Axiom *prop_ext* : *prop_extensionality*.

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).
```

Lemma 5 from paper

```
Lemma compose_local : ∀ f1 f2, local f1 → local f2 → local (compose f1 f2).
```

Lemma 6 from paper

```
Lemma union_local {A} (p : inhabited A) : ∀ fs, (∀ a : A, local (fs a)) → local (union p fs).

Lemma id_local : local id_act.
```

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).

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* : $\forall$ C n, *local* (*iter_n* C n).

Lemma *compose_iter* : $\forall$ C n, *compose* (*L.cmd_sem* C) (*iter_n* C n) = *compose* (*iter_n* C n) (*L.cmd_sem* C).

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*).

Lemma *disj_eq* : ∀ (*p* : assert) (*I* : Set), *inhabited I* → *p* = *disj I* (fun _ ⇒ *p*).

Lemma *disj_emp* : ∀ (*ps* : *False* → assert), *emp* = *disj False ps*.

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*).

0.5 Soundness and completeness

`Lemma` *soundness* : ∀ *t*, *derivable t* → *valid t*.

`Lemma` *completeness* : ∀ *t*, *valid t* → *derivable t*.

Theorem 3 from paper
`Theorem` *soundness_and_completeness* : ∀ *t*, *derivable t* ↔ *valid t*.