```
Require Import FunctionalExtensionality.
Require Import ClassicalFacts.

Axiom prop_ext : prop_extensionality.

Ltac inv H := inversion H; subst; clear H.
Ltac dup H := generalize H; intro.
```

0.1 Definition of a Separation Algebra

```
Module Type SepAlgebra.

Parameter A : Set.
Parameter u : A.

Parameter dot : option A → option A → option A.
Notation "a @ b" := (dot a b) (at level 2).

Definition disj (st0 st1 : A) : Prop := (Some st0) @ (Some st1) ≠ None.
Notation "st0 # st1" := (disj st0 st1) (at level 2).

Definition substate (st0 st : A) : Prop := ∃ st1, (Some st0) @ (Some st1) = Some st.
Notation "st0 «= st" := (substate st0 st) (at level 2).

Axiom dot_none : ∀ a, None@a = None.
Axiom dot_unit : ∀ a, (Some u)@a = a.
Axiom dot_comm : ∀ a b, a@b = b@a.
Axiom dot_assoc : ∀ a b c, (a@b)@c = a@(b@c).
Axiom dot_cancel : ∀ st a b, (Some st)@a = (Some st)@b → a = b.

End SepAlgebra.
```

0.2 Definition/facts for local actions

```
Declare Module S : SepAlgebra.
Notation "st0 @ st1" := (S.dot (Some st0) (Some st1)) (at level 2).

Inductive state :=
| St : S.A → state
| Bad : state
| Div : state.

Definition action := state → state → Prop.

Definition local (f : action) : Prop :=
  (∀ st, f Bad st ↔ st = Bad) ∧ (∀ st, f Div st ↔ st = Div) ∧ (∀ st, ∃ st', f st st') ∧
  (∀ st0 st1 st, ¬ f (St st0) Bad → st0 @ st1 = Some st →
    (¬ f (St st) Bad ∧ (f (St st0) Div ↔ f (St st) Div)) ∧
```

(∀ *st0'*, *f* (*St st0*) (*St st0'*) → ∃ *st'*, *st0'* @ *st1* = *Some st'* ∧ *f* (*St st*) (*St st'*)) ∧
(∀ *st'*, *f* (*St st*) (*St st'*) → ∃ *st0'*, *st0'* @ *st1* = *Some st'* ∧ *f* (*St st0*) (*St st0'*))).

```
Definition id_act : action := fun st st' ⇒ st = st'.
Definition compose (f1 f2 : action) : action := fun st st' ⇒ ∃ st'', f1 st st'' ∧ f2 st'' st'.
Definition union {A} (_ : inhabited A) (fs : A → action) : action := fun st st' ⇒ ∃ a :
A, fs a st st'.
```

```
Lemma compose_assoc : ∀ f1 f2 f3, compose (compose f1 f2) f3 = compose f1 (compose f2
f3).
Proof.
intros; extensionality st; extensionality st'; apply prop_ext; split; intros.
destruct H as [st1 [H] ].
destruct H as [st0 [H] ].
∃ st0; intuition.
∃ st1; intuition.
destruct H as [st0 [H] ].
destruct H0 as [st1 [H0] ].
∃ st1; intuition.
∃ st0; intuition.
Qed.
```

Lemma 5 from paper

```
Lemma compose_local : ∀ f1 f2, local f1 → local f2 → local (compose f1 f2).
Proof.
unfold local; unfold compose; intuition; subst.

destruct H5 as [st' [H5] ].
apply H1 in H5; subst.
apply H in H8; auto.

∃ Bad; split.
apply (H1 Bad); auto.
apply (H Bad); auto.

destruct H5 as [st' [H5] ].
apply H0 in H5; subst.
apply H2 in H8; auto.

∃ Div; split.
apply (H0 Div); auto.
apply (H2 Div); auto.

destruct (H3 st) as [st'].
destruct (H4 st') as [st''].
∃ st''; ∃ st'; split; auto.

destruct H9 as [ [st' | | ] [H9] ]; apply H5.
apply H6 in H8; intuition.
```

```
apply H14 in H9; destruct H9 as [st0' [H9] ].
∃ (St st0'); intuition.
apply H7 in H9; intuition.
apply H5; ∃ (St st0'); intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); auto.
∃ Bad; intuition.
apply H6 in H8; intuition.
apply H5; ∃ Bad; intuition.
apply H2 in H10; inv H10.
destruct H9 as [ [st0' | | ] [H9] ].
apply H6 in H8; intuition.
dup H9; apply H11 in H9; destruct H9 as [st' [H9] ].
∃ (St st'); intuition.
apply H7 in H9; intuition.
apply H5; ∃ (St st0'); intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); auto.
apply H in H10; inv H10.
∃ Div; intuition.
apply H6 in H8; intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); intuition.
destruct H9 as [ [st' | | ] [H9] ].
apply H6 in H8; intuition.
apply H14 in H9; destruct H9 as [st0' [H9] ].
∃ (St st0'); intuition.
apply H7 in H9; intuition.
apply H5; ∃ (St st0'); intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); auto.
apply H in H10; inv H10.
∃ Div; intuition.
apply H6 in H8; intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); intuition.
destruct H9 as [ [st0" | | ] [H9] ].
apply H6 in H8; intuition.
dup H9; apply H11 in H9; destruct H9 as [st" [H9] ].
apply H7 in H9; intuition.
apply H17 in H10; destruct H10 as [st' [H10] ].
∃ st'; intuition.
```

```
∃ (St st”); intuition.
apply H5; ∃ (St st0”); intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); intuition.
apply H in H10; inv H10.
apply H2 in H10; inv H10.

destruct H9 as [ [st” | | ] [H9] ].
apply H6 in H8; intuition.
apply H14 in H9; destruct H9 as [st0” [H9] ].
apply H7 in H9; intuition.
apply H19 in H10; destruct H10 as [st0’ [H10] ].
∃ st0’; intuition.
∃ (St st0”); intuition.
apply H5; ∃ (St st0”); intuition.
apply H5; ∃ Bad; intuition.
apply (H Bad); intuition.
apply H in H10; inv H10.
apply H2 in H10; inv H10.
Qed.
```

Lemma 6 from paper

```
Lemma union_local {A} (p : inhabited A) : ∀ fs, (∀ a : A, local (fs a)) → local (union p fs).
Proof.
unfold local; unfold union; intuition; subst.

destruct H0 as [a].
apply H in H0; auto.

destruct p as [a]; ∃ a.
specialize (H a); intuition.
apply (H0 Bad); auto.

destruct H0 as [a].
apply H in H0; auto.

destruct p as [a]; ∃ a.
specialize (H a); intuition.
apply (H Div); auto.

destruct p as [a].
specialize (H a); intuition.
destruct (H1 st) as [st’]; ∃ st’; ∃ a; auto.

destruct H2 as [a].
apply (H a) in H1; intuition.
apply H0; ∃ a; auto.

destruct H2 as [a]; ∃ a.
```

```
apply (H a) in H1; intuition.
apply H0; ∃ a; auto.
destruct H2 as [a]; ∃ a.
apply (H a) in H1; intuition.
apply H0; ∃ a; auto.
destruct H2 as [a].
apply (H a) in H1; intuition.
apply H3 in H2; destruct H2 as [st']; ∃ st'; intuition.
∃ a; auto.
apply H0; ∃ a; auto.
destruct H2 as [a].
apply (H a) in H1; intuition.
apply H6 in H2; destruct H2 as [st0']; ∃ st0'; intuition.
∃ a; auto.
apply H0; ∃ a; auto.
Qed.
Lemma id_local : local id_act.
Proof.
unfold local; unfold id_act; intuition.
∃ st; auto.
inv H1.
inv H1.
inv H1.
inv H1.
∃ st; auto.
inv H1.
∃ st0; auto.
Qed.
```

0.3 Definition and semantics of the program language

```
Module Type Language.
Parameter prim : Set.
Parameter prim_sem : prim → {f : action | local f}.
Inductive cmd :=
| Prim : prim → cmd
| Seq : cmd → cmd → cmd
| Choice : cmd → cmd → cmd
| Iter : cmd → cmd.
Fixpoint cmd_sem (C : cmd) : action :=
```

```
  match C with
  | Prim c ⇒ let (f,_) := prim_sem c in f
  | Seq C1 C2 ⇒ compose (cmd_sem C1) (cmd_sem C2)
  | Choice C1 C2 ⇒ union (inhabits true) (fun b : bool ⇒ if b then cmd_sem C1 else
cmd_sem C2)
  | Iter C ⇒ union (inhabits 0) (fix rec (n : nat) := match n with
                                                        | 0 ⇒ id_act
                                                        | S n ⇒ compose (cmd_sem
C) (rec n)

                                                        end)
  end.

End Language.

Declare Module L : Language.

Lemma sem_local : ∀ C : L.cmd, local (L.cmd_sem C).
Proof.
induction C; simpl.

destruct (L.prim_sem p); auto.

apply compose_local; auto.

apply union_local.
destruct a; auto.

apply union_local.
induction a.
apply id_local.
apply compose_local; auto.
Qed.

Definition iter_n C n := (fix rec n' := match n' with
                                          | 0 ⇒ id_act
                                          | S n' ⇒ compose (L.cmd_sem C) (rec n')
                                          end) n.

Lemma iter_n_local : ∀ C n, local (iter_n C n).
Proof.
induction n; intros; simpl.
apply id_local.
apply compose_local; auto.
apply sem_local.
Qed.

Lemma compose_iter : ∀ C n, compose (L.cmd_sem C) (iter_n C n) = compose (iter_n C
n) (L.cmd_sem C).
Proof.
```

```
induction n; simpl; extensionality st; extensionality st'; apply prop_ext; split;
intros.
destruct H as [st" [H] ].
inv H0; ∃ st; intuition.
unfold id_act; auto.
destruct H as [st" [H] ].
inv H; ∃ st'; intuition.
unfold id_act; auto.
rewrite compose_assoc; rewrite ← IHn; auto.
rewrite compose_assoc in H; rewrite ← IHn in H; auto.
Qed.
```

0.4 Assertions, triples, and inference rules

```
Definition assert := S.A → Prop.
Inductive triple := Trip : assert → L.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.
```

```
Definition emp : assert := fun _ ⇒ False.
Definition star (p q : assert) : assert := fun st ⇒ ∃ st0, ∃ st1, p st0 ∧ q st1 ∧ st0 @
st1 = Some st.
Notation "p ** q" := (star p q) (at level 2).
Definition implies (p q : assert) : Prop := ∀ st, p st → q st.
Definition disj (I : Set) (ps : I → assert) : assert := fun st ⇒ ∃ i : I, ps i st.
Definition conj (I : Set) (ps : I → assert) : assert := fun st ⇒ ∀ i : I, ps i st.
```

```
Axiom emp_dec : ∀ p, {p = emp} + {∃ st, p st}.
```

```
Lemma disj_canonical : ∀ p : assert, p = disj {st : S.A | p st} (fun stp ⇒ let (st,_) := stp
in eq st).
Proof.
intros; unfold disj.
extensionality st.
apply prop_ext; split; intros.
∃ (exist (fun st ⇒ p st) st H); auto.
destruct H as [ [st' H0] ]; subst; auto.
Qed.
```

```
Lemma disj_eq : ∀ (p : assert) (I : Set), inhabited I → p = disj I (fun _ ⇒ p).
Proof.
unfold disj; intros.
extensionality st.
apply prop_ext; split; intros.
```

```
destruct H as [i]; ∃ i; auto.
destruct H0; auto.
Qed.

Lemma disj_emp : ∀ (ps : False → assert), emp = disj False ps.
Proof.
unfold disj; unfold emp; intros.
extensionality st.
apply prop_ext; split; intros.
inv H.
destruct H; auto.
Qed.

Definition valid (t : triple) : Prop := let (p,C,q) := t in
  ∀ st, p st → ¬ L.cmd_sem C (St st) Bad ∧ ∀ st', L.cmd_sem C (St st) (St st') → q st'.

Definition prim_act c := let (f,_) := L.prim_sem c in f.

Inductive derivable : triple → Prop :=
| Derive_prim : ∀ st c,
    ¬ prim_act c (St st) Bad → derivable (Trip (eq st) (L.Prim c) (fun st' ⇒ prim_act c
(St st) (St st')))
| Derive_seq : ∀ p q r C1 C2,
    derivable (Trip p C1 q) → derivable (Trip q C2 r) → derivable (Trip p (L.Seq C1 C2)
r)
| Derive_choice : ∀ p q C1 C2,
    derivable (Trip p C1 q) → derivable (Trip p C2 q) → derivable (Trip p (L.Choice C1
C2) q)
| Derive_iter : ∀ p C,
    derivable (Trip p C p) → derivable (Trip p (L.Iter C) p)
| Derive_frame : ∀ p q r C,
    derivable (Trip p C q) → derivable (Trip p**r C q**r)
| Derive_conseq : ∀ p p' q q' C,
    derivable (Trip p C q) → implies p' p → implies q q' → derivable (Trip p' C q')
| Derive_disj : ∀ (I : Set) ps qs C,
    (∀ i : I, derivable (Trip (ps i) C (qs i))) → derivable (Trip (disj I ps) C (disj I qs))
| Derive_conj : ∀ (I : Set) ps qs C,
    inhabited I → (∀ i : I, derivable (Trip (ps i) C (qs i))) → derivable (Trip (conj I ps)
C (conj I qs)).

Lemma derive_emp : ∀ C p, derivable (Trip emp C p).
Proof.
intros.
apply Derive_conseq with (p := emp) (q := emp).
rewrite (disj_emp (fun _ ⇒ emp)).
apply Derive_disj; intuition.
```

unfold *implies*; intuition.
unfold *implies*; intuition.
inv H.
Qed.

0.5 Soundness and completeness

Lemma *soundness* : ∀ *t*, *derivable t* → *valid t*.
Proof.
intros; induction *H*; unfold *valid*; intuition; subst.

Prim

auto.
auto.

Seq

simpl in *H2*; destruct *H2* as [*st'* [*H2*]].
apply *IHderivable1* in *H1*; intuition.
apply *H4*; destruct *st'* as [*st'* | |]; auto.
apply *H5* in *H2*.
apply *IHderivable2* in *H2*; intuition.
apply *sem_local* in *H3*; *inv H3*.

simpl in *H2*; destruct *H2* as [[*st"* | |] [*H2*]].
apply *IHderivable1* in *H2*; auto.
apply *IHderivable2* in *H3*; auto.
apply *sem_local* in *H3*; *inv H3*.
apply *sem_local* in *H3*; *inv H3*.

Choice

simpl in *H2*; destruct *H2* as [*b*]; destruct *b*.
apply *IHderivable1* in *H2*; auto.
apply *IHderivable2* in *H2*; auto.

simpl in *H2*; destruct *H2* as [*b*]; destruct *b*.
apply *IHderivable1* in *H2*; auto.
apply *IHderivable2* in *H2*; auto.

Iter

simpl in *H1*; destruct *H1* as [*n*].
generalize *st H0 H1*; clear *st H0 H1*.
induction *n*; intros.
inv H1.
destruct *H1* as [[*st'* | |] [*H1*]].
apply (*IHn st'*); auto.
apply *IHderivable* in *H1*; auto.

```
apply IHderivable in H1; auto.
change (iter_n C n Div Bad) in H2; apply iter_n_local in H2; inv H2.

simpl in H1; destruct H1 as [n].
generalize st H0 H1; clear st H0 H1.
induction n; intros.
inv H1; auto.
destruct H1 as [ [st" | | ] [H1] ].
apply (IHn st"); auto.
apply IHderivable in H1; auto.
apply IHderivable in H1; auto; inv H1.
change (iter_n C n Div (St st')) in H2; apply iter_n_local in H2; inv H2.
```

Frame

```
destruct H0 as [st0 [st1] ]; intuition.
apply (sem_local C) in H4; intuition.
apply IHderivable in H2; intuition.

destruct H0 as [st0 [st1] ]; intuition.
apply (sem_local C) in H4; intuition.
apply H7 in H1; destruct H1 as [st0' [H1] ].
∃ st0'; ∃ st1; intuition.
apply IHderivable in H6; intuition.
apply IHderivable in H2; intuition.
```

Conseq

```
apply H0 in H2; apply IHderivable in H2; intuition.
apply H0 in H2; apply IHderivable in H2; intuition.
```

Disj

```
destruct H1 as [i].
apply H0 in H1; intuition.

destruct H1 as [i]; ∃ i.
apply H0 in H1; intuition.
```

Conj

```
destruct H as [i].
apply (H1 i) in H2; intuition.

intro i.
apply (H1 i) in H2; intuition.
Qed.

Lemma completeness : ∀ t, valid t → derivable t.
Proof.
destruct t as [p C q].
generalize p q; clear p q.
induction C; simpl; intros.
```

Prim

```
rename p into c; rename p0 into p.
destruct (emp_dec p); subst; try solve [apply derive_emp].
rewrite (disj_canonical p).
rewrite (disj_eq q {st : S.A | p st}).
apply Derive_disj.
intro i; destruct i as [st H0].
apply Derive_conseq with (p := eq st) (q := fun st' ⇒ prim_act c (St st) (St st')).
apply Derive_prim.
apply (H st); auto.
unfold implies; intuition.
unfold implies; intros.
apply H in H1; auto.
destruct e as [st]; apply (inhabits (exist (fun st ⇒ p st) st H0)).
```

Seq

```
destruct (emp_dec p); subst; try solve [apply derive_emp].
rewrite (disj_canonical p).
rewrite (disj_eq q {st : S.A | p st}).
apply Derive_disj.
intro i; destruct i as [st H0].
apply Derive_seq with (q := fun st' ⇒ L.cmd_sem C1 (St st) (St st')).
apply IHC1.
simpl; intuition; subst; auto.
apply (H st0); auto.
∃ Bad; intuition.
assert (local (L.cmd_sem C2)).
apply sem_local.
unfold local in H1; intuition.
apply (H3 Bad); auto.
apply IHC2.
simpl; intuition.
apply (H st); auto.
∃ (St st0); auto.
apply (H st); auto.
∃ (St st0); auto.
destruct e as [st]; apply (inhabits (exist (fun st ⇒ p st) st H0)).
```

Choice

```
destruct (emp_dec p); subst; try solve [apply derive_emp].
rewrite (disj_canonical p).
rewrite (disj_eq q {st : S.A | p st}).
apply Derive_disj.
intro i; destruct i as [st H0].
```

```
apply Derive_choice.
apply IHC1.
simpl; intuition; subst.
apply (H st0); auto.
∃ true; auto.
apply (H st0); auto.
∃ true; auto.
apply IHC2.
simpl; intuition; subst.
apply (H st0); auto.
∃ false; auto.
apply (H st0); auto.
∃ false; auto.
destruct e as [st]; apply (inhabits (exist (fun st ⇒ p st) st H0)).
```

Iter

```
destruct (emp_dec p); subst; try solve [apply derive_emp].
rewrite (disj_canonical p).
rewrite (disj_eq q {st : S.A | p st}).
apply Derive_disj.
intro i; destruct i as [st H0].
apply Derive_conseq with (p := disj nat (fun n st' ⇒ iter_n C n (St st) (St st')))
                         (q := disj nat (fun n st' ⇒ iter_n C n (St st) (St st'))).
apply Derive_iter.
apply Derive_conseq with (p := disj nat (fun n st' ⇒ iter_n C n (St st) (St st')))
                         (q := disj nat (fun n st' ⇒ iter_n C (S n) (St st) (St st'))).
apply Derive_disj; intro n.
apply IHC.
simpl; intuition.
apply (H st); auto.
∃ (S n).
change (compose (L.cmd_sem C) (iter_n C n) (St st) Bad); rewrite compose_iter.
∃ (St st0); intuition.
rewrite compose_iter; ∃ (St st0); intuition.
unfold implies; auto.
unfold implies; intros.
destruct H1 as [n [st' [H1] ] ].
∃ (S n); unfold iter_n.
∃ st'; intuition.
unfold implies; intros; subst.
∃ 0; simpl.
unfold id_act; auto.
unfold implies; intros.
```

```
apply (H st); auto.
destruct e as [st]; apply (inhabits (exist (fun st ⇒ p st) st H0)).
Qed.
```

Theorem 3 from paper

```
Theorem soundness_and_completeness : ∀ t, derivable t ↔ valid t.
Proof.
split; [apply soundness | apply completeness].
Qed.
```