Chapter 1

Library ddifc-coq

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*.

Proposition *in_app_iff* {*A*} : ∀ (*l1 l2* : *list A*) *x*, *In x* (*l1*++*l2*) ↔ *In x l1* ∨ *In x l2*.

Proposition *app_nil_r* {*A*} : ∀ *l* : *list A*, *l* ++ [] = *l*.

Proposition *list_finite* {*A*} : ∀ (*l* : *list A*) *x*, *l* ≠ *x* :: *l*.

Proposition *list_finite'* {*A*} : ∀ *l l'* : *list A*, *l'* ≠ [] → *l* ≠ *l'* ++ *l*.

Proposition *app_cancel_l* {*A*} : ∀ *l l1 l2* : *list A*, *l* ++ *l1* = *l* ++ *l2* → *l1* = *l2*.

Proposition *app_cancel_r_help* {*A*} : ∀ (*l1 l2* : *list A*) *x*, *l1* ++ [*x*] = *l2* ++ [*x*] → *l1* = *l2*.

Proposition *app_cancel_r* {*A*} : ∀ *l l1 l2* : *list A*, *l1* ++ *l* = *l2* ++ *l* → *l1* = *l2*.

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

Definition *nat_of_Z* (*v* : *Z*) (*pf* : *v* ≥ 0) : *nat*.
Defined.

Proposition *Zneg_dec* : ∀ *v* : *Z*, {*v* ≥ 0} + {*v* < 0}.

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.
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.
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.

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

Definition glbl_poset : poset (A:=glbl).
Defined.

Definition glbl_join_semi : join_semi (A:=glbl).
Defined.

Definition glbl_lattice : bounded_join_semi (A:=glbl).
```

```
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.

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.

Proposition glub_homo : ∀ l l1 l2, grp l (llub l1 l2) = glub (grp l l1) (grp l l2).

Proposition glub_leq : ∀ l l1 l2, glub (grp l l1) (grp l l2) = Lo ↔ grp l l1 = Lo ∧ grp l l2
= Lo.

Proposition glub_lo : ∀ l1 l2, glub l1 l2 = Lo ↔ l1 = Lo ∧ l2 = Lo.


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}.

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}.

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

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.
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).

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.

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*).

Proposition *edenZ_none* : ∀ *e i s*, *edenZ e i s* = *None* ↔ *eden e i s* = *None*.

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*).

Proposition *bdenZ_none* : ∀ *b i s*, *bdenZ b i s* = *None* ↔ *bden b i s* = *None*.

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*.

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*.

```
Proposition eden_no_lvars : ∀ e i i' s, no_lvars_exp e → eden e i s = eden e i' s.

Proposition bden_no_lvars : ∀ b i i' s, no_lvars_bexp b → bden b i s = bden b i' s.

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.

Proposition dvg_ex_mid : ∀ cf,
  (∀ n st, ¬ hstepn n cf (Cf st Skip [])) ∨ ∃ n, ∃ st, hstepn n cf (Cf st Skip []).

Lemma hstep_trans : ∀ n1 n2 cf1 cf2 cf3, hstepn n1 cf1 cf2 → hstepn n2 cf2 cf3 → hstepn
(n1+n2) cf1 cf3.

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).

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)).
```

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*)).

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*.

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*.

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*.

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*.

Lemma *hstep_trans_inv'* : ∀ *a b cf cf'*,
hstepn (*a*+*b*) *cf cf'* → ∃ *cf''*, *hstepn a cf cf''* ∧ *hstepn b cf'' cf'*.

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*.

Lemma *hstep_det* : ∀ *cf cf1 cf2*, *hstep cf cf1* → *hstep cf cf2* → *cf1* = *cf2*.

Lemma *hstepn_det* : ∀ *n cf cf1 cf2*, *hstepn n cf cf1* → *hstepn n cf cf2* → *cf1* = *cf2*.

Lemma *hstepn_det_term* : ∀ *n1 n2 cf st1 st2*, *hstepn n1 cf* (*Cf st1 Skip* []) → *hstepn n2 cf* (*Cf st2 Skip* []) → *n1* = *n2*.

Lemma *lstep_det* : ∀ *cf cf1 cf2 o1 o2*, *lstep cf cf1 o1* → *lstep cf cf2 o2* → *cf1* = *cf2* ∧ *o1* = *o2*.

Lemma *lstepn_det* : ∀ *n cf cf1 cf2 o1 o2*, *lstepn n cf cf1 o1* → *lstepn n cf cf2 o2* → *cf1* = *cf2* ∧ *o1* = *o2*.

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*.

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

`Corollary` *diverge_halt* : $\forall$ *n cf st o*, *diverge cf* $\rightarrow$ *lstepn n cf* (*Cf st Skip* []) *o* $\rightarrow$ *False*.

`Proposition` *diverge_same_cf* : $\forall$ *cf o*, *lstep cf cf o* $\rightarrow$ *diverge cf*.

`Lemma` *diverge_seq1* : $\forall$ *C1 C2 K st*, *diverge* (*Cf st C1* []) $\rightarrow$ *diverge* (*Cf st* (*Seq C1 C2*) *K*).

`Lemma` *diverge_seq2* : $\forall$ *C1 C2 K st st' n o*,
lstepn n (*Cf st C1* []) (*Cf st' Skip* []) *o* $\rightarrow$ *diverge* (*Cf st' C2 K*) $\rightarrow$ *diverge* (*Cf st* (*Seq C1 C2*) *K*).

`Lemma` *hstep_ff* : $\forall$ *C K C' K' i s h1 h2 h3 i' s' h1'*,
mydot h1 h2 h3 $\rightarrow$ *hstep* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) $\rightarrow$
$\exists$ *h3'*, *mydot h1' h2 h3'* $\wedge$ *hstep* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*).

`Lemma` *hstepn_ff* : $\forall$ *n C K C' K' i s h1 h2 h3 i' s' h1'*,
mydot h1 h2 h3 $\rightarrow$ *hstepn n* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) $\rightarrow$
$\exists$ *h3'*, *mydot h1' h2 h3'* $\wedge$ *hstepn n* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*).

`Lemma` *hstep_bf* : $\forall$ *C K C' K' i s h1 h2 h3 i' s' h3'*,
mydot h1 h2 h3 $\rightarrow$ *hstep* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) $\rightarrow$ *hsafe* (*Cf* (*St i s h1*) *C K*) $\rightarrow$
$\exists$ *h1'*, *mydot h1' h2 h3'* $\wedge$ *hstep* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*).

`Lemma` *hstepn_bf* : $\forall$ *n C K C' K' i s h1 h2 h3 i' s' h3'*,
mydot h1 h2 h3 $\rightarrow$ *hstepn n* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) $\rightarrow$ *hsafe* (*Cf* (*St i s h1*) *C K*) $\rightarrow$
$\exists$ *h1'*, *mydot h1' h2 h3'* $\wedge$ *hstepn n* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*).

`Lemma` *lstep_ff* : $\forall$ *C K C' K' i s h1 h2 h3 i' s' h1' o*,
mydot h1 h2 h3 $\rightarrow$ *lstep* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) *o* $\rightarrow$
$\exists$ *h3'*, *mydot h1' h2 h3'* $\wedge$ *lstep* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) *o*.

`Lemma` *lstepn_ff* : $\forall$ *n C K C' K' i s h1 h2 h3 i' s' h1' o*,
mydot h1 h2 h3 $\rightarrow$ *lstepn n* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) *o* $\rightarrow$
$\exists$ *h3'*, *mydot h1' h2 h3'* $\wedge$ *lstepn n* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) *o*.

`Corollary` *lstepn_nonincreasing* : $\forall$ *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* $\rightarrow$ *h a* = *None* $\rightarrow$ *h' a* = *None*.

`Lemma` *lstep_bf* : $\forall$ *C K C' K' i s h1 h2 h3 i' s' h3' o*,
mydot h1 h2 h3 $\rightarrow$ *lstep* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) *o* $\rightarrow$ *lsafe* (*Cf* (*St i s h1*) *C K*) $\rightarrow$
$\exists$ *h1'*, *mydot h1' h2 h3'* $\wedge$ *lstep* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) *o*.

`Lemma` *lstepn_bf* : $\forall$ *n C K C' K' i s h1 h2 h3 i' s' h3' o*,
mydot h1 h2 h3 $\rightarrow$ *lstepn n* (*Cf* (*St i s h3*) *C K*) (*Cf* (*St i' s' h3'*) *C' K'*) *o* $\rightarrow$ *lsafe* (*Cf* (*St i s h1*) *C K*) $\rightarrow$
$\exists$ *h1'*, *mydot h1' h2 h3'* $\wedge$ *lstepn n* (*Cf* (*St i s h1*) *C K*) (*Cf* (*St i' s' h1'*) *C' K'*) *o*.

`Lemma` *hstep_modifies_monotonic* : $\forall$ *st st' C C' K K' x*,
hstep (*Cf st C K*) (*Cf st' C' K'*) $\rightarrow$ *In x* (*modifies* (*C'*::*K'*)) $\rightarrow$ *In x* (*modifies* (*C*::*K*)).

Lemma *hstepn_modifies_monotonic* : $\forall$ *n st st' C C' K K' x*,
hstepn n (*Cf st C K*) (*Cf st' C' K'*) $\rightarrow$ *In x* (*modifies* (*C'*::*K'*)) $\rightarrow$ *In x* (*modifies* (*C*::*K*)).

Lemma *lstep_modifies_monotonic* : $\forall$ *st st' C C' K K' x o*,
lstep (*Cf st C K*) (*Cf st' C' K'*) *o* $\rightarrow$ *In x* (*modifies* (*C'*::*K'*)) $\rightarrow$ *In x* (*modifies* (*C*::*K*)).

Lemma *lstepn_modifies_monotonic* : $\forall$ *n st st' C C' K K' x o*,
lstepn n (*Cf st C K*) (*Cf st' C' K'*) *o* $\rightarrow$ *In x* (*modifies* (*C'*::*K'*)) $\rightarrow$ *In x* (*modifies* (*C*::*K*)).

Lemma *hstep_modifies_const* : $\forall$ *st st' C C' K K' x*,
hstep (*Cf st C K*) (*Cf st' C' K'*) $\rightarrow \neg$ *In x* (*modifies* (*C*::*K*)) $\rightarrow$ (*st*:*store*) *x* = (*st'*:*store*) *x*.

Lemma *hstepn_modifies_const* : $\forall$ *n st st' C C' K K' x*,
hstepn n (*Cf st C K*) (*Cf st' C' K'*) $\rightarrow \neg$ *In x* (*modifies* (*C*::*K*)) $\rightarrow$ (*st*:*store*) *x* = (*st'*:*store*) *x*.

Lemma *lstep_modifies_const* : $\forall$ *st st' C C' K K' x o*,
lstep (*Cf st C K*) (*Cf st' C' K'*) *o* $\rightarrow \neg$ *In x* (*modifies* (*C*::*K*)) $\rightarrow$ (*st*:*store*) *x* = (*st'*:*store*) *x*.

Lemma *lstepn_modifies_const* : $\forall$ *n st st' C C' K K' x o*,
lstepn n (*Cf st C K*) (*Cf st' C' K'*) *o* $\rightarrow \neg$ *In x* (*modifies* (*C*::*K*)) $\rightarrow$ (*st*:*store*) *x* = (*st'*:*store*) *x*.

Lemma *hstep_taints_s* : $\forall$ *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'*) $\rightarrow$
s x $\neq$ *s' x* $\rightarrow \exists$ *v*, *s' x* = *Some* (*v*,*Hi*).

Lemma *hstepn_taints_s* : $\forall$ *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'*) $\rightarrow$
s x $\neq$ *s' x* $\rightarrow \exists$ *v*, *s' x* = *Some* (*v*,*Hi*).

Lemma *hstep_taints_h* : $\forall$ *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'*) $\rightarrow$
h a $\neq$ *h' a* $\rightarrow \exists$ *v*, *h' a* = *Some* (*v*,*Hi*).

Lemma *hstepn_taints_h* : $\forall$ *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'*) $\rightarrow$
h a $\neq$ *h' a* $\rightarrow \exists$ *v*, *h' a* = *Some* (*v*,*Hi*).

Proposition *hstep_i_const* : $\forall$ *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'*) $\rightarrow$ *i'* = *i*.

Proposition *hstepn_i_const* : $\forall$ *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'*) $\rightarrow$ *i'* = *i*.

Proposition *lstep_i_const* : $\forall$ *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* $\rightarrow$ *i'* = *i*.

Proposition *lstepn_i_const* : $\forall$ *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.

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.

Proposition obs_eq_h_refl : ∀ h, obs_eq_h h h.

Proposition obs_eq_refl : ∀ st, obs_eq st st.

Proposition obs_eq_s_sym : ∀ s1 s2, obs_eq_s s1 s2 → obs_eq_s s2 s1.

Proposition obs_eq_h_sym : ∀ h1 h2, obs_eq_h h1 h2 → obs_eq_h h2 h1.

Proposition obs_eq_sym : ∀ st1 st2, obs_eq st1 st2 → obs_eq st2 st1.

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.

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.

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).

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.

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.

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.

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.
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.
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.
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.
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).
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'.
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'.
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'.
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'.
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).
Proposition expvars_none : ∀ e i s x v l, eden e i s = Some (v,l) → s x = None →
expvars e x = false.
Proposition bexpvars_none : ∀ b i s x v l, bden b i s = Some (v,l) → s x = None →
bexpvars b x = false.
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).
```

```
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'.

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'.

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.

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.

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).

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.

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.

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).

Proposition taint_vars_same_values : ∀ K s, same_values s (taint_vars K s) (modifies
K).

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).

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*.

`Proposition` *taint_vars_idempotent* : ∀ *K s*, *taint_vars K* (*taint_vars K s*) = *taint_vars K s*.

`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*.

`Lemma` *soundness_output* : ∀ *e*, *sound Lo* (*LblExp e Lo* ‘*AND*‘ *Emp*) (*Output e*) (*LblExp e Lo* ‘*AND*‘ *Emp*).

`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*).

`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).

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)).

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.

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.

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).

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’.

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).

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).

Proposition mydot_comm {A} : ∀ h1 h2 h3, mydot(A:=A) h1 h2 h3 → mydot h2 h1 h3.

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.

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).

Theorem soundness : ∀ N ct P C Q, judge N ct P C Q → sound ct P C Q.

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.

Proposition erase_fill_upd : ∀ f x v l, erase_fill (upd f x (v,l)) = upd (erase_fill f) x v.

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.

Proposition bden_erase : ∀ b i s, no_lvars_bexp b → bdenZ b i s ≠ None → bden' b (erase_fill s) = bdenZ b i s.

Proposition erase_taint_vars : ∀ K s, erase_fill (taint_vars K s) = erase_fill s.

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) [].

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*).

`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*.

`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*.

`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*.

`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*.

`Lemma` *step_det* : ∀ *cf cf1 cf2 o1 o2*, *step cf cf1 o1* → *step cf cf2 o2* → *cf1* = *cf2* ∧ *o1* = *o2*.

`Lemma` *stepn_det* : ∀ *n cf cf1 cf2 o1 o2*, *stepn n cf cf1 o1* → *stepn n cf cf2 o2* → *cf1* = *cf2* ∧ *o1* = *o2*.

`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*.

`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*.

`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'*).

`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'*).

`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'*).

```
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').

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') [].

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') [].

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.

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.

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.

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'.

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'.

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.

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 []) [].

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.

`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.

`Print Assumptions` *noninterference.*