eden_vars_same with (s' := s') in H0; auto.
rewrite bden_vars_same with (s' := s') in H0; auto.
split; [apply IHP1 with (s := s) | apply IHP2 with (s := s)]; intuit.
destruct H0; [left; apply IHP1 with (s := s) | right; apply IHP2 with (s := s)]; intuit.
destruct H0 as [h1 [h2]]; ∃ h1; ∃ h2; intuition.
apply IHP1 with (s := s); intuit.
apply IHP2 with (s := s); intuit.
Qed.

Proposition expvars_none : ∀ e i s x v l, eden e i s = Some (v,l) → s x = None →
expvars e x = false.
Proof.
induction e; simpl; intros; auto.
destruct (eq_nat_dec v x); subst; auto.
rewrite H in H0; inv H0.
case_eq (eden e1 i s); intros.
case_eq (eden e2 i s); intros.
destruct v1 as [v2 l2]; destruct v0 as [v1 l1].
rewrite H1 in H; rewrite H2 in H; inv H.
apply IHe1 with (x := x) in H1; auto.
```

```
apply IHe2 with (x := x) in H2; auto.
rewrite H1; rewrite H2; auto.
rewrite H2 in H; destruct (eden e1 i s); inv H.
rewrite H1 in H; inv H.
Qed.
```

```
Proposition bexpvars_none : ∀ b i s x v l, bden b i s = Some (v,l) → s x = None →
bexpvars b x = false.
Proof.
induction b; simpl; intros; auto.
case_eq (eden e i s); intros.
case_eq (eden e0 i s); intros.
destruct v1 as [v2 l2]; destruct v0 as [v1 l1].
rewrite H1 in H; rewrite H2 in H; inv H.
apply expvars_none with (x := x) in H1; auto.
apply expvars_none with (x := x) in H2; auto.
rewrite H1; rewrite H2; auto.
rewrite H2 in H; destruct (eden e i s); inv H.
rewrite H1 in H; inv H.
case_eq (bden b i s); intros.
destruct p; apply IHb with (x := x) in H1; auto.
rewrite H1 in H; inv H.
case_eq (bden b2 i s); intros.
case_eq (bden b3 i s); intros.
destruct p0 as [v2 l2]; destruct p as [v1 l1].
rewrite H1 in H; rewrite H2 in H; inv H.
apply IHb1 with (x := x) in H1; auto.
apply IHb2 with (x := x) in H2; auto.
rewrite H1; rewrite H2; auto.
rewrite H2 in H; destruct (bden b2 i s); inv H.
rewrite H1 in H; inv H.
Qed.
```

```
Proposition aden_upd_none : ∀ P x i s h v l, s x = None → aden P (St i s h) → aden
P (St i (upd s x (v,l)) h).
Proof.
induction P; simpl; intros; intuit.
rewrite edenZ_upd; auto.
destruct H0 as [v1 [pf [H0]]].
rewrite edenZ_some in H0; destruct H0 as [l1].
apply expvars_none with (x := x) in H0; auto.
repeat rewrite edenZ_upd; auto.
destruct H0 as [v1 [pf [H0 [v2 [H1]]]]].
rewrite edenZ_some in H1; destruct H1 as [l2].
```

```
apply expvars_none with (x := x) in H1; auto.
destruct H0 as [v1 [pf [H0]]].
rewrite edenZ_some in H0; destruct H0 as [l1].
apply expvars_none with (x := x) in H0; auto.
rewrite bdenZ_upd; auto.
rewrite bdenZ_some in H0; destruct H0 as [l1].
apply bexpvars_none with (x := x) in H0; auto.
unfold upd; destruct (eq_nat_dec v x); subst; auto.
destruct H0 as [v]; rewrite H in H0; inv H0.
unfold upd; destruct (eq_nat_dec v x); subst; auto.
destruct H0 as [v1 [l1 [H0]]]; rewrite H in H0; inv H0.
unfold upd; destruct (eq_nat_dec v x); subst; auto.
destruct H0 as [v1 [l1 [H0]]]; rewrite H in H0; inv H0.
rewrite eden_upd; auto.
destruct H0 as [v1]; apply expvars_none with (x := x) in H0; auto.
rewrite bden_upd; auto.
destruct H0 as [v1]; apply bexpvars_none with (x := x) in H0; auto.
destruct H0 as [h1 [h2]]; ∃ h1; ∃ h2; intuit.
Qed.

Proposition eden_taint_vars : ∀ e i s K v l, eden e i s = Some (v,l) →
  ∃ l', eden e i (taint_vars K s) = Some (v,l') ∧ l «= l'.
Proof.
induction e; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec v (modifies K)).
∃ Hi; rewrite H; split; auto.
destruct l; auto.
∃ l; rewrite H; split; auto.
destruct l; auto.
inv H; ∃ Lo; split; auto.
inv H; ∃ Lo; split; auto.
case_eq (eden e1 i s); case_eq (eden e2 i s); intros.
rewrite H1 in H; rewrite H0 in H; simpl in H; inv H.
destruct v1 as [v1 l1]; destruct v0 as [v2 l2].
apply IHe1 with (K := K) in H1; apply IHe2 with (K := K) in H0.
destruct H1 as [l1' [H1]]; destruct H0 as [l2' [H0]].
∃ (l1' \_/ l2'); simpl; split.
rewrite H1; rewrite H0; simpl; auto.
destruct l1; destruct l1'; destruct l2; destruct l2'; intuit.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
```

```
Qed.
Proposition bden_taint_vars : ∀ b i s K v l, bden b i s = Some (v,l) →
  ∃ l', bden b i (taint_vars K s) = Some (v,l') ∧ l «= l'.
Proof.
induction b; simpl; intros.
inv H; ∃ Lo; split; auto.
inv H; ∃ Lo; split; auto.
case_eq (eden e i s); case_eq (eden e0 i s); intros.
destruct v1 as [v1 l1]; destruct v0 as [v2 l2].
rewrite H1 in H; rewrite H0 in H; inv H.
apply eden_taint_vars with (K := K) in H1; apply eden_taint_vars with (K := K) in
H0.
destruct H1 as [l1' [H1]]; destruct H0 as [l2' [H0]].
∃ (l1' \_/ l2'); split.
rewrite H1; rewrite H0; auto.
destruct l1; destruct l1'; destruct l2; destruct l2'; intuit.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
case_eq (bden b i s); intros.
destruct p as [v' l']; rewrite H0 in H; inv H.
apply IHb with (K := K) in H0; destruct H0 as [l' [H0]].
∃ l'; split; auto.
rewrite H0; auto.
rewrite H0 in H; inv H.
case_eq (bden b2 i s); case_eq (bden b3 i s); intros.
rewrite H1 in H; rewrite H0 in H; simpl in H; inv H.
destruct p0 as [v1 l1]; destruct p as [v2 l2].
apply IHb1 with (K := K) in H1; apply IHb2 with (K := K) in H0.
destruct H1 as [l1' [H1]]; destruct H0 as [l2' [H0]].
∃ (l1' \_/ l2'); simpl; split.
rewrite H1; rewrite H0; simpl; auto.
destruct l1; destruct l1'; destruct l2; destruct l2'; intuit.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
rewrite H1 in H; rewrite H0 in H; inv H.
Qed.
Proposition edenZ_ignores_lbl : ∀ e i s x v l l',
  s x = Some (v,l) → edenZ e i (upd s x (v,l')) = edenZ e i s.
Proof.
induction e; simpl; intros; auto.
unfold upd; destruct (eq_nat_dec v x); subst; auto.
```

```
rewrite H; auto.
rewrite IHe1 with (l := l); auto.
rewrite IHe2 with (l := l); auto.
Qed.

Proposition bdenZ_ignores_lbl : ∀ b i s x v l l',
  s x = Some (v,l) → bdenZ b i (upd s x (v,l')) = bdenZ b i s.
Proof.
induction b; simpl; intros; auto.
repeat rewrite edenZ_ignores_lbl with (l := l); auto.
rewrite IHb with (l := l); auto.
rewrite IHb1 with (l := l); auto.
rewrite IHb2 with (l := l); auto.
Qed.

Proposition aden_haslbl : ∀ P x i s h v l l', haslbl P x = false → s x = Some (v,l) →
  aden P (St i s h) → aden P (St i (upd s x (v,l')) h).
Proof.
induction P; simpl; intros; auto.
rewrite edenZ_ignores_lbl with (l := l); auto.
repeat rewrite edenZ_ignores_lbl with (l := l0); auto.
rewrite bdenZ_ignores_lbl with (l := l); auto.
unfold upd; destruct (eq_nat_dec v x); auto; inv H.
unfold upd; destruct (eq_nat_dec v x); auto; inv H.
unfold upd; destruct (eq_nat_dec v x); auto; inv H.
rewrite eden_upd; auto.
rewrite bden_upd; auto.
apply orb_false_elim in H; destruct H; destruct H1; split.
apply IHP1 with (l := l); auto.
apply IHP2 with (l := l); auto.
apply orb_false_elim in H; destruct H; destruct H1; [left | right].
apply IHP1 with (l := l); auto.
apply IHP2 with (l := l); auto.
apply orb_false_elim in H; destruct H; destruct H1 as [h1 [h2]]; decomp H1.
∃ h1; ∃ h2; repeat (split; auto).
apply IHP1 with (l := l); auto.
apply IHP2 with (l := l); auto.
Qed.

Definition taint_vars_assert (P : assert) (xs : list var) (l1 l2 : glbl) : assert :=
  if gleq l1 l2 then P else P `AND` fold_right (fun x P ⇒ P `AND` LblLeq' (glub l1 l2)
x) TrueA xs.

Proposition aden_fold : ∀ (f : var → assert) xs st,
  (∀ x, In x xs → aden (f x) st) → aden (fold_right (fun x P ⇒ P `AND` f x) TrueA xs)
```

```
st.
Proof.
induction xs; destruct st as [i s h]; simpl; intros; auto.
Qed.

Proposition aden_fold_inv : ∀ (f : var → assert) xs st,
  aden (fold_right (fun x P ⇒ P `AND` f x) TrueA xs) st → ∀ x, In x xs → aden (f x) st.
Proof.
induction xs; destruct st as [i s h]; simpl; intros; intuit.
destruct H0; subst; intuit.
Qed.

Fixpoint no_lbls (P : assert) (xs : list var) :=
  match xs with
  | [] ⇒ true
  | x::xs ⇒ andb (negb (haslbl P x)) (no_lbls P xs)
  end.

Definition same_values (s1 s2 : store) (xs : list var) := ∀ x,
  if In_dec eq_nat_dec x xs then
    match s1 x, s2 x with
    | Some (v1,_), Some (v2,_) ⇒ v1 = v2
    | Some _, None ⇒ False
    | _, _ ⇒ True
    end
  else s1 x = s2 x.

Proposition no_lbls_same_values : ∀ P xs i s1 s2 h,
  no_lbls P xs = true → same_values s1 s2 xs → aden P (St i s1 h) → aden P (St i s2
h).
Proof.
induction xs; simpl; intros.
assert (s1 = s2).
extensionality x; specialize (H0 x); simpl in H0; auto.
subst; auto.
rewrite andb_true_iff in H; destruct H.
destruct (In_dec eq_nat_dec a xs).
apply IHxs with (s1 := s1); auto; intro x; specialize (H0 x).
simpl in H0.
destruct (eq_nat_dec a x); subst.
destruct (In_dec eq_nat_dec x xs); try contradiction; auto.
destruct (In_dec eq_nat_dec x xs); auto.
dup H0; specialize (H0 a).
simpl in H0; destruct (eq_nat_dec a a).
case_eq (s1 a); case_eq (s2 a); intros.
```

```
destruct v as [v2 l2]; destruct v0 as [v1 l1].
rewrite H4 in H0; rewrite H5 in H0; subst.
apply IHxs with (s1 := upd s1 a (v2,l2)); auto.
intro x; specialize (H3 x); simpl in H3; unfold upd.
destruct (In_dec eq_nat_dec x xs).
destruct (eq_nat_dec a x); destruct (eq_nat_dec x a); try subst; try contradiction; auto.
subst x; rewrite H5 in H3; auto.
destruct (eq_nat_dec a x); destruct (eq_nat_dec x a); try subst; try contradiction; auto.
subst x; auto.
apply aden_haslbl with (l := l1); auto.
destruct (haslbl P a); auto; inv H.
destruct v; rewrite H4 in H0; rewrite H5 in H0; inv H0.
destruct v as [v l]; apply IHxs with (s1 := upd s1 a (v,l)); auto.
intro x; specialize (H3 x); simpl in H3; unfold upd.
destruct (In_dec eq_nat_dec x xs).
destruct (eq_nat_dec a x); destruct (eq_nat_dec x a); try subst; try contradiction; auto.
subst x; rewrite H4; auto.
destruct (eq_nat_dec a x); destruct (eq_nat_dec x a); try subst; try contradiction; auto.
subst x; auto.
apply aden_upd_none; auto.
apply IHxs with (s1 := s1); auto; intro x; specialize (H3 x).
simpl in H3.
destruct (eq_nat_dec a x); subst.
destruct (In_dec eq_nat_dec x xs); try contradiction.
rewrite H4; rewrite H5; auto.
destruct (In_dec eq_nat_dec x xs); auto.
contradiction n0; auto.
Qed.

Proposition taint_vars_same_values : ∀ K s, same_values s (taint_vars K s) (modifies
K).
Proof.
intros; intro x; unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies K)); destruct (s x) as [[v l]|]; auto.
Qed.

Proposition no_lbls_taint_vars : ∀ P K i s h,
  no_lbls P (modifies K) = true → aden P (St i s h) → aden P (St i (taint_vars K s) h).
Proof.
intros; apply no_lbls_same_values with (xs := modifies K) (s1 := s); auto.
apply taint_vars_same_values.
Qed.

Proposition taint_vars_assert_inv : ∀ P K l l' i s h, gleq l l' = false →
  aden (taint_vars_assert P (modifies K) l l') (St i s h) → s = taint_vars K s.
```

```
Proof.
unfold taint_vars_assert; intros.
rewrite H in H0; simpl in H0; destruct H0.
extensionality x; unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies K)); auto.
apply aden_fold_inv with (x := x) in H1; auto.
simpl in H1.
destruct H1 as [vx [lx [H1]]].
rewrite H1.
destruct l; destruct l'; destruct lx; auto; inv H2; inv H.
Qed.

Proposition taint_vars_idempotent : ∀ K s, taint_vars K (taint_vars K s) = taint_vars K
s.
Proof.
unfold taint_vars; intros.
extensionality x; destruct (In_dec eq_nat_dec x (modifies K)); auto.
destruct (s x) as [[v l]|]; auto.
Qed.

Inductive judge : nat → context → assert → cmd → assert → Prop :=
| Judge_skip : ∀ pc, judge 0 pc Emp Skip Emp
| Judge_output : ∀ e, judge 0 Lo (LblExp e Lo ‘AND‘ Emp) (Output e) (LblExp e Lo ‘AND‘
Emp)
| Judge_assign : ∀ x e e' pc L, expvars e' x = false →
    judge 0 pc (BoolExp (Eq e e') ‘AND‘ LblExp e L ‘AND‘ Emp) (Assign x e)
                (BoolExp (Eq (Var x) e') ‘AND‘ LblEq x (Lub L pc) ‘AND‘ Emp)
| Judge_read : ∀ x e e1 e2 pc L1 L2, expvars e1 x = false → expvars e2 x = false →
    judge 0 pc (BoolExp (Eq (Var x) e1) ‘AND‘ LblExp e L1 ‘AND‘ Mapsto e e2 L2)
(Read x e)
                (BoolExp (Eq (Var x) e2) ‘AND‘ LblEq x (Lub (Lub L1 L2) pc) ‘AND‘
Mapsto (ereplace e x e1) e2 L2)
| Judge_write : ∀ e1 e2 pc L1 L2,
    judge 0 pc (LblExp e1 L1 ‘AND‘ LblExp e2 L2 ‘AND‘ Allocated e1) (Write e1 e2)
                (Mapsto e1 e2 (Lub (Lub L1 L2) pc))
| Judge_seq : ∀ N1 N2 P Q R C1 C2 pc, judge N1 pc P C1 Q → judge N2 pc Q C2 R →
judge (S (N1+N2)) pc P (Seq C1 C2) R
| Judge_if : ∀ N1 N2 P Q b C1 C2 pc (lt lf : glbl),
    implies P (BoolExp b ‘OR‘ BoolExp (Not b)) →
    implies (BoolExp b ‘AND‘ P) (LblBexp b lt) → implies (BoolExp (Not b) ‘AND‘ P)
(LblBexp b lf) →
    (gleq (glub lt lf) pc = false → no_lbls P (modifies [If b C1 C2]) = true) →
    judge N1 (glub lt pc) (BoolExp b ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) lt
pc) C1 Q →
```

judge N2 (glub lf pc) (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) lf pc) C2 Q →
judge (S (N1+N2)) pc P (If b C1 C2) Q
| *Judge_while* : ∀ *N P b C pc (l : glbl),*
implies P (LblBexp b l) → (gleq l pc = false → no_lbls P (modifies [While b C]) = true) →
judge N (glub l pc) (BoolExp b ‘AND‘ taint_vars_assert P (modifies [While b C]) l pc) C
(taint_vars_assert P (modifies [While b C]) l pc)
→
judge (S N) pc P (While b C) (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [While b C]) l pc)
| *Judge_conseq* : ∀ *N P P’ Q Q’ C pc, implies P’ P → implies Q Q’ → judge N pc P C Q → judge (S N) pc P’ C Q’*
| *Judge_conj* : ∀ *N1 N2 P1 P2 Q1 Q2 C pc, judge N1 pc P1 C Q1 → judge N2 pc P2 C Q2 →*
judge (S (N1+N2)) pc (P1 ‘AND‘ P2) C (Q1 ‘AND‘ Q2)
| *Judge_frame* : ∀ *N P Q R C pc, judge N pc P C Q → (∀ x, In x (modifies [C]) → vars R x = false) →*
*judge (S N) pc (P ** R) C (Q ** R).*

`Inductive` *sound* : `context` → `assert` → *cmd* → `assert` → `Prop` :=
| *Jden_hi* : ∀ *P C Q,*
(∀ *st, aden P st → hsafe (Cf st C [])) →*
(∀ *n st st’, aden P st → hstepn n (Cf st C []) (Cf st’ Skip []) → aden Q st’) →*
sound Hi P C Q
| *Jden_lo* : ∀ *P C Q,*
(∀ *st, aden P st → lsafe (Cf st C [])) →*
(∀ *n st st’ o, aden P st → lstepn n (Cf st C []) (Cf st’ Skip []) o → aden Q st’) →*
(∀ *n st1 st2 st1’ st2’ C’ K’ o1 o2, aden2 P st1 st2 →*
lstepn n (Cf st1 C []) (Cf st1’ C’ K’) o1 → lstepn n (Cf st2 C []) (Cf st2’ C’ K’) o2 →
diverge (Cf st1 C []) ∨ diverge (Cf st2 C []) ∨ side_condition C’ st1’ st2’) →
(∀ *n1 n2 st1 st2 st1’ st2’ o1 o2, aden2 P st1 st2 → side_condition C st1 st2 →*
lstepn n1 (Cf st1 C []) (Cf st1’ Skip []) o1 → lstepn n2 (Cf st2 C []) (Cf st2’ Skip []) o2 →
obs_eq st1’ st2’ ∧ o1 = o2) →
(∀ *n st1 st2 st1’ C’ K’ o1, aden2 P st1 st2 →*
lstepn n (Cf st1 C []) (Cf st1’ C’ K’) o1 →
diverge (Cf st1 C []) ∨ diverge (Cf st2 C []) ∨
∃ *st2’,* ∃ *o2, lstepn n (Cf st2 C []) (Cf st2’ C’ K’) o2) →*
(∀ *n1 n2 i1 s1 h1 i1’ s1’ h1’ i2 s2 h2 i2’ s2’ h2’ o1 o2 a,*
aden2 P (St i1 s1 h1) (St i2 s2 h2) →

```
    lstepn n1 (Cf (St i1 s1 h1) C []) (Cf (St i1' s1' h1') Skip []) o1 →
    lstepn n2 (Cf (St i2 s2 h2) C []) (Cf (St i2' s2' h2') Skip []) o2 →
    h1 a ≠ h1' a → (∃ v, h1' a = Some (v,Lo)) → h2 a ≠ None) →
  sound Lo P C Q.

Lemma soundness_skip : ∀ ct, sound ct Emp Skip Emp.
Proof.
destruct ct.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H0.
inv H1.
inv H2.
inv H0; auto.
inv H1.
right; right; inv H0; simpl; auto.
inv H2.
inv H1.
inv H2.
inv H; intuit.
inv H1.
inv H3.
right; right; inv H0.
∃ st2; ∃ []; apply LStep_zero.
inv H1.
inv H0.
contradiction H2; auto.
inv H4.

apply Jden_hi; intros.
unfold hsafe; intros.
inv H0.
inv H1.
inv H2.
inv H0; auto.
inv H1.
Qed.

Lemma soundness_output : ∀ e, sound Lo (LblExp e Lo `AND` Emp) (Output e) (LblExp e
Lo `AND` Emp).
Proof.
intros.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H0.
```

```
destruct st as [i s h].
destruct H as [[v]].
apply (Can_lstep _ (Cf (St i s h) Skip []) [v]); apply LStep_output; auto.
inv H2.
inv H3.
inv H1.
inv H0.
inv H0.
inv H1.
inv H2; auto.
inv H0.
right; right; inv H0; simpl; auto.
inv H2.
inv H3; simpl; auto.
inv H0.
inv H1.
inv H3.
inv H4.
inv H2.
inv H1.
inv H3.
inv H.
destruct H2.
dup (obs_eq_exp e _ _ _ _ _ _ H2).
rewrite H9 in H3; rewrite H8 in H3; destruct H3.
apply H4 in H3; subst; split; auto.
inv H1.
inv H1.
right; right; inv H0.
∃ st2; ∃ []; apply LStep_zero.
inv H1.
inv H2.
destruct H.
destruct H0.
destruct st2 as [i2 s2 h2].
destruct H0 as [[v2]].
∃ (St i2 s2 h2); ∃ ([v2]++[]); apply LStep_succ with (cf' := Cf (St i2 s2 h2) Skip []).
apply LStep_output; auto.
apply LStep_zero.
inv H0.
inv H0.
inv H4.
```

```
inv H5.
contradiction H2; auto.
inv H0.
Qed.

Lemma soundness_assign : ∀ e e' x L ct, expvars e' x = false →
  sound ct (BoolExp (Eq e e') ‘AND‘ LblExp e L ‘AND‘ Emp) (Assign x e)
           (BoolExp (Eq (Var x) e') ‘AND‘ LblEq x (Lub L ct) ‘AND‘ Emp).
Proof.
intros; destruct ct.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H1.
destruct st as [i s h]; destruct H0 as [[H0 [v]]].
apply (Can_lstep _ (Cf (St i (upd s x (v, lden L i)) h) Skip []) []).
apply LStep_assign; auto.
inv H3.
inv H4.
inv H2.
inv H1.
inv H1.
inv H2.
inv H3.
simpl in *.
decomp H0; subst.
destruct H4 as [v'].
rewrite H0 in H9; inv H9.
repeat (split; auto).
rewrite edenZ_upd; auto.
unfold upd.
destruct (eq_nat_dec x x); simpl.
destruct (edenZ e' i s).
assert (∃ l, eden e i s = Some (v,l)).
∃ (lden L i); auto.
rewrite ← edenZ_some in H1; rewrite H1 in H3; simpl in H3.
destruct (Z_eq_dec v z); auto.
destruct (edenZ e i s); inv H3.
contradiction n; auto.
∃ v; unfold upd.
destruct (eq_nat_dec x x).
destruct (lden L i); auto.
contradiction n; auto.
inv H1.
```

```
right; right; inv H1; simpl; auto.
inv H3.
inv H4; simpl; auto.
inv H1.
inv H2.
inv H4.
inv H5.
inv H3.
inv H2.
inv H4.
split; auto.
inv H0.
destruct H3.
dup (obs_eq_exp e _ _ _ _ _ _ H3).
rewrite H11 in H4; rewrite H10 in H4.
destruct H4; subst.
dup H3; inv H3.
simpl in *; subst; destruct H7.
repeat (split; auto).
intro y; simpl.
unfold upd; destruct (eq_nat_dec y x); subst; intuit.
apply H4.
inv H2.
inv H2.
right; right; inv H1.
∃ st2; ∃ []; apply LStep_zero.
inv H2.
inv H3.
destruct st2 as [i' s' h'].
inv H0.
destruct H2.
simpl in H0; decomp H0.
destruct H6 as [v']; ∃ (St i' (upd s' x (v',lden L i')) h'); ∃ ([]++[]).
apply LStep_succ with (cf' := Cf (St i' (upd s' x (v',lden L i')) h') Skip []).
apply LStep_assign; auto.
apply LStep_zero.
inv H1.
inv H1.
inv H5.
inv H6.
contradiction H3; auto.
inv H1.
```

```
apply Jden_hi; intros.
unfold hsafe; intros.
inv H1.
destruct st as [i s h]; destruct H0 as [[H0 [v]]].
apply (Can_hstep _ (Cf (St i (upd s x (v,Hi)) h) Skip [])).
apply HStep_assign with (l := lden L i); auto.
inv H3.
inv H4.
inv H2.
inv H1.
inv H1.
inv H2.
inv H3.
simpl in *.
decomp H0; subst.
destruct H4 as [v'].
rewrite H0 in H8; inv H8.
repeat (split; auto).
rewrite edenZ_upd; auto.
unfold upd.
destruct (eq_nat_dec x x); simpl.
destruct (edenZ e' i s).
assert (∃ l, eden e i s = Some (v,l)).
∃ (lden L i); auto.
rewrite ← edenZ_some in H1; rewrite H1 in H3; simpl in H3.
destruct (Z_eq_dec v z); auto.
destruct (edenZ e i s); inv H3.
contradiction n; auto.
∃ v; unfold upd.
destruct (eq_nat_dec x x).
destruct (lden L i); auto.
contradiction n; auto.
inv H1.
Qed.

Lemma soundness_read : ∀ ct e e1 e2 x L1 L2, expvars e1 x = false → expvars e2 x = false
→
  sound ct (BoolExp (Eq (Var x) e1) `AND` LblExp e L1 `AND` Mapsto e e2 L2)
    (Read x e) (BoolExp (Eq (Var x) e2) `AND` LblEq x (Lub (Lub L1 L2) ct)
                `AND` Mapsto (ereplace e x e1) e2 L2).
Proof.
destruct ct; intros.
apply Jden_lo; intros.
```

```
unfold lsafe; intros.
inv H2.
destruct st as [i s h].
destruct H1 as [[H1 [v]]].
destruct H4 as [v' [pf [H4 [v'' [H5]]]]].
apply (Can_lstep _ (Cf (St i (upd s x (v'', lden L1 i \_/ lden L2 i)) h) Skip []) []).
rewrite edenZ_some in H4; destruct H4 as [l'].
rewrite H4 in H2; inv H2.
apply LStep_read with (v1 := v) (pf := pf); auto.
destruct (eq_nat_dec (nat_of_Z v pf) (nat_of_Z v pf)); auto.
contradiction n; auto.
inv H4.
inv H5.
inv H3.
inv H2.
inv H2.
inv H3.
inv H4.
inv H1.
destruct H2 as [H2 [v]].
destruct H3 as [v' [pf' [H3 [v'' [H4]]]]].
rewrite edenZ_some in H3; destruct H3 as [l'].
rewrite H3 in H1; inv H1.
rewrite H3 in H10; inv H10.
rewrite (proof_irrelevance _ pf' pf) in H11.
destruct (eq_nat_dec (nat_of_Z v1 pf) (nat_of_Z v1 pf)); inv H11.
simpl; repeat split.
unfold upd at 1; destruct (eq_nat_dec x x); simpl.
rewrite edenZ_upd; auto; rewrite H4.
destruct (Z_eq_dec v2 v2); auto.
contradiction n; auto.
contradiction n; auto.
∃ v2; unfold upd; destruct (eq_nat_dec x x).
destruct (lden L1 i \_/ lden L2 i); auto.
contradiction n; auto.
∃ v1; ∃ pf; split.
rewrite edenZ_upd.
rewrite edenZ_ereplace.
rewrite edenZ_some; ∃ (lden L1 i); auto.
simpl in H2 ⊢ ×.
destruct (s x); destruct (edenZ e1 i s); simpl in H2 ⊢ *; try solve [inv H2].
destruct (Z_eq_dec (fst v) z); inv H2; auto.
```

```
apply ereplace_deletes; auto.
∃ v2; split.
rewrite edenZ_upd; auto.
rewrite (proof_irrelevance _ pf' pf); auto.
inv H2.
right; right; inv H2.
inv H3; simpl.
inv H1.
destruct H3.
destruct st1' as [i1 s1 h1]; destruct st2' as [i2 s2 h2]; simpl in *.
decomp H2; decomp H1.
destruct H5 as [v1 [pf1 [H5 [v1' [H10]]]]].
destruct H4 as [v2 [pf2 [H4 [v2' [H11]]]]].
apply edenZ_some in H5; destruct H5 as [l1].
apply edenZ_some in H4; destruct H4 as [l2].
destruct H7 as [v3]; destruct H9 as [v4].
rewrite H5 in H7; inv H7; rewrite H4 in H9; inv H9.
rewrite H5; rewrite H4.
destruct (Zneg_dec v3); try contradiction.
destruct (Zneg_dec v4); try contradiction.
rewrite (proof_irrelevance _ g pf1); rewrite (proof_irrelevance _ g0 pf2).
destruct (eq_nat_dec (nat_of_Z v3 pf1) (nat_of_Z v3 pf1)); intuit.
destruct (eq_nat_dec (nat_of_Z v4 pf2) (nat_of_Z v4 pf2)); intuit.
destruct H3.
simpl in H1; subst; auto.
inv H4.
inv H5; simpl; auto.
inv H2.
inv H3.
inv H5.
inv H6.
inv H4.
inv H3.
inv H5.
split; auto.
simpl in H2.
rewrite H12 in H2; rewrite H11 in H2.
destruct (Zneg_dec v1); try contradiction.
destruct (Zneg_dec v0); try contradiction.
rewrite (proof_irrelevance _ g pf) in H2; rewrite H13 in H2.
rewrite (proof_irrelevance _ g0 pf0) in H2; rewrite H14 in H2; subst.
destruct H1.
```

```
destruct H2.
dup H3; destruct H3.
destruct H5; repeat (split; auto).
intro y; simpl.
unfold upd; destruct (eq_nat_dec y x); subst.
dup (obs_eq_exp e _ _ _ _ _ _ H4).
rewrite H12 in H7; rewrite H11 in H7.
destruct H7; subst; intuition.
glub_simpl H7; subst.
specialize (H8 (refl_equal _)); subst.
rewrite (proof_irrelevance _ pf0 pf) in H14.
specialize (H6 (nat_of_Z v0 pf)); simpl in H6.
rewrite H13 in H6; rewrite H14 in H6; intuit.
apply H5.
inv H3.
inv H3.
right; right; inv H2.
∃ st2; ∃ []; apply LStep_zero.
inv H3.
inv H4.
inv H1.
destruct H3.
destruct st2 as [i' s' h']; simpl in H1.
decomp H1.
destruct H5 as [v1' [pf1 [H5 [v1'' [H12]]]]].
∃ (St i' (upd s' x (v1'', lden L1 i' \_/ lden L2 i')) h'); ∃ ([]++[]).
apply LStep_succ with (cf' := Cf (St i' (upd s' x (v1'', lden L1 i' \_/ lden L2 i')) h')
Skip []).
apply LStep_read with (v1 := v1') (pf := pf1).
apply edenZ_some in H5.
destruct H7 as [v']; destruct H5 as [l'].
rewrite H5 in H4; inv H4; auto.
subst; destruct (eq_nat_dec (nat_of_Z v1' pf1) (nat_of_Z v1' pf1)); auto.
contradiction n; auto.
apply LStep_zero.
inv H2.
inv H2.
inv H6.
inv H7.
contradiction H4; auto.
inv H2.
apply Jden_hi; intros.
```

```
unfold hsafe; intros.
inv H2.
destruct st as [i s h].
destruct H1 as [[H1 [v]]].
destruct H4 as [v' [pf [H4 [v'' [H5]]]]].
apply (Can_hstep _ (Cf (St i (upd s x (v'',Hi)) h) Skip [])).
rewrite edenZ_some in H4; destruct H4 as [l'].
rewrite H4 in H2; inv H2.
apply HStep_read with (v1 := v) (pf := pf) (l1 := lden L1 i) (l2 := lden L2 i); auto.
destruct (eq_nat_dec (nat_of_Z v pf) (nat_of_Z v pf)); auto.
contradiction n; auto.
inv H4.
inv H5.
inv H3.
inv H2.
inv H2.
inv H3.
inv H4.
inv H1.
destruct H2 as [H2 [v]].
destruct H3 as [v' [pf' [H3 [v'' [H4]]]]].
rewrite edenZ_some in H3; destruct H3 as [l'].
rewrite H3 in H1; inv H1.
rewrite H3 in H9; inv H9.
rewrite (proof_irrelevance _ pf' pf) in H10.
destruct (eq_nat_dec (nat_of_Z v1 pf) (nat_of_Z v1 pf)); inv H10.
simpl; repeat split.
unfold upd at 1; destruct (eq_nat_dec x x); simpl.
rewrite edenZ_upd; auto; rewrite H4.
destruct (Z_eq_dec v2 v2); auto.
contradiction n; auto.
contradiction n; auto.
∃ v2; unfold upd; destruct (eq_nat_dec x x).
destruct (lden L1 i \_/ lden L2 i); auto.
contradiction n; auto.
∃ v1; ∃ pf; split.
rewrite edenZ_upd.
rewrite edenZ_ereplace.
rewrite edenZ_some; ∃ (lden L1 i); auto.
simpl in H2 ⊢ ×.
destruct (s x); destruct (edenZ e1 i s); simpl in H2 ⊢ *; try solve [inv H2].
destruct (Z_eq_dec (fst v) z); inv H2; auto.
```

```
apply ereplace_deletes; auto.
∃ v2; split.
rewrite edenZ_upd; auto.
rewrite (proof_irrelevance _ pf' pf); auto.
inv H2.
Qed.

Lemma soundness_write : ∀ e1 e2 ct L1 L2,
  sound ct (LblExp e1 L1 `AND` LblExp e2 L2 `AND` Allocated e1) (Write e1 e2)
        (Mapsto e1 e2 (Lub (Lub L1 L2) ct)).
Proof.
destruct ct; intros.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H0.
destruct st as [i s h].
simpl in H; decomp H.
destruct H2 as [v' [pf [H2 [v'' [l'']]]]].
destruct H3 as [v1]; destruct H4 as [v2].
apply (Can_lstep _ (Cf (St i s (upd h (nat_of_Z v' pf) (v2, lden L1 i \_/ lden L2 i)))
Skip []) []).
rewrite edenZ_some in H2; destruct H2 as [l'].
rewrite H0 in H2; inv H2.
apply LStep_write; auto.
destruct (eq_nat_dec (nat_of_Z v' pf) (nat_of_Z v' pf)); auto; try discriminate.
inv H2.
inv H3.
inv H1.
inv H0.
inv H0.
inv H1.
inv H2.
simpl in H; decomp H.
destruct H2 as [v1']; destruct H3 as [v2'].
destruct H1 as [v' [pf' [H1 [v'' [l'']]]]].
rewrite edenZ_some in H1; destruct H1 as [l'].
rewrite H1 in H; inv H.
rewrite H0 in H9; inv H9.
rewrite H1 in H8; inv H8.
simpl.
∃ v1; ∃ pf; split.
rewrite edenZ_some; ∃ (lden L1 i); auto.
∃ v2; split.
```

```
rewrite edenZ_some; ∃ (lden L2 i); auto.
unfold upd; rewrite (proof_irrelevance _ pf' pf); extensionality n.
destruct (eq_nat_dec n (nat_of_Z v1 pf)); auto.
destruct (lden L1 i \_/ lden L2 i); auto.
inv H0.
right; right; inv H0.
inv H1; simpl; auto.
inv H2.
inv H3; simpl; auto.
inv H0.
inv H1.
inv H3.
inv H4.
inv H2.
inv H1.
inv H3.
split; auto.
destruct H.
destruct H1.
dup H2; destruct H2.
destruct H4; repeat (split; auto).
intro n; simpl.
dup (obs_eq_exp e1 _ _ _ _ _ _ H3).
dup (obs_eq_exp e2 _ _ _ _ _ _ H3).
rewrite H10 in H6; rewrite H9 in H6.
rewrite H13 in H7; rewrite H11 in H7.
destruct H6; destruct H7; subst.
simpl in H2; unfold upd; destruct (eq_nat_dec n (nat_of_Z v1 pf)); subst.
destruct l0.
specialize (H8 (refl_equal _)); subst.
rewrite (proof_irrelevance _ pf0 pf).
destruct (eq_nat_dec (nat_of_Z v0 pf) (nat_of_Z v0 pf)); auto.
contradiction n; auto.
destruct (eq_nat_dec (nat_of_Z v1 pf) (nat_of_Z v0 pf0)); intros.
inv H2.
destruct (h0 (nat_of_Z v1 pf)) as [[v l]|]; auto; intros.
inv H2.
specialize (H5 n); simpl in H5.
destruct (h n) as [[v l]|]; auto.
destruct l0.
specialize (H8 (refl_equal _)); subst.
destruct (eq_nat_dec n (nat_of_Z v0 pf0)); subst.
```

```
contradiction n0; rewrite (proof_irrelevance _ pf0 pf); auto.
destruct (h0 n) as [[v' l']|]; auto.
destruct (eq_nat_dec n (nat_of_Z v0 pf0)); subst; auto.
intros.
inv H6.
inv H1.
inv H1.
right; right; inv H0.
∃ st2; ∃ []; apply LStep_zero.
inv H1.
inv H2.
inv H.
destruct H1.
destruct st2 as [i' s' h']; simpl in H.
decomp H.
destruct H3 as [v1' [pf1 [H3 [v1'' [l1'']]]]].
destruct H4 as [v3]; destruct H5 as [v4].
∃ (St i' s' (upd h' (nat_of_Z v1' pf1) (v4, lden L1 i' \_/ lden L2 i'))); ∃ ([]++[]).
apply LStep_succ with (cf' := Cf (St i' s' (upd h' (nat_of_Z v1' pf1) (v4, lden L1 i' \_/
lden L2 i'))) Skip []).
apply LStep_write; auto.
rewrite edenZ_some in H3; destruct H3 as [l''].
rewrite H2 in H3; inv H3; auto.
subst.
destruct (eq_nat_dec (nat_of_Z v1' pf1) (nat_of_Z v1' pf1)); auto; try discriminate.
apply LStep_zero.
inv H0.
inv H0; inv H1.
inv H4; inv H0.
inv H5.
inv H6.
unfold upd in H2, H3.
destruct H3 as [v]; destruct (eq_nat_dec a (nat_of_Z v1 pf)).
inv H0.
glub_simpl H4; subst.
inv H.
destruct H1.
dup (obs_eq_exp e1 _ _ _ _ _ _ H1).
rewrite H13 in H3; rewrite H14 in H3; destruct H3.
specialize (H4 (refl_equal _)); subst.
rewrite (proof_irrelevance _ pf pf0); auto.
contradiction H2; auto.
```

```
inv H0.
inv H0.
apply Jden_hi; intros.
unfold hsafe; intros.
inv H0.
destruct st as [i s h].
simpl in H; decomp H.
destruct H2 as [v' [pf [H2 [v'' [l'']]]]].
destruct H3 as [v1]; destruct H4 as [v2].
apply (Can_hstep _ (Cf (St i s (upd h (nat_of_Z v' pf) (v2, Hi))) Skip [])).
rewrite edenZ_some in H2; destruct H2 as [l'].
rewrite H0 in H2; inv H2.
apply HStep_write with (l1 := lden L1 i) (l2 := lden L2 i); auto.
destruct (eq_nat_dec (nat_of_Z v' pf) (nat_of_Z v' pf)); auto; try discriminate.
inv H2.
inv H3.
inv H1.
inv H0.
inv H0.
inv H1.
inv H2.
simpl in H; decomp H.
destruct H2 as [v1']; destruct H3 as [v2'].
destruct H1 as [v' [pf' [H1 [v'' [l'']]]]].
rewrite edenZ_some in H1; destruct H1 as [l'].
rewrite H1 in H; inv H.
rewrite H0 in H8; inv H8.
rewrite H1 in H7; inv H7.
simpl.
∃ v1; ∃ pf; split.
rewrite edenZ_some; ∃ (lden L1 i); auto.
∃ v2; split.
rewrite edenZ_some; ∃ (lden L2 i); auto.
unfold upd; rewrite (proof_irrelevance _ pf' pf); extensionality n.
destruct (eq_nat_dec n (nat_of_Z v1 pf)); auto.
destruct (lden L1 i \_/ lden L2 i); auto.
inv H0.
Qed.

Lemma soundness_seq : ∀ N1 N2 P Q R C1 C2 ct,
  (∀ y : nat, y < S (N1 + N2) →
    ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
    judge y ct P C Q → sound ct P C Q) →
```

```
  judge N1 ct P C1 Q → judge N2 ct Q C2 R → sound ct P (Seq C1 C2) R.
Proof.
intros.
rename H1 into H2; rename H0 into H1; destruct ct.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H3.
apply (Can_lstep _ (Cf st C1 [C2]) []); apply LStep_seq.
inv H5.
change (lstepn n0 (Cf st C1 ([]++[C2])) cf' o') in H6.
destruct cf' as [st' C' K']; apply lstep_trans_inv in H6.
destruct H6.
destruct H3 as [K [H3]]; subst.
apply H in H1; try omega; inv H1.
case_eq (halt_config (Cf st' C' K)); intros.
destruct C'; destruct K; inv H1.
apply (Can_lstep _ (Cf st' C2 []) []); apply LStep_skip.
specialize (H5 st H0 _ _ _ H3 H1).
inv H5.
destruct cf' as [st'' C'' K''].
apply lstep_extend with (K0 := [C2]) in H11.
apply (Can_lstep _ (Cf st'' C'' (K''++[C2])) o); auto.
destruct H3 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
apply H in H1; try omega; apply H in H2; try omega; inv H1; inv H2.
apply H6 in H5; auto.
apply H1 in H5.
inv H7.
apply (Can_lstep _ (Cf st' C2 []) []); apply LStep_skip.
inv H2.
apply H5 in H17; auto.
inv H3.
inv H4.
change (lstepn n0 (Cf st C1 ([]++[C2])) (Cf st' Skip []) o') in H5.
apply lstep_trans_inv in H5; destruct H5.
destruct H3 as [K [H3]].
apply sym_eq in H4; apply app_eq_nil in H4; destruct H4.
inv H5.
destruct H3 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
apply H in H1; try omega; apply H in H2; try omega; inv H1; inv H2.
inv H6.
inv H2.
apply H5 in H4; auto; apply H11 in H16; auto.
```

```
inv H3; simpl; auto.
inv H5.
inv H4.
inv H5.
change (lstepn n0 (Cf st1 C1 ([]++[C2])) (Cf st1' C' K') o') in H6.
change (lstepn n0 (Cf st2 C1 ([]++[C2])) (Cf st2' C' K') o'0) in H7.
apply lstep_trans_inv in H6; apply lstep_trans_inv in H7.
destruct H6.
destruct H7.
destruct H3 as [K1 [H3]]; destruct H4 as [K2 [H4]]; subst.
apply app_cancel_r in H6; subst.
apply H in H1; try omega; inv H1.
dup (H7 _ _ _ _ _ _ _ _ _ H0 H3 H4).
decomp H1; auto.
left; apply diverge_seq1; auto.
right; left; apply diverge_seq1; auto.
destruct H3 as [K1 [H3]].
destruct H4 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H4; subst.
apply H in H1; try omega; inv H1.
apply H10 with (st2 := st1) in H6.
decomp H6.
right; left; apply diverge_seq1; auto.
left; apply diverge_seq1; auto.
destruct H12 as [st2'' [o2']].
apply lstep_trans_inv' in H3.
destruct H3 as [cf'' [o1'' [o2'']]]; decomp H3.
destruct (lstepn_det _ _ _ _ _ _ H6 H1); subst.
inv H13; simpl; auto.
inv H3.
inv H0.
destruct H12; split; auto; split; auto.
apply obs_eq_sym; auto.
destruct H7.
destruct H4 as [K1 [H5]].
destruct H3 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
apply H in H1; try omega; inv H1.
apply H10 with (st2 := st2) in H6; auto.
decomp H6.
left; apply diverge_seq1; auto.
right; left; apply diverge_seq1; auto.
destruct H12 as [st2'' [o2']].
apply lstep_trans_inv' in H5.
```

```
destruct H5 as [cf'' [o1'' [o2'']]]; decomp H5.
destruct (lstepn_det _ _ _ _ _ _ H6 H1); subst.
inv H13; simpl; auto.
inv H5.
destruct H3 as [st1'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
destruct H4 as [st2'' [n3 [n4 [o3 [o4]]]]].
decomp H3; subst.
apply H in H1; try omega; inv H1.
apply H in H2; try omega; inv H2.
assert (n1 = n3).
dup H5; apply H12 with (st2 := st2) in H5; auto.
decomp H5.
apply (False_ind _ (diverge_halt _ _ _ _ H19 H2)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H4)).
destruct H20 as [st2''' [o2']].
apply (lstepn_det_term _ _ _ _ _ _ _ H5 H4).
assert (n2 = n4); subst; try omega.
destruct n4.
inv H8; simpl; auto.
inv H8.
inv H19.
inv H7.
inv H8.
assert (aden2 Q st1'' st2'').
dup H0; inv H0.
destruct H8; split; try split.
apply (H9 _ _ _ _ H7 H5).
apply (H9 _ _ _ _ H0 H4).
apply (H11 n3 n3 st1 st2 st1'' st2'' o1 o3); auto.
repeat (split; auto).
decomp (H10 0 st1 st2 st1 st2 C1 [] [] [] H2 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H21 H5)).
apply (False_ind _ (diverge_halt _ _ _ _ H22 H4)).
decomp (H15 n4 st1'' st2'' st1' st2' C' K' o'0 o' H2 H19 H20); auto.
left; apply diverge_seq2 with (st' := st1'') (n := n3) (o := o1); auto.
right; left; apply diverge_seq2 with (st' := st2'') (n := n3) (o := o3); auto.
inv H4.
inv H6.
inv H5.
inv H4.
change (lstepn n (Cf st1 C1 ([]++[C2])) (Cf st1' Skip []) o') in H7.
change (lstepn n0 (Cf st2 C1 ([]++[C2])) (Cf st2' Skip []) o'0) in H6.
```

```
apply lstep_trans_inv in H7; apply lstep_trans_inv in H6.
destruct H7.
destruct H4 as [K1 [H4]].
apply f_equal with (f := fun l ⇒ length l) in H5; simpl in H5.
destruct K1; inv H5.
destruct H6.
destruct H5 as [K2 [H5]].
apply f_equal with (f := fun l ⇒ length l) in H6; simpl in H6.
destruct K2; inv H6.
destruct H4 as [st1'' [n1 [n2 [o1 [o2]]]]]; decomp H4; subst.
destruct H5 as [st2'' [n3 [n4 [o3 [o4]]]]].
decomp H4; subst.
apply H in H1; try omega; inv H1.
apply H in H2; try omega; inv H2.
assert (n1 = n3).
dup H6; apply H12 with (st2 := st2) in H6; auto.
decomp H6.
apply (False_ind _ (diverge_halt _ _ _ _ H19 H2)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H5)).
destruct H20 as [st [o]].
apply (lstepn_det_term _ _ _ _ _ _ _ H6 H5).
subst.
inv H8.
inv H2.
inv H9.
inv H2.
assert (obs_eq st1' st2' ∧ o' = o'0).
apply (H16 n n0 st1'' st2'' st1' st2' o' o'0); auto.
dup H0; inv H0.
destruct H20; split; try split.
apply H7 in H6; auto.
apply H7 in H5; auto.
apply (H11 n3 n3 st1 st2 st1'' st2'' o1 o3); auto.
repeat (split; auto).
decomp (H10 0 st1 st2 st1 st2 C1 [] [] [] H2 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H21 H6)).
apply (False_ind _ (diverge_halt _ _ _ _ H22 H5)).
assert (aden2 Q st1'' st2'').
dup H0; inv H0.
destruct H20; split; try split.
apply (H7 _ _ _ _ H9 H6).
apply (H7 _ _ _ _ H0 H5).
```

```
apply (H11 n3 n3 st1 st2 st1'' st2'' o1 o3); auto.
decomp (H10 0 st1 st2 st1 st2 C1 [] [] [] H2 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H21 H6)).
apply (False_ind _ (diverge_halt _ _ _ _ H22 H5)).
decomp (H15 0 st1'' st2'' st1'' st2'' C2 [] [] [] H2 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H9 H19)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H8)).
destruct H2; subst; split; auto.
assert (o1 = o3).
apply (H11 n3 n3 st1 st2 st1'' st2'' o1 o3); auto.
decomp (H10 0 st1 st2 st1 st2 C1 [] [] [] H0 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H9 H6)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H5)).
subst; auto.
inv H3.
right; right; ∃ st2; ∃ []; apply LStep_zero.
inv H4.
change (lstepn n0 (Cf st1 C1 ([]++[C2])) (Cf st1' C' K') o') in H5.
apply lstep_trans_inv in H5; destruct H5.
destruct H3 as [K'' [H3]]; subst.
apply H in H1; try omega; inv H1.
apply H8 with (st2 := st2) in H3; auto.
decomp H3.
left; apply diverge_seq1; auto.
right; left; apply diverge_seq1; auto.
right; right; destruct H10 as [st2' [o2]]; ∃ st2'; ∃ ([]++o2).
apply LStep_succ with (cf' := Cf st2 C1 [C2]); auto.
apply LStep_seq.
apply lstepn_extend with (K0 := [C2]) in H1; auto.
destruct H3 as [st1'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
apply H in H1; apply H in H2; try omega; inv H1; inv H2.
dup H4; apply H9 with (st2 := st2) in H4; auto.
decomp H4.
left; apply diverge_seq1; auto.
right; left; apply diverge_seq1; auto.
destruct H17 as [st2'' [o2']].
inv H6.
right; right; ∃ st2''; ∃ ([]++o2').
apply LStep_succ with (cf' := Cf st2 C1 [C2]).
apply LStep_seq.
assert (n1 + 0 = n1); try omega.
rewrite H6; apply lstepn_extend with (K0 := [C2]) in H4; auto.
```

```
inv H16.
apply H14 with (st2 := st2'') in H17.
decomp H17.
left; apply diverge_seq2 with (st' := st1'') (n := n1) (o := o1); auto.
right; left; apply diverge_seq2 with (st' := st2'') (n := n1) (o := o2'); auto.
right; right; destruct H16 as [st2' [o2'']].
∃ st2'; ∃ ([]++o2'++[]++o2'').
apply LStep_succ with (cf' := Cf st2 C1 [C2]).
apply LStep_seq.
apply lstep_trans with (cf2 := Cf st2'' Skip [C2]).
apply lstepn_extend with (K0 := [C2]) in H4; auto.
apply LStep_succ with (cf' := Cf st2'' C2 []); auto.
apply LStep_skip.
dup H0; inv H0.
destruct H18; split; try split.
apply H5 in H2; auto.
apply H5 in H4; auto.
apply (H8 n1 n1 st1 st2 st1'' st2'' o1 o2'); auto.
decomp (H7 0 st1 st2 st1 st2 C1 [] [] [] H6 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H19 H2)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H4)).
apply H in H1; try omega; inv H1.
apply H in H2; try omega; inv H2.
inv H3; inv H4.
inv H2; inv H3.
change (lstepn n (Cf (St i1 s1 h1) C1 ([]++[C2])) (Cf (St i1' s1' h1') Skip []) o') in H18.
change (lstepn n0 (Cf (St i2 s2 h2) C1 ([]++[C2])) (Cf (St i2' s2' h2') Skip []) o'0) in
H19.
apply lstep_trans_inv in H18; apply lstep_trans_inv in H19.
destruct H18.
destruct H2 as [K [H2]].
apply f_equal with (f := fun l ⇒ length l) in H3; simpl in H3.
destruct K; inv H3.
destruct H19.
destruct H3 as [K [H3]].
apply f_equal with (f := fun l ⇒ length l) in H4; simpl in H4.
destruct K; inv H4.
destruct H2 as [[i1'' s1'' h1''] [n1 [n2 [o1 [o2]]]]]; decomp H2.
destruct H3 as [[i2'' s2'' h2''] [n1' [n2' [o1' [o2']]]]].
decomp H2; subst.
inv H19; inv H22.
inv H2; inv H19.
```

```
destruct (opt_eq_dec val_eq_dec (h1 a) (h1'' a)).
destruct (opt_eq_dec val_eq_dec (h1'' a) (h1' a)).
rewrite e0 in e; contradiction.
apply (H17 _ n0 _ _ _ _ _ _ i2'' s2'' h2'' i2' s2' h2' _ o'0 a) in H18; auto.
intro; apply lstepn_nonincreasing with (a := a) in H3; auto.
split; try split.
destruct H0; apply H8 in H4; intuit.
destruct H0; apply H8 in H3; intuit.
decomp (H9 _ _ _ _ _ _ _ _ _ H0 (LStep_zero _) (LStep_zero _)).
apply (False_ind _ (diverge_halt _ _ _ _ H2 H4)).
apply (False_ind _ (diverge_halt _ _ _ _ H19 H3)).
destruct (H10 _ _ _ _ _ _ _ _ H0 H19 H4 H3); auto.
destruct (opt_eq_dec val_eq_dec (h1'' a) (h1' a)).
apply (H12 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H0 H4 H3 n2).
rewrite e; auto.
apply (H17 _ n0 _ _ _ _ _ _ i2'' s2'' h2'' i2' s2' h2' _ o'0 a) in H18; auto.
intro; apply lstepn_nonincreasing with (a := a) in H3; auto.
split; try split.
destruct H0; apply H8 in H4; intuit.
destruct H0; apply H8 in H3; intuit.
decomp (H9 _ _ _ _ _ _ _ _ _ H0 (LStep_zero _) (LStep_zero _)).
apply (False_ind _ (diverge_halt _ _ _ _ H2 H4)).
apply (False_ind _ (diverge_halt _ _ _ _ H19 H3)).
destruct (H10 _ _ _ _ _ _ _ _ H0 H19 H4 H3); auto.

apply Jden_hi; intros.
unfold hsafe; intros.
inv H3.
apply (Can_hstep _ (Cf st C1 [C2])); apply HStep_seq.
inv H5.
change (hstepn n0 (Cf st C1 ([]++[C2])) cf') in H6.
destruct cf' as [st' C' K']; apply hstep_trans_inv in H6.
destruct H6.
destruct H3 as [K [H3]]; subst.
apply H in H1; try omega; inv H1.
case_eq (halt_config (Cf st' C' K)); intros.
destruct C'; destruct K; inv H1.
apply (Can_hstep _ (Cf st' C2 [])); apply HStep_skip.
specialize (H5 st H0 _ _ H3 H1).
inv H5.
destruct cf' as [st'' C'' K''].
apply hstep_extend with (K0 := [C2]) in H7.
apply (Can_hstep _ (Cf st'' C'' (K''++[C2]))); auto.
```

```
destruct H3 as [st'' [n1 [n2]]]; decomp H3; subst.
apply H in H1; try omega; apply H in H2; try omega; inv H1; inv H2.
apply H6 in H5; auto.
apply H1 in H5.
inv H7.
apply (Can_hstep _ (Cf st' C2 [])); apply HStep_skip.
inv H2.
apply H5 in H9; auto.
inv H3.
inv H4.
change (hstepn n0 (Cf st C1 ([]++[C2])) (Cf st' Skip [])) in H5.
apply hstep_trans_inv in H5; destruct H5.
destruct H3 as [K [H3]].
apply sym_eq in H4; apply app_eq_nil in H4; destruct H4.
inv H5.
destruct H3 as [st'' [n1 [n2]]]; decomp H3; subst.
apply H in H1; try omega; apply H in H2; try omega; inv H1; inv H2.
inv H6.
inv H2.
apply H5 in H4; auto; apply H7 in H8; auto.
Qed.

Lemma soundness_if : ∀ N1 N2 P Q b C1 C2 ct (lt lf : glbl),
  (∀ y : nat, y < S (N1 + N2) →
    ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
    judge y ct P C Q → sound ct P C Q) →
  implies P (BoolExp b ‘OR‘ BoolExp (Not b)) →
  implies (BoolExp b ‘AND‘ P) (LblBexp b lt) → implies (BoolExp (Not b) ‘AND‘ P)
(LblBexp b lf) →
  (gleq (glub lt lf) ct = false → no_lbls P (modifies [If b C1 C2]) = true) →
  judge N1 (glub lt ct) (BoolExp b ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) lt
ct) C1 Q →
  judge N2 (glub lf ct) (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [If b C1
C2]) lf ct) C2 Q →
  sound ct P (If b C1 C2) Q.
Proof.
intros.
rename H5 into H6; rename H4 into H5; rename H3 into H4;
 rename H2 into H3; rename H1 into H2; rename H0 into H1; destruct ct.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H7.
dup H0; apply H1 in H0.
```

```
destruct st as [i s h]; simpl in H0; destruct H0.
assert (aden (LblBexp b lt) (St i s h)).
apply H2; simpl; split; auto.
destruct H9 as [v].
rewrite bdenZ_some in H0; destruct H0 as [l].
rewrite H9 in H0; inv H0.
destruct l.
apply (Can_lstep _ (Cf (St i s h) C1 []) []).
apply LStep_if_true; auto.
destruct (dvg_ex_mid (taint_vars_cf (Cf (St i s h) (If b C1 C2) []))).
apply (Can_lstep _ (Cf (St i s h) (If b C1 C2) []) []).
apply LStep_if_hi_dvg with (v := true); auto.
unfold hsafe; intros.
apply H in H5; try omega; inv H5.
inv H10.
apply (Can_hstep _ (Cf (St i (taint_vars [If b C1 C2] s) h) C1 [])).
apply HStep_if_true with (l := Hi).
apply bden_taint_vars with (K := [If b C1 C2]) in H9; destruct H9 as [l [H9]].
destruct l; inv H5; auto.
inv H5.
apply H12 in H14; intuit.
simpl; split; try split.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; intros.
simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]|].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
apply bden_taint_vars with (K := [If b C1 C2]) in H9.
destruct H9 as [l' [H9]].
rewrite H9 in H22; inv H22.
destruct H0 as [n [st]].
apply (Can_lstep _ (Cf st Skip []) []).
apply LStep_if_hi with (v := true) (n := n); auto.
unfold hsafe; intros.
apply H in H5; try omega; inv H5.
inv H10.
apply (Can_hstep _ (Cf (St i (taint_vars [If b C1 C2] s) h) C1 [])).
apply HStep_if_true with (l := Hi).
```

```
apply bden_taint_vars with (K := [If b C1 C2]) in H9; destruct H9 as [l [H9]].
destruct l; inv H5; auto.
inv H5.
apply H12 in H14; intuit.
simpl; split; try split.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; intros.
simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]|].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
apply bden_taint_vars with (K := [If b C1 C2]) in H9.
destruct H9 as [l' [H9]].
rewrite H9 in H22; inv H22.
case_eq (bdenZ b i s); intros.
rewrite H9 in H0; inv H0.
destruct b0; inv H11.
apply bdenZ_some in H9; destruct H9 as [l]; destruct l.
apply (Can_lstep _ (Cf (St i s h) C2 []) []).
apply LStep_if_false; auto.
destruct (dvg_ex_mid (taint_vars_cf (Cf (St i s h) (If b C1 C2) []))).
apply (Can_lstep _ (Cf (St i s h) (If b C1 C2) []) []).
apply LStep_if_hi_dvg with (v := false); auto.
unfold hsafe; intros.
apply H in H6; try omega; inv H6.
inv H10.
apply (Can_hstep _ (Cf (St i (taint_vars [If b C1 C2] s) h) C2 [])).
apply HStep_if_false with (l := Hi).
apply bden_taint_vars with (K := [If b C1 C2]) in H0; destruct H0 as [l [H0]].
destruct l; inv H6; auto.
inv H6.
apply bden_taint_vars with (K := [If b C1 C2]) in H0.
destruct H0 as [l' [H0]].
rewrite H0 in H23; inv H23.
apply H13 in H15; intuit.
destruct lf; inv H12; simpl; split; try split.
assert (∃ l, bden b i (taint_vars [If b C1 C2] s) = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
```

```
apply no_lbls_taint_vars; auto.
apply H4.
destruct lt; auto.
apply aden_fold; intros.
simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]|].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
destruct lf; inv H12.
specialize (H3 (St i s h)); simpl in H3.
assert (∃ v, bden b i s = Some (v,Lo)).
apply H3; split; auto.
case_eq (bdenZ b i s); intros.
destruct b0; simpl; auto.
apply bdenZ_some in H6; destruct H6 as [l].
rewrite H6 in H0; inv H0.
apply bdenZ_none in H6; rewrite H6 in H0; inv H0.
destruct H6 as [v]; rewrite H6 in H0; inv H0.
destruct H9 as [n [st]].
apply (Can_lstep _ (Cf st Skip []) []).
apply LStep_if_hi with (v := false) (n := n); auto.
unfold hsafe; intros.
apply H in H6; try omega; inv H6.
inv H10.
apply (Can_hstep _ (Cf (St i (taint_vars [If b C1 C2] s) h) C2 [])).
apply HStep_if_false with (l := Hi).
apply bden_taint_vars with (K := [If b C1 C2]) in H0; destruct H0 as [l [H0]].
destruct l; inv H6; auto.
inv H6.
apply bden_taint_vars with (K := [If b C1 C2]) in H0; destruct H0 as [l' [H0]].
rewrite H23 in H0; inv H0.
apply H13 in H15; intuit.
destruct lf; inv H12.
simpl; split; try split.
assert (∃ l, bden b i (taint_vars [If b C1 C2] s) = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
apply no_lbls_taint_vars; auto.
apply H4; destruct lt; auto.
apply aden_fold; intros.
```

```
simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]||].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
destruct lf; inv H12.
specialize (H3 (St i s h)); simpl in H3.
assert (∃ v, bden b i s = Some (v,Lo)).
apply H3; split; auto.
case_eq (bdenZ b i s); intros.
destruct b0; simpl; auto.
apply bdenZ_some in H6; destruct H6 as [l].
rewrite H6 in H0; inv H0.
apply bdenZ_none in H6; rewrite H6 in H0; inv H0.
destruct H6 as [v]; rewrite H6 in H0; inv H0.
rewrite H9 in H0; inv H0.
inv H9.
apply H in H5; try omega; inv H5.
destruct lt; inv H7.
specialize (H2 (St i s h)); simpl in H2.
assert (∃ v, bden b i s = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v]; rewrite H5 in H17; inv H17.
apply H9 in H10; intuit.
destruct lt; inv H7.
unfold taint_vars_assert; simpl; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
apply H in H6; try omega; inv H6.
destruct lf; inv H7.
specialize (H3 (St i s h)); simpl in H3.
assert (∃ v, bden b i s = Some (v,Hi)).
apply H3; split; auto.
case_eq (bdenZ b i s); intros.
destruct b0; auto.
apply bdenZ_some in H6; destruct H6 as [l].
rewrite H6 in H17; inv H17.
apply bdenZ_none in H6; rewrite H6 in H17; inv H17.
destruct H6 as [v]; rewrite H6 in H17; inv H17.
apply H9 in H10; intuit.
destruct lf; inv H7.
```

```
unfold taint_vars_assert; simpl; split; auto.
case_eq (bdenZ b i s); intros.
destruct b0; auto.
apply bdenZ_some in H6; destruct H6 as [l].
rewrite H6 in H17; inv H17.
apply bdenZ_none in H6; rewrite H6 in H17; inv H17.
inv H10.
inv H8.
inv H7.
generalize cf' o' H8 H10; clear cf' o' H8 H10.
induction n0; intros.
inv H10.
apply (Can_lstep _ (Cf (St i s h) (If b C1 C2) []) []).
apply LStep_if_hi_dvg with (v := v); auto.
inv H10.
inv H9.
rewrite H22 in H17; inv H17.
rewrite H22 in H17; inv H17.
inv H11.
inv H8.
inv H7.
apply IHn0 with (o' := o'0); auto.
inv H7.
inv H8.
apply H in H5; try omega; inv H5.
destruct lt; inv H7.
specialize (H2 (St i s h)); simpl in H2.
assert (∃ v, bden b i s = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v]; rewrite H5 in H16; inv H16.
apply H10 in H9; auto.
destruct lt; inv H7.
simpl; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
apply H in H6; try omega; inv H6.
destruct lf; inv H7.
specialize (H3 (St i s h)); simpl in H3.
assert (∃ v, bden b i s = Some (v,Hi)).
apply H3; split; auto.
case_eq (bdenZ b i s); intros.
destruct b0; auto.
```

```
rewrite bdenZ_some in H6; destruct H6 as [l].
rewrite H6 in H16; inv H16.
rewrite bdenZ_none in H6; rewrite H6 in H16; inv H16.
destruct H6 as [v]; rewrite H6 in H16; inv H16.
apply H10 in H9; auto.
destruct lf; inv H7.
simpl; split; auto.
assert (∃ l, bden b i s = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
inv H9.
inv H18.
inv H7.
apply H in H5; try omega; inv H5.
apply H10 in H8; auto.
destruct lt; inv H7.
simpl; split; try split.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]|].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
destruct lt; inv H7.
dup H16; apply bden_taint_vars with (K := [If b C1 C2]) in H16.
destruct H16 as [l' [H16]].
rewrite H16 in H19; inv H19.
destruct l; inv H7.
specialize (H2 (St i s h)); simpl in H2.
assert (∃ v, bden b i s = Some (v,Lo)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Hi; auto.
destruct H7 as [v].
rewrite H7 in H5; inv H5.
apply H in H6; try omega; inv H6.
apply H10 in H8; auto.
destruct lf; inv H7.
simpl; split; try split.
assert (∃ l, bden b i (taint_vars [If b C1 C2] s) = Some (false,l)).
∃ l; auto.
```

```
rewrite ← bdenZ_some in H6; rewrite H6; auto.
apply no_lbls_taint_vars; auto.
apply H4; destruct lt; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s x) as [[v1 l1]||].
∃ v1; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
destruct lf; inv H7.
dup H16; apply bden_taint_vars with (K := [If b C1 C2]) in H16.
destruct H16 as [l' [H16]].
rewrite H16 in H19; inv H19.
destruct l; inv H7.
specialize (H3 (St i s h)); simpl in H3.
assert (∃ v, bden b i s = Some (v,Lo)).
apply H3; split; auto.
assert (∃ l, bden b i s = Some (false,l)).
∃ Hi; auto.
rewrite ← bdenZ_some in H7; rewrite H7; auto.
destruct H7 as [v].
rewrite H7 in H6; inv H6.
inv H7.
generalize st' o' H9 H18; clear st' o' H9 H18.
induction n0; intros.
inv H9.
inv H9.
inv H8.
rewrite H21 in H16; inv H16.
rewrite H21 in H16; inv H16.
contradiction (H18 n st'0).
apply IHn0 with (o' := o'0); auto.
inv H7; simpl; auto.
inv H8.
inv H0.
destruct H8; destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2].
dup (obs_eq_bexp b _ _ _ _ _ _ H8).
inv H9.
inv H11.
apply H in H5; try omega; inv H5.
destruct lt; inv H9.
specialize (H2 (St i1 s1 h1)); simpl in H2.
```

```
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v]; rewrite H5 in H23; inv H23.
assert (aden2 (BoolExp b ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) lt Lo) (St i1
s1 h1) (St i2 s2 h2)).
destruct lt; inv H9; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
decomp (H15 _ _ _ _ _ _ _ _ _ H5 H10 H12); auto.
left; intro n.
destruct n.
∃ (Cf (St i1 s1 h1) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H19 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C1 []); auto.
apply LStep_if_true; auto.
right; left; intro n.
destruct n.
∃ (Cf (St i2 s2 h2) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H20 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C1 []); auto.
apply LStep_if_true; auto.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
specialize (H11 (refl_equal _)); inv H11.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
inv H9.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
inv H9.
inv H11.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
specialize (H11 (refl_equal _)); inv H11.
apply H in H6; try omega; inv H6.
destruct lf; inv H9.
specialize (H3 (St i1 s1 h1)); simpl in H3.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H3; split; auto.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct H6 as [v]; rewrite H6 in H23; inv H23.
assert (aden2 (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) lf Lo)
(St i1 s1 h1) (St i2 s2 h2)).
```

```
destruct lf; inv H9; repeat (split; auto); simpl.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
assert (∃ l, bden b i2 s2 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
decomp (H15 _ _ _ _ _ _ _ _ _ H6 H10 H12); auto.
left; intro n.
destruct n.
∃ (Cf (St i1 s1 h1) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H19 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C2 []); auto.
apply LStep_if_false; auto.
right; left; intro n.
destruct n.
∃ (Cf (St i2 s2 h2) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H20 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C2 []); auto.
apply LStep_if_false; auto.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
inv H9.
rewrite H23 in H13; rewrite H22 in H13; destruct H13.
inv H9.
inv H10; simpl; auto.
inv H9.
left; apply diverge_same_cf with (o := []).
apply LStep_if_hi_dvg with (v := v); auto.
inv H0.
destruct H11; destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2].
dup (obs_eq_bexp b _ _ _ _ _ _ H11).
inv H8; inv H9.
inv H13.
inv H8.
apply H in H5; try omega; inv H5.
destruct lt; inv H8.
specialize (H2 (St i1 s1 h1)); simpl in H2.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v]; rewrite H5 in H24; inv H24.
assert (obs_eq st1' st2' ∧ o' = o'0).
```

```
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [If b C1 C2]) lt Lo) (St i1
s1 h1) (St i2 s2 h2)).
destruct lt; inv H8; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
apply (H17 n n0 (St i1 s1 h1) (St i2 s2 h2)); auto.
decomp (H16 _ _ _ _ _ _ _ _ _ H5 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H20 H14)).
apply (False_ind _ (diverge_halt _ _ _ _ H21 H15)).
destruct H5; subst; auto.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
specialize (H9 (refl_equal _)); inv H9.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
inv H8.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
specialize (H9 (refl_equal _)); inv H9.
apply H in H6; try omega; inv H6.
destruct lf; inv H8.
specialize (H3 (St i1 s1 h1)); simpl in H3.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H3; split; auto.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct H6 as [v]; rewrite H6 in H24; inv H24.
assert (obs_eq st1' st2' ∧ o' = o'0).
assert (aden2 (BoolExp (Not b) `AND` taint_vars_assert P (modifies [If b C1 C2]) lf Lo)
(St i1 s1 h1) (St i2 s2 h2)).
destruct lf; inv H8; repeat (split; auto); simpl.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
assert (∃ l, bden b i2 s2 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
apply (H17 n n0 (St i1 s1 h1) (St i2 s2 h2)); auto.
decomp (H16 _ _ _ _ _ _ _ _ _ H6 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H20 H14)).
apply (False_ind _ (diverge_halt _ _ _ _ H21 H15)).
```

```
destruct H6; subst; auto.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
inv H8.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
rewrite H23 in H12; rewrite H24 in H12; destruct H12.
inv H8.
inv H14.
inv H15.
split; auto.
destruct st1' as [i1' s1' h1']; destruct st2' as [i2' s2' h2']; split; try split; simpl.
apply hstepn_i_const in H26; apply hstepn_i_const in H28; subst; inv H11; auto.
intro x.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])).
assert (∃ v, s1' x = Some (v,Hi)).
destruct (opt_eq_dec val_eq_dec (taint_vars [If b C1 C2] s1 x) (s1' x)).
unfold taint_vars in e.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s1 x) as [[v1 l1]||].
∃ v1; auto.
∃ 0%Z; auto.
apply hstepn_taints_s with (x := x) in H26; auto.
assert (∃ v, s2' x = Some (v,Hi)).
destruct (opt_eq_dec val_eq_dec (taint_vars [If b C1 C2] s2 x) (s2' x)).
unfold taint_vars in e.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s2 x) as [[v2 l2]||].
∃ v2; auto.
∃ 0%Z; auto.
apply hstepn_taints_s with (x := x) in H28; auto.
destruct H8 as [v1]; destruct H9 as [v2].
rewrite H8; rewrite H9; split; auto; intros.
inv H13.
apply hstepn_modifies_const with (x := x) in H26; auto; simpl in H26.
apply hstepn_modifies_const with (x := x) in H28; auto; simpl in H28.
rewrite ← H26; rewrite ← H28; unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
inv H11.
destruct H9.
```

```
apply H9.
intro n.
case_eq (h1' n); intros; auto.
destruct v1 as [v1 l1]; case_eq (h2' n); intros; auto.
destruct v2 as [v2 l2]; intros; subst.
destruct (opt_eq_dec val_eq_dec (h1 n) (h1' n)).
destruct (opt_eq_dec val_eq_dec (h2 n) (h2' n)).
rewrite H8 in e; rewrite H9 in e0; unfold obs_eq in H11; decomp H11.
specialize (H16 n); simpl in H16.
rewrite e in H16; rewrite e0 in H16; auto.
apply hstepn_taints_h with (a := n) in H28; auto.
destruct H28.
rewrite H13 in H9; inv H9.
apply hstepn_taints_h with (a := n) in H26; auto.
destruct H26.
rewrite H13 in H8; inv H8.
inv H8.
inv H8.
generalize o'0 H15 H28; clear o'0 H15 H28.
induction n0; intros.
inv H15.
inv H15.
inv H9.
rewrite H29 in H23; inv H23.
rewrite H29 in H23; inv H23.
contradiction (H28 n2 st'0).
apply IHn0; auto.
generalize o' H14 H26; clear o' H14 H26.
induction n; intros.
inv H14.
inv H14.
inv H13.
rewrite H27 in H24; inv H24.
rewrite H27 in H24; inv H24.
contradiction (H26 n1 st').
apply IHn; auto.
inv H0.
destruct H9.
dup (obs_eq_bexp b _ _ _ _ _ _ H9).
inv H7.
right; right; ∃ st2; ∃ []; apply LStep_zero.
destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2]; simpl in H10.
```

```
assert ((∃ v, bden b i1 s1 = Some (v,Hi)) → hsafe (taint_vars_cf (Cf (St i2 s2 h2) (If b
C1 C2) []))).
intros.
destruct H7 as [v].
dup H0; apply H1 in H0; unfold aden in H0; destruct H0.
rewrite bdenZ_some in H0; destruct H0 as [l].
rewrite H7 in H10; rewrite H0 in H10; destruct H10; subst.
clear H14; apply H in H5; try omega; inv H5.
unfold hsafe; intros.
inv H5.
apply (Can_hstep _ (Cf (St i2 (taint_vars [If b C1 C2] s2) h2) C1 [])).
apply bden_taint_vars with (K := [If b C1 C2]) in H0; destruct H0 as [l' [H0]].
apply HStep_if_true with (l := l'); auto.
inv H17.
apply H14 in H18; auto.
destruct lt; inv H10; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; intros; simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s2 x) as [[v2 l2]|].
∃ v2; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
apply bden_taint_vars with (K := [If b C1 C2]) in H0.
destruct H0 as [l' [H0]]; rewrite H26 in H0; inv H0.
destruct lt; inv H10.
specialize (H2 (St i2 s2 h2)); simpl in H2.
assert (∃ v, bden b i2 s2 = Some (v,Lo)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Hi; auto.
destruct H5 as [v']; rewrite H5 in H0; inv H0.
rewrite bdenZ_some in H0; destruct H0 as [l].
apply H in H6; try omega; inv H6.
unfold hsafe; intros.
inv H6.
apply (Can_hstep _ (Cf (St i2 (taint_vars [If b C1 C2] s2) h2) C2 [])).
apply bden_taint_vars with (K := [If b C1 C2]) in H0; destruct H0 as [l' [H0]].
apply HStep_if_false with (l := l'); auto.
simpl in H0; destruct (bden b i2 (taint_vars [If b C1 C2] s2)) as [[v2 l2]|]; inv H0.
destruct v2; auto; inv H19.
inv H18.
```

```
apply bden_taint_vars with (K := [If b C1 C2]) in H0.
destruct H0 as [l' [H0]]; simpl in H0; rewrite H27 in H0; inv H0.
apply H15 in H19; auto.
destruct lf; inv H14; repeat (split; auto); simpl.
assert (∃ l, bden b i2 (taint_vars [If b C1 C2] s2) = Some (false,l)).
∃ l0; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
apply no_lbls_taint_vars; destruct lt; auto.
apply aden_fold; intros; simpl.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [If b C1 C2])); try contradiction.
destruct (s2 x) as [[v2 l2]|].
∃ v2; ∃ Hi; split; auto.
∃ 0%Z; ∃ Hi; split; auto.
destruct lf; inv H14.
simpl in H0; case_eq (bden b i2 s2); intros.
destruct p as [v2 l2]; rewrite H6 in H0; inv H0.
destruct v2; inv H21.
rewrite H7 in H10; rewrite H6 in H10; destruct H10; subst.
specialize (H3 (St i2 s2 h2)); simpl in H3.
assert (∃ v, bden b i2 s2 = Some (v,Lo)).
apply H3; split; auto.
assert (∃ l, bden b i2 s2 = Some (false,l)).
∃ Hi; auto.
rewrite ← bdenZ_some in H0; rewrite H0; auto.
destruct H0 as [v']; rewrite H0 in H6; inv H6.
rewrite H7 in H10; rewrite H6 in H10; inv H10.
rename H7 into hi_hsafe.
inv H11.
apply H in H5; try omega; inv H5.
destruct lt; inv H7.
specialize (H2 (St i1 s1 h1)); simpl in H2.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v]; rewrite H5 in H21; inv H21.
assert (bden b i2 s2 = Some (true,Lo)).
rewrite H21 in H10; destruct (bden b i2 s2) as [[v2 l2]|].
destruct H10; subst.
specialize (H10 (refl_equal _)); subst; auto.
inv H10.
apply H16 with (st2 := St i2 s2 h2) in H12.
```

```
decomp H12.
left; intro n.
destruct n.
∃ (Cf (St i1 s1 h1) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H18 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C1 []); auto.
apply LStep_if_true; auto.
right; left; intro n.
destruct n.
∃ (Cf (St i2 s2 h2) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H19 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C1 []); auto.
apply LStep_if_true; auto.
destruct H19 as [st2' [o2]]; right; right; ∃ st2'; ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C1 []); auto.
apply LStep_if_true; auto.
destruct lt; inv H7.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
apply H in H6; try omega; inv H6.
destruct lf; inv H7.
specialize (H3 (St i1 s1 h1)); simpl in H3.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H3; split; auto.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct H6 as [v]; rewrite H6 in H21; inv H21.
assert (bden b i2 s2 = Some (false,Lo)).
rewrite H21 in H10; destruct (bden b i2 s2) as [[v2 l2]|].
destruct H10; subst.
specialize (H10 (refl_equal _)); subst; auto.
inv H10.
apply H16 with (st2 := St i2 s2 h2) in H12.
decomp H12.
left; intro n.
destruct n.
∃ (Cf (St i1 s1 h1) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H18 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C2 []); auto.
apply LStep_if_false; auto.
```

```
right; left; intro n.
destruct n.
∃ (Cf (St i2 s2 h2) (If b C1 C2) []); ∃ []; apply LStep_zero.
destruct (H19 n) as [cf [o]]; ∃ cf; ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C2 []); auto.
apply LStep_if_false; auto.
destruct H19 as [st2' [o2]]; right; right; ∃ st2'; ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C2 []); auto.
apply LStep_if_false; auto.
destruct lf; inv H7.
repeat (split; auto); simpl.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H7; rewrite H7; auto.
assert (∃ l, bden b i2 s2 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H7; rewrite H7; auto.
case_eq (bden b i2 s2); intros.
destruct p as [v' l]; rewrite H21 in H10; rewrite H7 in H10; destruct H10; subst.
clear H11; destruct (dvg_ex_mid (taint_vars_cf (Cf (St i2 s2 h2) (If b C1 C2) []))).
right; left; apply diverge_same_cf with (o := []).
apply LStep_if_hi_dvg with (v := v'); auto.
apply hi_hsafe; ∃ v; auto.
inv H12.
destruct H10 as [n' [st]]; right; right; ∃ st; ∃ ([]++[]).
apply LStep_succ with (cf' := Cf st Skip []).
apply LStep_if_hi with (v := v') (n := n'); auto.
apply hi_hsafe; ∃ v; auto.
apply LStep_zero.
inv H11.
rewrite H21 in H10; rewrite H7 in H10; inv H10.
left; apply diverge_same_cf with (o := []).
apply LStep_if_hi_dvg with (v := v); auto.
inv H0.
destruct H12.
dup (obs_eq_bexp b _ _ _ _ _ _ H12).
inv H7; inv H8.
inv H14.
inv H7.
apply H in H5; try omega; inv H5.
destruct lt; inv H7.
specialize (H2 (St i1 s1 h1)); simpl in H2.
```

```
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H2; split; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H5 as [v H5]; rewrite H5 in H25; inv H25.
destruct lt; inv H7.
assert (aden2 (BoolExp b ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) Lo Lo) (St
i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
apply (H20 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H5 H15 H16 H9 H10).
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
specialize (H8 (refl_equal _)); inv H8.
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
inv H7.
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
inv H7.
inv H7.
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
specialize (H8 (refl_equal _)); inv H8.
apply H in H6; try omega; inv H6.
destruct lf; inv H7.
specialize (H3 (St i1 s1 h1)); simpl in H3.
assert (∃ v, bden b i1 s1 = Some (v,Hi)).
apply H3; split; auto.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct H6 as [v H6]; rewrite H6 in H25; inv H25.
destruct lf; inv H7.
assert (aden2 (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [If b C1 C2]) Lo
Lo) (St i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto); simpl.
assert (∃ l, bden b i1 s1 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
assert (∃ l, bden b i2 s2 = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
apply (H20 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H6 H15 H16 H9 H10).
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
inv H7.
```

```
rewrite H25 in H13; rewrite H24 in H13; destruct H13.
inv H7.
inv H15.
apply hstepn_taints_h with (a := a) in H27; auto.
destruct H27 as [v1 H27]; destruct H10 as [v2 H10]; rewrite H10 in H27; inv H27.
inv H8.
assert (diverge (Cf (St i1 s1 h1) (If b C1 C2) [])).
apply diverge_same_cf with (o := []).
apply LStep_if_hi_dvg with (v := v); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H8 H15)).
apply Jden_hi; intros.
unfold hsafe; intros.
inv H7.
apply H1 in H0; destruct st as [i s h]; unfold aden in H0; destruct H0.
apply (Can_hstep _ (Cf (St i s h) C1 [])).
rewrite bdenZ_some in H0; destruct H0 as [l].
apply HStep_if_true with (l := l); auto.
apply (Can_hstep _ (Cf (St i s h) C2 [])).
rewrite bdenZ_some in H0; destruct H0 as [l].
apply HStep_if_false with (l := l); auto.
simpl in H0; destruct (bden b i s) as [[v l']|]; auto; inv H0.
destruct v; auto; inv H9.
inv H9.
apply H in H5; try omega; inv H5.
apply H9 in H10; auto.
destruct lt; inv H7; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
rewrite bdenZ_some; ∃ l; auto.
destruct lt; inv H7.
apply H in H6; try omega; inv H6.
apply H9 in H10; auto.
destruct lf; inv H7; repeat (split; auto); simpl.
assert (∃ l, bden b i s = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
assert (∃ l, bden b i s = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct lf; inv H7.
inv H7.
inv H8.
apply H in H5; try omega; inv H5.
```

```
apply H10 in H9; auto.
destruct lt; inv H7; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
rewrite bdenZ_some; ∃ l; auto.
destruct lt; inv H7.
apply H in H6; try omega; inv H6.
apply H10 in H9; auto.
destruct lf; inv H7; repeat (split; auto); simpl.
assert (∃ l, bden b i s = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
assert (∃ l, bden b i s = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
destruct lf; inv H7.
Qed.

Lemma soundness_while : ∀ N P b C ct (l : glbl),
  (∀ y : nat,
     y < S N →
     ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
     judge y ct P C Q → sound ct P C Q) →
  implies P (LblBexp b l) → (gleq l ct = false → no_lbls P (modifies [While b C]) = true)
→
  judge N (glub l ct) (BoolExp b ‘AND‘ taint_vars_assert P (modifies [While b C]) l ct) C
                                          (taint_vars_assert P (modifies [While b C]) l ct)
→
  sound ct P (While b C) (BoolExp (Not b) ‘AND‘ taint_vars_assert P (modifies [While b
C]) l ct).
Proof.
intros.
rename C into C0; rename H2 into H3; rename H1 into H2; rename H0 into H1; destruct
ct; intros.
apply Jden_lo; intros.
unfold lsafe; intros.
inv H4.
dup H0; apply H1 in H0.
destruct st as [i s h]; simpl in H0; destruct H0 as [v].
destruct l.
destruct v.
apply (Can_lstep _ (Cf (St i s h) C0 [While b C0]) []).
apply LStep_while_true; auto.
apply (Can_lstep _ (Cf (St i s h) Skip []) []).
```

```
apply LStep_while_false; auto.
assert (hsafe (taint_vars_cf (Cf (St i s h) (While b C0) []))).
unfold hsafe; intros.
generalize i s h cf' v H0 H4 H6 H7; clear i s h cf' v H0 H4 H5 H6 H7; unfold
taint_vars_cf.
induction n using (well_founded_induction lt_wf); intros.
inv H6.
destruct v.
apply (Can_hstep _ (Cf (St i (taint_vars [While b C0] s) h) C0 [While b C0])).
apply HStep_while_true with (l := Hi); auto.
apply bden_taint_vars with (K := [While b C0]) in H4.
destruct H4 as [l' [H4]]; rewrite H4.
destruct l'; inv H6; auto.
apply (Can_hstep _ (Cf (St i (taint_vars [While b C0] s) h) Skip [])).
apply HStep_while_false with (l := Hi); auto.
apply bden_taint_vars with (K := [While b C0]) in H4.
destruct H4 as [l' [H4]]; rewrite H4.
destruct l'; inv H6; auto.
inv H8.
change (hstepn n0 (Cf (St i (taint_vars [While b C0] s) h) C0 ([]++[While b C0])) cf')
in H9.
destruct cf' as [st C K]; apply hstep_trans_inv in H9; destruct H9.
destruct H6 as [K'' [H6]]; subst.
apply H in H3; try omega; inv H3.
case_eq (halt_config (Cf st C K'')); intros.
destruct C; destruct K''; inv H3.
apply (Can_hstep _ (Cf st (While b C0) [])); apply HStep_skip.
apply H8 in H6.
specialize (H6 H3); inv H6.
destruct cf' as [st' C' K''']; apply (Can_hstep _ (Cf st' C' (K'''++[While b C0]))).
apply hstep_extend; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s x) as [[v' l']||].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
destruct H6 as [st'' [n1 [n2]]]; decomp H6; subst.
apply H in H3; try omega; inv H3.
```

```
apply H9 in H8.
inv H10.
apply (Can_hstep _ (Cf st (While b C0) [])); apply HStep_skip.
inv H3.
dup H8; destruct st'' as [i'' s'' h'']; apply taint_vars_assert_inv in H8; auto.
simpl in H3; destruct H3.
dup H3; apply H1 in H3; simpl in H3; destruct H3 as [v'].
rewrite H8 in H11; apply H0 with (v := v') in H11; auto; try omega.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s x) as [[v' l']||].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
inv H9.
inv H7.
inv H6.
destruct (dvg_ex_mid (taint_vars_cf (Cf (St i s h) (While b C0) []))).
apply (Can_lstep _ (Cf (St i s h) (While b C0) []) []).
apply LStep_while_hi_dvg with (v := v); auto.
destruct H7 as [n [st]].
apply (Can_lstep _ (Cf st Skip []) []).
apply LStep_while_hi with (v := v) (n := n); auto.
generalize st cf' o' cf'0 o0 H6 H5 H7 H0; clear st cf' o' cf'0 o0 H6 H5 H7 H0.
induction n0 using (well_founded_induction lt_wf); intros.
inv H6.
apply H in H3; try omega; inv H3.
destruct l; inv H6.
apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H14; inv H14.
destruct cf' as [st' C' K'].
change (lstepn n0 (Cf (St i s h) C0 ([]++[While b C0])) (Cf st' C' K') o') in H7.
apply lstep_trans_inv in H7; destruct H7.
destruct H3 as [K'' [H3]]; subst.
apply H8 in H3.
case_eq (halt_config (Cf st' C' K'')); intros.
destruct C'; destruct K''; inv H7.
apply (Can_lstep _ (Cf st' (While b C0) []) []); apply LStep_skip.
specialize (H3 H7); inv H3.
```

```
destruct cf' as [st'' C'' K'''].
apply (Can_lstep _ (Cf st'' C'' (K'''++[While b C0])) o).
apply lstep_extend; auto.
destruct l; inv H6; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H3 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H3; subst.
inv H16.
apply (Can_lstep _ (Cf st' (While b C0) []) []); apply LStep_skip.
inv H3.
apply H9 in H7.
destruct l; inv H6; unfold taint_vars_assert in H7; simpl in H7.
destruct n.
inv H15.
destruct st' as [i' s' h']; apply H1 in H7; simpl in H7.
destruct H7 as [v]; destruct v.
apply (Can_lstep _ (Cf (St i' s' h') C0 [While b C0]) []).
apply LStep_while_true; auto.
apply (Can_lstep _ (Cf (St i' s' h') Skip []) []).
apply LStep_while_false; auto.
inv H15.
apply H0 with (st := st'') (o0 := o) in H16; auto; try omega.
destruct l; inv H6; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
inv H7.
inv H5.
inv H6.
inv H7.
inv H5.
inv H6.
destruct n0.
inv H7.
apply (Can_lstep _ (Cf (St i s h) (While b C0) []) []).
apply LStep_while_hi_dvg with (v := v); auto.
inv H7.
apply H0 with (st := St i s h) (o0 := o) in H9; auto.
generalize st st' o H0 H4; clear st st' o H0 H4.
induction n using (well_founded_induction lt_wf); intros.
inv H5.
inv H6.
destruct st' as [i' s' h'].
change (lstepn n0 (Cf (St i s h) C0 ([]++[While b C0])) (Cf (St i' s' h') Skip []) o') in
H7.
```

```
apply lstep_trans_inv in H7; destruct H7.
destruct H5 as [K' [H5]].
apply f_equal with (f := fun l ⇒ length l) in H6; simpl in H6.
destruct K'; inv H6.
destruct H5 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H5; subst.
apply H in H3; try omega; inv H3.
destruct l; inv H5.
apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H13; inv H13.
apply H9 in H6.
destruct l; inv H5; unfold taint_vars_assert in H6; simpl in H6.
inv H8.
inv H3.
apply H0 in H5; auto; try omega.
destruct l; inv H5; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
inv H7.
dup H4; apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H13; inv H13.
repeat (split; auto); simpl.
assert (∃ l, bden b i s = Some (false,l)).
∃ Lo; auto.
rewrite ← bdenZ_some in H6; rewrite H6; auto.
inv H5.
inv H7.
assert (l = Hi).
apply H1 in H4; simpl in H4.
destruct H4 as [v' H4]; rewrite H4 in H13; inv H13; auto.
subst; clear H0; generalize st' i s h v H13 H14 H4 H15; clear st' i s h v H13 H14
H4 H15.
induction n using (well_founded_induction lt_wf); intros.
inv H15.
inv H5.
destruct st' as [i' s' h'].
change (hstepn n0 (Cf (St i (taint_vars [While b C0] s) h) C0 ([]++[While b C0])) (Cf
(St i' s' h') Skip [])) in H6.
apply hstep_trans_inv in H6; destruct H6.
destruct H5 as [K' [H5]].
apply f_equal with (f := fun l ⇒ length l) in H6; simpl in H6.
destruct K'; inv H6.
destruct H5 as [st'' [n1 [n2]]]; decomp H5; subst.
inv H8.
```

```
inv H5.
apply H in H3; try omega; inv H3.
dup H6; apply H8 in H6.
destruct st'' as [i'' s'' h'']; dup H6; apply taint_vars_assert_inv in H6; auto.
simpl in H9; destruct H9.
dup H9; apply H1 in H9; simpl in H9.
destruct H9 as [v']; rewrite H6 in H7; apply H0 with (v := v') in H7; auto; try omega.
unfold hsafe; intros.
rewrite H6 in H3; apply hstepn_extend with (K0 := [While b C0]) in H3.
apply (H14 (S (n1 + S n0)) cf'); auto.
apply HStep_succ with (cf' := Cf (St i (taint_vars [While b C0] s) h) C0 [While b C0]).
apply HStep_while_true with (l := l); auto.
apply hstep_trans with (cf2 := Cf (St i'' (taint_vars [While b C0] s'') h'') Skip [While b
C0]); auto.
apply HStep_succ with (cf' := Cf (St i'' (taint_vars [While b C0] s'') h'') (While b C0)
[]); auto.
apply HStep_skip.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s x) as [[v' l']|].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
inv H6.
repeat (split; auto); simpl.
assert (∃ l, bden b i (taint_vars [While b C0] s) = Some (false,l)).
∃ l; auto.
rewrite ← bdenZ_some in H5; rewrite H5; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s x) as [[v' l']|].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
inv H5.
inv H5.
assert (diverge (Cf (St i s h) (While b C0) [])).
apply diverge_same_cf with (o := []).
```

```
apply LStep_while_hi_dvg with (v := v); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H5 H7)).
generalize st1 st2 st1' st2' C' K' o1 o2 H0 H4 H5; clear st1 st2 st1' st2' C' K' o1 o2
H0 H4 H5.
induction n using (well_founded_induction lt_wf); rename H0 into IHn; intros.
inv H4; simpl; auto.
inv H5.
inv H0.
destruct H5; destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2].
dup (obs_eq_bexp b _ _ _ _ _ _ H5).
inv H6.
inv H8.
apply H in H3; try omega; inv H3.
destruct l; inv H6.
apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H19; inv H19.
destruct st1' as [i1' s1' h1']; destruct st2' as [i2' s2' h2'].
change (lstepn n0 (Cf (St i1 s1 h1) C0 ([]++[While b C0])) (Cf (St i1' s1' h1') C' K')
o') in H7.
change (lstepn n0 (Cf (St i2 s2 h2) C0 ([]++[While b C0])) (Cf (St i2' s2' h2') C' K')
o'0) in H9.
apply lstep_trans_inv in H7; apply lstep_trans_inv in H9.
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [While b C0]) l Lo) (St i1
s1 h1) (St i2 s2 h2)).
destruct l; inv H6; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
assert (∀ i s h, bden b i s = Some (true,Lo) → diverge (Cf (St i s h) C0 []) → diverge
(Cf (St i s h) (While b C0) [])).
intros; intro n.
destruct n.
∃ (Cf (St i s h) (While b C0) []); ∃ []; apply LStep_zero.
destruct (H17 n) as [[st1 C1 K1] [o]].
∃ (Cf st1 C1 (K1++[While b C0])); ∃ ([]++o).
apply LStep_succ with (cf' := Cf (St i s h) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
apply lstepn_extend; auto.
destruct H7.
destruct H9.
destruct H7 as [K''' [H7]]; destruct H9 as [K'' [H9]]; subst.
apply app_cancel_r in H20; subst.
decomp (H12 _ _ _ _ _ _ _ _ _ H3 H7 H9); auto.
```

```
destruct H7 as [K''' [H7]]; subst.
destruct H9 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H9; subst.
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [While b C0]) l Lo) (St i2
s2 h2) (St i1 s1 h1)).
destruct l; inv H6; apply obs_eq_sym in H5; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
decomp (H14 _ _ _ _ _ _ _ H9 H17); auto.
destruct H22 as [st2' [o2']].
apply lstep_trans_inv' in H7.
destruct H7 as [cf'' [o1'' [o2'']]]; decomp H7.
destruct (lstepn_det _ _ _ _ _ _ H20 H22); subst.
inv H24; simpl; auto.
inv H7.
destruct H9.
destruct H9 as [K''' [H9]]; subst.
destruct H7 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H7; subst.
decomp (H14 _ _ _ _ _ _ _ H3 H17); auto.
destruct H20 as [st2' [o2']].
apply lstep_trans_inv' in H9.
destruct H9 as [cf'' [o1'' [o2'']]]; decomp H9.
destruct (lstepn_det _ _ _ _ _ _ H7 H20); subst.
inv H23; simpl; auto.
inv H9.
destruct H7 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H7; subst.
destruct H9 as [st''' [n1' [n2' [o1' [o2']]]]].
decomp H7; subst.
decomp (H14 _ _ _ _ _ _ _ H3 H17); auto.
destruct H23 as [st2' [o2'']].
dup (lstepn_det_term _ _ _ _ _ _ _ H9 H7); subst.
destruct (lstepn_det _ _ _ _ _ _ H7 H9); subst.
inv H23.
assert (n2' = n2); try omega; subst.
inv H22; simpl; auto.
inv H21; simpl; auto.
inv H23; inv H25.
assert (n < S (n1 + S n)); try omega.
assert (side_condition C0 (St i1 s1 h1) (St i2 s2 h2)).
decomp (H12 _ _ _ _ _ _ _ _ _ H3 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H22 H17)).
apply (False_ind _ (diverge_halt _ _ _ _ H23 H9)).
assert (aden2 P st'' st''').
```

```
split; try split.
destruct l; inv H6; apply H11 in H17; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
destruct l; inv H6; apply H11 in H9; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
destruct (H13 _ _ _ _ _ _ _ _ H3 H22 H17 H9); auto.
decomp (IHn _ H21 _ _ _ _ _ _ _ _ H23 H26 H24); auto.
left; intro n'.
destruct n'.
∃ (Cf (St i1 s1 h1) (While b C0) []); ∃ []; apply LStep_zero.
assert (n' ≤ n1 ∨ n' > n1); try omega.
destruct H27.
assert (n1 = n' + (n1-n')); try omega.
rewrite H28 in H17; apply lstep_trans_inv' in H17.
destruct H17 as [[st C K] [o2 [o3]]]; decomp H17.
∃ (Cf st C (K++[While b C0])); ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
apply lstepn_extend; auto.
destruct n'.
inv H27.
destruct (H25 (n'-n1)) as [[st C K] [o]].
∃ (Cf st C K); ∃ ([]++o1++[]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
assert (S n' = n1 + S (n'-n1)); try omega.
rewrite H29; apply lstep_trans with (cf2 := Cf st'' Skip ([]++[While b C0])).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st'' (While b C0) []); auto.
apply LStep_skip.
right; left; intro n'.
destruct n'.
∃ (Cf (St i2 s2 h2) (While b C0) []); ∃ []; apply LStep_zero.
assert (n' ≤ n1 ∨ n' > n1); try omega.
destruct H25.
assert (n1 = n' + (n1-n')); try omega.
rewrite H28 in H9; apply lstep_trans_inv' in H9.
destruct H9 as [[st C K] [o2 [o3]]]; decomp H9.
∃ (Cf st C (K++[While b C0])); ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
```

```
apply LStep_while_true; auto.
apply lstepn_extend; auto.
destruct n'.
inv H25.
destruct (H27 (n'-n1)) as [[st C K] [o]].
∃ (Cf st C K); ∃ ([]++o1'++[]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
assert (S n' = n1 + S (n'-n1)); try omega.
rewrite H29; apply lstep_trans with (cf2 := Cf st''' Skip ([]++[While b C0])).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st''' (While b C0) []); auto.
apply LStep_skip.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
specialize (H8 (refl_equal _)); inv H8.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
inv H6.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
inv H6.
inv H8.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
specialize (H8 (refl_equal _)); inv H8.
inv H7; simpl; auto.
inv H6.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
inv H6.
rewrite H19 in H10; rewrite H18 in H10; destruct H10.
inv H6.
inv H7; simpl; auto.
inv H6.
left; apply diverge_same_cf with (o := []).
apply LStep_while_hi_dvg with (v := v); auto.
clear H4; generalize n2 st1 st2 st1' st2' o1 o2 H0 H5 H6; clear n2 st1 st2 st1' st2'
o1 o2 H0 H5 H6.
induction n1 using (well_founded_induction lt_wf); intros.
inv H4.
destruct H8; destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2].
dup (obs_eq_bexp b _ _ _ _ _ _ H8).
inv H5; inv H6.
inv H10.
inv H5.
apply H in H3; try omega; inv H3.
```

```
destruct l; inv H5.
apply H1 in H7; simpl in H7.
destruct H7 as [v H7]; rewrite H7 in H20; inv H20.
destruct l; inv H5.
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [While b C0]) Lo Lo) (St
i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct st1' as [i1' s1' h1']; destruct st2' as [i2' s2' h2'].
change (lstepn n (Cf (St i1 s1 h1) C0 ([]++[While b C0])) (Cf (St i1' s1' h1') Skip [])
o') in H11.
change (lstepn n0 (Cf (St i2 s2 h2) C0 ([]++[While b C0])) (Cf (St i2' s2' h2') Skip [])
o'0) in H12.
apply lstep_trans_inv in H11; apply lstep_trans_inv in H12.
destruct H11.
destruct H5 as [K1 [H5]].
apply f_equal with (f := fun l ⇒ length l) in H11; simpl in H11.
destruct K1; inv H11.
destruct H12.
destruct H11 as [K2 [H11]].
apply f_equal with (f := fun l ⇒ length l) in H12; simpl in H12.
destruct K2; inv H12.
destruct H5 as [st1 [n1 [n2 [o1 [o2]]]]]; decomp H5; subst.
destruct H11 as [st2 [n1' [n2' [o1' [o2']]]]].
decomp H5; subst.
inv H18; inv H21.
inv H5; inv H18.
assert (n < S (n1 + S n)); try omega.
assert (side_condition C0 (St i1 s1 h1) (St i2 s2 h2)).
decomp (H13 _ _ _ _ _ _ _ _ _ H3 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H18 H12)).
apply (False_ind _ (diverge_halt _ _ _ _ H21 H11)).
assert (aden2 P st1 st2).
split; try split.
apply H10 in H12; inv H3; intuit.
apply H10 in H11; inv H3; intuit.
apply proj1 with (B := o1 = o1').
apply (H14 _ _ _ _ _ _ _ _ H3 H18 H12 H11).
destruct (H0 _ H5 _ _ _ _ _ _ _ H21 H17 H22); subst; split; auto.
destruct (H14 _ _ _ _ _ _ _ _ H3 H18 H12 H11); subst; auto.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
```

```
specialize (H6 (refl_equal _)); inv H6.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
specialize (H6 (refl_equal _)); inv H6.
inv H11.
inv H12; auto.
inv H5.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
rewrite H20 in H9; rewrite H19 in H9; destruct H9.
inv H5.
inv H11.
inv H12.
split; auto.
destruct st1' as [i1' s1' h1']; destruct st2' as [i2' s2' h2']; split; try split; simpl.
apply hstepn_i_const in H22; apply hstepn_i_const in H24; subst; inv H8; auto.
intro x.
destruct (In_dec eq_nat_dec x (modifies [While b C0])).
assert (∃ v, s1' x = Some (v,Hi)).
destruct (opt_eq_dec val_eq_dec (taint_vars [While b C0] s1 x) (s1' x)).
unfold taint_vars in e.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s1 x) as [[v1 l1]||].
∃ v1; auto.
∃ 0%Z; auto.
apply hstepn_taints_s with (x := x) in H22; auto.
assert (∃ v, s2' x = Some (v,Hi)).
destruct (opt_eq_dec val_eq_dec (taint_vars [While b C0] s2 x) (s2' x)).
unfold taint_vars in e.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s2 x) as [[v2 l2]||].
∃ v2; auto.
```

```
∃ 0%Z; auto.
apply hstepn_taints_s with (x := x) in H24; auto.
destruct H5 as [v1]; destruct H6 as [v2].
rewrite H5; rewrite H6; split; auto; intros.
inv H10.
apply hstepn_modifies_const with (x := x) in H22; auto; simpl in H22.
apply hstepn_modifies_const with (x := x) in H24; auto; simpl in H24.
rewrite ← H22; rewrite ← H24; unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
inv H8.
destruct H6.
apply H6.
intro n.
case_eq (h1' n); intros; auto.
destruct v1 as [v1 l1]; case_eq (h2' n); intros; auto.
destruct v2 as [v2 l2]; intros; subst.
destruct (opt_eq_dec val_eq_dec (h1 n) (h1' n)).
destruct (opt_eq_dec val_eq_dec (h2 n) (h2' n)).
rewrite H5 in e; rewrite H6 in e0; unfold obs_eq in H8; decomp H8.
specialize (H13 n); simpl in H13.
rewrite e in H13; rewrite e0 in H13; auto.
apply hstepn_taints_h with (a := n) in H24; auto.
destruct H24.
rewrite H10 in H6; inv H6.
apply hstepn_taints_h with (a := n) in H22; auto.
destruct H22.
rewrite H10 in H5; inv H5.
inv H5.
inv H5.
generalize o'0 H12; clear o'0 H12.
induction n0; intros.
inv H12.
inv H12.
inv H6.
rewrite H25 in H19; inv H19.
rewrite H25 in H19; inv H19.
contradiction (H24 n2 st'0).
apply IHn0; auto.
clear H0; generalize o' H11; clear o' H11.
induction n; intros.
inv H11.
inv H11.
```

```
inv H6.
rewrite H19 in H20; inv H20.
rewrite H19 in H20; inv H20.
contradiction (H22 n1 st').
apply IHn; auto.
assert (∀ i s h, (∃ v, bden b i s = Some (v,Hi)) → aden P (St i s h) →
  hsafe (taint_vars_cf (Cf (St i s h) (While b C0) []))).
clear n st1 st2 st1' C' K' o1 H0 H4; intros; unfold hsafe; intros.
generalize i s h H0 H4 cf' H5 H6; clear i s h H0 H4 cf' H5 H6.
induction n using (well_founded_induction lt_wf); intros.
destruct H4 as [v].
dup H5; apply H1 in H5; simpl in H5.
destruct H5 as [v' H5]; rewrite H5 in H4; inv H4.
apply H in H3; try omega; inv H3.
dup H5; apply bden_taint_vars with (K := [While b C0]) in H5; destruct H5 as [l [H5]].
destruct l; inv H10.
inv H6.
destruct v.
apply (Can_hstep _ (Cf (St i (taint_vars [While b C0] s) h) C0 [While b C0])).
apply HStep_while_true with (l := Hi); auto.
apply (Can_hstep _ (Cf (St i (taint_vars [While b C0] s) h) Skip [])).
apply HStep_while_false with (l := Hi); auto.
inv H10.
rewrite H18 in H5; inv H5.
destruct cf' as [st C K].
change (hstepn n0 (Cf (St i (taint_vars [While b C0] s) h) C0 ([]++[While b C0])) (Cf st
C K)) in H11.
apply hstep_trans_inv in H11; destruct H11.
destruct H5 as [K' [H5]]; subst.
apply H4 in H5.
case_eq (halt_config (Cf st C K')); intros.
destruct C; destruct K'; inv H6.
apply (Can_hstep _ (Cf st (While b C0) [])); apply HStep_skip.
specialize (H5 H6); inv H5.
destruct cf' as [st' C' K'']; ∃ (Cf st' C' (K''++[While b C0])).
apply hstep_extend; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Hi; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
```

```
destruct (s x) as [[v' l']||].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
destruct H5 as [st' [n1 [n2]]]; decomp H5; subst.
inv H11.
apply (Can_hstep _ (Cf st (While b C0) [])); apply HStep_skip.
inv H5.
apply H9 in H6.
dup H6; destruct st' as [i' s' h']; apply taint_vars_assert_inv in H6; auto.
rewrite H6 in H10; apply H0 in H10; auto; try omega.
apply (H1 (St i' s' h')); simpl in H5; intuit.
simpl in H5; intuit.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Hi; auto.
apply no_lbls_taint_vars; auto.
apply aden_fold; simpl; intros.
unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies [While b C0])); try contradiction.
destruct (s x) as [[v' l']||].
∃ v'; ∃ Hi; auto.
∃ 0%Z; ∃ Hi; auto.
rewrite H18 in H5; inv H5.
inv H11.
inv H7.
inv H5.
rename H5 into hsafe_help.
generalize st1 st2 st1' C' K' o1 H0 H4; clear st1 st2 st1' C' K' o1 H0 H4.
induction n using (well_founded_induction lt_wf); rename H0 into IHn; intros.
inv H4.
right; right; ∃ st2; ∃ []; apply LStep_zero.
destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2]; inv H0.
destruct H7.
dup (obs_eq_bexp b _ _ _ _ _ _ H7).
assert (∀ i s h, bden b i s = Some (true,Lo) → diverge (Cf (St i s h) C0 []) → diverge
(Cf (St i s h) (While b C0) [])).
intros; intro n.
destruct n.
∃ (Cf (St i s h) (While b C0) []); ∃ []; apply LStep_zero.
destruct (H10 n) as [[st1 C1 K1] [o1]].
∃ (Cf st1 C1 (K1++[While b C0])); ∃ ([]++o1).
apply LStep_succ with (cf' := Cf (St i s h) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
```

```
apply lstepn_extend; auto.
inv H5.
dup H4; apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H18; inv H18.
apply H in H3; try omega; inv H3.
assert (bden b i2 s2 = Some (true,Lo)).
rewrite H4 in H8; destruct (bden b i2 s2) as [[v l]|]; destruct H8; subst.
specialize (H8 (refl_equal _)); subst; auto.
change (lstepn n0 (Cf (St i1 s1 h1) C0 ([]++[While b C0])) (Cf st1' C' K') o') in H6.
apply lstep_trans_inv in H6; destruct H6.
destruct H6 as [K [H6]]; subst.
apply H14 with (st2 := St i2 s2 h2) in H6.
decomp H6; auto.
destruct H17 as [st2' [o2]].
right; right; ∃ st2'; ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
apply lstepn_extend; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
destruct H6 as [st'' [n1 [n2 [o1 [o2]]]]]; decomp H6; subst.
dup H16; apply H14 with (st2 := St i2 s2 h2) in H16.
decomp H16; auto.
destruct H19 as [st2' [o2']].
inv H18.
right; right; ∃ st2'; ∃ ([]++o2').
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 [While b C0]).
apply LStep_while_true; auto.
assert (n1 + 0 = n1); try omega.
rewrite H17; apply lstepn_extend with (K0 := [While b C0]) in H16; auto.
inv H17.
apply IHn with (st2 := st2') in H19; try omega.
decomp H19.
left; intro n'.
destruct n'.
∃ (Cf (St i1 s1 h1) (While b C0) []); ∃ []; apply LStep_zero.
assert (n' ≤ n1 ∨ n' > n1); try omega.
destruct H18.
assert (n1 = n' + (n1-n')); try omega.
rewrite H19 in H6; apply lstep_trans_inv' in H6.
destruct H6 as [[st C K] [o2 [o3]]]; decomp H6.
```

```
∃ (Cf st C (K++[While b C0])); ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
apply lstepn_extend; auto.
destruct n'.
inv H18.
destruct (H17 (n'-n1)) as [[st C K] [o]].
∃ (Cf st C K); ∃ ([]++o1++[]++o).
apply LStep_succ with (cf' := Cf (St i1 s1 h1) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
assert (S n' = n1 + S (n'-n1)); try omega.
rewrite H20; apply lstep_trans with (cf2 := Cf st'' Skip ([]++[While b C0])).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st'' (While b C0) []); auto.
apply LStep_skip.
right; left; intro n'.
destruct n'.
∃ (Cf (St i2 s2 h2) (While b C0) []); ∃ []; apply LStep_zero.
assert (n' ≤ n1 ∨ n' > n1); try omega.
destruct H17.
assert (n1 = n' + (n1-n')); try omega.
rewrite H19 in H16; apply lstep_trans_inv' in H16.
destruct H16 as [[st C K] [o2 [o3]]]; decomp H16.
∃ (Cf st C (K++[While b C0])); ∃ ([]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
apply lstepn_extend; auto.
destruct n'.
inv H17.
destruct (H18 (n'-n1)) as [[st C K] [o]].
∃ (Cf st C K); ∃ ([]++o2'++[]++o).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
assert (S n' = n1 + S (n'-n1)); try omega.
rewrite H20; apply lstep_trans with (cf2 := Cf st2' Skip ([]++[While b C0])).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st2' (While b C0) []); auto.
apply LStep_skip.
destruct H18 as [st2'' [o2]].
right; right; ∃ st2''; ∃ ([]++o2'++[]++o2).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) C0 ([]++[While b C0])).
apply LStep_while_true; auto.
```

```
apply lstep_trans with (cf2 := Cf st2' Skip ([]++[While b C0])).
apply lstepn_extend; auto.
apply LStep_succ with (cf' := Cf st2' (While b C0) []); auto.
apply LStep_skip.
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [While b C0]) Lo Lo) (St
i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
assert (side_condition C0 (St i1 s1 h1) (St i2 s2 h2)).
decomp (H12 _ _ _ _ _ _ _ _ _ H17 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H18 H6)).
apply (False_ind _ (diverge_halt _ _ _ _ H20 H16)).
split; try split.
apply H11 in H6; inv H17; intuit.
apply H11 in H16; inv H17; intuit.
destruct (H13 _ _ _ _ _ _ _ _ H17 H18 H6 H16); auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Lo; auto.
inv H6.
right; right; ∃ (St i2 s2 h2); ∃ ([]++[]).
apply LStep_succ with (cf' := Cf (St i2 s2 h2) Skip []).
apply LStep_while_false; auto.
destruct (bden b i2 s2) as [[v2 l2]|]; rewrite H18 in H8; destruct H8; subst.
specialize (H6 (refl_equal _)); subst; auto.
apply LStep_zero.
inv H5.
inv H6.
case_eq (bden b i2 s2); intros.
destruct p as [v' l'].
rewrite H18 in H8; rewrite H5 in H8; destruct H8; subst.
destruct (dvg_ex_mid (Cf (St i2 (taint_vars [While b C0] s2) h2) (While b C0) [])).
right; left; apply diverge_same_cf with (o := []).
apply LStep_while_hi_dvg with (v := v'); auto.
apply hsafe_help; auto.
∃ v'; auto.
destruct H6 as [n' [st]].
right; right; ∃ st; ∃ ([]++[]).
apply LStep_succ with (cf' := Cf st Skip []).
apply LStep_while_hi with (v := v') (n := n'); auto.
apply hsafe_help; auto.
```

```
∃ v'; auto.
apply LStep_zero.
rewrite H18 in H8; rewrite H5 in H8; inv H8.
inv H5.
left; apply diverge_same_cf with (o := []).
apply LStep_while_hi_dvg with (v := v); auto.
generalize n2 i1 s1 h1 i1' s1' h1' i2 s2 h2 i2' s2' h2' o1 o2 a H0 H4 H5 H6 H7;
  clear n2 i1 s1 h1 i1' s1' h1' i2 s2 h2 i2' s2' h2' o1 o2 a H0 H4 H5 H6 H7.
induction n1 using (well_founded_induction lt_wf); intros.
inv H4.
destruct H10.
dup (obs_eq_bexp b _ _ _ _ _ _ H10).
inv H5; inv H6.
inv H12.
inv H5.
apply H in H3; try omega; inv H3.
destruct l; inv H5.
apply H1 in H9; simpl in H9.
destruct H9 as [v1 H9]; rewrite H9 in H22; inv H22.
destruct l; inv H5.
change (lstepn n (Cf (St i1 s1 h1) C0 ([]++[While b C0])) (Cf (St i1' s1' h1') Skip [])
o') in H13.
change (lstepn n0 (Cf (St i2 s2 h2) C0 ([]++[While b C0])) (Cf (St i2' s2' h2') Skip [])
o'0) in H14.
apply lstep_trans_inv in H13; apply lstep_trans_inv in H14.
destruct H13.
destruct H3 as [K'' [H3]].
apply f_equal with (f := fun l ⇒ length l) in H5; simpl in H5.
destruct K''; inv H5.
destruct H14.
destruct H5 as [K'' [H5]].
apply f_equal with (f := fun l ⇒ length l) in H13; simpl in H13.
destruct K''; inv H13.
destruct H3 as [[i1'' s1'' h1''] [n1 [n2 [o1 [o2]]]]]; decomp H3.
destruct H5 as [[i2'' s2'' h2''] [n1' [n2' [o1' [o2']]]]].
decomp H3; subst.
inv H19; inv H24.
inv H3; inv H19.
assert (aden2 (BoolExp b `AND` taint_vars_assert P (modifies [While b C0]) Lo Lo) (St
i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
```

```
rewrite bdenZ_some; ∃ Lo; auto.
assert (aden2 P (St i1'' s1'' h1'') (St i2'' s2'' h2'')).
split; try split.
apply H12 in H13; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
apply H12 in H5; auto.
repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ Lo; auto.
decomp (H15 _ _ _ _ _ _ _ _ _ H3 (LStep_zero _) (LStep_zero _)).
apply (False_ind _ (diverge_halt _ _ _ _ H19 H13)).
apply (False_ind _ (diverge_halt _ _ _ _ H23 H5)).
destruct (H16 _ _ _ _ _ _ _ _ H3 H23 H13 H5); auto.
assert (n < S (n1 + S n)); try omega.
destruct (opt_eq_dec val_eq_dec (h1'' a) (h1' a)).
destruct (opt_eq_dec val_eq_dec (h1 a) (h1'' a)).
rewrite e in e0; contradiction.
assert (aden2 P (St i1 s1 h1) (St i2 s2 h2)).
repeat (split; auto).
apply (H18 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H3 H13 H5 n2).
rewrite e; auto.
dup (H0 _ H23 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H19 H14 H20 n2 H8).
intro; apply lstepn_nonincreasing with (a := a) in H5; auto.
rewrite H22 in H11; rewrite H21 in H11; destruct H11.
specialize (H6 (refl_equal _)); inv H6.
rewrite H22 in H11; rewrite H21 in H11; destruct H11.
inv H5.
rewrite H22 in H11; rewrite H21 in H11; destruct H11.
inv H5.
inv H13.
contradiction H7; auto.
inv H6.
inv H13.
apply hstepn_taints_h with (a := a) in H24; auto.
destruct H24 as [v1 H24]; destruct H8 as [v2 H8]; rewrite H8 in H24; inv H24.
inv H6.
assert (diverge (Cf (St i1 s1 h1) (While b C0) [])).
apply diverge_same_cf with (o := []).
apply LStep_while_hi_dvg with (v := v); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H6 H13)).
apply Jden_hi; intros.
unfold hsafe; intros.
```

```
generalize st H0 cf' H4 H5; clear st H0 cf' H4 H5.
induction n using (well_founded_induction lt_wf); rename H0 into IHn; intros.
inv H4.
dup H0; apply H1 in H0.
destruct st as [i s h]; simpl in H0; destruct H0 as [v].
destruct v.
apply (Can_hstep _ (Cf (St i s h) C0 [While b C0])).
apply HStep_while_true with (l := l); auto.
apply (Can_hstep _ (Cf (St i s h) Skip [])).
apply HStep_while_false with (l := l); auto.
inv H6.
apply H in H3; try omega; inv H3.
destruct cf' as [st' C' K'].
change (hstepn n0 (Cf (St i s h) C0 ([]++[While b C0])) (Cf st' C' K')) in H7.
apply hstep_trans_inv in H7; destruct H7.
destruct H3 as [K'' [H3]]; subst.
apply H6 in H3.
case_eq (halt_config (Cf st' C' K'')); intros.
destruct C'; destruct K''; inv H7.
apply (Can_hstep _ (Cf st' (While b C0) [])); apply HStep_skip.
specialize (H3 H7); inv H3.
destruct cf' as [st'' C'' K'''].
apply (Can_hstep _ (Cf st'' C'' (K'''++[While b C0]))).
apply hstep_extend; auto.
destruct l; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l0; auto.
rewrite bdenZ_some; ∃ l0; auto.
destruct H3 as [st'' [n1 [n2]]]; decomp H3; subst.
inv H10.
apply (Can_hstep _ (Cf st' (While b C0) [])); apply HStep_skip.
inv H3.
apply IHn in H9; auto; try omega.
apply H8 in H7.
destruct l; auto.
destruct l; repeat (split; auto); simpl.
rewrite bdenZ_some; ∃ l0; auto.
rewrite bdenZ_some; ∃ l0; auto.
destruct l; inv H4.
inv H7.
inv H5.
inv H4.
generalize st st' H0 H4; clear st st' H0 H4.
```

```
induction n using (well_founded_induction lt_wf); intros.
inv H5.
inv H6.
destruct st' as [i' s' h'].
change (hstepn n0 (Cf (St i s h) C0 ([]++[While b C0])) (Cf (St i' s' h') Skip [])) in H7.
apply hstep_trans_inv in H7; destruct H7.
destruct H5 as [K' [H5]].
apply f_equal with (f := fun l ⇒ length l) in H6; simpl in H6.
destruct K'; inv H6.
destruct H5 as [st'' [n1 [n2]]]; decomp H5; subst.
apply H in H3; try omega; inv H3.
dup H4; apply H1 in H4; simpl in H4.
destruct H4 as [v H4]; rewrite H4 in H12; inv H12.
inv H8.
inv H10.
apply H0 in H11; auto; try omega.
apply H9 in H6.
destruct l0; auto.
destruct l0; repeat (split; auto); unfold taint_vars_assert; simpl; auto.
rewrite bdenZ_some; ∃ Lo; auto.
rewrite bdenZ_some; ∃ Hi; auto.
destruct l; inv H5.
inv H7.
destruct l; repeat (split; auto); simpl.
assert (∃ l, bden b i s = Some (false,l)).
∃ l0; auto.
rewrite ← bdenZ_some in H5; rewrite H5; auto.
assert (∃ l, bden b i s = Some (false,l)).
∃ l0; auto.
rewrite ← bdenZ_some in H5; rewrite H5; auto.
inv H5.
Qed.

Lemma soundness_conseq : ∀ N P P' Q Q' C ct,
  (∀ y : nat,
     y < S N →
     ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
     judge y ct P C Q → sound ct P C Q) →
  implies P' P → implies Q Q' → judge N ct P C Q → sound ct P' C Q'.
Proof.
intros.
apply H in H2; auto.
destruct ct; inv H2.
```

```
apply Jden_lo; intros.
apply H3; apply H0; auto.
apply H4 in H9.
apply H1; auto.
apply H0; auto.
assert (aden2 P st1 st2).
inv H2; repeat (split; intuit).
apply (H5 _ _ _ _ _ _ _ _ _ _ H11 H9 H10).
assert (aden2 P st1 st2).
inv H2; repeat (split; intuit).
apply (H6 _ _ _ _ _ _ _ _ _ H12 H9 H10 H11).
apply H7 with (st2 := st2) in H9; auto.
inv H2; repeat (split; intuit).
assert (aden2 P (St i1 s1 h1) (St i2 s2 h2)).
inv H2; repeat (split; intuit).
apply (H8 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H13 H9 H10 H11 H12).

apply Jden_hi; intros.
apply H3; apply H0; auto.
apply H4 in H5.
apply H1; auto.
apply H0; auto.
Qed.

Lemma soundness_conj : ∀ N1 N2 P1 P2 Q1 Q2 C ct,
  (∀ y : nat,
     y < S (N1 + N2) →
     ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
     judge y ct P C Q → sound ct P C Q) →
  judge N1 ct P1 C Q1 → judge N2 ct P2 C Q2 → sound ct (P1 `AND` P2) C (Q1
`AND` Q2).
Proof.
intros.
apply H in H0; try omega.
apply H in H1; try omega.
destruct ct; inv H0; inv H1.
apply Jden_lo; intros.
apply H2.
destruct st; simpl in H1; intuit.
dup H13; apply H3 in H13.
apply H8 in H14.
destruct st'; simpl; intuit.
destruct st; simpl in H1; intuit.
destruct st; simpl in H1; intuit.
```

```
assert (aden2 P1 st1 st2).
destruct st1; destruct st2; inv H1; simpl in *; repeat (split; intuit).
apply (H4 _ _ _ _ _ _ _ _ _ H15 H13 H14).
assert (aden2 P1 st1 st2).
destruct st1; destruct st2; inv H1; simpl in *; repeat (split; intuit).
apply (H5 _ _ _ _ _ _ _ _ H16 H13 H14 H15).
apply H6 with (st2 := st2) in H13; auto.
destruct st1; destruct st2; inv H1; simpl in *; repeat (split; intuit).
assert (aden2 P1 (St i1 s1 h1) (St i2 s2 h2)).
inv H1; simpl in *; repeat (split; intuit).
apply (H7 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H17 H13 H14 H15 H16).

apply Jden_hi; intros.
apply H2.
destruct st; simpl in H1; intuit.
dup H5; apply H3 in H5.
apply H4 in H6.
destruct st'; simpl; intuit.
destruct st; simpl in H1; intuit.
destruct st; simpl in H1; intuit.
Qed.

Proposition aden2_star_inv : ∀ P Q i1 s1 h1 i2 s2 h2,
  aden2 (P**Q) (St i1 s1 h1) (St i2 s2 h2) →
  ∃ ha, ∃ hb, ∃ hc, ∃ hd,
    mydot ha hb h1 ∧ mydot hc hd h2 ∧ aden2 P (St i1 s1 ha) (St i2 s2 hc).
Proof.
intros.
inv H.
destruct H1; simpl in *.
destruct H0 as [ha [hb H0]]; decomp H0.
destruct H as [hc [hd H]]; decomp H.
inv H1.
destruct H3; simpl in H; subst.
∃ ha; ∃ hb; ∃ hc; ∃ hd; repeat (split; auto).
intro n; simpl.
case_eq (ha n); intros; auto.
destruct v as [v1 l1].
case_eq (hc n); intros; auto.
destruct v as [v2 l2]; intros; subst.
specialize (H2 n); rewrite H in H2.
specialize (H0 n); rewrite H8 in H0.
specialize (H3 n); simpl in H3.
destruct (h1 n) as [[v3 l3]|].
```

```
decomp H2.
inv H10; destruct (h2 n) as [[v4 l4]||].
decomp H0.
inv H9; apply H3; auto.
inv H9.
destruct H0; inv H0.
inv H10.
destruct H2; inv H2.
Qed.

Proposition mydot_comm {A} : ∀ h1 h2 h3, mydot(A:=A) h1 h2 h3 → mydot h2 h1 h3.
Proof.
intros; intro n.
specialize (H n).
destruct (h1 n); destruct (h2 n); destruct (h3 n); intuit.
Qed.

Proposition obs_eq_mydot_inv : ∀ ha hb hc hd h1 h2,
  mydot ha hb h1 → mydot hc hd h2 → obs_eq_h h1 h2 → obs_eq_h ha hc.
Proof.
intros; intro n.
specialize (H n); specialize (H0 n); specialize (H1 n).
destruct (ha n) as [[va la]||]; auto.
destruct (hc n) as [[vc lc]||]; auto.
destruct (h1 n) as [[v1 l1]||].
decomp H.
inv H3; destruct (h2 n) as [[v2 l2]||].
decomp H0.
inv H2; auto.
inv H2.
destruct H0 as [H0]; inv H0.
inv H3.
destruct H; inv H.
Qed.

Lemma soundness_frame : ∀ N P Q R C ct,
  (∀ y : nat,
     y < S N →
     ∀ (ct : context) (P : assert) (C : cmd) (Q : assert),
     judge y ct P C Q → sound ct P C Q) →
  judge N ct P C Q → (∀ x, In x (modifies [C]) → vars R x = false) → sound ct (P **
R) C (Q ** R).
Proof.
intros.
apply H in H0; auto.
```

```
destruct ct; inv H0.
apply Jden_lo; intros.
unfold lsafe; intros.
destruct st as [i s h]; simpl in H0; destruct H0 as [h1 [h2 H0]]; decomp H0.
destruct cf' as [[i' s' h'] C' K']; apply lstepn_bf with (h1 := h1) (h2 := h2) in H8;
auto.
destruct H8 as [h1' [H8]].
apply H2 in H0; auto.
specialize (H0 H9); inv H0.
destruct cf' as [[i'' s'' h''] C'' K''].
apply lstep_ff with (h2 := h2) (h3 := h') in H11; auto.
destruct H11 as [h3' [H11]].
apply (Can_lstep _ (Cf (St i'' s'' h3') C'' K'') o0); auto.
destruct st as [i s h]; simpl in H0; destruct H0 as [h1 [h2 H0]]; decomp H0.
destruct st' as [i' s' h']; apply lstepn_bf with (h1 := h1) (h2 := h2) in H8; auto.
destruct H8 as [h1' [H8]].
dup H0; apply lstepn_i_const in H0; subst.
assert (aden R (St i s' h2)).
apply aden_vars_same with (s := s); auto; intros.
apply lstepn_modifies_const with (x := x) in H10; auto; intro.
apply H1 in H13; rewrite H13 in H0; inv H0.
apply H3 in H10; auto.
∃ h1'; ∃ h2; repeat (split; auto).
destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2]; dup H0; apply aden2_star_inv in
H0.
destruct H0 as [ha [hb [hc [hd H0]]]]; decomp H0.
destruct st1' as [i1' s1' h1']; apply lstepn_bf with (h1 := ha) (h2 := hb) in H8; auto.
destruct st2' as [i2' s2' h2']; apply lstepn_bf with (h1 := hc) (h2 := hd) in H9; auto.
destruct H8 as [ha' [H8]]; destruct H9 as [hc' [H9]].
decomp (H4 _ _ _ _ _ _ _ _ _ H14 H0 H12).
left; intro n'.
destruct (H15 n') as [[[i s h] C1 K1] [o]].
apply lstepn_ff with (h2 := hb) (h3 := h1) in H16; auto.
destruct H16 as [h3' [H16]].
∃ (Cf (St i s h3') C1 K1); ∃ o; auto.
right; left; intro n'.
destruct (H16 n') as [[[i s h] C2 K2] [o]].
apply lstepn_ff with (h2 := hd) (h3 := h2) in H15; auto.
destruct H15 as [h3' [H15]].
∃ (Cf (St i s h3') C2 K2); ∃ o; auto.
right; right.
destruct C'; auto; simpl in H16 ⊢ ×.
```

```
destruct (eden e i1' s1') as [[v1' l1']||]; auto.
destruct (eden e i2' s2') as [[v2' l2']||]; auto.
destruct (Zneg_dec v1'); auto.
destruct (Zneg_dec v2'); auto.
specialize (H8 (nat_of_Z v1' g)).
destruct (h1' (nat_of_Z v1' g)) as [[v1'' l1'']||]; auto.
specialize (H9 (nat_of_Z v2' g0)).
destruct (h2' (nat_of_Z v2' g0)) as [[v2'' l2'']||]; auto.
decomp H8.
decomp H9.
rewrite H17 in H16; rewrite H15 in H16; auto.
rewrite H17 in H16; rewrite H15 in H16; inv H16.
rewrite H17 in H16; inv H16.
destruct H9; rewrite H9 in H16.
destruct (ha' (nat_of_Z v1' g)) as [[va la]||]; inv H16.
destruct H8; rewrite H8 in H16; inv H16.
apply H2; inv H14; intuit.
apply H2; inv H14; intuit.
destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2]; dup H0; apply aden2_star_inv in
H0.
destruct H0 as [ha [hb [hc [hd H0]]]]; decomp H0.
destruct st1' as [i1' s1' h1']; apply lstepn_bf with (h1 := ha) (h2 := hb) in H9; auto.
destruct st2' as [i2' s2' h2']; apply lstepn_bf with (h1 := hc) (h2 := hd) in H10; auto.
destruct H9 as [ha' [H9]]; destruct H10 as [hc' [H10]].
assert (side_condition C (St i1 s1 ha) (St i2 s2 hc)).
decomp (H4 _ _ _ _ _ _ _ _ _ H15 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H16 H0)).
apply (False_ind _ (diverge_halt _ _ _ _ H17 H13)).
destruct (H5 _ _ _ _ _ _ _ _ H15 H16 H0 H13); subst; split; auto.
inv H17.
destruct H19; simpl in H18; subst.
repeat (split; auto).
simpl in H19 ⊢ *; assert (obs_eq_h h1 h2).
inv H11.
destruct H20.
inv H20; intuit.
intro n; specialize (H9 n); specialize (H10 n).
destruct (h1' n) as [[v1' l1']||]; auto.
destruct (h2' n) as [[v2' l2']||]; auto; intros; subst.
decomp H9.
decomp H10.
specialize (H19 n); rewrite H21 in H19; rewrite H20 in H19; auto.
```

```
destruct (opt_eq_dec val_eq_dec (ha n) (ha' n)).
apply mydot_comm in H14.
dup (obs_eq_mydot_inv _ _ _ _ _ _ H12 H14 H18).
rewrite ← e in H21; specialize (H9 n).
rewrite H21 in H9; rewrite H23 in H9; auto.
assert (∃ v, ha' n = Some (v,Lo)).
∃ v1'; auto.
dup (H7 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H15 H0 H13 n0 H9).
contradiction H10; specialize (H14 n).
rewrite H23 in H14; destruct (h2 n); destruct (hc n); auto.
decomp H14.
inv H26.
inv H25.
destruct H14; inv H14.
decomp H10.
destruct (opt_eq_dec val_eq_dec (hc n) (hc' n)).
apply mydot_comm in H12.
dup (obs_eq_mydot_inv _ _ _ _ _ _ H12 H14 H18).
rewrite ← e in H20; specialize (H9 n).
rewrite H20 in H9; rewrite H22 in H9; auto.
assert (∃ v, hc' n = Some (v,Lo)).
∃ v2'; auto.
assert (aden2 P (St i2 s2 hc) (St i1 s1 ha)).
inv H15.
destruct H24.
apply obs_eq_sym in H24; repeat (split; auto).
dup (H7 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H10 H13 H0 n0 H9).
contradiction H24; specialize (H12 n).
rewrite H22 in H12; destruct (h1 n); destruct (ha n); auto.
decomp H12.
inv H27.
inv H26.
destruct H12; inv H12.
apply mydot_comm in H12; apply mydot_comm in H14.
dup (obs_eq_mydot_inv _ _ _ _ _ _ H12 H14 H18).
specialize (H9 n); rewrite H22 in H9; rewrite H23 in H9; auto.
apply H2; inv H15; intuit.
apply H2; inv H15; intuit.
destruct st1 as [i1 s1 h1]; destruct st2 as [i2 s2 h2]; dup H0; apply aden2_star_inv in
H0.
destruct H0 as [ha [hb [hc [hd H0]]]]; decomp H0.
destruct st1' as [i1' s1' h1']; apply lstepn_bf with (h1 := ha) (h2 := hb) in H8; auto.
```

```
destruct H8 as [ha' [H8]].
decomp (H6 _ _ _ _ _ _ _ H13 H0).
left; intro n'.
destruct (H11 n') as [[[i s h] C1 K1] [o]].
apply lstepn_ff with (h2 := hb) (h3 := h1) in H14; auto.
destruct H14 as [h3' [H14]].
∃ (Cf (St i s h3') C1 K1); ∃ o; auto.
right; left; intro n'.
destruct (H14 n') as [[[i s h] C2 K2] [o]].
apply lstepn_ff with (h2 := hd) (h3 := h2) in H11; auto.
destruct H11 as [h3' [H11]].
∃ (Cf (St i s h3') C2 K2); ∃ o; auto.
right; right; destruct H14 as [[i2' s2' hc'] [o2]].
apply lstepn_ff with (h2 := hd) (h3 := h2) in H11; auto.
destruct H11 as [h3' [H11]].
∃ (St i2' s2' h3'); ∃ o2; auto.
apply H2; inv H13; intuit.
dup H0; apply aden2_star_inv in H0.
destruct H0 as [ha [hb [hc [hd H0]]]]; decomp H0.
apply lstepn_bf with (h1 := ha) (h2 := hb) in H8; auto.
apply lstepn_bf with (h1 := hc) (h2 := hd) in H9; auto.
destruct H8 as [ha' [H8]]; destruct H9 as [hc' [H9]].
assert (hc a ≠ None).
apply (H7 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ H16 H0 H14).
intro; contradiction H10.
specialize (H13 a); specialize (H8 a).
destruct H11 as [v]; rewrite H11 in H8 ⊢ *; decomp H8.
rewrite H19 in H17; rewrite H17 in H13.
destruct (h1 a); decomp H13; auto.
inv H18.
rewrite H19 in H17; rewrite H17 in H13.
destruct (h1 a); decomp H13.
inv H18.
rewrite H21 in H20; auto.
rewrite H18 in H20; inv H20.
specialize (H8 a); destruct H11 as [v]; rewrite H11 in H8; decomp H8.
∃ v; auto.
contradiction H10; rewrite H11.
specialize (H13 a); destruct (h1 a).
decomp H13.
rewrite H20 in H19; inv H19.
rewrite H20 in H19; auto.
```

```
destruct H13.
rewrite H19 in H13; inv H13.
intro; contradiction H17; specialize (H15 a).
rewrite H18 in H15; intuit.
apply H2; inv H16; intuit.
apply H2; inv H16; intuit.

apply Jden_hi; intros.
unfold hsafe; intros.
destruct st as [i s h]; simpl in H0; destruct H0 as [h1 [h2 H0]]; decomp H0.
destruct cf' as [[i' s' h'] C' K']; apply hstepn_bf with (h1 := h1) (h2 := h2) in H4;
auto.
destruct H4 as [h1' [H4]].
apply H2 in H0; auto.
specialize (H0 H5); inv H0.
destruct cf' as [[i'' s'' h''] C'' K''].
apply hstep_ff with (h2 := h2) (h3 := h') in H7; auto.
destruct H7 as [h3' [H7]].
apply (Can_hstep _ (Cf (St i'' s'' h3') C'' K'')); auto.
destruct st as [i s h]; simpl in H0; destruct H0 as [h1 [h2 H0]]; decomp H0.
destruct st' as [i' s' h']; apply hstepn_bf with (h1 := h1) (h2 := h2) in H4; auto.
destruct H4 as [h1' [H4]].
dup H0; apply hstepn_i_const in H0; subst.
assert (aden R (St i s' h2)).
apply aden_vars_same with (s := s); auto; intros.
apply hstepn_modifies_const with (x := x) in H6; auto; intro.
apply H1 in H9; rewrite H9 in H0; inv H0.
apply H3 in H6; auto.
∃ h1'; ∃ h2; repeat (split; auto).
Qed.

Theorem soundness : ∀ N ct P C Q, judge N ct P C Q → sound ct P C Q.
Proof.
induction N using (well_founded_induction lt_wf); intros.
inv H0.
apply soundness_skip.
apply soundness_output.
apply soundness_assign; auto.
apply soundness_read; auto.
apply soundness_write.
apply soundness_seq with (N1 := N1) (N2 := N2) (Q := Q0); auto.
apply soundness_if with (N1 := N1) (N2 := N2) (lt0 := lt) (lf := lf); auto.
apply soundness_while with (N := N0); auto.
apply soundness_conseq with (N := N0) (P := P0) (Q := Q0); auto.
```

```
apply soundness_conj with (N1 := N1) (N2 := N2); auto.
apply soundness_frame with (N := N0); auto.
Qed.

Definition store' := var → option Z.
Definition heap' := addr → option Z.
Inductive state' := St' : store' → heap' → state'.
Inductive config' := Cf' : state' → cmd → list cmd → config'.

Definition erase (f : nat → option val) : nat → option Z := fun x ⇒ option_map (fun
v ⇒ fst v) (f x).
Definition erase_fill (f : nat → option val) : nat → option Z :=
  fun x ⇒ match f x with Some v ⇒ Some (fst v) | None ⇒ Some 0%Z end.
Definition erase_st (st : state) : state' := St' (erase_fill (st:store)) (erase (st:heap)).

Proposition erase_upd : ∀ f x v l, erase (upd f x (v,l)) = upd (erase f) x v.
Proof.
intros; extensionality n.
unfold erase; unfold upd.
destruct (eq_nat_dec n x); auto.
Qed.

Proposition erase_fill_upd : ∀ f x v l, erase_fill (upd f x (v,l)) = upd (erase_fill f) x v.
Proof.
intros; extensionality n.
unfold erase_fill; unfold upd.
destruct (eq_nat_dec n x); auto.
Qed.

Fixpoint eden' (e : exp) (s : store') : option Z :=
  match e with
  | Var x ⇒ s x
  | LVar _ ⇒ None
  | Num n ⇒ Some n
  | BinOp op e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ opden op v1 v2) (eden' e1 s) (eden' e2
s)
  end.

Fixpoint bden' (b : bexp) (s : store') : option bool :=
  match b with
  | FF ⇒ Some false
  | TT ⇒ Some true
  | Eq e1 e2 ⇒ option_map2 (fun v1 v2 ⇒ if Z_eq_dec v1 v2 then true else false) (eden'
e1 s) (eden' e2 s)
  | Not b ⇒ option_map (fun v ⇒ negb v) (bden' b s)
  | BBinOp bop b1 b2 ⇒ option_map2 (fun v1 v2 ⇒ bopden bop v1 v2) (bden' b1 s) (bden'
b2 s)
```

```
  end.
Proposition eden_erase : ∀ e i s, no_lvars_exp e → edenZ e i s ≠ None → eden' e
(erase_fill s) = edenZ e i s.
Proof.
induction e; simpl; intros; intuit.
unfold erase_fill; destruct (s v); auto.
contradiction H0; auto.
rewrite IHe1 with (i := i); intuit.
rewrite IHe2 with (i := i); intuit.
intro H1; contradiction H0; rewrite H1; destruct (edenZ e1 i s); auto.
intro H1; contradiction H0; rewrite H1; auto.
Qed.
Proposition bden_erase : ∀ b i s, no_lvars_bexp b → bdenZ b i s ≠ None → bden' b
(erase_fill s) = bdenZ b i s.
Proof.
induction b; simpl; intros; intuit.
rewrite eden_erase with (i := i); intuit.
rewrite eden_erase with (i := i); intuit.
intro H1; contradiction H0; rewrite H1; destruct (edenZ e i s); auto.
intro H1; contradiction H0; rewrite H1; auto.
rewrite IHb with (i := i); auto.
intro H1; contradiction H0; rewrite H1; auto.
rewrite IHb1 with (i := i); intuit.
rewrite IHb2 with (i := i); intuit.
intro H1; contradiction H0; rewrite H1; destruct (bdenZ b2 i s); auto.
intro H1; contradiction H0; rewrite H1; auto.
Qed.
Proposition erase_taint_vars : ∀ K s, erase_fill (taint_vars K s) = erase_fill s.
Proof.
intros; extensionality x.
unfold erase_fill; unfold taint_vars.
destruct (In_dec eq_nat_dec x (modifies K)); auto.
destruct (s x) as [[v l]|]; auto.
Qed.
Open Scope Z_scope.
Inductive step : config' → config' → list Z → Prop :=
| Step_skip : ∀ st C K, step (Cf' st Skip (C::K)) (Cf' st C K) []
| Step_output : ∀ s h K e v, eden' e s = Some v →
    step (Cf' (St' s h) (Output e) K) (Cf' (St' s h) Skip K) [v]
| Step_assign : ∀ s h K x e v, eden' e s = Some v →
    step (Cf' (St' s h) (Assign x e) K) (Cf' (St' (upd s x v) h) Skip K) []
```

```
| Step_read : ∀ s h K x e v1 v2 (pf : v1 ≥ 0), eden' e s = Some v1 → h (nat_of_Z v1 pf)
= Some v2 →
    step (Cf' (St' s h) (Read x e) K) (Cf' (St' (upd s x v2) h) Skip K) []
| Step_write : ∀ s h K e1 e2 v1 v2 (pf : v1 ≥ 0), eden' e1 s = Some v1 →
    eden' e2 s = Some v2 → h (nat_of_Z v1 pf) ≠ None →
    step (Cf' (St' s h) (Write e1 e2) K) (Cf' (St' s (upd h (nat_of_Z v1 pf) v2)) Skip
K) []
| Step_seq : ∀ st C1 C2 K, step (Cf' st (Seq C1 C2) K) (Cf' st C1 (C2::K)) []
| Step_if_true : ∀ s h C1 C2 K b, bden' b s = Some true →
    step (Cf' (St' s h) (If b C1 C2) K) (Cf' (St' s h) C1 K) []
| Step_if_false : ∀ s h C1 C2 K b, bden' b s = Some false →
    step (Cf' (St' s h) (If b C1 C2) K) (Cf' (St' s h) C2 K) []
| Step_while_true : ∀ s h C K b, bden' b s = Some true →
    step (Cf' (St' s h) (While b C) K) (Cf' (St' s h) C (While b C :: K)) []
| Step_while_false : ∀ s h C K b, bden' b s = Some false →
    step (Cf' (St' s h) (While b C) K) (Cf' (St' s h) Skip K) [].

Close Scope Z_scope.

Inductive stepn : nat → config' → config' → list Z → Prop :=
| Step_zero : ∀ cf, stepn 0 cf cf []
| Step_succ : ∀ n cf cf' cf'' o o', step cf cf' o → stepn n cf' cf'' o' → stepn (S n) cf cf''
(o++o').

Lemma step_trans : ∀ n1 n2 cf1 cf2 cf3 o1 o2, stepn n1 cf1 cf2 o1 → stepn n2 cf2 cf3 o2
→ stepn (n1+n2) cf1 cf3 (o1++o2).
Proof.
induction n1 using (well_founded_induction lt_wf); intros.
inv H0; simpl; auto.
rewrite app_assoc; apply Step_succ with (cf' := cf'); auto.
apply H with (cf2 := cf2); auto.
Qed.

Lemma step_extend : ∀ st C K st' C' K' K0 o,
  step (Cf' st C K) (Cf' st' C' K') o → step (Cf' st C (K++K0)) (Cf' st' C' (K'++K0))
o.
Proof.
intros.
inv H.
apply Step_skip.
apply Step_output; auto.
apply Step_assign; auto.
apply Step_read with (v1 := v1) (pf := pf); auto.
apply Step_write; auto.
apply Step_seq.
```

```
apply Step_if_true; auto.
apply Step_if_false; auto.
apply Step_while_true; auto.
apply Step_while_false; auto.
Qed.

Lemma stepn_extend : ∀ n st C K st' C' K' K0 o,
  stepn n (Cf' st C K) (Cf' st' C' K') o → stepn n (Cf' st C (K++K0)) (Cf' st' C'
(K'++K0)) o.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
apply Step_zero.
destruct cf' as [st'' C'' K''].
apply Step_succ with (cf' := Cf' st'' C'' (K''++K0)).
apply step_extend; auto.
apply H; auto.
Qed.

Lemma step_trans_inv : ∀ n st st' C C' K0 K K' o,
  stepn n (Cf' st C (K0++K)) (Cf' st' C' K') o →
  (∃ K'', stepn n (Cf' st C K0) (Cf' st' C' K'') o ∧ K' = K''++K) ∨
  ∃ st'', ∃ n1, ∃ n2, ∃ o1, ∃ o2,
    stepn n1 (Cf' st C K0) (Cf' st'' Skip []) o1 ∧ stepn n2 (Cf' st'' Skip K) (Cf' st' C'
K') o2 ∧
    n = n1 + n2 ∧ o = o1 ++ o2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
left; ∃ K0.
split; auto; apply Step_zero.
inv H1.

destruct K0.
simpl in H5; subst.
right; ∃ st; ∃ 0; ∃ (S n0); ∃ []; ∃ ([]++o'); repeat (split; auto).
apply Step_zero.
apply Step_succ with (cf' := Cf' st C0 K1); auto.
apply Step_skip.
inv H5.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' st c K0); auto.
```

```
apply Step_skip.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' st c K0); auto.
apply Step_skip.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' s h) Skip K0); auto.
apply Step_output; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([v]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s h) Skip K0); auto.
apply Step_output; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' (upd s x v) h) Skip K0); auto.
apply Step_assign; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' (upd s x v) h) Skip K0); auto.
apply Step_assign; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' (upd s x v2) h) Skip K0); auto.
apply Step_read with (v1 := v1) (pf := pf); auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' (upd s x v2) h) Skip K0); auto.
apply Step_read with (v1 := v1) (pf := pf); auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' s (upd h (nat_of_Z v1 pf) v2)) Skip K0); auto.
apply Step_write; auto.
```

```
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s (upd h (nat_of_Z v1 pf) v2)) Skip K0); auto.
apply Step_write; auto.

change (stepn n0 (Cf' st C1 ((C2 :: K0) ++ K)) (Cf' st' C' K') o') in H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' st C1 (C2::K0)); auto.
apply Step_seq.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' st C1 (C2::K0)); auto.
apply Step_seq.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' s h) C1 K0); auto.
apply Step_if_true; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s h) C1 K0); auto.
apply Step_if_true; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' s h) C2 K0); auto.
apply Step_if_false; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s h) C2 K0); auto.
apply Step_if_false; auto.

change (stepn n0 (Cf' (St' s h) C0 ((While b C0 :: K0) ++ K)) (Cf' st' C' K') o') in
H2.
apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
```

```
apply Step_succ with (cf' := Cf' (St' s h) C0 (While b C0 :: K0)); auto.
apply Step_while_true; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s h) C0 (While b C0 :: K0)); auto.
apply Step_while_true; auto.

apply H in H2; auto.
destruct H2.
destruct H0 as [K'' [H0]]; subst.
left; ∃ K''; split; auto.
apply Step_succ with (cf' := Cf' (St' s h) Skip K0); auto.
apply Step_while_false; auto.
destruct H0 as [st'' [n1 [n2 [o1 [o2 [H0 [H1 [H2]]]]]]]]; subst.
right; ∃ st''; ∃ (S n1); ∃ n2; ∃ ([]++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := Cf' (St' s h) Skip K0); auto.
apply Step_while_false; auto.
Qed.

Lemma step_trans_inv' : ∀ a b cf cf' o, stepn (a + b) cf cf' o →
  ∃ cf'', ∃ o1, ∃ o2, stepn a cf cf'' o1 ∧ stepn b cf'' cf' o2 ∧ o = o1 ++ o2.
Proof.
induction a using (well_founded_induction lt_wf); intros.
inv H0.
assert (a = 0); try omega.
assert (b = 0); try omega; subst.
∃ cf'; ∃ []; ∃ []; repeat (split; auto); apply Step_zero.
destruct a; simpl in H1; subst.
∃ cf; ∃ []; ∃ (o0++o'); repeat (split; auto).
apply Step_zero.
apply Step_succ with (cf' := cf'0); auto.
inv H1.
apply H in H3; auto.
destruct H3 as [cf'' [o1 [o2 [H3 [H4]]]]]; ∃ cf''; ∃ (o0++o1); ∃ o2; repeat (split; auto).
apply Step_succ with (cf' := cf'0); auto.
subst; rewrite app_assoc; auto.
Qed.

Lemma step_det : ∀ cf cf1 cf2 o1 o2, step cf cf1 o1 → step cf cf2 o2 → cf1 = cf2 ∧ o1 =
o2.
Proof.
intros.
inv H.
inv H0; auto.
inv H0.
```

```
rewrite H7 in H1; inv H1; auto.
inv H0.
rewrite H8 in H1; inv H1; auto.
inv H0.
rewrite H9 in H1; inv H1.
rewrite (proof_irrelevance _ pf0 pf) in H10; rewrite H10 in H2; inv H2; auto.
inv H0.
rewrite H10 in H1; inv H1; rewrite H11 in H2; inv H2.
rewrite (proof_irrelevance _ pf0 pf); auto.
inv H0; auto.
inv H0; auto.
rewrite H9 in H1; inv H1.
inv H0; auto.
rewrite H9 in H1; inv H1.
inv H0; auto.
rewrite H8 in H1; inv H1.
inv H0; auto.
rewrite H8 in H1; inv H1.
Qed.

Lemma stepn_det : ∀ n cf cf1 cf2 o1 o2, stepn n cf cf1 o1 → stepn n cf cf2 o2 → cf1 =
cf2 ∧ o1 = o2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0; inv H1; auto.
destruct (step_det _ _ _ _ _ H2 H4); subst.
assert (n0 < S n0); try omega.
destruct (H _ H0 _ _ _ _ _ H3 H5); subst; auto.
Qed.

Lemma step_output_inv : ∀ n st st' C C' K K' v,
  stepn n (Cf' st C K) (Cf' st' C' K') [v] →
  ∃ n1, ∃ n2, ∃ st'', ∃ e, ∃ K'',
    stepn n1 (Cf' st C K) (Cf' st'' (Output e) K'') [] ∧
    stepn n2 (Cf' st'' (Output e) K'') (Cf' st' C' K') [v] ∧ n = n1 + n2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
destruct cf' as [st'' C'' K''].
destruct o; inv H1.
simpl in H0; subst.
apply H in H6; auto.
destruct H6 as [n1 [n2 [st''' [e [K''' H6]]]]]; decomp H6; subst.
∃ (S n1); ∃ n2; ∃ st'''; ∃ e; ∃ K'''; intuition.
```

```
rewrite ← app_nil_r; apply Step_succ with (cf' := Cf' st'' C'' K''); auto.
destruct o; inv H4.
destruct o'; inv H0.
inv H2.
∃ 0; ∃ (S n0); ∃ (St' s h); ∃ e; ∃ K''; intuition.
apply Step_zero.
apply Step_succ with (cf' := Cf' (St' s h) Skip K''); auto.
apply Step_output; auto.
Qed.

Lemma step_output_inv' : ∀ n st st' C C' K K' o1 o2,
  stepn n (Cf' st C K) (Cf' st' C' K') (o1++o2) →
  ∃ n1, ∃ n2, ∃ st'', ∃ C'', ∃ K'',
    stepn n1 (Cf' st C K) (Cf' st'' C'' K'') o1 ∧
    stepn n2 (Cf' st'' C'' K'') (Cf' st' C' K') o2 ∧ n = n1 + n2.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H0.
destruct o1; inv H7.
destruct o2; inv H0.
∃ 0; ∃ 0; ∃ st'; ∃ C'; ∃ K'; intuition; apply Step_zero.
destruct cf' as [st'' C'' K''].
destruct o.
simpl in H1; subst; apply H in H6; auto.
destruct H6 as [n1 [n2 [st''' [C''' [K''' H6]]]]]; decomp H6; subst.
∃ (S n1); ∃ n2; ∃ st'''; ∃ C'''; ∃ K'''; intuition.
fold ([]++o1); apply Step_succ with (cf' := Cf' st'' C'' K''); auto.
dup H2; inv H2.
destruct o1; simpl in H1; subst.
∃ 0; ∃ (S n0); ∃ (St' s h); ∃ (Output e); ∃ K''; intuition.
apply Step_zero.
fold ([z]++o'); apply Step_succ with (cf' := Cf' (St' s h) Skip K''); auto.
inv H1.
apply H in H6; auto.
destruct H6 as [n1 [n2 [st''' [C''' [K''' H6]]]]]; decomp H6; subst.
∃ (S n1); ∃ n2; ∃ st'''; ∃ C'''; ∃ K'''; intuition.
fold ([z0]++o1); apply Step_succ with (cf' := Cf' (St' s h) Skip K''); auto.
Qed.

Proposition hstep_no_lvars_monotonic : ∀ st st' C C' K K',
  hstep (Cf st C K) (Cf st' C' K') → no_lvars (C::K) → no_lvars (C'::K').
Proof.
intros.
inv H; simpl in *; intuit.
```

```
Qed.

Proposition hstepn_no_lvars_monotonic : ∀ n st st' C C' K K',
  hstepn n (Cf st C K) (Cf st' C' K') → no_lvars (C::K) → no_lvars (C'::K').
Proof.
induction n; intros.
inv H; auto.
inv H.
destruct cf' as [st'' C'' K''].
apply hstep_no_lvars_monotonic in H2; auto.
apply IHn in H3; auto.
Qed.

Proposition lstep_no_lvars_monotonic : ∀ st st' C C' K K' o,
  lstep (Cf st C K) (Cf st' C' K') o → no_lvars (C::K) → no_lvars (C'::K').
Proof.
intros.
inv H; simpl in *; intuit.
Qed.

Proposition lstepn_no_lvars_monotonic : ∀ n st st' C C' K K' o,
  lstepn n (Cf st C K) (Cf st' C' K') o → no_lvars (C::K) → no_lvars (C'::K').
Proof.
induction n; intros.
inv H; auto.
inv H.
destruct cf' as [st'' C'' K''].
apply lstep_no_lvars_monotonic in H2; auto.
apply IHn in H3; auto.
Qed.

Lemma hstep_erase : ∀ st st' C C' K K', no_lvars (C::K) → hstep (Cf st C K) (Cf st' C'
K') →
  ∃ n, stepn n (Cf' (erase_st st) C K) (Cf' (erase_st st') C' K') [].
Proof.
intros.
inv H0; simpl in H.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st st') C' K').
apply Step_skip.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i (upd s x (v,Hi)) h)) Skip K').
unfold erase_st; simpl.
rewrite erase_fill_upd; apply Step_assign.
```

```
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H2; inv H2.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i (upd s x (v2,Hi)) h)) Skip K').
unfold erase_st; simpl.
rewrite erase_fill_upd; apply Step_read with (v1 := v1) (pf := pf).
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l1; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H3; inv H3.
unfold erase; rewrite H8; auto.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s (upd h (nat_of_Z v1 pf) (v2,Hi)))) Skip
K').
unfold erase_st; simpl.
rewrite erase_upd; apply Step_write.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l1; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H5; inv H5.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l2; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
unfold erase; destruct (h (nat_of_Z v1 pf)); auto; discriminate.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st st') C' (C2::K)).
apply Step_seq.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' K').
apply Step_if_true.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ l; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H2; inv H2.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' K').
apply Step_if_false.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ l; auto.
```

```
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H2; inv H2.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' (While b C' :: K)).
apply Step_while_true.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ l; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H2; inv H2.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) Skip K').
apply Step_while_false.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ l; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H2; inv H2.
apply Step_zero.
Qed.

Lemma hstepn_erase : ∀ n st st' C C' K K', no_lvars (C::K) → hstepn n (Cf st C K) (Cf
st' C' K') →
  ∃ n', stepn n' (Cf' (erase_st st) C K) (Cf' (erase_st st') C' K') [].
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H1.
∃ 0; apply Step_zero.
destruct cf' as [st'' C'' K''].
apply H in H3; auto.
apply hstep_erase in H2; auto.
destruct H2 as [n1]; destruct H3 as [n2]; ∃ (n1+n2).
rewrite ← app_nil_r; apply step_trans with (cf2 := Cf' (erase_st st'') C'' K''); auto.
apply hstep_no_lvars_monotonic in H2; auto.
Qed.

Lemma lstep_erase : ∀ st st' C C' K K' o, no_lvars (C::K) → lstep (Cf st C K) (Cf st'
C' K') o →
  ∃ n, stepn n (Cf' (erase_st st) C K) (Cf' (erase_st st') C' K') o.
Proof.
intros.
inv H0; simpl in H.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st st') C' K').
apply Step_skip.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
```

```
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) Skip K').
apply Step_output.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ Lo; auto.
simpl; intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i (upd s x (v,l)) h)) Skip K').
unfold erase_st; simpl.
rewrite erase_fill_upd; apply Step_assign.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i (upd s x (v2,l1 \_/ l2)) h)) Skip K').
unfold erase_st; simpl.
rewrite erase_fill_upd; apply Step_read with (v1 := v1) (pf := pf).
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l1; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
unfold erase; rewrite H9; auto.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s (upd h (nat_of_Z v1 pf) (v2,l1 \_/
l2)))) Skip K').
unfold erase_st; simpl.
rewrite erase_upd; apply Step_write.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l1; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
rewrite eden_erase with (i := i); intuit.
rewrite edenZ_some; ∃ l2; auto.
intro H0; rewrite edenZ_none in H0; rewrite H0 in H9; inv H9.
unfold erase; destruct (h (nat_of_Z v1 pf)); auto; discriminate.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st st') C' (C2::K)).
apply Step_seq.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' K').
```

```
apply Step_if_true.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ Lo; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' K').
apply Step_if_false.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ Lo; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) C' (While b C' :: K)).
apply Step_while_true.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ Lo; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
∃ 1; rewrite ← app_nil_r.
apply Step_succ with (cf' := Cf' (erase_st (St i s h)) Skip K').
apply Step_while_false.
rewrite bden_erase with (i := i); intuit.
rewrite bdenZ_some; ∃ Lo; auto.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply Step_zero.
apply hstepn_erase in H10; simpl; intuit.
destruct H10 as [n']; ∃ n'.
unfold erase_st in H0 ⊢ *; simpl in H0 ⊢ ×.
rewrite erase_taint_vars in H0.
change (stepn n' (Cf' (St' (erase_fill s) (erase h)) (If b C1 C2) ([]++K'))
                 (Cf' (St' (erase_fill (st':store)) (erase (st':heap))) Skip ([]++K')) []).
apply stepn_extend; auto.
∃ 0; apply Step_zero.
apply hstepn_erase in H10; simpl; intuit.
destruct H10 as [n']; ∃ n'.
unfold erase_st in H0 ⊢ *; simpl in H0 ⊢ ×.
rewrite erase_taint_vars in H0.
change (stepn n' (Cf' (St' (erase_fill s) (erase h)) (While b C0) ([]++K'))
                 (Cf' (St' (erase_fill (st':store)) (erase (st':heap))) Skip ([]++K')) []).
apply stepn_extend; auto.
∃ 0; apply Step_zero.
```

```
Qed.

Lemma lstepn_erase : ∀ n st st' C C' K K' o, no_lvars (C::K) → lstepn n (Cf st C K) (Cf
st' C' K') o →
  ∃ n', stepn n' (Cf' (erase_st st) C K) (Cf' (erase_st st') C' K') o.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H1.
∃ 0; apply Step_zero.
destruct cf' as [st'' C'' K''].
apply H in H3; auto.
apply lstep_erase in H2; auto.
destruct H2 as [n1]; destruct H3 as [n2]; ∃ (n1+n2).
apply step_trans with (cf2 := Cf' (erase_st st'') C'' K''); auto.
apply lstep_no_lvars_monotonic in H2; auto.
Qed.

Theorem step_erase : ∀ n st st' C o, no_lvars_cmd C → lstepn n (Cf st C []) (Cf st' Skip
[]) o →
  ∃ n', stepn n' (Cf' (erase_st st) C []) (Cf' (erase_st st') Skip []) o.
Proof.
intros.
apply lstepn_erase in H0; simpl; auto.
Qed.

Lemma hstep_instrument : ∀ st est C C' K K' o, no_lvars (C::K) →
  step (Cf' (erase_st st) C K) (Cf' est C' K') o → hsafe (Cf st C K) →
  ∃ n, ∃ st', hstepn n (Cf st C K) (Cf st' C' K') ∧ est = erase_st st'.
Proof.
intros.
inv H0; simpl in H.
∃ 1; ∃ st; split; auto.
apply HStep_succ with (cf' := Cf st C' K').
apply HStep_skip.
apply HStep_zero.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
destruct st as [i s h].
∃ 1; ∃ (St i (upd s x (v,Hi)) h); split.
apply HStep_succ with (cf' := Cf (St i (upd s x (v,Hi)) h) Skip K').
rewrite eden_erase with (i := i) in H10; intuit.
rewrite edenZ_some in H10; destruct H10 as [l].
apply HStep_assign with (l := l); auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
```

```
simpl; intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
apply HStep_zero.
unfold erase_st; simpl; rewrite erase_fill_upd; auto.
destruct st as [i s h].
∃ 1; ∃ (St i (upd s x (v2,Hi)) h); split.
apply HStep_succ with (cf' := Cf (St i (upd s x (v2,Hi)) h) Skip K').
rewrite eden_erase with (i := i) in H10; intuit.
rewrite edenZ_some in H10; destruct H10 as [l1].
unfold erase in H11; simpl in H11; case_eq (h (nat_of_Z v1 pf)); intros.
destruct v as [v2' l2].
apply HStep_read with (v1 := v1) (pf := pf) (l1 := l1) (l2 := l2); auto.
rewrite H2 in H11; inv H11; auto.
rewrite H2 in H11; inv H11.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H0; rewrite edenZ_none in H0; rewrite H0 in H8; inv H8.
apply HStep_zero.
unfold erase_st; simpl; rewrite erase_fill_upd; auto.
destruct st as [i s h].
∃ 1; ∃ (St i s (upd h (nat_of_Z v1 pf) (v2,Hi))); split.
apply HStep_succ with (cf' := Cf (St i s (upd h (nat_of_Z v1 pf) (v2,Hi))) Skip K').
rewrite eden_erase with (i := i) in H10; intuit.
rewrite edenZ_some in H10; destruct H10 as [l1].
rewrite eden_erase with (i := i) in H11; intuit.
rewrite edenZ_some in H11; destruct H11 as [l2].
apply HStep_write with (l1 := l1) (l2 := l2); auto.
intro H3; contradiction H12; unfold erase; simpl; rewrite H3; auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H2.
simpl; intro H3; rewrite edenZ_none in H3; rewrite H3 in H10; inv H10.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H3; rewrite edenZ_none in H3; rewrite H3 in H8; inv H8.
apply HStep_zero.
unfold erase_st; simpl; rewrite erase_upd; auto.
∃ 1; ∃ st; split; auto.
apply HStep_succ with (cf' := Cf st C' (C2::K)).
apply HStep_seq.
apply HStep_zero.
destruct st as [i s h].
∃ 1; ∃ (St i s h); split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C' K').
```

```
rewrite bden_erase with (i := i) in H10; intuit.
rewrite bdenZ_some in H10; destruct H10 as [l].
apply HStep_if_true with (l := l); auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H9; inv H9.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H9; inv H9.
apply HStep_zero.
destruct st as [i s h].
∃ 1; ∃ (St i s h); split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C' K').
rewrite bden_erase with (i := i) in H10; intuit.
rewrite bdenZ_some in H10; destruct H10 as [l].
apply HStep_if_false with (l := l); auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H9; inv H9.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H9; inv H9.
apply HStep_zero.
destruct st as [i s h].
∃ 1; ∃ (St i s h); split; auto.
apply HStep_succ with (cf' := Cf (St i s h) C' (While b C' :: K)).
rewrite bden_erase with (i := i) in H10; intuit.
rewrite bdenZ_some in H10; destruct H10 as [l].
apply HStep_while_true with (l := l); auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply HStep_zero.
destruct st as [i s h].
∃ 1; ∃ (St i s h); split; auto.
apply HStep_succ with (cf' := Cf (St i s h) Skip K').
rewrite bden_erase with (i := i) in H10; intuit.
rewrite bdenZ_some in H10; destruct H10 as [l].
apply HStep_while_false with (l := l); auto.
specialize (H1 _ _ (HStep_zero _) (refl_equal _)); inv H1.
inv H0.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
simpl; intro H0; rewrite bdenZ_none in H0; rewrite H0 in H8; inv H8.
apply HStep_zero.
Qed.
```

```
Lemma hstepn_instrument : ∀ n st est C C' K K' o, no_lvars (C::K) →
  stepn n (Cf' (erase_st st) C K) (Cf' est C' K') o → hsafe (Cf st C K) →
  ∃ n', ∃ st', hstepn n' (Cf st C K) (Cf st' C' K') ∧ est = erase_st st'.
Proof.
induction n using (well_founded_induction lt_wf); intros.
inv H1.
∃ 0; ∃ st; split; auto; apply HStep_zero.
destruct cf' as [est'' C'' K'']; apply hstep_instrument in H3; auto.
destruct H3 as [n [st'' [H3]]]; subst.
apply H in H4; auto.
destruct H4 as [n' [st' [H4]]]; subst.
∃ (n+n'); ∃ st'; split; auto.
apply hstep_trans with (cf2 := Cf st'' C'' K''); auto.
apply hstepn_no_lvars_monotonic in H3; auto.
unfold hsafe; intros.
assert (hstepn (n+n1) (Cf st C K) cf').
apply hstep_trans with (cf2 := Cf st'' C'' K''); auto.
apply H2 in H6; auto.
Qed.

Lemma lstepn_instrument : ∀ n st est C K K' e, no_lvars (C::K) →
  stepn (S n) (Cf' (erase_st st) C K) (Cf' est (Output e) K') [] → lsafe (Cf st C K) →
  ∃ n1, ∃ n2, ∃ st', ∃ C'', ∃ K'',
    stepn n1 (Cf' (erase_st st) C K) (Cf' (erase_st st') C'' K'') [] ∧
    stepn n2 (Cf' (erase_st st') C'' K'') (Cf' est (Output e) K') [] ∧
    lstep (Cf st C K) (Cf st' C'' K'') [] ∧ S n = n1 + n2.
Proof.
intros.
case_eq (halt_config (Cf st C K)); intros.
destruct C; destruct K; inv H2.
inv H0.
inv H4.
specialize (H1 _ _ _ (LStep_zero _) H2); inv H1.
destruct o.
destruct cf' as [st' C'' K'']; dup H3; apply lstep_erase in H3; auto.
destruct H3 as [n'].
assert (n' ≤ S n ∨ n' > S n); try omega.
destruct H4.
∃ n'; ∃ (S n - n'); ∃ st'; ∃ C''; ∃ K''; intuition.
assert (S n = n' + (S n - n')); try omega.
rewrite H5 in H0; apply step_trans_inv' in H0.
destruct H0 as [cf [o1 [o2 H0]]]; decomp H0.
destruct (stepn_det _ _ _ _ _ _ H3 H6); subst.
```

```
destruct o2; inv H9; auto.
assert (n' = S n + (n' - S n)); try omega.
rewrite H5 in H3; apply step_trans_inv' in H3.
destruct H3 as [cf [o1 [o2 H3]]]; decomp H3.
destruct (stepn_det _ _ _ _ _ _ H0 H6); subst cf o1.
destruct o2; inv H9.
inv H8.
assert False; try omega; intuit.
inv H9; subst o; inv H7.
inv H3.
inv H0.
inv H4; inv H3.
Qed.

Fixpoint size (C : cmd) :=
  match C with
  | Seq C1 C2 ⇒ S (size C1 + size C2)
  | If _ C1 C2 ⇒ S (size C1 + size C2)
  | While _ C ⇒ S (size C)
  | _ ⇒ 0
  end.

Lemma step_instrument_term : ∀ n st est C K, no_lvars (C::K) →
  stepn n (Cf' (erase_st st) C K) (Cf' est Skip []) [] → lsafe (Cf st C K) →
  ∃ n', ∃ st', lstepn n' (Cf st C K) (Cf st' Skip []) [].
Proof.
induction n using (well_founded_induction lt_wf); intros.
case_eq (halt_config (Cf st C K)); intros.
destruct C; destruct K; inv H3.
∃ 0; ∃ st; apply LStep_zero.
dup H2; specialize (H2 _ _ _ (LStep_zero _) H3); inv H2.
rename H4 into H'; rename H5 into H4.
destruct cf' as [st' C' K']; dup H4; apply lstep_erase in H4; auto.
destruct H4 as [n'].
assert (n' < n ∨ n' ≥ n); try omega.
destruct H5.
destruct n'.
inv H4.
inv H2.
apply f_equal with (f := fun l ⇒ length l) in H13; simpl in H13.
assert False; try omega; intuit.
apply f_equal with (f := fun l ⇒ length l) in H13; simpl in H13.
assert False; try omega; intuit.
apply f_equal with (f := fun C ⇒ size C) in H12; simpl in H12.
```

```
assert (False); try omega; intuit.
apply f_equal with (f := fun C ⇒ size C) in H12; simpl in H12.
assert (False); try omega; intuit.
apply f_equal with (f := fun l ⇒ length l) in H13; simpl in H13.
assert False; try omega; intuit.
assert (∀ n est, ¬ stepn n (Cf' (erase_st (St i s h)) (If b C1 C2) []) (Cf' est Skip []) []);
intros.
intro.
unfold erase_st in H2; rewrite ← erase_taint_vars with (K := [If b C1 C2]) in H2;
simpl in H2.
change (stepn n0 (Cf' (erase_st (St i (taint_vars [If b C1 C2] s) h)) (If b C1 C2) []) (Cf'
est0 Skip []) []) in H2.
apply hstepn_instrument in H2; auto.
destruct H2 as [n' [st' [H2]]]; contradiction (H13 n' st').
simpl in H0 ⊢ *; intuit.
change (stepn n (Cf' (erase_st (St i s h)) (If b C1 C2) ([]++K')) (Cf' est Skip []) ([]++[]))
in H1.
apply step_trans_inv in H1; destruct H1.
destruct H1 as [K'' [H1]].
destruct K''; inv H4.
contradiction (H2 n est).
destruct H1 as [st'' [n1 [n2 [o1 [o2 H1]]]]]; decomp H1.
destruct o1; inv H14; contradiction (H2 n1 st'').
assert (∀ n est, ¬ stepn n (Cf' (erase_st (St i s h)) (While b C) []) (Cf' est Skip []) []);
intros.
intro.
unfold erase_st in H2; rewrite ← erase_taint_vars with (K := [While b C]) in H2; simpl
in H2.
change (stepn n0 (Cf' (erase_st (St i (taint_vars [While b C] s) h)) (While b C) []) (Cf'
est0 Skip []) []) in H2.
apply hstepn_instrument in H2; auto.
destruct H2 as [n' [st' [H2]]]; contradiction (H13 n' st').
simpl in H0 ⊢ *; intuit.
change (stepn n (Cf' (erase_st (St i s h)) (While b C) ([]++K')) (Cf' est Skip []) ([]++[]))
in H1.
apply step_trans_inv in H1; destruct H1.
destruct H1 as [K'' [H1]].
destruct K''; inv H4.
contradiction (H2 n est).
destruct H1 as [st'' [n1 [n2 [o1 [o2 H1]]]]]; decomp H1.
destruct o1; inv H14; contradiction (H2 n1 st'').
assert (n = S n' + (n-S n')); try omega.
```

```
rewrite H6 in H1; clear H6; apply step_trans_inv' in H1.
destruct H1 as [cf [o1 [o2 H1]]]; decomp H1.
destruct o1; inv H9.
destruct o2; inv H1.
destruct (stepn_det _ _ _ _ _ _ H4 H6); subst.
apply H in H8; try omega.
destruct H8 as [n1 [st1]]; ∃ (S n1); ∃ st1.
rewrite ← app_nil_r; apply LStep_succ with (cf' := Cf st' C' K'); auto.
apply lstep_no_lvars_monotonic in H2; auto.
unfold lsafe; intros.
apply (H' (S n0) _ ([]++o)); auto.
apply LStep_succ with (cf' := Cf st' C' K'); auto.
assert (n' = n + (n'-n)); try omega.
rewrite H6 in H4; clear H6; apply step_trans_inv' in H4.
destruct H4 as [cf [o1 [o2 H4]]]; decomp H4; subst.
destruct (stepn_det _ _ _ _ _ _ H1 H6); subst.
inv H8.
∃ 1; ∃ st'.
rewrite ← app_nil_r; apply LStep_succ with (cf' := Cf st' Skip []); auto.
apply LStep_zero.
inv H7.
Qed.

Theorem step_instrument : ∀ n st est C K o, no_lvars (C::K) →
  stepn n (Cf' (erase_st st) C K) (Cf' est Skip []) o → lsafe (Cf st C K) →
  ∃ n', ∃ st', lstepn n' (Cf st C K) (Cf st' Skip []) o.
Proof.
induction n using (well_founded_induction lt_wf); intros.
destruct o as [|v].
apply step_instrument_term in H1; auto.
fold ([v]++o) in H1; apply step_output_inv' in H1.
destruct H1 as [n1 [n2 [st' [C' [K' H1]]]]]; decomp H1; subst.
apply step_output_inv in H3.
destruct H3 as [n3 [n4 [st'' [e [K'' H3]]]]]; decomp H3; subst.
destruct n3.
inv H6.
inv H4.
inv H3.
inv H1.
assert (stepn (n+n2) (Cf' (erase_st st) Skip K'') (Cf' est Skip []) ([]++o)).
apply step_trans with (cf2 := Cf' st' C' K'); auto.
assert (lstep (Cf st (Output e) K'') (Cf st Skip K'') [v]).
destruct st as [i s h]; apply LStep_output.
```

```
specialize (H2 _ _ _ (LStep_zero _) (refl_equal _)); inv H2.
inv H3.
rewrite eden_erase with (i := i) in H12.
simpl in H12; rewrite edenZ_some in H12; destruct H12 as [l].
rewrite H13 in H2; inv H2; auto.
simpl in H0; intuit.
simpl; intro H2; rewrite edenZ_none in H2; rewrite H2 in H13; inv H13.
apply H in H1; auto.
destruct H1 as [n1 [st1]]; ∃ (S n1); ∃ st1.
fold ([v]++o); apply LStep_succ with (cf' := Cf st Skip K''); auto.
simpl in H0 ⊢ *; intuit.
unfold lsafe; intros.
apply (H2 (S n0) _ ([v]++o0)); auto.
apply LStep_succ with (cf' := Cf st Skip K''); auto.
apply lstepn_instrument in H1; auto.
destruct H1 as [n1' [n2' [st1 [C1 [K1 H1]]]]]; decomp H1.
assert (stepn (n2'+(n4+n2)) (Cf' (erase_st st1) C1 K1) (Cf' est Skip []) ([]++([v]++o))).
apply step_trans with (cf2 := Cf' st'' (Output e) K''); auto.
apply step_trans with (cf2 := Cf' st' C' K'); auto.
destruct n1'.
inv H3; inv H4.
apply f_equal with (f := fun l ⇒ length l) in H16; simpl in H16.
assert False; try omega; intuit.
apply f_equal with (f := fun l ⇒ length l) in H16; simpl in H16.
assert False; try omega; intuit.
apply f_equal with (f := fun C ⇒ size C) in H15; simpl in H15.
assert (False); try omega; intuit.
apply f_equal with (f := fun C ⇒ size C) in H15; simpl in H15.
assert (False); try omega; intuit.
apply f_equal with (f := fun l ⇒ length l) in H16; simpl in H16.
assert False; try omega; intuit.
assert (∀ n est o, ¬ stepn n (Cf' (erase_st (St i s h)) (If b C0 C2) []) (Cf' est Skip [])
o); intros.
intro.
unfold erase_st in H3; rewrite ← erase_taint_vars with (K := [If b C0 C2]) in H3;
simpl in H3.
change (stepn n (Cf' (erase_st (St i (taint_vars [If b C0 C2] s) h)) (If b C0 C2) []) (Cf'
est0 Skip []) o0) in H3.
apply hstepn_instrument in H3; auto.
destruct H3 as [n'' [st2 [H3]]]; contradiction (H16 n'' st2).
simpl in H0 ⊢ *; intuit.
change (stepn (n2'+(n4+n2)) (Cf' (erase_st (St i s h)) (If b C0 C2) ([]++K1)) (Cf' est
```

```
Skip []) ([]++([v]++o))) in H1.
apply step_trans_inv in H1; destruct H1.
destruct H1 as [K''' [H1]].
destruct K'''; inv H4.
contradiction (H3 (n2'+(n4+n2)) est ([]++[v]++o)).
destruct H1 as [st2 [n1'' [n2'' [o1 [o2 H1]]]]]; decomp H1.
contradiction (H3 n1'' st2 o1).
assert (∀ n est o, ¬ stepn n (Cf' (erase_st (St i s h)) (While b C) []) (Cf' est Skip []) o);
intros.
intro.
unfold erase_st in H3; rewrite ← erase_taint_vars with (K := [While b C]) in H3; simpl
in H3.
change (stepn n (Cf' (erase_st (St i (taint_vars [While b C] s) h)) (While b C) []) (Cf'
est0 Skip []) o0) in H3.
apply hstepn_instrument in H3; auto.
destruct H3 as [n'' [st2 [H3]]]; contradiction (H16 n'' st2).
simpl in H0 ⊢ *; intuit.
change (stepn (n2'+(n4+n2)) (Cf' (erase_st (St i s h)) (While b C) ([]++K1)) (Cf' est
Skip []) ([]++([v]++o))) in H1.
apply step_trans_inv in H1; destruct H1.
destruct H1 as [K''' [H1]].
destruct K'''; inv H4.
contradiction (H3 (n2'+(n4+n2)) est ([]++[v]++o)).
destruct H1 as [st2 [n1'' [n2'' [o1 [o2 H1]]]]]; decomp H1.
contradiction (H3 n1'' st2 o1).
apply H in H1; try omega.
destruct H1 as [n' [st2]]; ∃ (S n'); ∃ st2.
change (lstepn (S n') (Cf st C K) (Cf st2 Skip []) ([]++[]++[v]++o)).
apply LStep_succ with (cf' := Cf st1 C1 K1); auto.
apply lstep_no_lvars_monotonic in H4; auto.
unfold lsafe; intros.
apply (H2 (S n) _ ([]++o0)); auto.
apply LStep_succ with (cf' := Cf st1 C1 K1); auto.
Qed.

Theorem noninterference : ∀ N P C Q st1 st2 st1' st2' n1 n2 o1 o2,
  no_lvars_cmd C → judge N Lo P C Q → aden2 P st1 st2 →
  stepn n1 (Cf' (erase_st st1) C []) (Cf' st1' Skip []) o1 →
  stepn n2 (Cf' (erase_st st2) C []) (Cf' st2' Skip []) o2 → o1 = o2.
Proof.
intros.
apply soundness in H0; inv H0.
apply step_instrument in H2; simpl; auto.
```

```
apply step_instrument in H3; simpl; auto.
destruct H2 as [n1' [st1'']]; destruct H3 as [n2' [st2'']].
assert (side_condition C st1 st2).
decomp (H6 _ _ _ _ _ _ _ _ _ _ H1 (LStep_zero _) (LStep_zero _)); auto.
apply (False_ind _ (diverge_halt _ _ _ _ H3 H0)).
apply (False_ind _ (diverge_halt _ _ _ _ H10 H2)).
destruct (H7 _ _ _ _ _ _ _ _ _ H1 H3 H0 H2); auto.
apply H4; inv H1; intuit.
apply H4; inv H1; intuit.
Qed.

Print Assumptions noninterference.
```