```
Require Import FMaps.
Require Import FSets.FMapFacts.
Require Import ZArith.
Require Import List.
Require Import ClassicalFacts.
Require Import FunctionalExtensionality.
Require Import Arith.Compare_dec.
Require Import ProofIrrelevance.

Ltac inv H := inversion H; subst; clear H.
Ltac dup H := generalize H; intro.
Ltac znat_simpl H H' := rewrite H' in H; apply inj_eq_rev in H; subst.
```

0.1 Random useful facts

```
Lemma double_neg : ∀ P : Prop, {P} + {¬ P} → ¬ ¬ P → P.
Proof.
intros.
destruct H; auto.
contradiction H0; auto.
Qed.

Lemma leq_dec : ∀ n m, {n ≤ m} + {n > m}.
Proof.
induction n; intros.
left; omega.
induction m.
right; omega.
destruct IHm.
left; omega.
destruct (IHn m).
left; omega.
right; omega.
Qed.

Lemma lt_dec : ∀ n m, {n < m} + {n ≥ m}.
Proof.
intros.
destruct (eq_nat_dec n m); subst.
right; auto.
destruct (leq_dec n m).
left; omega.
right; omega.
Qed.
```

0.2 Language Definition

```
Definition var := nat.
Definition val := Z.

Inductive binop :=
| Plus
| Minus
| Mult
| Div
| Mod.

Open Scope Z_scope.

Fixpoint op_val op v1 v2 :=
  match op with
  | Plus => v1+v2
  | Minus => v1-v2
  | Mult => v1*v2
  | Div => v1/v2
  | Mod => v1 mod v2
  end.

Inductive bbinop :=
| And
| Or
| Impl.

Fixpoint bop_val bop a1 a2 :=
  match bop with
  | And => andb a1 a2
  | Or => orb a1 a2
  | Impl => orb (negb a1) a2
  end.

Inductive exp :=
| Exp_val : val -> exp
| Exp_nil : exp
| Exp_var : var -> exp
| Exp_op : binop -> exp -> exp -> exp.

Inductive bexp :=
| BExp_eq : exp -> exp -> bexp
| BExp_false : bexp
| BExp_bop : bbinop -> bexp -> bexp -> bexp.

Definition bnot (b : bexp) : bexp := BExp_bop Impl b BExp_false.

Close Scope Z_scope.
```

```
Inductive cmd :=
| Skip : cmd
| Assgn : var → exp → cmd
| Read : var → exp → cmd
| Write : exp → exp → cmd
| Cons : var → list exp → cmd
| Free : exp → cmd
| Seq : cmd → cmd → cmd
| If : bexp → cmd → cmd → cmd
| While : bexp → cmd → cmd.

Notation "c1 ;; c2" := (Seq c1 c2) (at level 81, left associativity).
Notation "'if_' b 'then' c1 'else' c2" := (If b c1 c2) (at level 1).
Notation "'while' b 'do' c" := (While b c) (at level 1).
Notation "x ::= e" := (Assgn x e) (at level 1).
Notation "x ::= [[ e ]]" := (Read x e) (at level 1).
Notation "x ::= 'cons' l" := (Cons x l) (at level 1).
Notation "[[ e1 ]] ::= e2" := (Write e1 e2) (at level 1).
Notation "[ ]" := nil (at level 1).
Notation "[ a ; .. ; b ]" := (a :: .. (b :: []) ..) (at level 1).
```

0.3 Definition and lemmas for natmap

```
Open Scope nat_scope.

Definition natmap (A : Type) := list (nat*A).

Fixpoint find {A} (m : natmap A) n :=
  match m with
  | [] ⇒ None
  | (n',v)::m ⇒ if eq_nat_dec n n' then Some v else find m n
  end.

Fixpoint del {A} (m : natmap A) n :=
  match m with
  | [] ⇒ []
  | (n',v)::m ⇒ if eq_nat_dec n n' then del m n else (n',v)::(del m n)
  end.

Fixpoint maxkey_help {A} (m : natmap A) n :=
  match m with
  | [] ⇒ n
  | (n',_)::m ⇒ if lt_dec n n' then maxkey_help m n' else maxkey_help m n
  end.

Definition maxkey {A} (m : natmap A) := maxkey_help m 0.
```

```
Definition upd {A} (m : natmap A) n v := (n,v)::m.
Definition union {A} (m1 m2 : natmap A) := m1 ++ m2.
Notation "m1 @ m2" := (union m1 m2) (at level 2).
Definition haskey {A} (m : natmap A) n := find m n ≠ None.
Definition mapsto {A} (m : natmap A) n v := find m n = Some v.
Definition disjoint {A} (m1 m2 : natmap A) := ∀ n, haskey m1 n → ¬ haskey m2 n.
Notation "m1 # m2" := (disjoint m1 m2) (at level 2).
Definition empmap {A} : natmap A := [].

Lemma maxkey_help_best {A} : ∀ (m : natmap A) n, maxkey_help m n ≥ n.
Proof.
induction m; intros; simpl; auto.
destruct a.
destruct (lt_dec n n0); auto.
specialize (IHm n0); omega.
Qed.

Lemma maxkey_help_monotonic {A} : ∀ (m : natmap A) a b, a ≤ b → maxkey_help m a ≤
maxkey_help m b.
Proof.
induction m; intros; simpl; auto.
destruct a; clear a; rename a0 into a.
destruct (lt_dec a n); destruct (lt_dec b n); auto; omega.
Qed.

Lemma maxkey_max {A} : ∀ (m : natmap A) n, haskey m n → n < S (maxkey m).
Proof.
induction m; intros.
unfold haskey in H; simpl in H; contradiction H; auto.
unfold haskey in H; simpl in H; destruct a.
destruct (eq_nat_dec n n0); subst.
unfold maxkey; simpl.
destruct (lt_dec 0 n0).
assert (maxkey_help m n0 ≥ n0).
apply maxkey_help_best.
omega.
omega.
apply IHm in H.
unfold maxkey in ×; simpl.
destruct (lt_dec 0 n0); auto.
assert (maxkey_help m 0 ≤ maxkey_help m n0).
apply maxkey_help_monotonic; omega.
omega.
Qed.
```

Lemma *natmap_finite* $\{A\}$: $\forall$ (m : *natmap* A), $\neg$ *haskey* m (S (*maxkey* m)).
Proof.
intros; intro.
apply *maxkey_max* in H.
omega.
Qed.

Lemma *mapsto_eq* $\{A\}$: $\forall$ (m : *natmap* A) n *v1* *v2*, *mapsto* m n *v1* $\rightarrow$ *mapsto* m n *v2* $\rightarrow$ *v1* = *v2*.
Proof.
intros.
unfold *mapsto* in $\times$.
rewrite H in *H0*.
inversion *H0*; auto.
Qed.

Lemma *mapsto_in* $\{A\}$: $\forall$ (m : *natmap* A) n v, *mapsto* m n v $\rightarrow$ *In* (n,v) m.
Proof.
intros.
generalize n v H; clear n v H.
induction m; intros.
inversion H.
destruct a.
unfold *mapsto* in H; simpl in H.
assert (n=n0 $\vee$ n<>n0).
omega.
destruct *H0*; subst.
destruct (*eq_nat_dec* *n0* *n0*).
inv H; *left*; auto.
contradiction n; auto.
destruct (*eq_nat_dec* n *n0*); subst.
contradiction *H0*; auto.
right; auto.
Qed.

Lemma *mapsto_haskey* $\{A\}$: $\forall$ (m : *natmap* A) n v, *mapsto* m n v $\rightarrow$ *haskey* m n.
Proof.
intros.
unfold *haskey*; unfold *mapsto* in H.
rewrite H; discriminate.
Qed.

Lemma *haskey_mapsto* $\{A\}$: $\forall$ (m : *natmap* A) n, *haskey* m n $\rightarrow$ $\exists$ v, *mapsto* m n v.
Proof.
intros.

```
unfold haskey in H.
case_eq (find m n); intros.
∃ a; auto.
rewrite H0 in H; contradiction H; auto.
Qed.
```

```
Lemma mapsto_union {A} : ∀ (m1 m2 : natmap A) n v, mapsto m1 n v → mapsto (union
m1 m2) n v.
Proof.
induction m1; intros.
inversion H.
simpl.
destruct a.
assert (n=n0 ∨ n<>n0).
omega.
destruct H0; subst.
unfold mapsto; simpl.
unfold mapsto in H; simpl in H.
destruct (eq_nat_dec n0 n0); auto.
contradiction n; auto.
unfold mapsto; simpl.
unfold mapsto in H; simpl in H.
destruct (eq_nat_dec n n0); subst; auto.
apply IHm1; auto.
Qed.
```

```
Lemma haskey_union {A} : ∀ (m1 m2 : natmap A) n, haskey m1 n → haskey (union m1
m2) n.
Proof.
intros.
apply haskey_mapsto in H; destruct H.
apply mapsto_haskey with (v := x).
apply mapsto_union; auto.
Qed.
```

```
Lemma mapsto_union_frame {A} : ∀ (m1 m2 : natmap A) n v, mapsto m2 n v → ¬ haskey
m1 n → mapsto (union m1 m2) n v.
Proof.
induction m1; intros; simpl; auto.
destruct a; unfold mapsto; simpl.
destruct (eq_nat_dec n n0); subst.
contradiction H0; unfold haskey; simpl.
destruct (eq_nat_dec n0 n0).
discriminate.
contradiction n; auto.
```

apply *IHm1*; auto.
intro; *contradiction H0*.
unfold *haskey*; simpl.
destruct (*eq_nat_dec n n0*); auto; try discriminate.
Qed.

Lemma *mapsto_union_inversion* $\{A\}$: $\forall$ (*m1 m2* : *natmap A*) *n v*, *mapsto* (*union m1 m2*) *n v* $\rightarrow$ (*mapsto m1 n v* $\vee$ ($\neg$ *haskey m1 n* $\wedge$ *mapsto m2 n v*)).
Proof.
induction *m1*; intros; auto.
simpl in *H*; destruct *a*.
destruct (*eq_nat_dec n n0*); subst.
left; unfold *mapsto* in *H* $\vdash \times$; simpl in $\times$.
destruct (*eq_nat_dec n0 n0*); auto.
contradiction n; auto.
unfold *mapsto* in *H*; simpl in *H*.
destruct (*eq_nat_dec n n0*); subst.
contradiction n1; auto.
destruct (*IHm1 _ _ _ H*); [*left* | *right*].
unfold *mapsto*; simpl.
destruct (*eq_nat_dec n n0*); subst; auto.
contradiction n1; auto.
intuition.
contradiction H1.
unfold *haskey* in *H0*; simpl in *H0*.
destruct (*eq_nat_dec n n0*); subst; auto.
contradiction n1; auto.
Qed.

Lemma *union_mapsto* $\{A\}$: $\forall$ (*m1 m2* : *natmap A*) *n v*, *mapsto* (*union m1 m2*) *n v* $\leftrightarrow$ *mapsto m1 n v* $\vee$ ($\neg$ *haskey m1 n* $\wedge$ *mapsto m2 n v*).
Proof.
intros; split; intros.
apply *mapsto_union_inversion*; auto.
destruct *H*.
apply *mapsto_union*; auto.
destruct *H*.
apply *mapsto_union_frame*; auto.
Qed.

Lemma *del_mapsto* $\{A\}$: $\forall$ (*m* : *natmap A*) *n n' v*, *mapsto* (*del m n*) *n' v* $\leftrightarrow$ *mapsto m n' v* $\wedge$ $n \neq n'$.
Proof.
induction *m*; intros; split; intros.
unfold *mapsto* in *H*; simpl in *H*; inversion *H*.

destruct *H*; auto.
destruct *a*; simpl in *H*.
destruct (*eq_nat_dec n n0*); subst.
apply *IHm* in *H*; intuition.
unfold *mapsto*; simpl.
destruct (*eq_nat_dec n' n0*); subst; auto.
contradiction H1; auto.
unfold *mapsto* in *H*; simpl in *H*.
destruct (*eq_nat_dec n' n0*); subst.
split; auto.
unfold *mapsto*; simpl.
destruct (*eq_nat_dec n0 n0*); auto.
contradiction n2; auto.
apply *IHm* in *H*; intuition.
unfold *mapsto*; simpl.
destruct (*eq_nat_dec n' n0*); subst; auto.
contradiction n2; auto.
simpl; destruct *a*; destruct *H*.
destruct (*eq_nat_dec n n0*); subst.
rewrite *IHm*; split; auto.
unfold *mapsto* in *H*; simpl in *H*.
destruct (*eq_nat_dec n' n0*); subst; auto.
contradiction H0; auto.
unfold *mapsto* in *H* $\vdash$ $\times$; simpl in $\times$.
destruct (*eq_nat_dec n' n0*); subst; auto.
change (*mapsto* (*del m n*) *n' v*); rewrite *IHm*; split; auto.
Qed.

Lemma *del_haskey* {*A*} : $\forall$ (*m* : *natmap A*) *n n'*, *haskey* (*del m n*) *n'* $\leftrightarrow$ *haskey m n'* $\wedge$ *n* $\neq$ *n'*.
Proof.
intros; split; intros.
apply *haskey_mapsto* in *H*; destruct *H*.
apply *del_mapsto* in *H*; destruct *H*.
split; auto.
apply *mapsto_haskey* with (*v* := *x*); auto.
destruct *H*.
apply *haskey_mapsto* in *H*; destruct *H*.
apply *mapsto_haskey* with (*v* := *x*).
rewrite *del_mapsto*; split; auto.
Qed.

Lemma *del_not_haskey* {*A*} : $\forall$ (*m* : *natmap A*) *n*, $\neg$ *haskey m n* $\rightarrow$ *del m n* = *m*.
Proof.

```
induction m; intros; auto.
destruct a; simpl.
destruct (eq_nat_dec n n0); subst.
contradiction H.
unfold haskey; simpl.
destruct (eq_nat_dec n0 n0); auto; try discriminate.
apply f_equal2; auto.
apply IHm.
intro; contradiction H.
unfold haskey; simpl.
destruct (eq_nat_dec n n0); auto.
Qed.
```

```
Lemma upd_mapsto {A} : ∀ (m : natmap A) n v n' v', mapsto (upd m n v) n' v' ↔ (n = n' ∧ v = v') ∨ (n ≠ n' ∧ mapsto m n' v').
Proof.
intros; split; intros.
unfold mapsto in H; simpl in H.
destruct (eq_nat_dec n' n); subst.
inv H; left; split; auto.
right; split; auto.
intuition; subst.
unfold mapsto; simpl.
destruct (eq_nat_dec n' n'); auto.
contradiction n; auto.
unfold mapsto; simpl.
destruct (eq_nat_dec n' n); subst; auto.
contradiction H; auto.
Qed.
```

```
Lemma upd_haskey {A} : ∀ (m : natmap A) n v n', haskey (upd m n v) n' ↔ n = n' ∨ (n ≠ n' ∧ haskey m n').
Proof.
intros; split; intros.
apply haskey_mapsto in H; destruct H.
apply upd_mapsto in H; destruct H; destruct H; subst; auto.
right; split; auto.
apply mapsto_haskey in H0; auto.
destruct H; subst.
unfold haskey; simpl.
destruct (eq_nat_dec n' n'); auto; try discriminate.
destruct H.
unfold haskey; simpl.
destruct (eq_nat_dec n' n); auto; try discriminate.
```

```
Qed.
```

Lemma *in_remove* : $\forall$ (l : *list nat*) n k, *In* k (*remove eq_nat_dec* n l) $\leftrightarrow$ $k \neq n \wedge$ *In* k l.

```
Proof.
induction l; intros; simpl.
intuition.
destruct (eq_nat_dec n a).
rewrite IHl; intuition.
contradiction H0; subst; auto.
simpl; rewrite IHl; intuition.
Qed.
```

Lemma *haskey_dec* {A} : $\forall$ (m : *natmap* A) n, {*haskey* m n} + {$\neg$ *haskey* m n}.

```
Proof.
induction m; intros.
right; auto.
destruct a.
destruct (eq_nat_dec n0 n); subst.
left.
unfold haskey; simpl.
destruct (eq_nat_dec n n); auto; try discriminate.
destruct (IHm n); [left | right].
change (haskey (upd m n0 a) n).
rewrite upd_haskey; intuition.
change (¬ haskey (upd m n0 a) n).
rewrite upd_haskey; intuition.
Qed.
```

Lemma *haskey_union_frame* {A} : $\forall$ ($m1$ $m2$: *natmap* A) n, *haskey* $m2$ n $\rightarrow$ *haskey* (*union* $m1$ $m2$) n.

```
Proof.
intros.
destruct (haskey_dec m1 n).
apply haskey_union; auto.
apply haskey_mapsto in H; destruct H.
apply mapsto_haskey with (v := x).
apply mapsto_union_frame; auto.
Qed.
```

0.4 Operational semantics and locality properties

0.4.1 Definition of state

Definition *store* := *var* $\rightarrow$ *val*.

```
Definition heap := natmap val.
Definition freelist := list nat.

Inductive state := St : store → heap → freelist → state.
Definition Store (st : state) := let (s,_,_) := st in s.
Definition Heap (st : state) := let (_,h,_) := st in h.
Definition Flst (st : state) := let (_,_,f) := st in f.
Definition config := prod state cmd.

Definition in_fl (f : freelist) n := ¬ In n f.
Definition disjhf (h : heap) (f : freelist) := ∀ n, haskey h n → ¬ in_fl f n.

Lemma in_fl_dec : ∀ f n, {in_fl f n} + {¬ in_fl f n}.
Proof.
induction f; intros; unfold in_fl in ×; simpl in ×; auto.
destruct (IHf n); destruct (eq_nat_dec a n); subst.
right; intuition.
left; intuition.
right; intuition.
right; intuition.
Qed.
```

0.4.2 Infiniteness of free list

```
Fixpoint maxaddr_help (f : freelist) n :=
  match f with
  | [] ⇒ n
  | k::f ⇒ if lt_dec n k then maxaddr_help f k else maxaddr_help f n
  end.

Definition maxaddr f := maxaddr_help f 0.

Lemma maxaddr_help_monotonic : ∀ f a b, a ≥ b → maxaddr_help f a ≥ maxaddr_help f b.
Proof.
induction f; intros; simpl; auto.
destruct (lt_dec a0 a); destruct (lt_dec b a); auto; omega.
Qed.

Lemma maxaddr_max_help1 : ∀ f n k, n > maxaddr (k::f) → n > k.
Proof.
induction f; intros.
unfold maxaddr in H; simpl in H.
destruct (lt_dec 0 k); auto; omega.
unfold maxaddr in H; simpl in H.
destruct (lt_dec 0 k); destruct (lt_dec k a); destruct (lt_dec 0 a); try omega.
cut (n > a); intros; try omega.
apply IHf.
```

```
unfold maxaddr; simpl.
destruct (lt_dec 0 a); auto; try omega.
apply IHf.
unfold maxaddr; simpl.
destruct (lt_dec 0 k); auto; try omega.
apply IHf.
unfold maxaddr; simpl.
destruct (lt_dec 0 k); auto; try omega.
Qed.
```

```
Lemma maxaddr_max_help2 : ∀ f n k, n > maxaddr (k::f) → n > maxaddr f.
Proof.
induction f; intros.
unfold maxaddr in H; simpl in H.
destruct (lt_dec 0 k); auto.
unfold maxaddr; simpl; omega.
unfold maxaddr in H ⊢ ×; simpl in ×.
destruct (lt_dec 0 k); destruct (lt_dec 0 a); destruct (lt_dec k a); auto; try omega.
assert (maxaddr_help f k ≥ maxaddr_help f a).
apply maxaddr_help_monotonic; auto.
omega.
assert (maxaddr_help f k ≥ maxaddr_help f 0).
apply maxaddr_help_monotonic; omega.
omega.
Qed.
```

```
Lemma maxaddr_max : ∀ f n, n > maxaddr f → in_fl f n.
Proof.
unfold in_fl; intros.
induction f; auto.
simpl; intuition; subst.
apply maxaddr_max_help1 in H; omega.
apply IHf; auto.
apply maxaddr_max_help2 in H; auto.
Qed.
```

0.4.3 Definitions and lemmas for updating state/blocks

```
Definition upd_s s x v : store := fun y ⇒ if eq_nat_dec y x then v else s y.
Notation "s [ x ↦ v ]" := (upd_s s x v) (at level 2).
```

```
Notation "h [ n → v ]" := (upd h n v) (at level 2).
```

```
Fixpoint upd_block (h : heap) n vs : heap :=
  match vs with
```

```
  | v::vs ⇒ (upd_block h (n+1) vs)[n→v]
  | [] ⇒ h
  end.
Notation "h [ n ⇒ vs ]" := (upd_block h n vs) (at level 2).

Fixpoint add_fl (f : freelist) n k : freelist :=
  match k with
  | 0 ⇒ f
  | S k ⇒ remove eq_nat_dec n (add_fl f (n+1) k)
  end.

Fixpoint del_fl (f : freelist) n k : freelist :=
  match k with
  | 0 ⇒ f
  | S k ⇒ n :: del_fl f (n+1) k
  end.

Lemma upd_s_simpl : ∀ s x v, s[x↦v] x = v.
Proof.
unfold upd_s; intros.
destruct (eq_nat_dec x x); auto.
contradiction n; auto.
Qed.

Lemma upd_s_simpl_neq : ∀ s x y v, y ≠ x → s[x↦v] y = s y.
Proof.
unfold upd_s; intros.
destruct (eq_nat_dec y x); auto.
contradiction.
Qed.

Lemma disj_upd : ∀ (h1 h2 : heap) n v, h1 # h2 → haskey h1 n → h1[n→v] # h2.
Proof.
unfold disjoint; intros.
destruct (eq_nat_dec n0 n); subst.
apply H; auto.
apply H.
rewrite upd_haskey in H1; intuition; subst.
contradiction n1; auto.
Qed.

Lemma disjhf_upd : ∀ h f n v, disjhf h f → haskey h n → disjhf h[n→v] f.
Proof.
unfold disjhf; intros.
destruct (eq_nat_dec n0 n); subst.
apply H; auto.
apply H.
```

rewrite *upd_haskey* in *H1*; intuition; subst.
contradiction n1; auto.
Qed.

Lemma *disjhf_dot* : $\forall$ *h1 h2 f*, *disjhf* (*h1@h2*) *f* $\leftrightarrow$ *disjhf h1 f* $\wedge$ *disjhf h2 f*.
Proof.
intros.
unfold *disjhf*; intuition.
apply (*H n*); auto.
apply *haskey_union*; auto.
apply (*H n*); auto.
apply *haskey_union_frame*; auto.
destruct (*haskey_dec h1 n*).
apply (*H0 n*); auto.
apply *haskey_mapsto* in *H*; destruct *H*.
apply *mapsto_union_inversion* in *H*; destruct *H*.
contradiction n0; apply *mapsto_haskey* in *H*; auto.
apply (*H1 n*); intuition.
apply *mapsto_haskey* in *H4*; auto.
Qed.

Lemma *dot_upd_comm* : $\forall$ (*h1 h2* : *heap*) *n v*, *h1*[$n$$\longrightarrow$v] @ *h2* = (*h1@h2*)[$n$$\longrightarrow$v].
Proof.
unfold *upd*; unfold *union*; intros; simpl; auto.
Qed.

Lemma *dot_upd_block_comm* : $\forall$ (*h1 h2* : *heap*) *vs n*, *h1*[$n$$\Rightarrow$vs] @ *h2* = (*h1@h2*)[$n$$\Rightarrow$vs].
Proof.
induction *vs*; intros; simpl; auto.
rewrite (*IHvs* (n+1)); auto.
Qed.

Lemma *upd_haskey_block* : $\forall$ (*h* : *heap*) *n vs k*, *haskey h*[$n$$\Rightarrow$vs] *k* $\leftrightarrow$ ($k \geq n \wedge k < n +$ *length vs*) $\vee$ *haskey h k*.
Proof.
intros *h n vs*.
generalize *h n*; clear *h n*.
induction *vs*; intros; simpl; split; intros; auto.
intuition.
assert *False*.
omega.
inv H0.
rewrite *upd_haskey* in *H*; destruct *H*; subst.
left; split; omega.
destruct *H*.

```
rewrite IHvs in H0; intuition.
rewrite upd_haskey.
rewrite IHvs.
destruct (eq_nat_dec n k); intuition.
Qed.
```

```
Lemma dot_del_comm : ∀ (h1 h2 : heap) n, ¬ haskey h2 n → (del h1 n)@h2 = del (h1@h2)
n.
Proof.
induction h1; intros; simpl.
apply sym_eq; apply del_not_haskey; auto.
destruct a.
destruct (eq_nat_dec n n0); simpl; auto.
rewrite (IHh1 _ _ H); auto.
Qed.
```

0.4.4 Expression Evaluation

```
Open Scope Z_scope.
```

```
Fixpoint exp_val (s : store) e :=
  match e with
  | Exp_val v ⇒ v
  | Exp_nil ⇒ -1
  | Exp_var x ⇒ s x
  | Exp_op op e1 e2 ⇒ op_val op (exp_val s e1) (exp_val s e2)
  end.
```

```
Fixpoint bexp_val (s : store) b :=
  match b with
  | BExp_eq e1 e2 ⇒ if Z_eq_dec (exp_val s e1) (exp_val s e2) then true else false
  | BExp_false ⇒ false
  | BExp_bop bop b1 b2 ⇒ bop_val bop (bexp_val s b1) (bexp_val s b2)
  end.
```

```
Open Scope list_scope.
Open Scope nat_scope.
```

0.4.5 Operational Semantics

```
Inductive step : config → config → Prop :=
| Step_skip :
    ∀ st C,
      step (st, Skip ;; C) (st, C)
| Step_assgn :
```

$\forall\ s\ h\ f\ x\ e,$
$\quad step\ (St\ s\ h\ f,\ x ::= e)\ (St\ s[x \mapsto exp\_val\ s\ e]\ h\ f,\ Skip)$

$|\ Step\_read :$
$\forall\ s\ h\ f\ x\ e\ n\ v,$
$\quad exp\_val\ s\ e = Z\_of\_nat\ n \rightarrow mapsto\ h\ n\ v \rightarrow$
$\quad step\ (St\ s\ h\ f,\ x ::= [[e]])\ (St\ s[x \mapsto v]\ h\ f,\ Skip)$

$|\ Step\_write :$
$\forall\ s\ h\ f\ e\ e'\ n,$
$\quad exp\_val\ s\ e = Z\_of\_nat\ n \rightarrow haskey\ h\ n \rightarrow$
$\quad step\ (St\ s\ h\ f,\ [[e]] ::= e')\ (St\ s\ h[n \rightarrow exp\_val\ s\ e']\ f,\ Skip)$

$|\ Step\_cons :$
$\forall\ s\ h\ f\ x\ es\ n,$
$\quad (\forall\ i : nat,\ i < length\ es \rightarrow in\_fl\ f\ (n+\mathrm{i})) \rightarrow$
$\quad step\ (St\ s\ h\ f,\ Cons\ x\ es)\ (St\ s[x \mapsto Z\_of\_nat\ n]\ h[n \Rightarrow map\ (exp\_val\ s)\ es]\ (del\_fl\ f\ n\ (length\ es)),\ Skip)$

$|\ Step\_free :$
$\forall\ s\ h\ f\ e\ n,$
$\quad exp\_val\ s\ e = Z\_of\_nat\ n \rightarrow haskey\ h\ n \rightarrow$
$\quad step\ (St\ s\ h\ f,\ Free\ e)\ (St\ s\ (del\ h\ n)\ (add\_fl\ f\ n\ 1),\ Skip)$

$|\ Step\_seq :$
$\forall\ st\ st'\ C\ C'\ C'',$
$\quad step\ (st,\ C)\ (st',\ C') \rightarrow step\ (st,\ C;;C'')\ (st',\ C';;C'')$

$|\ Step\_if\_true :$
$\forall\ st\ b\ C1\ C2,$
$\quad bexp\_val\ (Store\ st)\ b = true \rightarrow step\ (st,\ if\_\ b\ \texttt{then}\ C1\ \texttt{else}\ C2)\ (st,\ C1)$

$|\ Step\_if\_false :$
$\forall\ st\ b\ C1\ C2,$
$\quad bexp\_val\ (Store\ st)\ b = false \rightarrow step\ (st,\ if\_\ b\ \texttt{then}\ C1\ \texttt{else}\ C2)\ (st,\ C2)$

$|\ Step\_while\_true :$
$\forall\ st\ b\ C',$
$\quad bexp\_val\ (Store\ st)\ b = true \rightarrow step\ (st,\ while\ b\ \texttt{do}\ C')\ (st,\ C' ;;\ while\ b\ \texttt{do}\ C')$

$|\ Step\_while\_false :$
$\forall\ st\ b\ C',$
$\quad bexp\_val\ (Store\ st)\ b = false \rightarrow step\ (st,\ while\ b\ \texttt{do}\ C')\ (st,\ Skip).$

$\texttt{Inductive}\ stepn : nat \rightarrow config \rightarrow config \rightarrow \texttt{Prop} :=$
$|\ Stepn\_zero : \forall\ cf,\ stepn\ O\ cf\ cf$
$|\ Stepn\_succ : \forall\ n\ cf\ cf'\ cf'',\ step\ cf\ cf' \rightarrow stepn\ n\ cf'\ cf'' \rightarrow stepn\ (S\ n)\ cf\ cf''.$

$\texttt{Definition}\ multi\_step\ cf\ cf' := \exists\ n,\ stepn\ n\ cf\ cf'.$
$\texttt{Definition}\ halt\_config\ (cf : config) := snd\ cf = Skip.$
$\texttt{Definition}\ safe\ (cf : config) := \forall\ cf',\ multi\_step\ cf\ cf' \rightarrow \neg\ halt\_config\ cf' \rightarrow \exists\ cf'',\ step\ cf'\ cf''.$
$\texttt{Definition}\ diverges\ (cf : config) := \forall\ n,\ \exists\ cf',\ stepn\ n\ cf\ cf'.$

0.4.6 Facts about stepping

Lemma *safe_step* : ∀ *cf*, *safe cf* → ¬ *halt_config cf* → ∃ *cf'*, *step cf cf'*.
Proof.
unfold *safe*; intros.
apply *H*; auto.
∃ 0; apply *Stepn_zero*.
Qed.

Lemma *safe_step_safe* : ∀ *cf cf'*, *safe cf* → *step cf cf'* → *safe cf'*.
Proof.
unfold *safe*; intros.
apply *H*; auto.
destruct *H1*.
∃ (*S x*); apply *Stepn_succ* with (*cf'* := *cf'*); auto.
Qed.

Lemma *safe_stepn_safe* : ∀ *n cf cf'*, *safe cf* → *stepn n cf cf'* → *safe cf'*.
Proof.
induction *n*; intros.
inversion *H0*; subst; auto.
inversion *H0*; subst.
apply *safe_step_safe* in *H2*; auto.
apply *IHn* with (*cf* := *cf'0*); auto.
Qed.

Lemma *safe_multi_step_safe* : ∀ *cf cf'*, *safe cf* → *multi_step cf cf'* → *safe cf'*.
Proof.
intros; destruct *H0*; apply *safe_stepn_safe* in *H0*; auto.
Qed.

Lemma *stepn_seq* : ∀ *n st st' C C' C”*, *stepn n* (*st*,C) (*st'*,C') → *stepn n* (*st*, *C*;;*C”*) (*st'*, *C'*;;*C”*).
Proof.
induction *n*; intros.
inversion *H*; subst; apply *Stepn_zero*.
inversion *H*; subst; clear *H*.
destruct *cf'*.
apply *Stepn_succ* with (*cf'* := (*s*,c;;*C”*)).
apply *Step_seq*; auto.
apply *IHn*; auto.
Qed.

Lemma *multi_step_seq* : ∀ *st st' C C' C”*, *multi_step* (*st*,C) (*st'*,C') → *multi_step* (*st*, *C*;;*C”*) (*st'*, *C'*;;*C”*).
Proof.

intros; unfold *multi_step* in ×.
destruct *H*.
∃ *x*; apply *stepn_seq*; auto.
Qed.

Lemma *safe_seq* : ∀ *st C C'*, *safe* (*st*, *C* ;; *C'*) → *safe* (*st*, *C*).
Proof.
unfold *safe*; intros.
destruct *cf'*.
assert (¬ *halt_config* (*s*, *c*;;*C'*)).
unfold *halt_config*; simpl; discriminate.
apply *H* in *H2*.
destruct *H2*; *inv H2*.
contradiction H1; unfold *halt_config*; auto.
∃ (*st'*,C'0); auto.
apply *multi_step_seq*; auto.
Qed.

0.4.7 Well-definedness of states (i.e., heap and free list don't overlap)

Definition *wd* (*st* : *state*) := let (_,h,f) := *st* in *disjhf h f*.

Lemma *wd_step* : ∀ *C C' st st'*, *step* (*st*,C) (*st'*,C') → *wd st* → *wd st'*.
Proof.
induction *C*; intros; *inv H*; auto; simpl in ×.

apply *disjhf_upd*; auto.

generalize *n H2*; clear *n H2*.
induction *l*; intros; simpl in ×; auto.
unfold *disjhf*; intros; unfold *in_fl*.
rewrite *upd_haskey* in *H*; destruct *H*; subst.
intro *H*; *contradiction H*.
simpl; auto.
destruct *H*.
apply *IHl* in *H1*.
intro.
simpl in *H3*; intuition.
intros.
rewrite ← *plus_assoc*.
apply *H2*; omega.

unfold *disjhf*; intros.
rewrite *del_haskey* in *H*; destruct *H*.
apply *H0* in *H*.

```
intro; contradiction H.
unfold in_fl in H2 ⊢ ×.
intro; contradiction H2.
rewrite in_remove; split; auto.
apply IHC1 in H2; intuition.
Qed.
```

```
Lemma wd_stepn : ∀ n C C' st st', stepn n (st,C) (st',C') → wd st → wd st'.
Proof.
induction n; intros.
inv H; auto.
inv H.
destruct cf'.
apply IHn in H3; auto.
apply wd_step in H2; auto.
Qed.
```

```
Lemma wd_multi_step : ∀ C C' st st', multi_step (st,C) (st',C') → wd st → wd st'.
Proof.
intros.
inv H.
apply wd_stepn in H1; auto.
Qed.
```

0.4.8 Major Theorems: Forwards and Backwards Frame Properties, Safety Monotonicity, and Termination Equivalence

```
Lemma forwards_frame_property_step :
  ∀ C C' s h0 f s' h0' f' h1,
    step (St s h0 f, C) (St s' h0' f', C') → h0 # h1 → disjhf h1 f →
      h0' # h1 ∧ step (St s h0@h1 f, C) (St s' h0'@h1 f', C').
Proof.
induction C; intros.
```

Skip

```
inv H.
```

Assgn

```
inv H; split; auto; apply Step_assgn.
```

Read

```
inv H; split; auto.
apply Step_read with (n := n); auto.
apply mapsto_union; auto.
```

Write

inv H; `split`.
`apply` *disj_upd*; `auto`.
`rewrite` *dot_upd_comm*.
`apply` *Step_write*; `auto`.
`apply` *haskey_union*; `auto`.

Cons

inv H; `split`.
`unfold` *disjoint* `in` ×; `unfold` *disjhf* `in` ×; `intros`.
`rewrite` *upd_haskey_block* `in` *H*; `destruct` *H*.
`rewrite` *map_length* `in` *H*; `destruct` *H*.
specialize (*H3* (*n0*-n)).
`assert` ($n + (n0\text{-n}) = n0$).
`omega`.
`rewrite` *H4* `in` *H3*.
`intro`.
contradiction (*H1 n0 H5*).
`apply` *H3*; `omega`.
`apply` *H0*; `auto`.
`rewrite` *dot_upd_block_comm*; `apply` *Step_cons*; `auto`.

Free

inv H; `split`.
`unfold` *disjoint*; `intros`.
`apply` *H0*.
`rewrite` *del_haskey* `in` *H*; `intuition`.
`rewrite` *dot_del_comm*.
`apply` *Step_free*; `auto`.
`apply` *haskey_union*; `auto`.
`apply` *H0*; `auto`.

Seq

inv H.
`split`; `auto`.
`apply` *Step_skip*.
`apply` *IHC1* `with` (*h1* := *h1*) `in` *H3*; `intuition`.
`apply` *Step_seq*; `auto`.

If

inv H; `split`; `auto`.
`apply` *Step_if_true*; `auto`.
`apply` *Step_if_false*; `auto`.

While

inv H; `split`; `auto`.

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

Theorem 2.1 from paper

```
Theorem forwards_frame_property :
  ∀ n C C' s h0 f s' h0' f' h1,
    stepn n (St s h0 f, C) (St s' h0' f', C') → wd (St s h0 f) → h0 # h1 → disjhf h1 f
→
      h0' # h1 ∧ stepn n (St s h0@h1 f, C) (St s' h0'@h1 f', C').
Proof.
induction n; unfold wd; intros.
inv H; split; auto.
apply Stepn_zero.
inv H.
destruct cf'; destruct s0.
apply IHn with (h1 := h1) in H5; intuition.
apply forwards_frame_property_step with (h1 := h1) in H4; intuition.
apply (Stepn_succ _ _ _ _ H6 H3).
apply wd_step in H4; auto.
apply forwards_frame_property_step with (h1 := h1) in H4; intuition.
apply forwards_frame_property_step with (h1 := h1) in H4; intuition.
apply wd_step in H3; simpl in ×.
rewrite disjhf_dot in H3; intuition.
rewrite disjhf_dot; intuition.
Qed.
```

```
Lemma forwards_frame_property_multi_step :
  ∀ C C' s h0 f s' h0' f' h1,
    multi_step (St s h0 f, C) (St s' h0' f', C') → wd (St s h0 f) → h0 # h1 → disjhf h1
f →
      h0' # h1 ∧ multi_step (St s h0@h1 f, C) (St s' h0'@h1 f', C').
Proof.
unfold multi_step; intros.
destruct H; apply forwards_frame_property with (h1 := h1) in H; intuition.
∃ x; auto.
Qed.
```

```
Lemma backwards_frame_property_step :
  ∀ C C' s h0 f s' f' h1 h',
    h0 # h1 → step (St s h0@h1 f, C) (St s' h' f', C') → wd (St s h0@h1 f) → safe (St
s h0 f, C) →
      ∃ h0', h0' # h1 ∧ h' = h0'@h1 ∧ step (St s h0 f, C) (St s' h0' f', C').
Proof.
induction C; unfold wd; intros.
```

Skip

inv H0.

Assgn

inv H0; ∃ *h0*; `intuition`.
`apply` *Step_assgn*.

Read

inv H0; ∃ *h0*; `intuition`.
`apply` *Step_read* `with` $(n := n)$; `auto`.
`apply` *safe_step* `in` *H2*; `try discriminate`.
`destruct` *H2*.
inv H0.
`apply` *mapsto_haskey* `in` *H10*; *znat_simpl H5 H9*.
`apply` *mapsto_union_inversion* `in` *H13*; `intuition`.

Write

inv H0.
∃ *h0*[*n*→exp_val *s' e0*].
`assert` (*haskey h0 n*).
`apply` *safe_step* `in` *H2*; `try discriminate`.
`destruct` *H2*.
inv H0.
znat_simpl H5 H9; `auto`.
`intuition`.
`apply` *disj_upd*; `auto`.
`apply` *Step_write*; `auto`.

Cons

inv H0.
∃ *h0*[*n*⇒map (*exp_val s*) *l*]; `intuition`.
`unfold` *disjoint* `in` ×; `unfold` *disjhf* `in` ×; `intros`.
`rewrite` *upd_haskey_block* `in` *H0*; `destruct` *H0*.
`rewrite` *map_length* `in` *H0*; `destruct` *H0*.
specialize (*H4* (*n0*-n)).
`assert` $(n + (n0\text{-n}) = n0)$.
`omega`.
`rewrite` *H5* `in` *H4*.
`intro`.
`apply` *haskey_union_frame* `with` $(m1 := h0)$ `in` *H6*.
contradiction (*H1* _ *H6*).
`apply` *H4*; `omega`.
`apply` *H*; `auto`.
`rewrite` *dot_upd_block_comm*; `auto`.
`apply` *Step_cons*; `auto`.

Free

```
inv H0.
assert (haskey h0 n).
apply safe_step in H2; try discriminate.
destruct H2.
inv H0.
znat_simpl H5 H8; auto.
∃ (del h0 n); intuition.
unfold disjoint; intros.
rewrite del_haskey in H3.
apply H; intuition.
rewrite dot_del_comm; auto.
apply Step_free; auto.
```

Seq

```
inv H0.
∃ h0; intuition.
apply Step_skip.
apply safe_seq in H2.
apply IHC1 in H4; auto.
destruct H4 as [h0']; ∃ h0'; intuition.
apply Step_seq; auto.
```

If

```
∃ h0.
inv H0; intuition.
apply Step_if_true; auto.
apply Step_if_false; auto.
```

While

```
∃ h0.
inv H0; intuition.
apply Step_while_true; auto.
apply Step_while_false; auto.
Qed.
```

Theorem 2.2 from paper

```
Theorem backwards_frame_property :
  ∀ n C C' s h0 f s' f' h1 h',
    h0 # h1 → stepn n (St s h0@h1 f, C) (St s' h' f', C') → wd (St s h0@h1 f) → safe
(St s h0 f, C) →
      ∃ h0', h0' # h1 ∧ h' = h0'@h1 ∧ stepn n (St s h0 f, C) (St s' h0' f', C').
Proof.
induction n; intros.
inv H0; ∃ h0; intuition.
```

apply *Stepn_zero*.
inv H0.
destruct *cf'*; destruct *s0*.
dup H4; apply *backwards_frame_property_step* in *H4*; auto.
destruct *H4* as [*h0'*]; intuition; subst.
apply *IHn* in *H5*; auto.
destruct *H5* as [*h0"*]; intuition; subst.
∃ *h0"*; intuition.
apply (*Stepn_succ _ _ _ _ H7 H8*).
unfold *wd* in *H1* ⊢ ×; rewrite *disjhf_dot* in *H1* ⊢ ×; intuition.
apply *wd_step* in *H7*; auto.
apply *wd_step* in *H0*.
simpl in *H0*; rewrite *disjhf_dot* in *H0*; intuition.
simpl; rewrite *disjhf_dot*; intuition.
apply (*safe_step_safe _ _ H2 H7*).
Qed.

Lemma *backwards_frame_property_multi_step* :
  ∀ *C C' s h0 f s' f' h1 h'*,
    h0 # h1 → *multi_step* (*St s h0@h1 f*, *C*) (*St s' h' f'*, *C'*) → *wd* (*St s h0@h1 f*) → *safe* (*St s h0 f*, *C*) →
      ∃ *h0'*, *h0' # h1* ∧ *h' = h0'@h1* ∧ *multi_step* (*St s h0 f*, *C*) (*St s' h0' f'*, *C'*).
Proof.
unfold *multi_step*; intros.
destruct *H0*.
apply *backwards_frame_property* in *H0*; auto.
destruct *H0* as [*h0'*]; ∃ *h0'*; intuition.
∃ *x*; auto.
Qed.

Lemma 3 from paper

Theorem *safety_monotonicity* :
  ∀ *C s f h0 h1*,
    safe (*St s h0 f*, *C*) → *wd* (*St s h0 f*) → *h0 # h1* → *disjhf h1 f* → *safe* (*St s h0@h1 f*, *C*).
Proof.
unfold *safe*; unfold *wd*; intros.
destruct *cf'*; destruct *s0*.
apply *backwards_frame_property_multi_step* in *H3*; auto.
destruct *H3* as [*h0'*]; intuition.
dup H7; apply *H* in *H7*; auto.
destruct *H7* as [*cf'*]; destruct *cf'*; destruct *s1*.
apply *forwards_frame_property_step* with (*h1* := *h1*) in *H7*; auto; subst; intuition.
∃ (*St s1 h2@h1 f1*, *c0*); auto.

```
apply forwards_frame_property_multi_step with (h1 := h1) in H6; auto; intuition.
apply wd_multi_step in H8; simpl in ×.
rewrite disjhf_dot in H8; intuition.
rewrite disjhf_dot; intuition.
simpl; rewrite disjhf_dot; intuition.
Qed.
```

Lemma 4 from paper

```
Theorem termination_equivalence :
  ∀ C s f h0 h1,
    safe (St s h0 f, C) → wd (St s h0 f) → h0 # h1 → disjhf h1 f →
    (diverges (St s h0 f, C) ↔ diverges (St s h0@h1 f, C)).
Proof.
unfold diverges; unfold wd; intuition.
destruct (H3 n) as [ [ [s' h0' f'] C'] ].
apply forwards_frame_property with (h1 := h1) in H4; auto; intuition.
∃ (St s' h0'@h1 f', C'); auto.
destruct (H3 n) as [ [ [s' h' f'] C'] ].
apply backwards_frame_property in H4; auto.
destruct H4 as [h0']; intuition.
∃ (St s' h0' f', C'); auto.
simpl; rewrite disjhf_dot; intuition.
Qed.
```

0.5 Soundness and Completeness, relative to standard Separation Logic

```
Inductive state_sl := St_sl : store → heap → state_sl.
Definition Store_sl (st : state_sl) := let (s,_) := st in s.
Definition Heap_sl (st : state_sl) := let (_,h) := st in h.
Definition config_sl := prod state_sl cmd.
```

```
Inductive step_sl : config_sl → config_sl → Prop :=
| Step_sl_skip :
    ∀ st C,
      step_sl (st, Skip ;; C) (st, C)
| Step_sl_assgn :
    ∀ s h x e,
      step_sl (St_sl s h, x ::= e) (St_sl s[x ↦ exp_val s e] h, Skip)
| Step_sl_read :
    ∀ s h x e n v,
      exp_val s e = Z_of_nat n → mapsto h n v →
      step_sl (St_sl s h, x ::= [[e]]) (St_sl s[x ↦ v] h, Skip)
```

| $Step\_sl\_write$:
  $\forall$ s h e e' n,
    $exp\_val\ s\ e = Z\_of\_nat\ n \rightarrow haskey\ h\ n \rightarrow$
    $step\_sl\ (St\_sl\ s\ h,\ [[e]] ::= e')\ (St\_sl\ s\ h[n \rightarrow exp\_val\ s\ e'],\ Skip)$
| $Step\_sl\_cons$:
  $\forall$ s h x es n,
    $(\forall\ i : nat,\ i < length\ es \rightarrow \neg\ haskey\ h\ (n{+}\mathrm{i})) \rightarrow$
    $step\_sl\ (St\_sl\ s\ h,\ Cons\ x\ es)\ (St\_sl\ s[x \mapsto Z\_of\_nat\ n]\ h[n \Rightarrow map\ (exp\_val\ s)\ es],$
$Skip)$
| $Step\_sl\_free$:
  $\forall$ s h e n,
    $exp\_val\ s\ e = Z\_of\_nat\ n \rightarrow haskey\ h\ n \rightarrow$
    $step\_sl\ (St\_sl\ s\ h,\ Free\ e)\ (St\_sl\ s\ (del\ h\ n),\ Skip)$
| $Step\_sl\_seq$:
  $\forall$ st st' C C' C'',
    $step\_sl\ (st,\ C)\ (st',\ C') \rightarrow step\_sl\ (st,\ C;;C'')\ (st',\ C';;C'')$
| $Step\_sl\_if\_true$:
  $\forall$ st b $C1$ $C2$,
    $bexp\_val\ (Store\_sl\ st)\ b = true \rightarrow step\_sl\ (st,\ if\_\ b$ `then` $C1$ `else` $C2)\ (st,\ C1)$
| $Step\_sl\_if\_false$:
  $\forall$ st b $C1$ $C2$,
    $bexp\_val\ (Store\_sl\ st)\ b = false \rightarrow step\_sl\ (st,\ if\_\ b$ `then` $C1$ `else` $C2)\ (st,\ C2)$
| $Step\_sl\_while\_true$:
  $\forall$ st b C',
    $bexp\_val\ (Store\_sl\ st)\ b = true \rightarrow step\_sl\ (st,\ while\ b$ `do` $C')\ (st,\ C'\ ;;\ while\ b$ `do` $C')$
| $Step\_sl\_while\_false$:
  $\forall$ st b C',
    $bexp\_val\ (Store\_sl\ st)\ b = false \rightarrow step\_sl\ (st,\ while\ b$ `do` $C')\ (st,\ Skip)$.

`Inductive` $stepn\_sl : nat \rightarrow config\_sl \rightarrow config\_sl \rightarrow$ `Prop` :=
| $Stepn\_sl\_zero : \forall\ cf,\ stepn\_sl\ O\ cf\ cf$
| $Stepn\_sl\_succ : \forall\ n\ cf\ cf'\ cf'',\ step\_sl\ cf\ cf' \rightarrow stepn\_sl\ n\ cf'\ cf'' \rightarrow stepn\_sl\ (S\ n)\ cf$
cf''.

`Definition` $multi\_step\_sl\ cf\ cf' := \exists\ n,\ stepn\_sl\ n\ cf\ cf'$.
`Definition` $halt\_config\_sl\ (cf : config\_sl) := snd\ cf = Skip$.
`Definition` $safe\_sl\ (cf : config\_sl) := \forall\ cf',\ multi\_step\_sl\ cf\ cf' \rightarrow \neg\ halt\_config\_sl\ cf' \rightarrow$
$\exists\ cf'',\ step\_sl\ cf'\ cf''$.
`Definition` $diverges\_sl\ (cf : config\_sl) := \forall\ n,\ \exists\ cf',\ stepn\_sl\ n\ cf\ cf'$.

0.5.1 Bisimulation between stepn and stepn_sl

(exclude h) represents a canonical-form freelist which has all locations except for those in the domain of h

```
Fixpoint remove_dup (f : freelist) : freelist :=
  match f with
  | [] ⇒ []
  | n::f ⇒ if in_fl_dec f n then n :: remove_dup f else remove_dup f
  end.

Definition exclude (h : heap) : freelist := remove_dup (map (fst (B:=val)) h).

Lemma in_dec : ∀ (f : freelist) n, {In n f} + {¬ In n f}.
Proof.
induction f; intros; simpl; auto.
destruct (IHf n); destruct (eq_nat_dec a n); intuition.
Qed.

Lemma remove_dup_in : ∀ f n, In n (remove_dup f) ↔ In n f.
Proof.
induction f; intros; split; intros; auto; simpl in ×.
destruct (eq_nat_dec a n); auto.
right; rewrite ← IHf.
destruct (in_fl_dec f a); auto.
simpl in H; intuition.
destruct H; subst.
destruct (in_fl_dec f n); simpl; auto.
unfold in_fl in n0.
rewrite IHf; apply (double_neg _ (in_dec _ _)); auto.
rewrite ← IHf in H.
destruct (in_fl_dec f a); simpl; auto.
Qed.

Lemma haskey_in_fst : ∀ (h : heap) n, In n (map (fst (B:=val)) h) ↔ haskey h n.
Proof.
induction h; intros; split; intros; simpl in ×; auto; try contradiction; destruct a as
[k v]; simpl in ×.
change (haskey h[k→v] n); rewrite upd_haskey.
rewrite IHh in H.
destruct (eq_nat_dec k n); intuition.
change (haskey h[k→v] n) in H; rewrite upd_haskey in H.
rewrite ← IHh in H.
destruct (eq_nat_dec k n); intuition.
Qed.

Lemma haskey_exclude : ∀ h n, ¬ in_fl (exclude h) n ↔ haskey h n.
Proof.
unfold in_fl; unfold exclude; intros; split; intros.
apply (double_neg _ (in_dec _ _)) in H.
rewrite remove_dup_in in H; rewrite haskey_in_fst in H; auto.
```

rewrite ← *haskey_in_fst* in H; rewrite ← *remove_dup_in* in H; auto.
Qed.

Lemma *exclude_write* : ∀ *h n v*, *haskey h n* → *exclude h*[*n*→v] = *exclude h*.
Proof.
intros; unfold *upd*; unfold *exclude*; simpl.
destruct (*in_fl_dec* (*map* (*fst* (*B*:=val)) *h*) *n*); auto.
rewrite ← *haskey_exclude* in H; *contradiction H*; unfold *exclude*.
clear H; unfold *in_fl* in ×.
intro; *contradiction i*.
rewrite *remove_dup_in* in H; auto.
Qed.

Lemma *exclude_cons_help* : ∀ *h n v*, ¬ *haskey h n* → *exclude h*[*n*→v] = *n* :: *exclude h*.
Proof.
intros; unfold *upd*; unfold *exclude*; simpl.
destruct (*in_fl_dec* (*map* (*fst* (*B*:=val)) *h*) *n*); auto.
rewrite ← *haskey_exclude* in H; *contradiction H*; unfold *exclude*.
clear H; unfold *in_fl* in ×.
rewrite *remove_dup_in*; auto.
Qed.

Lemma *exclude_cons* : ∀ *vs h n*,
(∀ *i*, *i* < *length vs* → ¬ *haskey h* (*n*+i)) → *exclude h*[*n*⇒vs] = *del_fl* (*exclude h*) *n* (*length vs*).
Proof.
induction *vs*; intros; simpl in ×; auto.
rewrite *exclude_cons_help*.
rewrite (*IHvs* _ (*n*+1)); auto.
intros.
rewrite ← *plus_assoc*; apply H; omega.
rewrite *upd_haskey_block*; intuition.
apply (H 0); try omega.
rewrite *plus_0_r*; auto.
Qed.

Lemma *exclude_free* : ∀ *h n*, *exclude* (*del h n*) = *add_fl* (*exclude h*) *n* 1.
Proof.
induction *h*; intros; unfold *exclude* in ×; simpl; auto; destruct *a* as [*k v*]; simpl in ×.
case_eq (*eq_nat_dec n k*); *case_eq* (*in_fl_dec* (*map* (*fst* (*B*:=val)) *h*) *k*); intros; simpl; auto.
rewrite *H0*; auto.
rewrite *H0*.
destruct (*in_fl_dec* (*map* (*fst* (*B*:=val)) (*del h n*)) *k*).

```
rewrite IHh; auto.
unfold in_fl in ×.
apply (double_neg _ (in_dec _ _)) in n1; contradiction i.
rewrite haskey_in_fst in n1 ⊢ ×.
rewrite del_haskey in n1; intuition.
destruct (in_fl_dec (map (fst (B:=val)) (del h n)) k); auto.
unfold in_fl in ×.
clear H; apply (double_neg _ (in_dec _ _)) in n0; contradiction i.
rewrite haskey_in_fst in n0 ⊢ ×.
rewrite del_haskey; intuition.
Qed.

Lemma exclude_wd : ∀ s h, wd (St s h (exclude h)).
Proof.
unfold wd; unfold disjhf; intros.
rewrite haskey_exclude; auto.
Qed.

Lemma step_bisim_forwards :
  ∀ C C' s h s' h',
    step_sl (St_sl s h, C) (St_sl s' h', C') → step (St s h (exclude h), C) (St s' h' (exclude
h'), C').
Proof.
induction C; intros; inv H; simpl in ×.
apply Step_assgn.
apply Step_read with (n := n); auto.
rewrite exclude_write; auto; apply Step_write; auto.
rewrite exclude_cons.
rewrite map_length; apply Step_cons.
intros.
apply H1 in H.
rewrite ← haskey_exclude in H; unfold in_fl in ×.
intro; contradiction H.
intro; contradiction.
rewrite map_length; auto.
rewrite exclude_free; apply Step_free; auto.
apply Step_skip.
apply Step_seq; apply IHC1 in H1; auto.
apply Step_if_true; auto.
apply Step_if_false; auto.
apply Step_while_true; auto.
apply Step_while_false; auto.
Qed.

Lemma stepn_bisim_forwards :
```

```
  ∀ n C C' s h s' h',
    stepn_sl n (St_sl s h, C) (St_sl s' h', C') → stepn n (St s h (exclude h), C) (St s' h'
(exclude h'), C').
Proof.
induction n; intros; inv H.
apply Stepn_zero.
destruct cf' as [ [s" h"] C"]; apply step_bisim_forwards in H1.
apply IHn in H2.
apply (Stepn_succ _ _ _ _ H1 H2).
Qed.

Lemma multi_step_bisim_forwards :
  ∀ C C' s h s' h',
    multi_step_sl (St_sl s h, C) (St_sl s' h', C') → multi_step (St s h (exclude h), C) (St
s' h' (exclude h'), C').
Proof.
unfold multi_step_sl; unfold multi_step; intros.
destruct H.
apply stepn_bisim_forwards in H; ∃ x; auto.
Qed.

Lemma step_bisim_backwards :
  ∀ C C' s h f s' h' f', wd (St s h f) →
    step (St s h f, C) (St s' h' f', C') → step_sl (St_sl s h, C) (St_sl s' h', C').
Proof.
induction C; intros; inv H0; simpl in ×.
apply Step_sl_assgn.
apply Step_sl_read with (n := n); auto.
apply Step_sl_write; auto.
apply Step_sl_cons.
intros; intro.
apply H2 in H0; apply H in H1; contradiction.
apply Step_sl_free; auto.
apply Step_sl_skip.
apply Step_sl_seq; apply IHC1 in H2; auto.
apply Step_sl_if_true; auto.
apply Step_sl_if_false; auto.
apply Step_sl_while_true; auto.
apply Step_sl_while_false; auto.
Qed.

Lemma stepn_bisim_backwards :
  ∀ n C C' s h f s' h' f', wd (St s h f) →
    stepn n (St s h f, C) (St s' h' f', C') → stepn_sl n (St_sl s h, C) (St_sl s' h', C').
Proof.
```

```
induction n; intros; inv H0.
apply Stepn_sl_zero.
destruct cf' as [ [s" h" f"] C"].
dup H2; apply wd_step in H2; auto.
apply step_bisim_backwards in H0; auto.
apply IHn in H3; auto.
apply (Stepn_sl_succ _ _ _ _ H0 H3).
Qed.
```

```
Lemma multi_step_bisim_backwards :
  ∀ C C' s h f s' h' f', wd (St s h f) →
    multi_step (St s h f, C) (St s' h' f', C') → multi_step_sl (St_sl s h, C) (St_sl s' h',
C').
Proof.
unfold multi_step; unfold multi_step_sl; intros.
destruct H0.
apply stepn_bisim_backwards in H0; auto; ∃ x; auto.
Qed.
```

```
Lemma stepn_bisim :
  ∀ n C C' s h s' h',
    stepn_sl n (St_sl s h, C) (St_sl s' h', C') ↔ ∃ f, ∃ f', wd (St s h f) ∧ stepn n (St s h
f, C) (St s' h' f', C').
Proof.
intros; split; intros.
∃ (exclude h); ∃ (exclude h'); split.
apply exclude_wd.
apply stepn_bisim_forwards; auto.
destruct H as [f [f' [H] ] ].
apply stepn_bisim_backwards in H0; auto.
Qed.
```

Lemma 2 from paper

```
Lemma step_all_freelists :
  ∀ C C' s h s' h' f,
    step_sl (St_sl s h, C) (St_sl s' h', C') → ∃ st, step (St s h f, C) (st, C').
Proof.
induction C; intros; inv H.
∃ (St s[v↦exp_val s e] h' f); apply Step_assgn.
∃ (St s[v↦v0] h' f); apply Step_read with (n := n); auto.
∃ (St s' h[n→exp_val s' e0] f); apply Step_write; auto.
set (a := S (maxaddr f)).
∃ (St s[v↦Z_of_nat a] h[a⇒map (exp_val s) l] (del_fl f a (length l))); apply Step_cons.
intros; simpl.
apply maxaddr_max; omega.
```

```
∃ (St s' (del h n) (add_fl f n 1)); apply Step_free; auto.
∃ (St s' h' f); apply Step_skip.
apply IHC1 with (f := f) in H1.
destruct H1 as [st]; ∃ st; apply Step_seq; auto.
∃ (St s' h' f); apply Step_if_true; auto.
∃ (St s' h' f); apply Step_if_false; auto.
∃ (St s' h' f); apply Step_while_true; auto.
∃ (St s' h' f); apply Step_while_false; auto.
Qed.
```

0.5.2 Definitions of assertions and triples

```
Definition assert := store → heap → Prop.
Definition sat (p : assert) (st : state) := let (s,h,f) := st in p s h ∧ wd (St s h f).
Definition sat_sl (p : assert) (st : state_sl) := let (s,h) := st in p s h.

Inductive triple := Trip : assert → cmd → assert → triple.
Definition Pre (t : triple) := let (p,_,_) := t in p.
Definition Cmd (t : triple) := let (_,C,_) := t in C.
Definition Post (t : triple) := let (_,_,q) := t in q.
```

Derivability is the same in both logics

```
Parameter derivable : triple → Prop.

Definition safe_triple (t : triple) := ∀ st, sat (Pre t) st → safe (st, Cmd t).
Definition correct_triple (t : triple) :=
  ∀ st st', sat (Pre t) st → multi_step (st, Cmd t) (st', Skip) → sat (Post t) st'.
Definition valid t := safe_triple t ∧ correct_triple t.

Definition safe_triple_sl (t : triple) := ∀ st, sat_sl (Pre t) st → safe_sl (st, Cmd t).
Definition correct_triple_sl (t : triple) :=
  ∀ st st', sat_sl (Pre t) st → multi_step_sl (st, Cmd t) (st', Skip) → sat_sl (Post t) st'.
Definition valid_sl t := safe_triple_sl t ∧ correct_triple_sl t.
```

0.5.3 Soundness and completeness proofs

Axiom: standard separation logic is assumed to be sound and complete

```
Axiom soundness_and_completeness_sl : ∀ t, derivable t ↔ valid_sl t.
```

Theorem 1 from the paper

```
Theorem soundness_and_completeness : ∀ t, derivable t ↔ valid t.
Proof.
intros; rewrite soundness_and_completeness_sl; split; intros.

destruct H; split.
unfold safe_triple; unfold sat; unfold safe; intros.
destruct st as [s h f]; intuition.
```

```
destruct cf' as [ [s' h' f'] C']; apply multi_step_bisim_backwards in H2; auto.
apply H in H2; auto.
destruct H2; auto.
destruct x as [ [s'' h''] C''].
apply step_all_freelists with (f := f') in H1; destruct H1 as [st].
∃ (st, C''); auto.
unfold correct_triple; unfold sat; intros.
destruct st as [s h f]; destruct st' as [s' h' f']; intuition.
apply multi_step_bisim_backwards in H2; auto.
apply H0 in H2; auto.
apply wd_multi_step in H2; auto.

destruct H; split.
unfold safe_triple_sl; unfold sat_sl; unfold safe_sl; intros.
destruct st as [s h f].
destruct cf' as [ [s' h'] C']; apply multi_step_bisim_forwards in H2.
apply H in H2.
destruct H2; auto.
destruct x as [ [s'' h'' f''] C''].
apply step_bisim_backwards in H2.
∃ (St_sl s'' h'', C''); auto.
apply exclude_wd.
split; auto; apply exclude_wd.
unfold correct_triple_sl; unfold sat_sl; intros.
destruct st as [s h]; destruct st' as [s' h'].
apply multi_step_bisim_forwards in H2.
apply H0 in H2.
unfold sat in H2; intuition.
split; auto; apply exclude_wd.
Qed.
```