0.1 Random useful facts

Lemma *double_neg* : $\forall$ P : Prop, $\{P\} + \{\neg\ P\} \rightarrow \neg\ \neg\ P \rightarrow P$.

Lemma *leq_dec* : $\forall$ n m, $\{n \leq m\} + \{n > m\}$.

Lemma *lt_dec* : $\forall$ n m, $\{n < m\} + \{n \geq m\}$.

0.2 Language Definition

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

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

```
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.
```

```
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 :: ..
```

0.3 Definition and lemmas for natmap

```
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.

`Lemma` *maxkey_help_monotonic* {A} : ∀ (m : *natmap* A) a b, $a \le b$ → *maxkey_help m a* ≤ *maxkey_help m b*.

`Lemma` *maxkey_max* {A} : ∀ (m : *natmap* A) n, *haskey m n* → n < S (*maxkey m*).

`Lemma` *natmap_finite* {A} : ∀ (m : *natmap* A), ¬ *haskey m* (S (*maxkey m*)).

`Lemma` *mapsto_eq* {A} : ∀ (m : *natmap* A) n *v1 v2*, *mapsto m n v1* → *mapsto m n v2* → *v1* = *v2*.

`Lemma` *mapsto_in* {A} : ∀ (m : *natmap* A) n v, *mapsto m n v* → *In* (n,v) m.

`Lemma` *mapsto_haskey* {A} : ∀ (m : *natmap* A) n v, *mapsto m n v* → *haskey m n*.

`Lemma` *haskey_mapsto* {A} : ∀ (m : *natmap* A) n, *haskey m n* → ∃ v, *mapsto m n v*.

`Lemma` *mapsto_union* {A} : ∀ (*m1 m2* : *natmap* A) n v, *mapsto m1 n v* → *mapsto* (*union m1 m2*) n v.

`Lemma` *haskey_union* {A} : ∀ (*m1 m2* : *natmap* A) n, *haskey m1 n* → *haskey* (*union m1 m2*) n.

`Lemma` *mapsto_union_frame* {A} : ∀ (*m1 m2* : *natmap* A) n v, *mapsto m2 n v* → ¬ *haskey m1 n* → *mapsto* (*union m1 m2*) n v.

`Lemma` *mapsto_union_inversion* {A} : ∀ (*m1 m2* : *natmap* A) n v, *mapsto* (*union m1 m2*) n v → (*mapsto m1 n v* ∨ (¬ *haskey m1 n* ∧ *mapsto m2 n v*)).

`Lemma` *union_mapsto* {A} : ∀ (*m1 m2* : *natmap* A) n v, *mapsto* (*union m1 m2*) n v ↔ *mapsto m1 n v* ∨ (¬ *haskey m1 n* ∧ *mapsto m2 n v*).

`Lemma` *del_mapsto* {A} : ∀ (m : *natmap* A) n n' v, *mapsto* (*del m n*) n' v ↔ *mapsto m n' v* ∧ n ≠ n'.

`Lemma` *del_haskey* {A} : ∀ (m : *natmap* A) n n', *haskey* (*del m n*) n' ↔ *haskey m n'* ∧ n ≠ n'.

`Lemma` *del_not_haskey* {A} : ∀ (m : *natmap* A) n, ¬ *haskey m n* → *del m n* = m.

Lemma *upd_mapsto* $\{A\}$: ∀ (m : *natmap A*) n v n' v', *mapsto* (*upd m n v*) n' v' ↔ ($n = n'$ ∧ $v = v'$) ∨ ($n \neq n'$ ∧ *mapsto m n' v'*).

Lemma *upd_haskey* $\{A\}$: ∀ (m : *natmap A*) n v n', *haskey* (*upd m n v*) n' ↔ $n = n'$ ∨ ($n \neq n'$ ∧ *haskey m n'*).

Lemma *in_remove* : ∀ (l : *list nat*) n k, *In k* (*remove eq_nat_dec n l*) ↔ $k \neq n$ ∧ *In k l*.

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

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

0.4 Operational semantics and locality properties

0.4.1 Definition of state

Definition *store* := *var* → *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*}.

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, $a \geq b$ → *maxaddr_help f a* ≥ *maxaddr_help f b*.

Lemma *maxaddr_max_help1* : ∀ f n k, $n >$ *maxaddr* (k::f) → $n > k$.

Lemma *maxaddr_max_help2* : ∀ f n k, $n >$ *maxaddr* (k::f) → $n >$ *maxaddr f*.

Lemma *maxaddr_max* : ∀ f n, $n >$ *maxaddr f* → *in_fl f n*.

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.

Lemma upd_s_simpl_neq : ∀ s x y v, y ≠ x → s[x↦v] y = s y.

Lemma disj_upd : ∀ (h1 h2 : heap) n v, h1 # h2 → haskey h1 n → h1[n→v] # h2.

Lemma disjhf_upd : ∀ h f n v, disjhf h f → haskey h n → disjhf h[n→v] f.

Lemma disjhf_dot : ∀ h1 h2 f, disjhf (h1@h2) f ↔ disjhf h1 f ∧ disjhf h2 f.

Lemma dot_upd_comm : ∀ (h1 h2 : heap) n v, h1[n→v] @ h2 = (h1@h2)[n→v].

Lemma dot_upd_block_comm : ∀ (h1 h2 : heap) vs n, h1[n⇒vs] @ h2 = (h1@h2)[n⇒vs].

Lemma upd_haskey_block : ∀ (h : heap) n vs k, haskey h[n⇒vs] k ↔ (k ≥ n ∧ k < n +
length vs) ∨ haskey h k.

Lemma dot_del_comm : ∀ (h1 h2 : heap) n, ¬ haskey h2 n → (del h1 n)@h2 = del (h1@h2)
n.
```

0.4.4 Expression Evaluation

```
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.
```

0.4.5 Operational Semantics

```
Inductive step : config → config → Prop :=
| Step_skip :
    ∀ st C,
      step (st, Skip ;; C) (st, C)
| Step_assgn :
    ∀ s h f x e,
      step (St s h f, x ::= e) (St s[x ↦ exp_val s e] h f, Skip)
| Step_read :
    ∀ s h f x e n v,
      exp_val s e = Z_of_nat n → mapsto h n v →
      step (St s h f, x ::= [[e]]) (St s[x ↦ v] h f, Skip)
| Step_write :
    ∀ s h f e e' n,
      exp_val s e = Z_of_nat n → haskey h n →
      step (St s h f, [[e]] ::= e') (St s h[n → exp_val s e'] f, Skip)
| Step_cons :
    ∀ s h f x es n,
      (∀ i : nat, i < length es → in_fl f (n+i)) →
      step (St s h f, Cons x es) (St s[x ↦ Z_of_nat n] h[n ⇒ map (exp_val s) es] (del_fl
f n (length es)), Skip)
| Step_free :
    ∀ s h f e n,
      exp_val s e = Z_of_nat n → haskey h n →
      step (St s h f, Free e) (St s (del h n) (add_fl f n 1), Skip)
| Step_seq :
    ∀ st st' C C' C",
      step (st, C) (st', C') → step (st, C;;C") (st', C';;C")
| Step_if_true :
    ∀ st b C1 C2,
```

bexp_val (*Store st*) *b* = *true* → *step* (*st*, *if_ b* then *C1* else *C2*) (*st*, *C1*)
| *Step_if_false* :
∀ *st b C1 C2*,
bexp_val (*Store st*) *b* = *false* → *step* (*st*, *if_ b* then *C1* else *C2*) (*st*, *C2*)
| *Step_while_true* :
∀ *st b C'*,
bexp_val (*Store st*) *b* = *true* → *step* (*st*, *while b* do *C'*) (*st*, *C'* ;; *while b* do *C'*)
| *Step_while_false* :
∀ *st b C'*,
bexp_val (*Store st*) *b* = *false* → *step* (*st*, *while b* do *C'*) (*st*, *Skip*).

Inductive *stepn* : *nat* → *config* → *config* → Prop :=
| *Stepn_zero* : ∀ *cf*, *stepn O cf cf*
| *Stepn_succ* : ∀ *n cf cf' cf"*, *step cf cf'* → *stepn n cf' cf"* → *stepn* (*S n*) *cf cf"*.

Definition *multi_step cf cf'* := ∃ *n*, *stepn n cf cf'*.
Definition *halt_config* (*cf* : *config*) := *snd cf* = *Skip*.
Definition *safe* (*cf* : *config*) := ∀ *cf'*, *multi_step cf cf'* → ¬ *halt_config cf'* → ∃ *cf"*, *step cf' cf"*.
Definition *diverges* (*cf* : *config*) := ∀ *n*, ∃ *cf'*, *stepn n cf cf'*.

0.4.6 Facts about stepping

Lemma *safe_step* : ∀ *cf*, *safe cf* → ¬ *halt_config cf* → ∃ *cf'*, *step cf cf'*.

Lemma *safe_step_safe* : ∀ *cf cf'*, *safe cf* → *step cf cf'* → *safe cf'*.

Lemma *safe_stepn_safe* : ∀ *n cf cf'*, *safe cf* → *stepn n cf cf'* → *safe cf'*.

Lemma *safe_multi_step_safe* : ∀ *cf cf'*, *safe cf* → *multi_step cf cf'* → *safe cf'*.

Lemma *stepn_seq* : ∀ *n st st' C C' C"*, *stepn n* (*st*,C) (*st'*,C') → *stepn n* (*st*, *C*;;*C"*) (*st'*, *C'*;;*C"*).

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

Lemma *safe_seq* : ∀ *st C C'*, *safe* (*st*, *C* ;; *C'*) → *safe* (*st*, *C*).

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'*.

Lemma *wd_stepn* : ∀ *n C C' st st'*, *stepn n* (*st*,C) (*st'*,C') → *wd st* → *wd st'*.

Lemma *wd_multi_step* : ∀ *C C' st st'*, *multi_step* (*st*,C) (*st'*,C') → *wd st* → *wd st'*.

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'*).

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'*).

`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'*).

`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'*).

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'*).

`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'*).

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*).

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)).
```

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 :
    ∀ s h e e' n,
      exp_val s e = Z_of_nat n → haskey h n →
      step_sl (St_sl s h, [[e]] ::= e') (St_sl s h[n → exp_val s e'], Skip)
| Step_sl_cons :
    ∀ s h x es n,
      (∀ i : nat, i < length es → ¬ haskey h (n+i)) →
      step_sl (St_sl s h, Cons x es) (St_sl s[x ↦ Z_of_nat n] h[n ⇒ map (exp_val s) es],
Skip)
| Step_sl_free :
    ∀ s h e n,
      exp_val s e = Z_of_nat n → haskey h n →
      step_sl (St_sl s h, Free e) (St_sl s (del h n), Skip)
| Step_sl_seq :
    ∀ st st' C C' C",
      step_sl (st, C) (st', C') → step_sl (st, C;;C") (st', C';;C")
| Step_sl_if_true :
    ∀ st b C1 C2,
      bexp_val (Store_sl st) b = true → step_sl (st, if_ b then C1 else C2) (st, C1)
```

| *Step_sl_if_false* :
  ∀ *st b C1 C2*,
    bexp_val (*Store_sl st*) *b* = *false* → *step_sl* (*st*, *if_ b* `then` *C1* `else` *C2*) (*st*, *C2*)
| *Step_sl_while_true* :
  ∀ *st b C'*,
    bexp_val (*Store_sl st*) *b* = *true* → *step_sl* (*st*, *while b* `do` *C'*) (*st*, *C'* ;; *while b* `do` *C'*)
| *Step_sl_while_false* :
  ∀ *st b C'*,
    bexp_val (*Store_sl st*) *b* = *false* → *step_sl* (*st*, *while b* `do` *C'*) (*st*, *Skip*).

`Inductive` *stepn_sl* : *nat* → *config_sl* → *config_sl* → `Prop` :=
| *Stepn_sl_zero* : ∀ *cf*, *stepn_sl O cf cf*
| *Stepn_sl_succ* : ∀ *n cf cf' cf"*, *step_sl cf cf'* → *stepn_sl n cf' cf"* → *stepn_sl* (*S n*) *cf cf"*.

`Definition` *multi_step_sl cf cf'* := ∃ *n*, *stepn_sl n cf cf'*.
`Definition` *halt_config_sl* (*cf* : *config_sl*) := *snd cf* = *Skip*.
`Definition` *safe_sl* (*cf* : *config_sl*) := ∀ *cf'*, *multi_step_sl cf cf'* → ¬ *halt_config_sl cf'* → ∃ *cf"*, *step_sl cf' cf"*.
`Definition` *diverges_sl* (*cf* : *config_sl*) := ∀ *n*, ∃ *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*}.

`Lemma` *remove_dup_in* : ∀ *f n*, *In n* (*remove_dup f*) ↔ *In n f*.

`Lemma` *haskey_in_fst* : ∀ (*h* : *heap*) *n*, *In n* (*map* (*fst* (*B*:=val)) *h*) ↔ *haskey h n*.

`Lemma` *haskey_exclude* : ∀ *h n*, ¬ *in_fl* (*exclude h*) *n* ↔ *haskey h n*.

`Lemma` *exclude_write* : ∀ *h n v*, *haskey h n* → *exclude h*[n→v] = *exclude h*.

`Lemma` *exclude_cons_help* : ∀ *h n v*, ¬ *haskey h n* → *exclude h*[n→v] = *n* :: *exclude h*.

`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*).

`Lemma` *exclude_free* : ∀ *h n*, *exclude* (*del h n*) = *add_fl* (*exclude h*) *n* 1.

Lemma *exclude_wd* : ∀ *s h*, *wd* (*St s h* (*exclude h*)).

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'*).

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'*).

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'*).

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'*).

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'*).

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'*).

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'*).

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'*).

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.