Chapter 1

Closure_calculus

```
Require Import List Omega.
```

General tactics

```
Ltac eapply2 H := eapply H; eauto.

Ltac split_all := simpl; intros;
match goal with
| H : _ ∧ _ ⊢ _ ⇒ inversion_clear H; split_all
| H : ∃ _, _ ⊢ _ ⇒ inversion H; clear H; split_all
| _ ⇒ try (split; split_all); try contradiction
end; try congruence; auto.

Ltac exist x := ∃ x; split_all.
Ltac invsub :=
match goal with
| H : _ = _ ⊢ _ ⇒ inversion H; subst; clear H; invsub
| _ ⇒ split_all
end.
```

The terms of closure calculus are:

- variables (given by natural numbers)
- tagged applications
- applications
- the identity operator
- extensions (constructor Add)
- closures (constructor Abs)

The variable names will become de Bruijn indices when doing meta-theory, but this will come after the reduction rules have been defined.

```
Inductive lambda: Set :=
| Ref : nat → lambda
| Tag : lambda → lambda → lambda
| App : lambda → lambda → lambda
| Iop : lambda
| Add : nat → lambda → lambda → lambda
| Abs : nat → list nat → lambda → lambda → lambda
            .

Hint Constructors lambda.

Definition termred := lambda → lambda → Prop.

Inductive multi_step : termred → termred :=
  | zero_red : ∀ red M, multi_step red M M
  | succ_red : ∀ (red: lambda→ lambda → Prop) M N P,
                         red M N → multi_step red N P → multi_step red M P
.

Hint Constructors multi_step.

Definition reflective red := ∀ (M: lambda), red M M.

Lemma refl_multi_step : ∀ (red: termred), reflective (multi_step red).
Proof. red; split_all. Qed.

Ltac reflect := match goal with
| ⊢ reflective (multi_step _) ⇒ eapply2 refl_multi_step
| ⊢ multi_step _ _ _ ⇒ try (eapply2 refl_multi_step)
| _ ⇒ split_all
end.

Definition transitive red := ∀ (M N P: lambda), red M N → red N P → red M P.

Lemma transitive_red : ∀ red, transitive (multi_step red).
Proof. red; induction 1; split_all.
apply succ_red with N; auto.
Qed.

Ltac one_step :=
  try red;
match goal with
| ⊢ multi_step _ _ ?N ⇒ apply succ_red with N; auto; try reflect
end.

Definition confluence (A : Set) (R : A → A → Prop) :=
  ∀ x y : A,
```

$R\ x\ y \to \forall\ z : A,\ R\ x\ z \to \exists\ u : A,\ R\ y\ u \land R\ z\ u.$

Definition *diamond0* (*red1 red2* : *termred*) :=
$\forall$ *M N*, *red1 M N* $\to \forall$ *P*, *red2 M P* $\to \exists$ *Q*, *red2 N Q* $\land$ *red1 P Q*.

Lemma *diamond0_flip*: $\forall$ *red1 red2*, *diamond0 red1 red2* $\to$ *diamond0 red2 red1*.
Proof.
unfold *diamond0*. intros *red1 red2 d M N r2 P r1*. elim (*d M P r1 N r2*); *split_all*.
exist x. Qed.

Lemma *diamond0_strip* :
$\forall$ *red1 red2*, *diamond0 red1 red2* $\to$ *diamond0 red1* (*multi_step red2*).
Proof.
intros *red1 red2 d*. *eapply2 diamond0_flip*. red; induction 1; intros *Q r*.
exist Q.
elim (*d M Q r N*); *split_all*.
elim(*IHmulti_step d x*); *split_all*.
exist x0.
apply *succ_red* with *x*; auto.
Qed.

Definition *diamond0_star* (*red1 red2*: *termred*) := $\forall$ *M N*, *red1 M N* $\to \forall$ *P*, *red2 M P*
$\to$
$\exists$ *Q*, *red1 P Q* $\land$ *multi_step red2 N Q*.

Lemma *diamond0_star_strip*:
$\forall$ *red1 red2*, *diamond0_star red1 red2* $\to$ *diamond0* (*multi_step red2*) *red1* .
Proof.
red. intros *red1 red2 d*. intros *M N r*; induction *r*; intros *Q r1*.
exist Q.
elim(*d M Q r1 N H*); *split_all*.
elim(*IHr d x*); *split_all*.
exist x0.
apply *transitive_red* with *x*; auto.
Qed.

Lemma *diamond0_tiling* :
$\forall$ *red1 red2*, *diamond0 red1 red2* $\to$ *diamond0* (*multi_step red1*) (*multi_step red2*).
Proof.
red. intros *red1 red2 d M N r*; induction *r*; intros *Q r2*.
exist Q.
elim(*diamond0_strip* red *red2 d M N H Q*); *split_all*.
elim(*IHr d x H1*); *split_all*.
exist x0.
apply *succ_red* with *x*; auto.
Qed.

Hint Resolve *diamond0_tiling*.

Definition *diamond* (*red1 red2* : *termred*) :=
∀ *M N P*, *red1 M N* → *red2 M P* → ∃ *Q*, *red2 N Q* ∧ *red1 P Q*.

Lemma *diamond_iff_diamond0* : ∀ *red1 red2*, *diamond red1 red2* ↔ *diamond0 red1 red2*.
Proof. intros; red; *split_all*; red; *split_all*; *eapply2 H*. Qed.

Lemma *diamond_tiling* : ∀ *red1 red2*, *diamond red1 red2* → *diamond* (*multi_step red1*) (*multi_step red2*).
Proof.
intros. *eapply2 diamond_iff_diamond0*. *eapply2 diamond0_tiling*. *eapply2 diamond_iff_diamond0*.
Qed.

Inductive *seq_red1* : *lambda* → *lambda* → Prop :=
| *tagl_seq_red* : ∀ *M M' N*, *seq_red1 M M'* → *seq_red1* (*Tag M N*) (*Tag M' N*)
| *tagr_seq_red* : ∀ *M N N'*, *seq_red1 N N'* → *seq_red1* (*Tag M N*) (*Tag M N'*)
| *appl_seq_red* : ∀ *M M' N*, *seq_red1 M M'* → *seq_red1* (*App M N*) (*App M' N*)
| *appr_seq_red* : ∀ *M N N'*, *seq_red1 N N'* → *seq_red1* (*App M N*) (*App M N'*)
| *addl_seq_red* : ∀ *i M M' sigma*, *seq_red1 M M'* → *seq_red1* (Add *i M sigma*) (Add *i M' sigma*)
| *addr_seq_red* : ∀ *i M sigma sigma'*, *seq_red1 sigma sigma'* →
seq_red1 (Add *i M sigma*) (Add *i M sigma'*)
| *absl_seq_red* : ∀ *sigma sigma' i is M*, *seq_red1 sigma sigma'* →
seq_red1 (*Abs i is sigma M*) (*Abs i is sigma' M*)
| *absr_seq_red* : ∀ *sigma i is M M'*, *seq_red1 M M'* →
seq_red1 (*Abs i is sigma M*) (*Abs i is sigma M'*)
| *app_ref_seq_red* : ∀ *i M*, *seq_red1* (*App* (*Ref i*) *M*) (*Tag* (*Ref i*) *M*)
| *app_tag_seq_red* : ∀ *M N P*, *seq_red1* (*App* (*Tag M N*) *P*) (*Tag* (*Tag M N*) *P*)
| *beta1_seq_red* : ∀ *sigma j M N*,
seq_red1 (*App* (*Abs j nil sigma M*) *N*)
(*App* (Add *j N sigma*) *M*)
| *beta2_seq_red* : ∀ *sigma j j2 js M N*,
seq_red1 (*App* (*Abs j* (*cons j2 js*) *sigma M*) *N*)
(*Abs j2 js* (Add *j N sigma*) *M*)
| *nil_seq_red* : ∀ *M*, *seq_red1* (*App Iop M*) *M*
| *subst_eq_seq_red* : ∀ *j sigma N*, *seq_red1* (*App* (Add *j N sigma*) (*Ref j*)) *N*
| *subst_uneq_seq_red* : ∀ *sigma i j N*, *i*≠ *j* →
seq_red1 (*App* (Add *i N sigma*) (*Ref j*)) (*App sigma* (*Ref j*))
| *subst_tag_seq_red* : ∀ *j U sigma M N*,
seq_red1 (*App* (Add *j U sigma*) (*Tag M N*))
(*App* (*App* (Add *j U sigma*) *M*) (*App* (Add *j U sigma*) *N*))

```
  | subst_nil_seq_red : ∀ j U sigma, seq_red1 (App (Add j U sigma) Iop) (App sigma Iop)
  | subst_add_seq_red : ∀ j N sigma j2 P sigma2,
                          seq_red1 (App (Add j N sigma) (Add j2 P sigma2))
                                   (Add j2 (App (Add j N sigma) P) (App (Add j N
sigma) sigma2))
  | subst_abs_seq_red : ∀ j N sigma j2 js sigma2 M,
                        seq_red1 (App (Add j N sigma) (Abs j2 js sigma2 M))
                                 (Abs j2 js (App (Add j N sigma) sigma2) M)
  .

Hint Constructors seq_red1 .
Definition seq_red := multi_step seq_red1.

Lemma reflective_seq_red: reflective seq_red.
Proof. red; red; reflect. Qed.
Hint Resolve reflective_seq_red.

Definition preserve (R : termred) (P : lambda → Prop) :=
  ∀ x : lambda, P x → ∀ y : lambda, R x y → P y.

Definition preserves_tagl (red : termred) :=
∀ M M' N, red M M' → red (Tag M N) (Tag M' N).

Definition preserves_tagr (red : termred) :=
∀ M N N', red N N' → red (Tag M N) (Tag M N').

Lemma preserves_tagl_multi_step : ∀ (red: termred), preserves_tagl red → preserves_tagl
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Tag N0 N); auto. Qed.

Lemma preserves_tagr_multi_step : ∀ (red: termred), preserves_tagr red → preserves_tagr
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Tag M N); auto. Qed.

Definition preserves_tag (red : termred) :=
∀ M M' N N', red M M' → red N N' → red (Tag M N) (Tag M' N').

Definition preserves_apl (red : termred) :=
∀ M M' N, red M M' → red (App M N) (App M' N).

Definition preserves_apr (red : termred) :=
∀ M N N', red N N' → red (App M N) (App M N').

Lemma preserves_apl_multi_step : ∀ (red: termred), preserves_apl red → preserves_apl
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (App N0 N); auto. Qed.

Lemma preserves_apr_multi_step : ∀ (red: termred), preserves_apr red → preserves_apr
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (App M N); auto. Qed.

Definition preserves_app (red : termred) :=
```

```
∀ M M' N N', red M M' → red N N' → red (App M N) (App M' N').

Definition preserves_adl (red : termred) :=
∀ i M M' N, red M M' → red (Add i M N) (Add i M' N).

Definition preserves_adr (red : termred) :=
∀ i M sigma sigma', red sigma sigma' → red (Add i M sigma) (Add i M sigma').

Lemma preserves_adl_multi_step : ∀ (red: termred), preserves_adl red → preserves_adl
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Add i N0 N); auto. Qed.

Lemma preserves_adr_multi_step : ∀ (red: termred), preserves_adr red → preserves_adr
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Add i M N); auto. Qed.

Definition preserves_add (red : termred) :=
∀ M M' N N' i, red M M' → red N N' → red (Add i M N) (Add i M' N').

Definition preserves_absl (red : termred) :=
∀ sigma sigma' j js M, red sigma sigma' → red (Abs j js sigma M) (Abs j js sigma' M).

Definition preserves_absr (red : termred) :=
∀ sigma j js M M', red M M' → red (Abs j js sigma M) (Abs j js sigma M').

Lemma preserves_absl_multi_step : ∀ (red: termred), preserves_absl red → preserves_absl
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Abs j js N M); auto. Qed.

Lemma preserves_absr_multi_step : ∀ (red: termred), preserves_absr red → preserves_absr
(multi_step red).
Proof. red. induction 2; split_all. apply succ_red with (Abs j js sigma N); auto. Qed.

Definition preserves_abs (red : termred) :=
∀ sigma sigma' j js M N, red sigma sigma' → red M N → red (Abs j js sigma M) (Abs
j js sigma' N).

Lemma preserves_tagl_seq_red: preserves_tagl seq_red.
Proof. eapply2 preserves_tagl_multi_step. red; split_all. Qed.
Hint Resolve preserves_tagl_seq_red.

Lemma preserves_tagr_seq_red: preserves_tagr seq_red.
Proof. eapply2 preserves_tagr_multi_step. red; split_all. Qed.
Hint Resolve preserves_tagr_seq_red.

Lemma preserves_tag_seq_red: preserves_tag seq_red.
Proof.
red; split_all.
apply transitive_red with (Tag M' N); split_all.
eapply2 preserves_tagl_seq_red.
eapply2 preserves_tagr_seq_red.
Qed.
```

```
Hint Resolve preserves_tag_seq_red.

Lemma preserves_apl_seq_red: preserves_apl seq_red.
Proof. eapply2 preserves_apl_multi_step. red; split_all. Qed.
Hint Resolve preserves_apl_seq_red.

Lemma preserves_apr_seq_red: preserves_apr seq_red.
Proof. eapply2 preserves_apr_multi_step. red; split_all. Qed.
Hint Resolve preserves_apr_seq_red.

Lemma preserves_app_seq_red: preserves_app seq_red.
Proof.
red; split_all.
apply transitive_red with (App M' N); split_all.
eapply2 preserves_apl_seq_red.
eapply2 preserves_apr_seq_red.
Qed.
Hint Resolve preserves_app_seq_red.

Lemma preserves_adl_seq_red: preserves_adl seq_red.
Proof. eapply2 preserves_adl_multi_step. red; split_all. Qed.
Hint Resolve preserves_adl_seq_red.

Lemma preserves_adr_seq_red: preserves_adr seq_red.
Proof. eapply2 preserves_adr_multi_step. red; split_all. Qed.
Hint Resolve preserves_adr_seq_red.

Lemma preserves_add_seq_red: preserves_add seq_red.
Proof.
red; split_all.
apply transitive_red with (Add i M' N); split_all.
eapply2 preserves_adl_seq_red.
eapply2 preserves_adr_seq_red.
Qed.
Hint Resolve preserves_add_seq_red.

Lemma preserves_absl_seq_red: preserves_absl seq_red.
Proof. eapply2 preserves_absl_multi_step. red; split_all. Qed.
Hint Resolve preserves_absl_seq_red.

Lemma preserves_absr_seq_red: preserves_absr seq_red.
Proof. eapply2 preserves_absr_multi_step. red; split_all. Qed.
Hint Resolve preserves_absr_seq_red.

Lemma preserves_abs_seq_red: preserves_abs seq_red.
Proof.
red; split_all.
apply transitive_red with (Abs j js sigma' M); split_all.
eapply2 preserves_absl_seq_red.
```

```
eapply2 preserves_absr_seq_red.
Qed.

Inductive dl_red1: termred :=
  | ref_red : ∀ i, dl_red1 (Ref i) (Ref i)
  | tag_red : ∀ M M' ,
      dl_red1 M M' →
      ∀ N N', dl_red1 N N' → dl_red1 (Tag M N) (Tag M' N')
  | app_red :
      ∀ M M' ,
      dl_red1 M M' →
      ∀ N N', dl_red1 N N' → dl_red1 (App M N) (App M' N')
  | Iop_red : dl_red1 Iop Iop
  | add_red : ∀ M M' ,
      dl_red1 M M' →
      ∀ sigma sigma', dl_red1 sigma sigma' → ∀ i, dl_red1 (Add i M sigma) (Add i M'
sigma')
  | abs_red :
      ∀ sigma sigma' j js M M', dl_red1 sigma sigma' → dl_red1 M M' →
                                       dl_red1 (Abs j js sigma M) (Abs j js sigma' M')
   | app_ref_red : ∀ i M M' , dl_red1 M M' →
                                       dl_red1 (App (Ref i) M) (Tag (Ref i) M')
  | app_tag_red : ∀ M M' N N' P P', dl_red1 M M' → dl_red1 N N' → dl_red1 P P' →
                                       dl_red1 (App (Tag M N) P) (Tag (Tag M' N') P')

  | beta1_red : ∀ sigma sigma' j M M' N N',
                      dl_red1 sigma sigma' → dl_red1 M M' → dl_red1 N N' →
                          dl_red1 (App (Abs j nil sigma M) N)
                                (App (Add j N' sigma') M')
  | beta2_red : ∀ sigma sigma' j j2 js M M' N N',
                      dl_red1 sigma sigma' → dl_red1 M M' → dl_red1 N N' →
                          dl_red1 (App (Abs j (cons j2 js) sigma M) N)
                                (Abs j2 js (Add j N' sigma') M')
  | nil_red : ∀ M M', dl_red1 M M' → dl_red1 (App Iop M) M'
   | subst_eq_red : ∀ j sigma N N', dl_red1 N N' → dl_red1 (App (Add j N sigma) (Ref
j)) N'
  | subst_uneq_red : ∀ sigma sigma' i j N, i≠ j → dl_red1 sigma sigma' →
                                       dl_red1 (App (Add i N sigma) (Ref j)) (App sigma'
(Ref j))
  | subst_tag_red : ∀ j U U' sigma sigma' M M' N N',
                          dl_red1 U U' → dl_red1 sigma sigma' → dl_red1 M M' →
dl_red1 N N' →
                                         dl_red1 (App (Add j U sigma) (Tag M N))
```

```
                                        (App (App (Add j U' sigma') M') (App (Add
j U' sigma') N'))
  | subst_nil_red : ∀ j U sigma sigma', dl_red1 sigma sigma' →
                                             dl_red1 (App (Add j U sigma) Iop) (App
sigma' Iop)
  | subst_add_red : ∀ j N N' sigma sigma' j2 P P' sigma2 sigma2' ,
                     dl_red1 sigma sigma' → dl_red1 P P' → dl_red1 N N' →dl_red1
sigma2 sigma2' →
                            dl_red1 (App (Add j N sigma) (Add j2 P sigma2))
                                  (Add j2 (App (Add j N' sigma') P') (App (Add j N'
sigma') sigma2'))
  | subst_abs_red : ∀ j N N' sigma sigma' j2 js M M' sigma2 sigma2' ,
                     dl_red1 sigma sigma' → dl_red1 M M' → dl_red1 N N' →dl_red1
sigma2 sigma2' →
                         dl_red1 (App (Add j N sigma) (Abs j2 js sigma2 M))
                                (Abs j2 js (App (Add j N' sigma') sigma2') M')
.

Hint Constructors dl_red1.
Definition dl_red := multi_step dl_red1.

Lemma refl_red1: reflective dl_red1.
Proof. red. induction M; split_all. Qed.
Hint Resolve refl_red1.

Ltac inv_dl_red :=
match goal with
| H: dl_red1 (Ref _) _ ⊢ _ ⇒ inversion H; clear H; subst; inv_dl_red
| H: dl_red1 (Tag _ _) _ ⊢ _ ⇒ inversion H; clear H; subst; inv_dl_red
| H: dl_red1 Iop _ ⊢ _ ⇒ inversion H; clear H; subst; inv_dl_red
| H: dl_red1 (Add _ _ _) _ ⊢ _ ⇒ inversion H; clear H; subst; inv_dl_red
| H: dl_red1 (Abs _ _ _ _) _ ⊢ _ ⇒ inversion H; clear H; subst; inv_dl_red
| _ ⇒ invsub; eauto
end.

Lemma diamond_red1 : diamond dl_red1 dl_red1.
Proof.
red. induction M; intros N P r1 r2; inv_dl_red.
elim(IHM1 M' M'0); split_all. elim(IHM2 N' N'0); split_all. eauto.
inversion r1; inversion r2; subst; inv_dl_red; eauto.
elim(IHM1 M' M'0); split_all. elim(IHM2 N' N'0); split_all. eauto.
elim(IHM2 N' M'0); split_all; eauto.
elim(IHM1 (Tag M'0 N'0) (Tag M'1 N'1)); elim(IHM2 N' P'); split_all; eauto.
inv_dl_red. exist (Tag (Tag M' N'2) x).
elim(IHM1 (Abs j nil sigma' M'0) (Abs j nil sigma'0 M'1)); split_all.
```

```
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (App (Add j x0 sigma'1) M').
elim(IHM1 (Abs j (j2 ::js) sigma' M'0) (Abs j (j2 ::js) sigma'0 M'1)); split_all.
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (Abs j2 js (Add j x0 sigma'1) M').
elim(IHM2 N' P); split_all; eauto.
elim(IHM1 (Add j P sigma) (Add j M'0 sigma)); split_all.
inv_dl_red.
elim(IHM1 (Add i N1 sigma') (Add i N1 sigma'0)); split_all.
inv_dl_red.
elim(IHM1 (Add j U' sigma') (Add j M'2 sigma'0)); split_all.
elim(IHM2 (Tag M'0 N'0) (Tag M'1 N'1)); split_all. inv_dl_red.
exist(App (App (Add j M'3 sigma'1) M') (App (Add j M'3 sigma'1) N')) .
elim(IHM1 (Add j U sigma') (Add j U sigma'0)); split_all. inv_dl_red.
elim(IHM1 (Add j N'0 sigma') (Add j M'1 sigma'1)); split_all.
elim(IHM2 (Add j2 P' sigma2')(Add j2 M'0 sigma'0)); split_all. inv_dl_red.
exist(Add j2 (App (Add j M'2 sigma'3) M') (App (Add j M'2 sigma'3) sigma'2)).
elim(IHM1 (Add j N'0 sigma') (Add j M'1 sigma'0)); split_all.
elim(IHM2 (Abs j2 js sigma2' M'0)(Abs j2 js sigma'1 M')); split_all. inv_dl_red.
exist(Abs j2 js (App (Add j M'2 sigma'2) sigma'3) M'3).
elim(IHM2 M' N'); split_all; eauto.
elim(IHM2 M' M'0); split_all; eauto.
elim(IHM1 (Tag M' N') (Tag M'1 N'1)); elim(IHM2 P' N'0); split_all.
inv_dl_red. exist(Tag (Tag M'0 N'2) x).
elim(IHM1 (Tag M' N') (Tag M'0 N'0)); elim(IHM2 P' P'0); split_all.
inv_dl_red. exist(Tag (Tag M'1 N'1) x).
elim(IHM1 (Abs j nil sigma' M') (Abs j nil sigma'0 M'1)); split_all.
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (App (Add j x0 sigma'1) M'0).
elim(IHM1 (Abs j nil sigma' M') (Abs j nil sigma'0 M'0)); split_all.
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (App (Add j x0 sigma'1) M'1).
elim(IHM1 (Abs j (j2::js) sigma' M') (Abs j (j2:: js) sigma'0 M'1)); split_all.
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (Abs j2 js (Add j x0 sigma'1) M'0).
elim(IHM1 (Abs j (j2::js) sigma' M') (Abs j (j2:: js) sigma'0 M'0)); split_all.
elim(IHM2 N' N'0); split_all.
inv_dl_red. exist (Abs j2 js (Add j x0 sigma'1) M'1).
elim(IHM2 N N'); split_all; eauto.
elim(IHM1 (Add j N sigma)(Add j M'0 sigma)); split_all; inv_dl_red; eauto.
elim(IHM1 (Add j N sigma)(Add j P sigma)); split_all; inv_dl_red; eauto.
elim(IHM1 (Add i N0 sigma')(Add i N0 sigma'0)); split_all; inv_dl_red; eauto.
```

```
elim(IHM1 (Add i N0 sigma')(Add i N0 sigma'0)); split_all; inv_dl_red; eauto.
elim(IHM1 (Add j U' sigma')(Add j M'2 sigma'0)); split_all.
elim(IHM2 (Tag M' N') (Tag M'1 N'1)); split_all. inv_dl_red; eauto.
exist(App (App (Add j M'3 sigma'1) M'0) (App (Add j M'3 sigma'1) N'0)) .
elim(IHM1 (Add j U' sigma')(Add j U'0 sigma'0)); split_all.
elim(IHM2 (Tag M' N') (Tag M'0 N'0)); split_all. inv_dl_red; eauto.
exist(App (App (Add j M'2 sigma'1) M'1) (App (Add j M'2 sigma'1) N'1)) .
elim(IHM1 (Add j U sigma') (Add j U sigma'0)); split_all. inv_dl_red.
elim(IHM1 (Add j U sigma') (Add j U sigma'0)); split_all. inv_dl_red.
elim(IHM1 (Add j N' sigma')(Add j M'1 sigma'1)); split_all.
elim(IHM2 (Add j2 P' sigma2') (Add j2 M'0 sigma'0)); split_all. inv_dl_red; eauto.
exist(Add j2 (App (Add j M'2 sigma'3) M') (App (Add j M'2 sigma'3) sigma'2)).
elim(IHM1 (Add j N' sigma')(Add j N'0 sigma'0)); split_all.
elim(IHM2 (Add j2 P' sigma2') (Add j2 P'0 sigma2'0)); split_all. inv_dl_red; eauto.
exist(Add j2 (App (Add j M'0 sigma'2) M') (App (Add j M'0 sigma'2) sigma'1)) .
elim(IHM2 (Abs j2 js sigma2' M')(Abs j2 js sigma'1 M'0)); split_all.
elim(IHM1 (Add j N' sigma') (Add j M'1 sigma'0)); split_all.
inv_dl_red. exist (Abs j2 js (App (Add j M'2 sigma'2) sigma'3) M'3).
elim(IHM2 (Abs j2 js sigma2' M')(Abs j2 js sigma2'0 M'0)); split_all.
elim(IHM1 (Add j N' sigma') (Add j N'0 sigma'0)); split_all.
inv_dl_red. exist (Abs j2 js (App (Add j M'1 sigma'1) sigma'2) M'2).
elim(IHM1 M' M'0); split_all. elim(IHM2 sigma' sigma'0); split_all. eauto.
elim(IHM1 sigma' sigma'0); split_all. elim(IHM2 M' M'0); split_all. eauto.
Qed.

Theorem tuple_parallel_confluence: confluence lambda dl_red.
Proof. red. eapply2 diamond0_tiling. eapply2 diamond_iff_diamond0. eapply2 diamond_red1.
Qed.

Definition implies_red (red1 red2: termred) := ∀ M N, red1 M N → red2 M N.

Lemma implies_red_multi_step: ∀ red1 red2, implies_red red1 (multi_step red2) →
                                        implies_red (multi_step red1) (multi_step
red2).
Proof. red.
intros red1 red2 IR M N R; induction R; split_all.
apply transitive_red with N; auto.
Qed.

Lemma seq_red1_to_red1 : implies_red seq_red1 dl_red1.
Proof.
  red. intros M N B; induction B; split_all; try (red; one_step; fail).
Qed.

Lemma seq_red_to_red: implies_red seq_red dl_red.
Proof.
```

```
  eapply2 implies_red_multi_step. red; split_all; one_step; eapply2 seq_red1_to_red1.
Qed.

Lemma to_seq_red_multi_step: ∀ red, implies_red red seq_red → implies_red (multi_step
red) seq_red.
Proof.
red. intros red B M N R; induction R; split_all.
red; split_all.
assert(seq_red M N) by eapply2 B.
apply transitive_red with N; auto.
eapply2 IHR.
Qed.

Hint Resolve preserves_app_seq_red preserves_abs_seq_red.

Lemma dl_red1_to_seq_red: implies_red dl_red1 seq_red .
Proof.
  red. intros M N OR; induction OR; split_all;
        try(eapply2 succ_red;
        try eapply2 preserves_ref_seq_red;
        try eapply2 beta_tag_seq_red;
        try eapply2 preserves_tag_seq_red;
        try eapply2 preserves_add_seq_red;
       try eapply2 preserves_abs_seq_red;
        try eapply2 preserves_app_seq_red;
        try eapply2 preserves_aps_seq_red; fail).
Qed.

Hint Resolve dl_red1_to_seq_red.

Lemma dl_red_to_seq_red: implies_red dl_red seq_red.
Proof. eapply2 to_seq_red_multi_step. Qed.

Lemma diamond_seq_red: diamond seq_red seq_red.
Proof.
red; split_all.
assert(dl_red M N) by eapply2 seq_red_to_red.
assert(dl_red M P) by eapply2 seq_red_to_red.
elim(tuple_parallel_confluence M N H1 P); split_all.
exist x; eapply2 dl_red_to_seq_red.
Qed.

Theorem simple_confluence: confluence lambda seq_red.
Proof. red. split_all. eapply2 diamond_seq_red. Qed.

Inductive normal : lambda → Prop :=
| nf_ref: ∀ i, normal (Ref i)
| nf_tag: ∀ s u, normal s → normal u → normal (Tag s u)
```

```
| nf_nil: normal Iop
| nf_add: ∀ s j u, normal s → normal u → normal (Add j u s)
| nf_abs : ∀ sigma j js M, normal sigma → normal M → normal (Abs j js sigma M)
.

Hint Constructors normal.

Definition irreducible M (red:termred) := ∀ N, red M N → False.

Lemma normal_is_irreducible:
∀ M, normal M → irreducible M seq_red1.
Proof.
  intros M nor; induction nor; split_all;
  intro; intro r; inversion r; subst; split_all;
  try (eapply2 IHnor1; fail); try (eapply2 IHnor2; fail).
Qed.

Theorem simple_progress :
∀ (M : lambda), (normal M) ∨ (∃ N, seq_red1 M N) .
Proof.
induction M; try (inversion IHM); subst; split_all; eauto.
inversion IHM1; inversion IHM2; split_all; try (right; eauto; fail).
inversion IHM1; inversion IHM2; split_all; eauto.
inversion H; subst; eauto. inversion H0; subst; eauto.
right; assert(i=j ∨ i≠j) by decide equality. inversion H3; subst; eauto.
right; case js; eauto.
inversion IHM1; inversion IHM2; split_all; eauto.
inversion IHM1; inversion IHM2; split_all; eauto.
Qed.

Lemma irreducible_is_normal:
∀ M, irreducible M seq_red1 → normal M.
Proof.
split_all. elim(simple_progress M); split_all. assert False by eapply2 H. inversion H0.
Qed.

Theorem irreducible_iff_normal:
  ∀ M, (irreducible M seq_red1 ↔ normal M).
Proof. split_all. eapply2 irreducible_is_normal. eapply2 normal_is_irreducible. Qed.

Definition stable M := ∀ N, dl_red M N → N = M.

Theorem normal_implies_stable: ∀ M, normal M → stable M.
Proof.
unfold stable; split_all.
assert(seq_red M N) by eapply2 dl_red_to_seq_red.
inversion H1; subst; auto.
assert(irreducible M seq_red1) by eapply2 irreducible_iff_normal.
```

```
assert False by eapply2 H4. inversion H5.
Qed.

Definition omega := Abs 1 (0::nil) Iop (Tag (Ref 0) (Tag (Tag (Ref 1) (Ref 1)) (Ref 0))).
Definition Ycomb := Abs 0 nil (Add 1 omega Iop) (Tag (Ref 0) (Tag (Tag (Ref 1) (Ref
1)) (Ref 0))).

Lemma fixpoint_property:
∀ f, seq_red (App Ycomb f) (App f (App Ycomb f)).
Proof. intros; unfold Ycomb at 1; unfold omega; subst. repeat eapply2 succ_red. Qed.

Definition omega2 :=
  Abs 1 (0::2::nil) Iop (Tag (Tag (Ref 0) (Tag (Tag (Ref 1) (Ref 1)) (Ref 0))) (Ref 2))
 .
Definition Y2 := App omega2 omega2.

Lemma fixpoint2_property:
∀ f N, seq_red (App (App Y2 f) N) (App (App f (App Y2 f)) N).
Proof.
intros; unfold Y2 at 1. unfold omega2 at 1.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red.
eapply2 preserves_app_seq_red. eapply2 preserves_app_seq_red.
repeat eapply2 succ_red.
Qed.

Definition abs j js := Abs j js Iop.

Definition tt := abs 1 (0::nil) (Ref 1).
Definition ff := abs 1 (0::nil) (Ref 0).

Lemma if_true : ∀ m n, seq_red (App (App tt m) n) m.
Proof. split_all; subst; unfold tt, abs. eapply2 succ_red. Qed.

Lemma if_false : ∀ m n, seq_red (App (App ff m) n) n.
Proof. split_all; subst; unfold ff, abs. eapply2 succ_red. Qed.

Definition zero:= tt. Definition succ := abs 2 (1::0::nil) (Tag (Ref 0) (Ref 2)). Definition
case := abs 2 (1::0::nil) (Tag (Tag (Ref 2) (Ref 1)) (Ref 0)).

Fixpoint scott n :=
  match n with
    | 0 ⇒ tt
    | S n ⇒ Abs 1 (0::nil) (Add 2 (scott n) Iop) (Tag (Ref 0) (Ref 2))
  end.

Lemma scott_numerals_are_normal: ∀ n, normal (scott n).
Proof.
```

```
induction n; unfold scott; fold scott; unfold zero, abs, value; split_all. unfold tt, abs;
auto.
Qed.

Hint Resolve scott_numerals_are_normal.

Lemma succ_scott: ∀ n, seq_red (App succ (scott n)) (scott (S n)).
Proof. intro; unfold succ, abs. eapply2 succ_red. Qed.

Definition is_zero :=
Abs 2 nil (Add 0 (abs 0 nil ff) (Add 1 tt Iop))
     (Tag (Tag (Ref 2) (Ref 1)) (Ref 0)) .

Lemma is_zero_zero: seq_red (App is_zero zero) tt .
Proof. unfold is_zero, zero, tt, abs; split_all. repeat eapply2 succ_red. Qed.

Lemma is_zero_succ: ∀ n, seq_red (App is_zero (scott (S n))) ff .
Proof.
intros. unfold is_zero, abs. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
unfold scott; fold scott. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
unfold ff, abs; split_all. repeat eapply2 succ_red.
Qed.

Definition my_pred :=
 abs 0 nil (Tag (Tag (Ref 0) zero) (abs 0 nil (Ref 0))).

Lemma pred_zero: seq_red (App my_pred zero) zero.
Proof.
unfold my_pred, zero, abs. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
eapply2 succ_red. eapply transitive_red.
eapply2 if_true. unfold tt, abs; split_all. eapply2 succ_red.
Qed.

Lemma pred_succ: ∀ n, seq_red (App my_pred (scott (S n))) (scott n).
Proof.
intro n; case n; unfold my_pred, abs; split_all; repeat eapply2 succ_red.
Qed.

Definition my_plus_aux :=
abs 3 (2::nil) (Tag (Tag (Ref 2) (abs 0 nil (Ref 0)))
(Abs 1 (0::nil) (Add 3 (Ref 3) Iop)
      (App succ (Tag (Tag (Ref 3) (Ref 1)) (Ref 0))))).
Definition my_plus := App Y2 my_plus_aux.

Lemma my_plus_scott:
∀ m n, seq_red (App (App my_plus (scott m)) (scott n)) (scott (m+n)).
```

Proof.
  induction m; intros.
  split_all. unfold *my_plus*, *zero*, *abs*; *split_all.*
  eapply *transitive_red.* eapply *preserves_app_seq_red.*
  eapply2 fixpoint2_property. auto.
  unfold *my_plus_aux*, *abs.* *eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.*
  eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
  eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
  eapply2 succ_red. eapply2 succ_red.
  eapply *transitive_red.* eapply *preserves_app_seq_red. eapply2 if_true.* auto.
  eapply2 succ_red.
  simpl. unfold *my_plus*, *abs.*
  eapply *transitive_red.* eapply *preserves_app_seq_red.*
  eapply2 fixpoint2_property. auto.
replace (*App Y2 my_plus_aux*) with *my_plus* by auto.
  unfold *my_plus_aux*, *abs.*
  eapply2 succ_red; repeat (*eapply2 succ_red*; *eapply2 succ_red*).
  eapply *transitive_red.*
  unfold *succ*, *abs. eapply2 succ_red.*
  eapply *transitive_red.* eapply *succ_red.* eapply *subst_abs_seq_red.* auto.
  eapply2 preserves_abs_seq_red. eapply2 succ_red.
  eapply2 preserves_add_seq_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
  eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red. eapply2 succ_red.
  eapply2 succ_red. eapply2 succ_red. eapply2 IHm.
eapply2 succ_red.
Qed.