Alea: a library for reasoning on randomized algorithms in Coq Version 5

Christine Paulin-Mohring
with contributions by Pierre Courtieu
PROVAL Team
LRI, UMR 8623 Univ. Paris-Sud 11, CNRS & INRIA Saclay - Île-de-France

November 24, 2010

Contents

1 Misc.v: Preliminaries

```
Require Export Arith.
Require Import Coq.Classes.SetoidTactics.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.Morphisms.

Open Local Scope signature_scope.

Lemma beq_nat_neq: ∀ x y : nat, x ≠ y → false = beq_nat x y.

Lemma if_beq_nat_nat_eq_dec : ∀ A (x y:nat) (a b:A),
  (if beq_nat x y then a else b) = if eq_nat_dec x y then a else b.

Definition ifte A (test:bool) (thn els:A) := if test then thn else els.

Add Parametric Morphism (A:Type) : (@ifte A)
  with signature (eq ⇒eq ⇒ eq ⇒ eq) as ifte_morphism1.

Add Parametric Morphism (A:Type) x : (@ifte A x)
  with signature (eq ⇒ eq ⇒ eq) as ifte_morphism2.

Add Parametric Morphism (A:Type) x y : (@ifte A x y)
  with signature (eq ⇒ eq) as ifte_morphism3.
```

1.1 Definition of iterator *compn*

compn f u n x is defined as (f (u (n-1)).. (f (u 0) x))

```
Fixpoint compn (A:Type)(f:A → A → A) (x:A) (u:nat → A) (n:nat) {struct n}: A :=
   match n with O ⇒ x | (S p) ⇒ f (u p) (compn f x u p) end.

Lemma comp0 : ∀ (A:Type) (f:A → A → A) (x:A) (u:nat → A), compn f x u 0 = x.

Lemma compS : ∀ (A:Type) (f:A → A → A) (x:A) (u:nat → A) (n:nat),
                compn f x u (S n) = f (u n) (compn f x u n).
```

1.2 Reducing if constructs

```
Lemma if_then : ∀ (P:Prop) (b:{ P }+{ ~P})(A:Type)(p q:A),
      P → (if b then p else q) = p.

Lemma if_else : ∀ (P:Prop) (b:{ P }+{ ~P})(A:Type)(p q:A),
      ~P → (if b then p else q) = q.
```

1.3 Classical reasoning

```
Definition class (A:Prop) := //A → A.
Lemma class_neg : ∀ A:Prop, class ( /A).
Lemma class_false : class False.
Hint Resolve class_neg class_false.
Definition orc (A B:Prop) := ∀ C:Prop, class C → (A → C) → (B → C) → C.
Lemma orc_left : ∀ A B:Prop, A → orc A B.
Lemma orc_right : ∀ A B:Prop, B → orc A B.
Hint Resolve orc_left orc_right.
Lemma class_orc : ∀ A B, class (orc A B).
Implicit Arguments class_orc [].
Lemma orc_intro : ∀ A B, ( /A → /B → False) → orc A B.
Lemma class_and : ∀ A B, class A → class B → class (A ∧ B).
Lemma excluded_middle : ∀ A, orc A ( /A).
Definition exc (A :Type)(P:A → Prop) :=
  ∀ C:Prop, class C → (∀ x:A, P x → C) → C.
Lemma exc_intro : ∀ (A :Type)(P:A → Prop) (x:A), P x → exc P.
Lemma class_exc : ∀ (A :Type)(P:A → Prop), class (exc P).
Lemma exc_intro_class : ∀ (A:Type) (P:A → Prop), ((∀ x, /P x) → False) → exc P.
Lemma not_and_elim_left : ∀ A B, /(A ∧ B) → A → /B.
Lemma not_and_elim_right : ∀ A B, /(A ∧ B) → B → /A.
Hint Resolve class_orc class_and class_exc excluded_middle.
Lemma class_double_neg : ∀ P Q: Prop, class Q → (P → Q) → //P → Q.
```

1.4 Extensional equality

```
Definition feq A B (f g : A → B) := ∀ x, f x = g x.
Lemma feq_refl : ∀ A B (f:A→B), feq f f.
Lemma feq_sym : ∀ A B (f g : A → B), feq f g → feq g f.
Lemma feq_trans : ∀ A B (f g h: A → B), feq f g → feq g h → feq f h.
Hint Resolve feq_refl.
Hint Immediate feq_sym.
Hint Unfold feq.
Add Parametric Relation (A B : Type) : (A → B) (feq (A:=A) (B:=B))
  reflexivity proved by (feq_refl (A:=A) (B:=B))
  symmetry proved by (feq_sym (A:=A) (B:=B))
  transitivity proved by (feq_trans (A:=A) (B:=B))
as feq_rel.
```

2 Ccpo.v: Specification and properties of a cpo

```
Require Export Arith.
Require Export Omega.
Require Export Coq.Classes.SetoidTactics.
```

```
Require Export Coq.Classes.SetoidClass.
Require Export Coq.Classes.Morphisms.

Open Local Scope signature_scope.
```

2.1 Ordered type

```
Definition eq_rel {A} (E1 E2:relation A) := ∀ x y, E1 x y ↔ E2 x y.

Class Order {A} (E:relation A) (R:relation A) :=
  {reflexive :> Reflexive R;
   order_eq : ∀ x y, R x y ∧ R y x ↔ E x y;
   transitive :> Transitive R }.

Instance OrderEqRefl `{Order A E R} : Reflexive E.
Save.

Instance OrderEqSym `{Order A E R} : Symmetric E.
Save.

Instance OrderEqTrans `{Order A E R} : Transitive E.
Save.

Instance OrderEquiv `{Order A E R} : Equivalence E.

Class ord A :=
   { Oeq : relation A;
     Ole : relation A;
     order_rel :> Order Oeq Ole }.

Lemma OrdSetoid `(o:ord A) : Setoid A.

Add Parametric Relation {A} {o:ord A} : A (@Oeq _ o)
reflexivity proved by OrderEqRefl
symmetry proved by OrderEqSym
transitivity proved by OrderEqTrans
as Oeq_setoid.

Infix "≤" := Ole.
Infix "≡" := Oeq : type_scope.

Lemma Ole_refl_eq : ∀ {O} {o:ord O} (x y:O), x ≡ y → x ≤ y.

Hint Immediate @Ole_refl_eq.

Lemma Ole_refl_eq_inv : ∀ {O} {o:ord O} (x y:O), x ≡ y → y ≤ x.

Hint Immediate @Ole_refl_eq_inv.

Lemma Ole_trans : ∀ {O} {o:ord O} (x y z:O), x ≤ y → y ≤ z → x ≤ z.

Lemma Ole_refl : ∀ {O} {o:ord O} (x:O), x ≤ x.

Hint Resolve @Ole_refl.

Add Parametric Relation {A} {o:ord A} : A (@Ole _ o)
reflexivity proved by Ole_refl
transitivity proved by Ole_trans
as Ole_setoid.

Lemma Ole_antisym : ∀ {O} {o:ord O} (x y:O), x ≤ y → y ≤ x → x ≡ y.
Hint Immediate @Ole_antisym.

Lemma Oeq_refl : ∀ {O} {o:ord O} (x:O), x ≡ x.
Hint Resolve @Oeq_refl.

Lemma Oeq_refl_eq : ∀ {O} {o:ord O} (x y:O), x = y → x ≡ y.
Hint Resolve @Oeq_refl_eq.
```

```
Lemma Oeq_sym : ∀ {O} {o:ord O} (x y:O), x ≡ y → y ≡ x.
Lemma Oeq_le : ∀ {O} {o:ord O} (x y:O), x ≡ y → x ≤ y.
Lemma Oeq_le_sym : ∀ {O} {o:ord O} (x y:O), x ≡ y → y ≤ x.
Hint Resolve @Oeq_le.
Hint Immediate @Oeq_sym @Oeq_le_sym.
Lemma Oeq_trans
  : ∀ {O} {o:ord O} (x y z:O), x ≡ y → y ≡ z → x ≡ z.
Hint Resolve @Oeq_trans.
Add Parametric Morphism `(o:ord A): (Ole (ord:=o))
with signature (Oeq (A:=A) ⇒ Oeq (A:=A) ⇒ iff) as Ole_eq_compat_iff.
Save.
```

Equivalence of orders

```
Definition eq_ord {O} (o1 o2:ord O) := eq_rel (Ole (ord:=o1)) (Ole (ord:=o2)).
Lemma eq_ord_equiv : ∀ {O} (o1 o2:ord O), eq_ord o1 o2 →
      eq_rel (Oeq (ord:=o1)) (Oeq (ord:=o2)).
Lemma Ole_eq_compat :
     ∀ {O} {o:ord O} (x1 x2 : O),
       x1 ≡ x2 → ∀ x3 x4 : O, x3 ≡ x4 → x1 ≤ x3 → x2 ≤ x4.
Lemma Ole_eq_right : ∀ {O} {o:ord O} (x y z: O),
            x ≤ y → y ≡ z → x ≤ z.
Lemma Ole_eq_left : ∀ {O} {o:ord O} (x y z: O),
            x ≡ y → y ≤ z → x ≤ z.
Add Parametric Morphism `{o:ord A} : (Oeq (A:=A))
       with signature Oeq ⟹ Oeq ⟹ iff as Oeq_iff_morphism.
Qed.
Add Parametric Morphism `{o:ord A} : (Ole (A:=A))
       with signature Oeq ⟹ Oeq ⟹ iff as Ole_iff_morphism.
Qed.
Add Parametric Morphism `{o:ord A} : (Ole (A:=A))
       with signature Ole -> Ole ⟹ Basics.impl as Ole_impl_morphism.
Qed.
```

2.1.1 Dual order

- *Iord x y = y ≤ x*

```
Definition Iord : ∀ O {o:ord O}, ord O.
Defined.
Implicit Arguments Iord [[o]].
```

2.1.2 Order on functions

```
Definition fun_ext A B (R:relation B) : relation (A → B) :=
                fun f g ⇒ ∀ x, R (f x) (g x).
Implicit Arguments fun_ext [B].
```

- *ford f g := ∀ x, f x ≤ g x*

```
Instance ford A O {o:ord O} : ord (A → O) :=
  {Oeq:=fun_ext A (Oeq (A:=O));Ole:=fun_ext A (Ole (A:=O))}.
Defined.

Lemma ford_le_elim : ∀ A O (o:ord O) (f g:A → O), f ≤ g → ∀ n, f n ≤ g n.
Hint Immediate ford_le_elim.

Lemma ford_le_intro : ∀ A O (o:ord O) (f g:A → O), ( ∀ n, f n ≤ g n ) → f ≤ g.
Hint Resolve ford_le_intro.

Lemma ford_eq_elim : ∀ A O (o:ord O) (f g:A → O), f ≡ g → ∀ n, f n ≡ g n.
Hint Immediate ford_eq_elim.

Lemma ford_eq_intro : ∀ A O (o:ord O) (f g:A → O), ( ∀ n, f n ≡ g n ) → f ≡ g.
Hint Resolve ford_eq_intro.
```

2.2 Monotonicity

2.2.1 Definition and properties

```
Class monotonic `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob) :=
      monotonic_def : ∀ x y, x ≤ y → f x ≤ f y.

Lemma monotonic_intro : ∀ `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob),
  (∀ x y, x ≤ y → f x ≤ f y) → monotonic f.
Hint Resolve @monotonic_intro.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob) {m:monotonic f} : f
with signature (Ole (A:=Oa) ⟹ Ole (A:=Ob))
as monotonic_morphism.
Save.

Class stable `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob) :=
      stable_def : ∀ x y, x ≡ y → f x ≡ f y.
Hint Unfold stable.

Lemma stable_intro : ∀ `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob),
  (∀ x y, x ≡ y → f x ≡ f y) → stable f.
Hint Resolve @stable_intro.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob) {s:stable f} : f
with signature (Oeq (A:=Oa) ⟹ Oeq (A:=Ob))
as stable_morphism.
Save.

Typeclasses Opaque monotonic stable.

Instance monotonic_stable `{o1:ord Oa} `{o2:ord Ob} (f : Oa → Ob) {m:monotonic f}
        : stable f.
Save.
```

2.2.2 Type of monotonic functions

```
Record fmon `{o1:ord Oa} `{o2:ord Ob}:= mon
          {fmont :> Oa → Ob;
           fmonotonic: monotonic fmont}.

Implicit Arguments mon [[Oa] [o1] [Ob] [o2] [fmonotonic]].
Implicit Arguments fmon [[o1] [o2]].

Hint Resolve @fmonotonic.

Notation "Oa -m> Ob" := (fmon Oa Ob)
```

```
    (right associativity, at level 30) : O_scope.
Notation "Oa –m> Ob" := (fmon Oa (o1:=lord Oa) Ob )
    (right associativity, at level 30) : O_scope.
Notation "Oa –m-> Ob" := (fmon Oa (o1:=lord Oa) Ob (o2:=lord Ob))
    (right associativity, at level 30) : O_scope.
Notation "Oa -m-> Ob" := (fmon Oa Ob (o2:=lord Ob))
    (right associativity, at level 30) : O_scope.

Open Scope O_scope.

Lemma mon_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob){mf: monotonic f} x,
      mon f x = f x.
Hint Resolve @mon_simpl.

Instance fstable `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> Ob) : stable f.
Save.

Hint Resolve @fstable.

Lemma fmon_le : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> Ob) x y,
                  x ≤ y → f x ≤ f y.
Hint Resolve @fmon_le.

Lemma fmon_eq : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> Ob) x y,
                  x ≡ y → f x ≡ f y.
Hint Resolve @fmon_eq.

Instance fmono Oa Ob {o1:ord Oa} {o2:ord Ob} : ord (Oa -m> Ob)
   := {Oeq := fun (f g : Oa-m> Ob)=> ∀ x, f x ≡ g x;
       Ole := fun (f g : Oa-m> Ob)=> ∀ x, f x ≤ g x}.
Defined.

Lemma mon_le_compat : ∀ `{o1:ord Oa} `{o2:ord Ob} (f g:Oa → Ob)
      {mf:monotonic f} {mg:monotonic g}, f ≤ g → mon f ≤ mon g.
Hint Resolve @ mon_le_compat.

Lemma mon_eq_compat : ∀ `{o1:ord Oa} `{o2:ord Ob} (f g:Oa→ Ob)
      {mf:monotonic f} {mg:monotonic g}, f ≡ g → mon f ≡ mon g.
Hint Resolve @ mon_eq_compat.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob}
       : (fmont (Oa:=Oa) (Ob:=Ob))
       with signature Oeq ⟹ Oeq ⟹ Oeq as fmont_eq_morphism.
Qed.
```

2.2.3 Monotonicity and dual order

```
Lemma lmonotonic `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) {m:monotonic f}
        : monotonic (o1:=lord Oa) (o2:=lord Ob) f.

Hint Extern 2 (@monotonic _ (lord _) _ (lord _) _) ⇒ apply @lmonotonic
  : typeclass_instances.

Definition imon `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) {m:monotonic f}
   : Oa -m→ Ob := mon (o1:=lord Oa) (o2:=lord Ob) f.

Lemma imon_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) {m:monotonic f} (x:Oa),
     imon f x = f x.
```

- *Iord* $(A \to U)$ corresponds to $A \to$ *Iord* U

```
Lemma lord_app {A} `{o1:ord Oa} (x: A) : ((A → Oa) -m→ Oa).
```

- *Imon f* uses f as monotonic function over the dual order.

```
Definition lmon : ∀ `{o1:ord Oa} `{o2:ord Ob}, (Oa -m> Ob) → (Oa -m→ Ob).
Defined.

Lemma lmon_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> Ob)(x:Oa),
                   lmon f x = f x.
```

2.2.4 Monotonicity and equality

```
Lemma mon_fun_eq_monotonic
  : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) (g:Oa -m> Ob),
          f ≡ g → monotonic f.

Definition mon_fun_subst `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) (g:Oa -m> Ob) (H:f ≡ g)
   : Oa -m> Ob := mon f (fmonotonic:= mon_fun_eq_monotonic _ _ H).

Lemma mon_fun_eq
  : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa → Ob) (g:Oa -m> Ob)
          (H:f ≡ g), g ≡ mon_fun_subst f g H.
```

2.2.5 Monotonic functions with 2 arguments

```
Class monotonic2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc) :=
    monotonic2_intro : ∀ (x y:Oa) (z t:Ob), x ≤ y → z ≤ t → f x z ≤ f y t.

Instance mon2_intro `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
    {m1:monotonic f} {m2: ∀ x, monotonic (f x)} : monotonic2 f | 10.
Save.

Lemma mon2_elim1 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
    {m:monotonic2 f} : monotonic f.

Lemma mon2_elim2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
    {m:monotonic2 f} : ∀ x, monotonic (f x).
Hint Immediate @mon2_elim1 @mon2_elim2: typeclass_instances.

Definition mon_comp {A} `{o1: ord Oa} `{o2: ord Ob}
         (f:A → Oa → Ob) {mf:∀ x, monotonic (f x)} : A → Oa -m> Ob
         := fun x ⇒ mon (f x).

Instance mon_fun_mon `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
    {m:monotonic2 f} : monotonic (fun x ⇒ mon (f x)).
Save.

Class stable2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc) :=
    stable2_intro : ∀ (x y:Oa) (z t:Ob), x≡y → z ≡ t → f x z ≡ f y t.

Instance monotonic2_stable2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
    (f:Oa → Ob → Oc) {m:monotonic2 f} : stable2 f.
Save.

Typeclasses Opaque monotonic2 stable2.

Definition mon2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
     {mf:monotonic2 f} : Oa -m> Ob -m> Oc := mon (fun x ⇒ mon (f x)).

Lemma mon2_simpl : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob → Oc)
     {mf:monotonic2 f} x y, mon2 f x y = f x y.
Hint Resolve @mon2_simpl.

Lemma mon2_le_compat : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
     (f g:Oa → Ob → Oc) {mf: monotonic2 f} {mg:monotonic2 g},
     f ≤ g → mon2 f ≤ mon2 g.

Definition fun2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob -m> Oc)
     : Oa → Ob → Oc := fun x ⇒ f x.
```

```
Instance fmon2_mon `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc} (f:Oa → Ob -m> Oc) :
      ∀ x:Oa, monotonic (fun2 f x).
Save.

Instance fun2_monotonic `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
        (f:Oa → Ob -m> Oc) {mf:monotonic f} : monotonic (fun2 f).
Save.
Hint Resolve @fun2_monotonic.

Instance fmonotonic2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc)
        : monotonic2 (fun2 f).
Save.
Hint Resolve @fmonotonic2.

Definition mfun2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc)
   : Oa-m> (Ob → Oc) := mon (fun2 f).

Lemma mfun2_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc) x y,
     mfun2 f x y = f x y.

Instance mfun2_mon `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
        (f:Oa -m> Ob -m> Oc) x : monotonic (mfun2 f x).
Save.

Lemma mon2_fun2 : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
     (f:Oa -m> Ob -m> Oc), mon2 (fun2 f) ≡ f.

Lemma fun2_mon2 : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
      (f:Oa → Ob → Oc) {mf:monotonic2 f} , fun2 (mon2 f) ≡ f.
Hint Resolve @mon2_fun2 @fun2_mon2.

Instance fstable2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc)
                : stable2 (fun2 f).
Save.
Hint Resolve @fstable2.

Definition lmon2 : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc},
     (Oa -m> Ob -m> Oc) → (Oa -m> Ob -m→ Oc).
Defined.

Lemma lmon2_simpl : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
      (f:Oa -m> Ob -m> Oc) (x:Oa) (y: Ob),
      lmon2 f x y = f x y.

Lemma lmonotonic2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
      (f:Oa → Ob → Oc){mf : monotonic2 f}
      : monotonic2 (o1:=lord Oa) (o2:=lord Ob) (o3:=lord Oc) f.

Hint Extern 2 (@monotonic2 _ (lord _) _ (lord _) _ (lord _) _) ⇒ apply @lmonotonic2
  : typeclass_instances.

Definition imon2 `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
      (f:Oa → Ob → Oc){mf : monotonic2 f} : Oa -m> Ob -m→ Oc :=
      mon2 (o1:=lord Oa) (o2:=lord Ob) (o3:=lord Oc) f.

Lemma imon2_simpl : ∀ `{o1: ord Oa} `{o2: ord Ob} `{o3:ord Oc}
      (f:Oa → Ob → Oc){mf : monotonic2 f} (x:Oa) (y:Ob),
      imon2 f x y = f x y.
```

2.3 Sequences

2.3.1 Usual order on natural numbers

```
Instance nat0 : ord nat :=
```

```
    { Oeq := fun n m : nat ⇒ n = m;
      Ole := fun n m : nat ⇒ (n ≤ m)%nat}.
Defined.

Lemma le_Ole : ∀ n m, ((n ≤ m)%nat)-> n ≤ m.
Hint Resolve le_Ole.

Lemma nat_monotonic : ∀ {O} {o:ord O}
                      (f:nat → O), (∀ n, f n ≤ f (S n)) → monotonic f.
Hint Resolve @nat_monotonic.

Definition fnatO_intro : ∀ {O} {o:ord O} (f:nat → O), (∀ n, f n ≤ f (S n)) → nat -m> O.
Defined.

Lemma fnatO_elim : ∀ {O} {o:ord O} (f:nat -m> O) (n:nat), f n ≤ f (S n).
Hint Resolve @fnatO_elim.
```

- (mseq_lift_left f n) k = f (n+k)

```
Definition seq_lift_left {O} (f:nat → O) n := fun k ⇒ f (n+k)%nat.

Instance mon_seq_lift_left
  : ∀ n {O} {o:ord O} (f:nat → O) {m:monotonic f}, monotonic (seq_lift_left f n).
Save.

Definition mseq_lift_left : ∀ {O} {o:ord O} (f:nat -m> O) (n:nat), nat -m> O.
Defined.

Lemma mseq_lift_left_simpl : ∀ {O} {o:ord O} (f:nat -m> O) (n k:nat),
    mseq_lift_left f n k = f (n+k)%nat.

Lemma mseq_lift_left_le_compat : ∀ {O} {o:ord O} (f g:nat -m> O) (n:nat),
                f ≤ g → mseq_lift_left f n ≤ mseq_lift_left g n.
Hint Resolve @mseq_lift_left_le_compat.

Add Parametric Morphism {O} {o:ord O} : (@mseq_lift_left _ o)
  with signature Oeq ⟹ eq ⟹ Oeq
  as mseq_lift_left_eq_compat.
Save.
Hint Resolve @mseq_lift_left_eq_compat.

Add Parametric Morphism {O} {o:ord O}: (@seq_lift_left O)
  with signature Oeq ⟹ eq ⟹ Oeq
  as seq_lift_left_eq_compat.
Save.
Hint Resolve @seq_lift_left_eq_compat.
```

- (mseq_lift_right f n) k = f (k+n)

```
Definition seq_lift_right {O} (f:nat → O) n := fun k ⇒ f (k+n)%nat.

Instance mon_seq_lift_right
   : ∀ n {O} {o:ord O} (f:nat → O) {m:monotonic f}, monotonic (seq_lift_right f n).
Save.

Definition mseq_lift_right : ∀ {O} {o:ord O} (f:nat -m> O) (n:nat), nat -m> O.
Defined.

Lemma mseq_lift_right_simpl : ∀ {O} {o:ord O} (f:nat -m> O) (n k:nat),
    mseq_lift_right f n k = f (k+n)%nat.

Lemma mseq_lift_right_le_compat : ∀ {O} {o:ord O} (f g:nat -m> O) (n:nat),
                f ≤ g → mseq_lift_right f n ≤ mseq_lift_right g n.
Hint Resolve @mseq_lift_right_le_compat.
```

```
Add Parametric Morphism {O} {o:ord O} : (mseq_lift_right (o:=o))
    with signature Oeq ⟹ eq ⟹ Oeq
    as mseq_lift_right_eq_compat.
Save.

Add Parametric Morphism {O} {o:ord O}: (@seq_lift_right O)
  with signature Oeq ⟹ eq ⟹ Oeq
  as seq_lift_right_eq_compat.
Save.
Hint Resolve @seq_lift_right_eq_compat.

Lemma mseq_lift_right_left : ∀ {O} {o:ord O} (f:nat -m> O) n,
        mseq_lift_left f n ≡ mseq_lift_right f n.
```

2.3.2 Monotonicity and functions

- (shift f x) n = f n x

```
Instance shift_mon_fun {A} `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> (A → Ob)) :
        ∀ x:A, monotonic (fun (y:Oa) ⇒ f y x).
Save.

Definition shift {A} `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> (A → Ob)) : A → Oa -m> Ob
    := fun x ⇒ (mon (fun y ⇒ f y x)).

Infix "◇" := shift (at level 30, no associativity) : O_scope.

Lemma shift_simpl : ∀ {A} `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> (A → Ob)) x y,
        (f <o> x) y = f y x.

Lemma shift_le_compat : ∀ {A} `{o1:ord Oa} `{o2:ord Ob} (f g:Oa -m> (A → Ob)),
                f ≤ g → shift f ≤ shift g.
Hint Resolve @shift_le_compat.

Add Parametric Morphism {A} `{o1:ord Oa} `{o2:ord Ob}
     : (shift (A:=A) (Oa:=Oa) (Ob:=Ob)) with signature Oeq ⟹ eq ⟹ Oeq
as shift_eq_compat.
Save.

Instance ishift_mon {A} `{o1:ord Oa} `{o2:ord Ob} (f:A → (Oa -m> Ob)) :
        monotonic (fun (y:Oa) (x:A) ⇒ f x y).
Save.

Definition ishift {A} `{o1:ord Oa} `{o2:ord Ob} (f:A → (Oa -m> Ob)) : Oa -m> (A → Ob)
    := mon (fun (y:Oa) (x:A) ⇒ f x y) (fmonotonic:=ishift_mon f).

Lemma ishift_simpl : ∀ {A} `{o1:ord Oa} `{o2:ord Ob} (f:A → (Oa -m> Ob)) x y,
        ishift f x y = f y x.

Lemma ishift_le_compat : ∀ {A} `{o1:ord Oa} `{o2:ord Ob} (f g:A → (Oa -m> Ob)),
                f ≤ g → ishift f ≤ ishift g.
Hint Resolve @ishift_le_compat.

Add Parametric Morphism {A} `{o1:ord Oa} `{o2:ord Ob}
     : (ishift (A:=A) (Oa:=Oa) (Ob:=Ob)) with signature Oeq ⟹ eq ⟹ Oeq
as ishift_eq_compat.
Save.

Instance shift_fun_mon `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> (Ob → Oc))
      {m:∀ x, monotonic (f x)} : monotonic (shift f).
Save.

Instance shift_mon2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc)
      : monotonic2 (fun x y ⇒ f y x).
```

```
Save.
Hint Resolve @shift_mon_fun @shift_fun_mon @shift_mon2.
Definition mshift `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Oa -m> Ob -m> Oc)
    : Ob -m> Oa -m> Oc := mon2 (fun x y ⇒ f y x).
```

- id c = c

```
Definition id O {o:ord O} : O → O := fun x ⇒ x.
Instance mon_id : ∀ {O:Type} {o:ord O}, monotonic (id O).
Save.
```

- (cte c) n = c

```
Definition cte A `{o1:ord Oa} (c:Oa) : A → Oa := fun x ⇒ c.
Instance mon_cte : ∀ `{o1:ord Oa} `{o2:ord Ob} (c:Ob), monotonic (cte Oa c).
Save.
Definition mseq_cte {O} {o:ord O} (c:O) : nat -m> O := mon (cte nat c).
Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} : (@cte Oa Ob _)
  with signature Ole ⟹ Ole as cte_le_compat.
Save.
Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} : (@cte Oa Ob _)
  with signature Oeq ⟹ Oeq as cte_eq_compat.
Save.
Instance mon_diag `{o1:ord Oa} `{o2:ord Ob}(f:Oa -m> (Oa -m> Ob))
    : monotonic (fun x ⇒ f x x).
Save.
Hint Resolve @mon_diag.
Definition diag `{o1:ord Oa} `{o2:ord Ob}(f:Oa -m> (Oa -m> Ob)) : Oa-m> Ob
    := mon (fun x ⇒ f x x).
Lemma fmon_diag_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} (f:Oa -m> (Oa -m> Ob)) (x:Oa),
              diag f x = f x x.
Lemma diag_le_compat : ∀ `{o1:ord Oa} `{o2:ord Ob} (f g:Oa -m> (Oa -m> Ob)),
              f ≤ g → diag f ≤ diag g.
Hint Resolve @diag_le_compat.
Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} : (diag (Oa:=Oa) (Ob:=Ob))
   with signature Oeq ⟹ Oeq as diag_eq_compat.
Save.
Lemma diag_shift : ∀ `{o1:ord Oa} `{o2:ord Ob} (f: Oa -m> Oa -m> Ob),
                    diag f ≡ diag (mshift f).
Hint Resolve @diag_shift.
Lemma mshift_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (h:Oa -m> Ob -m> Oc) (x : Ob) (y:Oa), mshift h x y = h y x.
Lemma mshift_le_compat : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (f g:Oa -m> Ob -m> Oc), f ≤ g → mshift f ≤ mshift g.
Hint Resolve @mshift_le_compat.
Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} : (@mshift Oa _ Ob _ Oc _)
     with signature Oeq ⟹ Oeq as mshift_eq_compat.
Save.
Lemma mshift2_eq : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (h : Oa -m> Ob -m> Oc),
              mshift (mshift h) ≡ h.
```

- (f@g) x = f (g x)

```
Instance monotonic_comp `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
  (f:Ob → Oc){mf : monotonic f} (g:Oa → Ob){mg:monotonic g} : monotonic (fun x ⇒ f (g x)).
Save.
Hint Resolve @monotonic_comp.

Instance monotonic_comp_mon `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
  (f:Ob -m> Oc)(g:Oa -m> Ob) : monotonic (fun x ⇒ f (g x)).
Save.
Hint Resolve @monotonic_comp_mon.

Definition comp `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f:Ob -m> Oc) (g:Oa -m> Ob)
      : Oa -m> Oc := mon (fun x ⇒ f (g x)).

Infix "@" := comp (at level 35) : O_scope.

Lemma comp_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (f:Ob -m> Oc) (g:Oa -m> Ob) (x:Oa), (f@g) x = f (g x).

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}: (@comp Oa _ Ob _ Oc _)
    with signature Ole ++> Ole ++> Ole
    as comp_le_compat.
Save.

Hint Immediate @comp_le_compat.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} : (@comp Oa _ Ob _ Oc _)
    with signature Oeq ⟹ Oeq ⟹ Oeq
    as comp_eq_compat.
Save.

Hint Immediate @comp_eq_compat.
```

- (f@2 g) h x = f (g x) (h x)

```
Instance mon_app2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}
      (f:Ob → Oc → Od) (g:Oa → Ob) (h:Oa → Oc)
      {mf:monotonic2 f}{mg:monotonic g} {mh:monotonic h}
      : monotonic (fun x ⇒ f (g x) (h x)).
Save.

Instance mon_app2_mon `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}
      (f:Ob -m> Oc -m> Od) (g:Oa -m> Ob) (h:Oa -m> Oc)
      : monotonic (fun x ⇒ f (g x) (h x)).
Save.

Definition app2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}
        (f:Ob -m> Oc -m> Od) (g:Oa -m> Ob) (h:Oa -m> Oc) : Oa -m> Od
        := mon (fun x ⇒ f (g x) (h x)).

Infix "@²" := app2 (at level 70) : O_scope.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}:
        (@app2 Oa _ Ob _ Oc _ Od _)
    with signature Ole ++> Ole ++> Ole ++> Ole
    as app2_le_compat.
Save.

Hint Immediate @app2_le_compat.

Add Parametric Morphism `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}:
        (@app2 Oa _ Ob _ Oc _ Od _)
    with signature Oeq ⟹ Oeq ⟹ Oeq ⟹ Oeq
```

```
    as app2_eq_compat.
Save.

Hint Immediate @app2_eq_compat.

Lemma app2_simpl :
    ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}
          (f:Ob -m> Oc -m> Od) (g:Oa -m> Ob) (h:Oa -m> Oc) (x:Oa),
    (f@2 g) h x = f (g x) (h x).

Lemma comp_monotonic_right :
      ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f: Ob -m> Oc) (g1 g2:Oa -m> Ob),
            g1≤ g2 → f @ g1 ≤ f @ g2.
Hint Resolve @comp_monotonic_right.

Lemma comp_monotonic_left :
      ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} (f1 f2: Ob -m> Oc) (g:Oa -m> Ob),
            f1≤ f2 → f1 @ g ≤ f2 @ g.
Hint Resolve @comp_monotonic_left.

Instance comp_monotonic2 : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc},
            monotonic2 (@comp Oa _ Ob _ Oc _).
Save.
Hint Resolve @comp_monotonic2.

Definition fcomp `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} :
   (Ob -m> Oc) -m> (Oa -m> Ob) -m> (Oa -m> Oc) := mon2 (@comp Oa _ Ob _ Oc _).

Implicit Arguments fcomp [[o1] [o2] [o3]].

Lemma fcomp_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (f:Ob -m> Oc) (g:Oa -m> Ob), fcomp _ _ _ f g = f@g.

Definition fcomp2 `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od} :
        (Oc -m> Od) -m> (Oa -m> Ob -m> Oc) -m> (Oa -m> Ob -m> Od):=
        (fcomp Oa (Ob -m> Oc) (Ob -m> Od))@(fcomp Ob Oc Od).

Implicit Arguments fcomp2 [[o1] [o2] [o3] [o4]].

Lemma fcomp2_simpl : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4:ord Od}
      (f:Oc -m> Od) (g:Oa -m> Ob -m> Oc) (x:Oa)(y:Ob), fcomp2 _ _ _ _ f g x y = f (g x y).

Lemma fmon_le_compat2 : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (f: Oa -m> Ob -m> Oc) (x y:Oa) (z t:Ob), x≤y → z ≤t → f x z ≤ f y t.
Hint Resolve fmon_le_compat2.

Lemma fmon_cte_comp : ∀ `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc}
      (c:Oc)(f:Oa -m> Ob), (mon (cte Ob c)) @ f ≡ mon (cte Oa c).
```

2.4 Abstract relational notion of lubs

```
Record islub O (o:ord O) I (f:I → O) (x:O) : Prop := mk_islub
     { le_islub : ∀ i, f i ≤ x;
       islub_le : ∀ y, (∀ i, f i ≤ y) → x ≤ y}.
Implicit Arguments islub [O o I].
Implicit Arguments le_islub [O o I f x].
Implicit Arguments islub_le [O o I f x].

Definition isglb O (o:ord O) I (f:I → O) (x:O) : Prop
     := islub (o:=lord O) f x.
Implicit Arguments isglb [O o I].

Lemma le_isglb O (o:ord O) I (f:I → O) (x:O) :
          isglb f x → ∀ i, x ≤ f i.

Lemma isglb_le O (o:ord O) I (f:I → O) (x:O) :
```

```
        isglb f x → ∀ y, (∀ i, y ≤ f i) → y ≤ x.
Implicit Arguments le_isglb [O o I f x].
Implicit Arguments isglb_le [O o I f x].

Lemma mk_isglb O (o:ord O) I (f:I → O) (x:O) :
      (∀ i, x ≤ f i) → (∀ y, (∀ i, y ≤ f i) → y ≤ x)
      → isglb f x.

Lemma islub_eq_compat O (o:ord O) I (f g:I → O) (x y:O):
      f≡g → x ≡ y → islub f x → islub g y.

Lemma isglb_eq_compat O (o:ord O) I (f g:I → O) (x y:O):
      f≡g → x ≡ y → isglb f x → isglb g y.

Add Parametric Morphism {O} {o:ord O} I : (@islub _ o I)
with signature Oeq ⟹ Oeq ⟹ iff
as islub_morphism.
Save.

Add Parametric Morphism {O} {o:ord O} I : (@isglb _ o I)
with signature Oeq ⟹ Oeq ⟹ iff
as isglb_morphism.
Save.
```

2.5 Basic operators of omega-cpos

- Constant : 0
  - lub : limit of monotonic sequences

2.5.1 Definition of cpos

```
Class cpo `{o:ord D} : Type := mk_cpo
  {D0 : D; lub: ∀ (f:nat -m> D), D;
   Dbot : ∀ x:D, D0 ≤ x;
   le_lub : ∀ (f : nat -m> D) (n:nat), f n ≤ lub f;
   lub_le : ∀ (f : nat -m> D) (x:D), (∀ n, f n ≤ x) → lub f ≤ x}.

Implicit Arguments cpo [[o]].

Notation "0" := D0 : O_scope.

Hint Resolve @Dbot @le_lub @lub_le.

Definition mon_ord_equiv : ∀ `{o:ord D1} `{o1:ord D2} {o2:ord D2},
      eq_ord o1 o2 → fmon D1 D2 (o2:=o2) → fmon D1 D2 (o2:=o1).
Defined.

Lemma mon_ord_equiv_simpl : ∀ `{o:ord D1} `{o1:ord D2} {o2:ord D2}
      (H:eq_ord o1 o2) (f:fmon D1 D2 (o2:=o2)) (x:D1),
      mon_ord_equiv H f x = f x.

Definition cpo_ord_equiv `{o1:ord D} (o2:ord D)
       : eq_ord o1 o2 → cpo (o:=o1) D → cpo (o:=o2) D.
Defined.
```

2.5.2 Least upper bounds

```
Add Parametric Morphism `{c:cpo D} : (lub (cpo:=c))
              with signature Ole ++> Ole as lub_le_compat.
Save.
```

```
Hint Resolve @lub_le_compat.
Add Parametric Morphism `{c:cpo D}: (lub (cpo:=c))
    with signature Oeq ⟹ Oeq as lub_eq_compat.
Save.
Hint Resolve @lub_eq_compat.
Notation "'mlub' f" := (lub (mon f)) (at level 60) : O_scope .
Lemma mlub_le_compat : ∀ `{c:cpo D} (f g:nat → D) {mf:monotonic f} {mg:monotonic g},
                f ≤ g → mlub f ≤ mlub g.
Hint Resolve @mlub_le_compat.
Lemma mlub_eq_compat : ∀ `{c:cpo D} (f g:nat → D) {mf:monotonic f} {mg:monotonic g},
                f ≡ g → mlub f ≡ mlub g.
Hint Resolve @mlub_eq_compat.
Lemma le_mlub : ∀ `{c:cpo D} (f:nat → D) {m:monotonic f} (n:nat), f n ≤ mlub f.
Lemma mlub_le : ∀ `{c:cpo D}(f:nat → D) {m:monotonic f}(x:D), (∀ n, f n ≤ x) → mlub f ≤ x.
Hint Resolve @le_mlub @mlub_le.
Instance lub_mon `{c:cpo D} : monotonic lub.
Save.
Definition Lub `{c:cpo D} : (nat -m> D) -m> D := mon lub.
Instance monotonic_lub_comp {O} {o:ord O} `{c:cpo D} (f:O → nat → D){mf:monotonic2 f}:
        monotonic (fun x ⇒ mlub (f x)).
Save.
Lemma lub_cte : ∀ `{c:cpo D} (d:D), mlub (cte nat d) ≡ d.
Hint Resolve @lub_cte.
Lemma mlub_lift_right : ∀ `{c:cpo D} (f:nat -m> D) n,
      lub f ≡ mlub (seq_lift_right f n).
Hint Resolve @mlub_lift_right.
Lemma mlub_lift_left : ∀ `{c:cpo D} (f:nat -m> D) n,
      lub f ≡ mlub (seq_lift_left f n).
Hint Resolve @mlub_lift_left.
Lemma lub_lift_right : ∀ `{c:cpo D} (f:nat -m> D) n,
      lub f ≡ lub (mseq_lift_right f n).
Hint Resolve @lub_lift_right.
Lemma lub_lift_left : ∀ `{c:cpo D} (f:nat -m> D) n,
      lub f ≡ lub (mseq_lift_left f n).
Hint Resolve @lub_lift_left.
Lemma lub_le_lift : ∀ `{c:cpo D} (f g:nat -m> D)
      (n:nat), (∀ k, n ≤ k → f k ≤ g k) → lub f ≤ lub g.
Lemma lub_eq_lift : ∀ `{c:cpo D} (f g:nat -m> D) {m:monotonic f} {m':monotonic g}
      (n:nat), (∀ k, n ≤ k → f k ≡ g k) → lub f ≡ lub g.
Lemma lub_seq_eq : ∀ `{c:cpo D} (f:nat → D) (g: nat-m> D) (H:f ≡ g),
      lub g ≡ lub (mon_fun_subst f g H).
```

- (lub_fun h) x = lub_n (h n x)

```
Definition lub_fun {A} `{c:cpo D} (h : nat -m> (A → D)) : A → D
                := fun x ⇒ mlub (h <o> x).
Instance lub_shift_mon {O} {o:ord O} `{c:cpo D} (h : nat -m> (O -m> D))
          : monotonic (fun (x:O) ⇒ lub (mshift h x)).
Save.
Hint Resolve @lub_shift_mon.
```

2.5.3 Functional cpos

```
Instance fcpo {A: Type} `(c:cpo D) : cpo (A → D) :=
  {D0 := fun x:A ⇒ (0:D);
   lub := fun f ⇒ lub_fun f}.
Defined.

Lemma fcpo_lub_simpl : ∀ {A} `{c:cpo D} (h:nat -m> (A → D))(x:A),
      (lub h) x = lub (h <o> x).

Lemma lub_ishift : ∀ {A} `{c:cpo D} (h:A → (nat -m> D)),
       lub (ishift h) ≡ fun x ⇒ lub (h x).
```

2.6 Cpo of monotonic functions

```
Instance fmon_cpo {O} {o:ord O} `{c:cpo D} : cpo (O -m> D) :=
    { D0 := mon (cte O (0:D));
      lub := fun h:nat -m> (O -m> D) ⇒ mon (fun (x:O) ⇒ lub (cpo:=c) (mshift h x))}.
Defined.

Lemma fmon_lub_simpl : ∀ {O} {o:ord O} `{c:cpo D}
      (h:nat -m> (O -m> D))(x:O), (lub h) x = lub (mshift h x).
Hint Resolve @fmon_lub_simpl.

Instance mon_fun_lub : ∀ {O} {o:ord O} `{c:cpo D}
         (h:nat -m> (O → D)) {mh:∀ n, monotonic (h n)}, monotonic (lub h).
Save.
```

Link between lubs on ordinary functions and monotonic functions

```
Lemma lub_mon_fcpo : ∀ {O} {o:ord O} `{c:cpo D} (h:nat -m> (O -m> D)),
      lub h ≡ mon (lub (mfun2 h)).

Lemma lub_fcpo_mon : ∀ {O} {o:ord O} `{c:cpo D} (h:nat -m> (O → D))
     {mh:∀ x, monotonic (h x)}, lub h ≡ lub (mon2 h).

Lemma double_lub_diag : ∀ `{c:cpo D} (h : nat -m> nat -m> D),
        lub (lub h) ≡ lub (diag h).
Hint Resolve @double_lub_diag.

Lemma double_lub_shift : ∀ `{c:cpo D} (h : nat -m> nat -m> D),
        lub (lub h) ≡ lub (lub (mshift h)).
Hint Resolve @double_lub_shift.
```

2.7 Continuity

```
Lemma lub_comp_le :
    ∀ `{c1:cpo D1} `{c2:cpo D2} (f:D1 -m> D2) (h : nat -m> D1),
                lub (f @ h) ≤ f (lub h).
Hint Resolve @lub_comp_le.

Lemma lub_app2_le : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
        (F:D1 -m> D2 -m> D3) (f : nat -m> D1) (g: nat -m> D2),
        lub ((F @² f) g) ≤ F (lub f) (lub g).
Hint Resolve @lub_app2_le.

Class continuous `{c1:cpo D1} `{c2:cpo D2} (f:D1 -m> D2) :=
    cont_intro : ∀ (h : nat -m> D1), f (lub h) ≤ lub (f @ h).

Typeclasses Opaque continuous.

Lemma continuous_eq_compat : ∀ `{c1:cpo D1} `{c2:cpo D2}(f g:D1 -m> D2),
```

$f \equiv g \to$ **continuous** $f \to$ **continuous** g.

```
Add Parametric Morphism `{c1:cpo D1} `{c2:cpo D2} : (@continuous D1 _ _ D2 _ _)
    with signature Oeq ==> iff
as continuous_eq_compat_iff.
Save.
```

```
Lemma lub_comp_eq :
  ∀ `{c1:cpo D1} `{c2:cpo D2} (f:D1 -m> D2) (h : nat -m> D1),
          continuous f → f (lub h) ≡ lub (f @ h).
Hint Resolve @lub_comp_eq.
```

- mon0 x == 0

```
Instance cont0 `{c1:cpo D1} `{c2:cpo D2} : continuous (mon (cte D1 (0:D2))).
Save.
Implicit Arguments cont0 [].
```

- double_app f g n m = f m (g n)

```
Definition double_app `{o1:ord Oa} `{o2:ord Ob} `{o3:ord Oc} `{o4: ord Od}
      (f:Oa -m> Oc -m> Od) (g:Ob -m> Oc)
        : Ob -m> (Oa -m> Od) := mon ((mshift f) @ g).
```

2.7.1 Continuity

```
Class continuous2 `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (F:D1 -m> D2 -m> D3) :=
continuous2_intro : ∀ (f : nat -m> D1) (g :nat -m> D2),
                    F (lub f) (lub g) ≤ lub ((F @² f) g).
```

```
Lemma continuous2_app : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
            (F : D1 -m> D2 -m> D3) {cF:continuous2 F} (k:D1), continuous (F k).
```

```
Typeclasses Opaque continuous2.
```

```
Lemma continuous2_eq_compat :
  ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f g : D1 -m> D2 -m> D3),
  f ≡ g → continuous2 f → continuous2 g.
```

```
Lemma continuous2_continuous : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
           (F : D1 -m> D2 -m> D3), continuous2 F → continuous F.
Hint Immediate @continuous2_continuous.
```

```
Lemma continuous2_left : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
            (F : D1 -m> D2 -m> D3) (h:nat -m> D1) (x:D2),
            continuous F → F (lub h) x ≤ lub (mshift (F @ h) x).
```

```
Lemma continuous2_right : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
            (F : D1 -m> D2 -m> D3) (x:D1)(h:nat -m> D2),
            continuous2 F → F x (lub h) ≤ lub (F x @ h).
```

```
Lemma continuous_continuous2 : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
      (F : D1 -m> D2 -m> D3) (cFr: ∀ k:D1, continuous (F k)) (cF: continuous F),
      continuous2 F.
```

```
Hint Resolve @continuous2_app @continuous2_continuous @continuous_continuous2.
```

```
Lemma lub_app2_eq : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
      (F : D1 -m> D2 -m> D3) {cFr:∀ k:D1, continuous (F k)} {cF : continuous F},
      ∀ (f:nat -m> D1) (g:nat -m> D2),
      F (lub f) (lub g) ≡ lub ((F@2 f) g).
```

```
Lemma lub_cont2_app2_eq : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
```

$(F : D1$ -m> $D2$ -m> $D3)\{cF :$ **continuous2** $F\}$,
$\forall\ (f$:**nat** -m> $D1)\ (g$:**nat** -m> $D2)$,
$F\ (\mathsf{lub}\ f)\ (\mathsf{lub}\ g) \equiv \mathsf{lub}\ ((F@2\ f)\ g)$.

Lemma mshift_continuous2 : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$
$(F : D1$ -m> $D2$ -m> $D3)$, **continuous2** $F \to$ **continuous2** (mshift F).
Hint Resolve @mshift_continuous2.

Lemma monotonic_sym : $\forall$ '$\{o1$:**ord** $D1\}$ '$\{o2$:**ord** $D2\}$ $(F : D1 \to D1 \to D2)$,
$(\forall\ x\ y,\ F\ x\ y \equiv F\ y\ x) \to (\forall\ k{:}D1,$ **monotonic** $(F\ k)) \to$ **monotonic** F.
Hint Immediate @monotonic_sym.

Lemma monotonic2_sym : $\forall$ '$\{o1$:**ord** $D1\}$ '$\{o2$:**ord** $D2\}$ $(F : D1 \to D1 \to D2)$,
$(\forall\ x\ y,\ F\ x\ y \equiv F\ y\ x) \to (\forall\ k{:}D1,$ **monotonic** $(F\ k)) \to$ **monotonic2** F.
Hint Immediate @monotonic2_sym.

Lemma continuous_sym : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ $(F : D1$ -m> $D1$ -m> $D2)$,
$(\forall\ x\ y,\ F\ x\ y \equiv F\ y\ x) \to (\forall\ k{:}D1,$ **continuous** $(F\ k)) \to$ **continuous** F.

Lemma continuous2_sym : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ $(F : D1$ -m>$D1$ -m>$D2)$,
$(\forall\ x\ y,\ F\ x\ y \equiv F\ y\ x) \to (\forall\ k,$ **continuous** $(F\ k)) \to$ **continuous2** F.
Hint Resolve @continuous2_sym.

- continuity is preserved by composition

Lemma continuous_comp : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$
$(f{:}D2$ -m> $D3)(g{:}D1$ -m> $D2)$, **continuous** $f \to$ **continuous** $g \to$ **continuous** (mon $(f@g)$).
Hint Resolve @continuous_comp.

Lemma continuous2_comp : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$ '$\{c4$:**cpo** $D4\}$
$(f{:}D1$ -m> $D2)(g{:}D2$ -m> $D3$ -m> $D4)$,
continuous $f \to$ **continuous2** $g \to$ **continuous2** $(g \circ f)$.
Hint Resolve @continuous2_comp.

Lemma continuous2_comp2 : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$ '$\{c4$:**cpo** $D4\}$
$(f{:}D3$ -m> $D4)(g{:}D1$ -m> $D2$ -m> $D3)$,
continuous $f \to$ **continuous2** $g \to$ **continuous2** (fcomp2 $D1\ D2\ D3\ D4\ f\ g$).
Hint Resolve @continuous2_comp2.

Lemma continuous2_app2 : $\forall$ '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$ '$\{c4$:**cpo** $D4\}$
$(F : D1$ -m> $D2$ -m> $D3)\ (f{:}D4$ -m> $D1)(g{:}D4$ -m> $D2)$, **continuous2** $F \to$
continuous $f \to$ **continuous** $g \to$ **continuous** $((F\ @^2\ f)\ g)$.
Hint Resolve @continuous2_app2.

2.8 Cpo of continuous functions

Instance lub_continuous '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ '$\{c3$:**cpo** $D3\}$
$(f$:**nat** -m> $(D1$ -m> $D2))\ \{cf{:}\forall\ n,$ **continuous** $(f\ n)\}$
: **continuous** (lub f).
Save.

Record **fcont** '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$: Type
$:=$ cont {fcontm :> $D1$ -m> $D2$; fcontinuous : **continuous** fcontm}.

Hint Resolve @fcontinuous.
Implicit Arguments **fcont** [[o][$c1$] [$o0$][$c2$]].
Implicit Arguments cont [[$D1$][o][$c1$] [$D2$][$o0$][$c2$] [$fcontinuous$]].

Infix "$\overset{c}{\leftarrow}$" := **fcont** (at *level* 30, *right associativity*) : *O_scope*.

Definition fcont_fun '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$ $(f{:}D1$ -c> $D2)$: $D1 \to D2 :=$ fun $x \Rightarrow f\ x$.

Instance fcont_ord '$\{c1$:**cpo** $D1\}$ '$\{c2$:**cpo** $D2\}$: **ord** $(D1$ -c> $D2)$

```
  := {Oeq := fun f g ⇒ ∀ x, f x ≡ g x; Ole := fun f g ⇒ ∀ x, f x ≤ g x}.
Defined.

Lemma fcont_le_intro : ∀ `{c1:cpo D1} `{c2:cpo D2} (f g : D1 -c> D2),
    (∀ x, f x ≤ g x) → f ≤ g.

Lemma fcont_le_elim : ∀ `{c1:cpo D1} `{c2:cpo D2} (f g : D1 -c> D2),
     f ≤ g → ∀ x, f x ≤ g x.

Lemma fcont_eq_intro : ∀ `{c1:cpo D1} `{c2:cpo D2} (f g : D1 -c> D2),
       (∀ x, f x ≡ g x) → f ≡ g.

Lemma fcont_eq_elim : ∀ `{c1:cpo D1} `{c2:cpo D2} (f g : D1 -c> D2),
        f ≡ g → ∀ x, f x ≡ g x.

Lemma fcont_le : ∀ `{c1:cpo D1} `{c2:cpo D2} (f : D1 -c> D2) (x y : D1),
             x ≤ y → f x ≤ f y.
Hint Resolve @fcont_le.

Lemma fcont_eq : ∀ `{c1:cpo D1} `{c2:cpo D2} (f : D1 -c> D2) (x y : D1),
             x ≡ y → f x ≡ f y.
Hint Resolve @fcont_eq.

Definition fcont0 D1 `{c1:cpo D1} D2 `{c2:cpo D2} : D1 -c> D2 := cont (mon (cte D1 (0:D2))).

Instance fcontm_monotonic : ∀ `{c1:cpo D1} `{c2:cpo D2},
          monotonic (fcontm (D1:=D1) (D2:=D2)).
Save.

Definition Fcontm D1 `{c1:cpo D1} D2 `{c2:cpo D2} : (D1 -c> D2) -m> (D1 -m> D2) :=
      mon (fcontm (D1:=D1) (D2:=D2)).

Instance fcont_lub_continuous :
     ∀ `{c1:cpo D1} `{c2:cpo D2} (f:nat -m> (D1 -c> D2)),
     continuous (lub (D:=D1 -m> D2) (Fcontm D1 D2 @ f)).
Save.

Definition fcont_lub `{c1:cpo D1} `{c2:cpo D2} : (nat -m> (D1 -c> D2)) → D1 -c> D2 :=
      fun f ⇒ cont (lub (D:=D1 -m> D2) (Fcontm D1 D2 @ f)).

Instance fcont_cpo `{c1:cpo D1} `{c2:cpo D2} : cpo (D1-c> D2) :=
       {D0:=fcont0 D1 D2; lub:=fcont_lub (D1:=D1) (D2:=D2)}.
Defined.

Definition fcont_app {O} {o:ord O} `{c1:cpo D1} `{c2:cpo D2} (f: O -m> D1 -c> D2) (x:D1) : O -m> D2
          := mshift (Fcontm D1 D2 @ f) x.

Infix "<_>" := fcont_app (at level 70) : O_scope.

Lemma fcont_app_simpl : ∀ {O} {o:ord O} `{c1:cpo D1} `{c2:cpo D2} (f: O -m> D1 -c> D2)(x:D1)(y:O),
             (f <_> x) y = f y x.

Instance ishift_continuous :
   ∀ {A:Type} `{c1:cpo D1} `{c2:cpo D2} (f: A → (D1 -c> D2)),
           continuous (ishift f).
Qed.

Definition fcont_ishift {A:Type} `{c1:cpo D1} `{c2:cpo D2} (f: A → (D1 -c> D2))
        : D1 -c> (A → D2) := cont _ (fcontinuous:=ishift_continuous f).

Instance mshift_continuous : ∀ {O} {o:ord O} `{c1:cpo D1} `{c2:cpo D2} (f: O -m> (D1 -c> D2)),
          continuous (mshift (Fcontm D1 D2 @ f)).
Save.

Definition fcont_mshift {O} {o:ord O} `{c1:cpo D1} `{c2:cpo D2} (f: O -m> (D1 -c> D2))
   : D1 -c> O -m> D2 := cont (mshift (Fcontm D1 D2 @ f)).

Lemma fcont_app_continuous :
```

```
    ∀ {O} {o:ord O} `{c1:cpo D1} `{c2:cpo D2} (f: O -m> D1 -c> D2) (h:nat -m> D1),
      f <_> (lub h) ≤ lub (D:=O -m> D2) ((fcont_mshift f) @ h).

Lemma fcont_lub_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} (h:nat -m> D1 -c> D2)(x:D1),
      lub h x = lub (h <_> x).

Instance cont_app_monotonic : ∀ `{o1:ord D1} `{c2:cpo D2} `{c3:cpo D3} (f:D1 -m> D2 -m> D3)
      (p:∀ k, continuous (f k)),
      monotonic (Ob:=D2 -c> D3) (fun (k:D1) ⇒ cont _ (fcontinuous:=p k)).
Qed.

Definition cont_app `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f:D1 -m> D2 -m> D3)
      (p:∀ k, continuous (f k)) : D1 -m> (D2 -c> D3)
  := mon (fun k ⇒ cont (f k) (fcontinuous:=p k)).

Lemma cont_app_simpl :
∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}(f:D1 -m> D2 -m> D3)(p:∀ k, continuous (f k))
    (k:D1), cont_app f p k = cont (f k).

Instance cont2_continuous `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f:D1 -m> D2 -m> D3)
      (p:continuous2 f) : continuous (cont_app f (continuous2_app f)).
Qed.

Definition cont2 `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f:D1 -m> D2 -m> D3)
      {p:continuous2 f} : D1 -c> (D2 -c> D3)
:= cont (cont_app f (continuous2_app f)).

Instance Fcontm_continuous `{c1:cpo D1} `{c2:cpo D2} : continuous (Fcontm D1 D2).
Save.
Hint Resolve @Fcontm_continuous.

Instance fcont_comp_continuous : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
    (f:D2 -c> D3) (g:D1 -c> D2), continuous (f @ g).
Save.

Definition fcont_comp `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f:D2 -c> D3) (g:D1 -c> D2)
  : D1 -c> D3 := cont (f @ g).

Infix "@_" := fcont_comp (at level 35) : O_scope.

Lemma fcont_comp_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
      (f:D2 -c> D3)(g:D1 -c> D2) (x:D1), (f @_ g) x = f (g x).

Lemma fcontm_fcont_comp_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
      (f:D2 -c> D3)(g:D1 -c> D2), fcontm (f @_ g) = f @ g.

Lemma fcont_comp_le_compat : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
     (f g : D2 -c> D3) (k l :D1 -c> D2),
     f ≤ g → k ≤ l → f @_ k ≤ g @_ l.
Hint Resolve @fcont_comp_le_compat.

Add Parametric Morphism `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
    : (@fcont_comp _ _ c1 _ _ c2 _ _ c3)
      with signature Ole ++> Ole ++> Ole as fcont_comp_le_morph.
Save.

Add Parametric Morphism `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
    : (@fcont_comp _ _ c1 _ _ c2 _ _ c3)
      with signature Oeq ⟹ Oeq ⟹ Oeq as fcont_comp_eq_compat.
Save.

Definition fcont_Comp D1 `{c1:cpo D1} D2 `{c2:cpo D2} D3 `{c3:cpo D3}
      : (D2 -c> D3) -m> (D1 -c> D2) -m> D1 -c> D3
      := mon2 _ (mf:=fcont_comp_le_compat (D1:=D1) (D2:=D2) (D3:=D3)).

Lemma fcont_Comp_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
```

```
                    (f:D2 -c> D3) (g:D1 -c> D2), fcont_Comp D1 D2 D3 f g = f @_ g.
Instance fcont_Comp_continuous2
  : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}, continuous2 (fcont_Comp D1 D2 D3).
Save.

Definition fcont_COMP D1 `{c1:cpo D1} D2 `{c2:cpo D2} D3 `{c3:cpo D3}
      : (D2 -c> D3) -c> (D1 -c> D2) -c> D1 -c> D3
      := cont2 (fcont_Comp D1 D2 D3).

Lemma fcont_COMP_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
        (f: D2 -c> D3) (g:D1 -c> D2),
        fcont_COMP D1 D2 D3 f g = f @_ g.

Definition fcont2_COMP D1 `{c1:cpo D1} D2 `{c2:cpo D2} D3 `{c3:cpo D3} D4 `{c4:cpo D4}
  : (D3 -c> D4) -c> (D1 -c> D2 -c> D3) -c> D1 -c> D2 -c> D4 :=
    (fcont_COMP D1 (D2 -c> D3) (D2 -c> D4)) @_ (fcont_COMP D2 D3 D4).

Definition fcont2_comp `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} `{c4:cpo D4}
          (f:D3 -c> D4)(F:D1 -c> D2 -c> D3) := fcont2_COMP D1 D2 D3 D4 f F.

Infix "@@_" := fcont2_comp (at level 35) : O_scope.

Lemma fcont2_comp_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} `{c4:cpo D4}
       (f:D3 -c> D4)(F:D1 -c> D2 -c> D3)(x:D1)(y:D2), (f @@_ F) x y = f (F x y).

Lemma fcont_le_compat2 : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f : D1 -c> D2 -c> D3)
    (x y : D1) (z t : D2), x ≤ y → z ≤ t → f x z ≤ f y t.
Hint Resolve @fcont_le_compat2.

Lemma fcont_eq_compat2 : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f : D1 -c> D2 -c> D3)
    (x y : D1) (z t : D2), x ≡ y → z ≡ t → f x z ≡ f y t.
Hint Resolve @fcont_eq_compat2.

Lemma fcont_continuous : ∀ `{c1:cpo D1} `{c2:cpo D2} (f:D1 -c> D2)(h:nat -m> D1),
            f (lub h) ≤ lub (f @ h).
Hint Resolve @fcont_continuous.

Instance fcont_continuous2 : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
         (f:D1 -c> D2 -c> D3), continuous2 (Fcontm D2 D3 @ f).
Save.
Hint Resolve @fcont_continuous2.

Instance cshift_continuous2 : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
         (f:D1 -c> D2 -c> D3), continuous2 (mshift (Fcontm D2 D3 @ f)).
Save.
Hint Resolve @cshift_continuous2.

Definition cshift `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3} (f:D1 -c> D2 -c> D3)
  : D2 -c> D1 -c> D3 := cont2 (mshift (Fcontm D2 D3 @ f)).

Lemma cshift_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
            (f:D1 -c> D2 -c> D3) (x:D2) (y:D1), cshift f x y = f y x.

Definition fcont_SEQ D1 `{c1:cpo D1} D2 `{c2:cpo D2} D3 `{c3:cpo D3}
  : (D1 -c> D2) -c> (D2 -c> D3) -c> D1 -c> D3 := cshift (fcont_COMP D1 D2 D3).

Lemma fcont_SEQ_simpl : ∀ `{c1:cpo D1} `{c2:cpo D2} `{c3:cpo D3}
        (f: D1 -c> D2) (g:D2 -c> D3), fcont_SEQ D1 D2 D3 f g = g @_ f.

Instance Id_mon : ∀ `{o1:ord Oa}, monotonic (fun x:Oa ⇒ x).
Save.

Definition Id Oa {o1:ord Oa} : Oa -m> Oa := mon (fun x ⇒ x).

Lemma Id_simpl : ∀ `{o1:ord Oa} (x:Oa), Id Oa x = x.
```

2.9 Fixpoints

```
Fixpoint iter_ {D} {o} '{c: @cpo D o} (f : D -m> D) n {struct n} : D
    := match n with O ⇒ 0 | S m ⇒ f (iter_ f m) end.

Lemma iter_incr : ∀ '{c: cpo D} (f : D -m> D) n, iter_ f n ≤ f (iter_ f n).
Hint Resolve @iter_incr.

Instance iter_mon : ∀ '{c: cpo D} (f : D -m> D), monotonic (iter_ f).
Save.

Definition iter '{c: cpo D} (f : D -m> D) : nat -m> D := mon (iter_ f).

Definition fixp '{c: cpo D} (f : D -m> D) : D := mlub (iter_ f).

Lemma fixp_le : ∀ '{c: cpo D} (f : D -m> D), fixp f ≤ f (fixp f).
Hint Resolve @fixp_le.

Lemma fixp_eq : ∀ '{c: cpo D} (f : D -m> D) {mf:continuous f},
      fixp f ≡ f (fixp f).

Lemma fixp_inv : ∀ '{c: cpo D} (f : D -m> D) g, f g ≤ g → fixp f ≤ g.

Definition fixp_cte : ∀ '{c:cpo D} (d:D), fixp (mon (cte D d)) ≡ d.
Save.
Hint Resolve @fixp_cte.

Lemma fixp_le_compat : ∀ '{c:cpo D} (f g : D -m> D),
      f ≤ g → fixp f ≤ fixp g.
Hint Resolve @fixp_le_compat.

Instance fixp_monotonic '{c:cpo D} : monotonic fixp.
Save.

Add Parametric Morphism '{c:cpo D} : (fixp (c:=c))
    with signature Oeq ⟹ Oeq as fixp_eq_compat.
Save.
Hint Resolve @fixp_eq_compat.

Definition Fixp D '{c:cpo D} : (D -m> D) -m> D := mon fixp.

Lemma Fixp_simpl : ∀ '{c:cpo D} (f:D-m>D), Fixp D f = fixp f.

Instance iter_monotonic '{c:cpo D} : monotonic iter.
Save.

Definition Iter D '{c:cpo D} : (D -m> D) -m> (nat -m> D) := mon iter.

Lemma IterS_simpl : ∀ '{c:cpo D} f n, Iter D f (S n) = f (Iter D f n).

Lemma IterO_simpl : ∀ '{c:cpo D} (f: D-m> D), iter f O = (0:D).

Lemma iterS_simpl : ∀ '{c:cpo D} f n, iter f (S n) = f (iter f n).

Lemma iter_continuous : ∀ '{c:cpo D} (h : nat -m> (D -m> D)),
      (∀ n, continuous (h n)) → iter (lub h) ≤ lub (mon iter @ h).

Hint Resolve @iter_continuous.

Lemma iter_continuous_eq : ∀ '{c:cpo D} (h : nat -m> (D -m> D)),
    (∀ n, continuous (h n)) → iter (lub h) ≡ lub (mon iter @ h).

Lemma fixp_continuous : ∀ '{c:cpo D} (h : nat -m> (D -m> D)),
      (∀ n, continuous (h n)) → fixp (lub h) ≤ lub (mon fixp @ h).
Hint Resolve @fixp_continuous.

Lemma fixp_continuous_eq : ∀ '{c:cpo D} (h : nat -m> (D -m> D)),
      (∀ n, continuous (h n)) → fixp (lub h) ≡ lub (mon fixp @ h).

Definition Fixp_cont D '{c:cpo D} : (D -c> D) -m> D := Fixp D @ (Fcontm D D).

Lemma Fixp_cont_simpl : ∀ '{c:cpo D} (f:D -c> D), Fixp_cont D f = fixp (fcontm f).
```

```
Instance Fixp_cont_continuous : ∀ D `{c:cpo D}, continuous (Fixp_cont D).
Save.

Definition FIXP D `{c:cpo D} : (D -c> D) -c> D := cont (Fixp_cont D).

Lemma FIXP_simpl : ∀ `{c:cpo D} (f:D -c> D), FIXP D f = Fixp D (fcontm f).

Lemma FIXP_le_compat : ∀ `{c:cpo D} (f g : D -c> D),
            f ≤ g → FIXP D f ≤ FIXP D g.
Hint Resolve @FIXP_le_compat.

Lemma FIXP_eq_compat : ∀ `{c:cpo D} (f g : D -c> D),
            f ≡ g → FIXP D f ≡ FIXP D g.
Hint Resolve @FIXP_eq_compat.

Lemma FIXP_eq : ∀ `{c:cpo D} (f:D -c> D), FIXP D f ≡ f (FIXP D f).
Hint Resolve @FIXP_eq.

Lemma FIXP_inv : ∀ `{c:cpo D} (f:D -c> D) (g : D), f g ≤ g → FIXP D f ≤ g.
```

2.9.1 Iteration of functional

```
Lemma FIXP_comp_com : ∀ `{c:cpo D} (f g:D-c>D),
        g @_ f ≤ f @_ g→ FIXP D g ≤ f (FIXP D g).

Lemma FIXP_comp : ∀ `{c:cpo D} (f g:D-c>D),
        g @_ f ≤ f @_ g → f (FIXP D g) ≤ FIXP D g → FIXP D (f @_ g) ≡ FIXP D g.

Fixpoint fcont_compn {D} {o} `{c:@cpo D o}(f:D -c> D) (n:nat) {struct n} : D -c> D :=
              match n with O ⇒ f | S p ⇒ fcont_compn f p @_ f end.

Lemma fcont_compn_Sn_simpl :
      ∀ `{c:cpo D}(f:D -c> D) (n:nat), fcont_compn f (S n) = fcont_compn f n @_ f.

Lemma fcont_compn_com : ∀ `{c:cpo D}(f:D-c>D) (n:nat),
            f @_ (fcont_compn f n) ≤ fcont_compn f n @_ f.

Lemma FIXP_compn :
      ∀ `{c:cpo D} (f:D-c>D) (n:nat), FIXP D (fcont_compn f n) ≡ FIXP D f.

Lemma fixp_double : ∀ `{c:cpo D} (f:D-c>D), FIXP D (f @_ f) ≡ FIXP D f.
```

2.9.2 Induction principle

```
Definition admissible `{c:cpo D}(P:D→Type) :=
          ∀ f : nat -m> D, (∀ n, P (f n)) → P (lub f).

Lemma fixp_ind : ∀ `{c:cpo D}(F:D -m> D)(P:D→Type),
        admissible P → P 0 → (∀ x, P x → P (F x)) → P (fixp F).

Definition admissible2 `{c1:cpo D1}`{c2:cpo D2}(R:D1 → D2 → Type) :=
    ∀ (f : nat -m> D1) (g:nat -m> D2), (∀ n, R (f n) (g n)) → R (lub f) (lub g).

Lemma fixp_ind_rel : ∀ `{c1:cpo D1}`{c2:cpo D2}(F:D1 -m> D1) (G:D2-m> D2)
        (R:D1 → D2 → Type),
        admissible2 R → R 0 0 → (∀ x y, R x y → R (F x) (G y)) → R (fixp F) (fixp G).

Ltac continuity cont Cont Hcont:=
  match goal with
 | ⊢ (Ole ?x1 (lub (mon (fun (n:nat) ⇒ cont (@?g n))))) ⇒
      let f := fresh "f" in (
           pose (f:=g); assert (monotonic f) ;
                                  [auto | (transitivity (lub (Cont@(mon f))); [rewrite ← Hcont | auto])]
           )
end.
```

```
Ltac gen_monotonic :=
match goal with ⊢ context [(@mon _ _ _ _ ?f ?mf)] ⇒ generalize (mf:monotonic f)
end.

Ltac gen_monotonic1 f :=
match goal with ⊢ context [(@mon _ _ _ _ f ?mf)] ⇒ generalize (mf:monotonic f)
end.
```

2.9.3 Function for conditionnal choice defined as a morphism

```
Definition fif {A} (b:bool) : A → A → A := fun e1 e2 ⇒ if b then e1 else e2.

Instance fif_mon2 `{o:ord A} (b:bool) : monotonic2 (@fif _ b).
Save.

Definition Fif `{o:ord A} (b:bool) : A -m> A -m> A := mon2 (@fif _ b).

Lemma Fif_simpl : ∀ `{o:ord A} (b:bool) (x y:A), Fif b x y = fif b x y.

Lemma Fif_continuous_right `{c:cpo A} (b:bool) (e:A) : continuous (Fif b e).

Lemma Fif_continuous_left `{c:cpo A} (b:bool) : continuous (Fif (A:=A) b).
Hint Resolve @Fif_continuous_right @Fif_continuous_left.

Lemma fif_continuous_left `{c:cpo A} (b:bool) (f:nat-m> A):
    fif b (lub f) ≡ lub (Fif b@f).

Lemma fif_continuous_right `{c:cpo A} (b:bool) e (f:nat-m> A):
    fif b e (lub f) ≡ lub (Fif b e@f).

Hint Resolve @fif_continuous_right @fif_continuous_left.

Instance Fif_continuous2 `{c:cpo A} (b:bool) : continuous2 (Fif (A:=A) b).
Save.

Lemma fif_continuous2 `{c:cpo A} (b:bool) (f g : nat-m> A):
      fif b (lub f) (lub g) ≡ lub ((Fif b@2 f) g).

Add Parametric Morphism `{o:ord A} (b:bool) : (@fif A b)
with signature Ole ⟹ Ole ⟹ Ole
as fif_le_compat.
Save.

Add Parametric Morphism `{o:ord A} (b:bool) : (@fif A b)
with signature Oeq ⟹ Oeq ⟹ Oeq
as fif_eq_compat.
Save.
```

3 Utheory.v: Specification of U, interval [0,1]

```
Require Export Misc.
Require Export Ccpo.
Open Local Scope O_scope.
```

3.1 Basic operators of U

- Constants : 0 and 1
- Constructor : [1/1+] n ($\equiv \frac{1}{n+1}$)
- Operations : x+y (=min (x+y,1)), x×y, [1-] x
- Relations : $x \leq y$, $x \equiv y$

Module Type UNIVERSE.
Parameter U : Type.
Declare Instance ordU: **ord** U.
Declare Instance cpoU:**cpo** U.
Delimit Scope *U_scope* with U.

Parameters *Uplus Umult Udiv*: $U \to U \to U$.
Parameter *Uinv* : $U \to U$.
Parameter *Unth* : **nat** $\to U$.

Infix "+" := *Uplus* : *U_scope*.
Infix "×" := *Umult* : *U_scope*.
Infix "/" := *Udiv* : *U_scope*.
Notation "[1-] x" := (*Uinv* x) (at *level* 35, *right associativity*) : *U_scope*.

Notation "[1/]1+ n" := (*Unth* n) (at *level* 35, *right associativity*) : *U_scope*.
Open Local Scope *U_scope*.

Definition U1 : U := [1-] 0.
Notation "1" := U1 : *U_scope*.

3.2 Basic Properties

Hypothesis *Udiff_0_1* : $0 \not\equiv 1$.

Hypothesis *Uplus_sym* : $\forall\ x\ y{:}U,\ x + y \equiv y + x$.
Hypothesis *Uplus_assoc* : $\forall\ x\ y\ z{:}U,\ x + (y + z) \equiv x + y + z$.
Hypothesis *Uplus_zero_left* : $\forall\ x{:}U,\ 0 + x \equiv x$.

Hypothesis *Umult_sym* : $\forall\ x\ y{:}U,\ x \times y \equiv y \times x$.
Hypothesis *Umult_assoc* : $\forall\ x\ y\ z{:}U,\ x \times (y \times z) \equiv x \times y \times z$.
Hypothesis *Umult_one_left* : $\forall\ x{:}U,\ 1 \times x \equiv x$.

Hypothesis *Uinv_one* : [1-] $1 \equiv 0$.

Hypothesis *Umult_div* : $\forall\ x\ y,\ y \not\equiv 0 \to x \leq y \to y \times (x/y) \equiv x$.
Hypothesis *Udiv_le_one* : $\forall\ x\ y,\ y \not\equiv 0 \to y \leq x \to (x/y) \equiv 1$.
Hypothesis *Udiv_by_zero* : $\forall\ x\ y,\ 0 \equiv y \to (x/y) \equiv 0$.

- Property : 1 - $(x + y) + x = 1$ - y holds when $x+y$ does not overflow

Hypothesis *Uinv_plus_left* : $\forall\ x\ y,\ y \leq$ [1-] $x \to$ [1-] $(x + y) + x \equiv$ [1-] y.

- Property : $(x + y) \times z = x \times z + y \times z$ holds when $x+y$ does not overflow

Hypothesis *Udistr_plus_right* : $\forall\ x\ y\ z,\ x \leq$ [1-] $y \to (x + y) \times z \equiv x \times z + y \times z$.

- Property : 1 - $(x\ y) = (1 - x) \times y + (1-y)$

Hypothesis *Udistr_inv_right* : $\forall\ x\ y{:}U,$ [1-] $(x \times y) \equiv$ ([1-] $x) \times y +$ [1-] y.

- Totality of the order

Hypothesis *Ule_class* : $\forall\ x\ y :\ U$, class $(x \leq y)$.
Hypothesis *Ule_total* : $\forall\ x\ y :\ U$, orc $(x \leq y)\ (y \leq x)$.
Implicit Arguments *Ule_total* [].

- The relation $x \leq y$ is compatible with operators

Declare Instance Uplus_mon_right :$\forall\ x$,**monotonic** (*Uplus* x).

Declare Instance Umult_mon_right : $\forall\ x$, **monotonic** (*Umult* x).
Hypothesis *Uinv_le_compat* : $\forall\ x\ y{:}U,\ x \leq y \to$ [1-] $y \leq$ [1-] x.

- Properties of simplification in case there is no overflow

Hypothesis *Uplus_le_simpl_right* : $\forall\ x\ y\ z,\ z \leq$ [1-] $x \to x$ + $z \leq y$ + $z \to x \leq y$.
Hypothesis *Umult_le_simpl_left* : $\forall\ x\ y\ z\colon U,\ /0 \equiv z \to z \times x \leq z \times y \to x \leq y$.

- Property of *Unth*: 1 / n+1 ≡ 1 - n × (1/n+1)

Hypothesis *Unth_prop* : $\forall\ n$, [1/]1+$n \equiv$ [1-](compn *Uplus* 0 (fun $k \Rightarrow$ [1/]1+n) n).

- Archimedian property

Hypothesis *archimedian* : $\forall\ x$, $0/ \equiv x \to$ exc (fun $n \Rightarrow$ [1/]1+$n \leq x$).

- Stability properties of lubs with respect to + and ×

Hypothesis *Uplus_right_continuous* : $\forall\ k$, **continuous** (mon (*Uplus* k)).
Hypothesis *Umult_right_continuous* : $\forall\ k$, **continuous** (mon (*Umult* k)).

End Universe.

Declare Module Univ:Universe.
Export *Univ*.

Hint Resolve *Udiff_0_1 Unth_prop*.
Hint Resolve *Uplus_sym Uplus_assoc Umult_sym Umult_assoc*.
Hint Resolve *Uinv_one Uinv_plus_left Umult_div Udiv_le_one Udiv_by_zero*.
Hint Resolve *Uplus_zero_left Umult_one_left Udistr_plus_right Udistr_inv_right*.
Hint Resolve *Uplus_mon_right Umult_mon_right Uinv_le_compat*.
Hint Resolve lub_le le_lub *Uplus_right_continuous Umult_right_continuous*.
Hint Resolve *Ule_total Ule_class*.

4 Uprop.v : Properties of operators on [0,1]

Require Export Utheory.
Require Export Arith.
Require Export Omega.

Open Local Scope *U_scope*.

4.1 Direct consequences of axioms

Lemma Uplus_le_compat_right : $\forall\ x\ y\ z\colon U,\ y \leq z \to x$ + $y \leq x$ + z.
Hint Resolve Uplus_le_compat_right.

Instance Uplus_mon2 : **monotonic2** *Uplus*.
Save.
Hint Resolve Uplus_mon2.

Lemma Uplus_le_compat_left : $\forall\ x\ y\ z\colon U,\ x \leq y \to x$ + $z \leq y$ + z.
Hint Resolve Uplus_le_compat_left.

Lemma Uplus_le_compat : $\forall\ x\ y\ z\ t,\ x \leq y \to z \leq t \to x$ + $z \leq y$ + t.
Hint Immediate Uplus_le_compat.

Lemma Uplus_eq_compat_left : $\forall\ x\ y\ z\colon U,\ x \equiv y \to x$ + $z \equiv y$ + z.
Hint Resolve Uplus_eq_compat_left.

Lemma Uplus_eq_compat_right : $\forall\ x\ y\ z\colon U,\ x \equiv y \to (z$ + $x) \equiv (z$ + $y)$.

Hint Resolve Uplus_eq_compat_left Uplus_eq_compat_right.

Add *Morphism Uplus* with *signature* Oeq ⟹ Oeq ⟹ Oeq as *Uplus_eq_compat*.

```
Qed.
Hint Immediate Uplus_eq_compat.

Add Morphism Uinv with signature Oeq ⟹ Oeq as Uinv_eq_compat.
Qed.
Hint Resolve Uinv_eq_compat.

Lemma Uplus_zero_right : ∀ x:U, x + 0 ≡ x.
Hint Resolve Uplus_zero_right.

Lemma Uinv_opp_left : ∀ x, [1-] x + x ≡ 1.
Hint Resolve Uinv_opp_left.

Lemma Uinv_opp_right : ∀ x, x + [1-] x ≡ 1.
Hint Resolve Uinv_opp_right.

Lemma Uinv_inv : ∀ x : U, [1-] [1-] x ≡ x.
Hint Resolve Uinv_inv.

Lemma Unit : ∀ x:U, x ≤ 1.
Hint Resolve Unit.

Lemma Uinv_zero : [1-] 0 = 1.

Lemma Ueq_class : ∀ x y:U, class (x≡y).

Lemma Ueq_double_neg : ∀ x y : U, //(x ≡ y) → x ≡ y.
Hint Resolve Ueq_class.
Hint Immediate Ueq_double_neg.

Lemma Ule_orc : ∀ x y : U, orc (x≤y) (/x≤y).
Implicit Arguments Ule_orc [].

Lemma Ueq_orc : ∀ x y:U, orc (x≡y) (/x≡y).
Implicit Arguments Ueq_orc [].

Lemma Upos : ∀ x:U, 0 ≤ x.

Lemma Ule_0_1 : 0 ≤ 1.

Hint Resolve Upos Ule_0_1.
```

4.2 Properties of ≡ derived from properties of ≤

```
Definition UPlus : U -m> U -m> U := mon2 Uplus.

Definition UPlus_simpl : ∀ x y, UPlus x y = x+y.
Save.

Instance Uplus_continuous2 : continuous2 (mon2 Uplus).
Save.

Hint Resolve Uplus_continuous2.

Lemma Uplus_lub_eq : ∀ f g : nat -m> U,
      lub f + lub g ≡ lub ((UPlus @² f) g).

Lemma Umult_le_compat_right : ∀ x y z:U, y ≤ z → x × y ≤ x × z.
Hint Resolve Umult_le_compat_right.

Instance Umult_mon2 : monotonic2 Umult.
Save.

Lemma Umult_le_compat_left : ∀ x y z:U, x ≤ y → x × z ≤ y × z.
Hint Resolve Umult_le_compat_left.

Lemma Umult_le_compat : ∀ x y z t, x ≤ y → z ≤ t → x × z ≤ y × t.
Hint Immediate Umult_le_compat.

Definition UMult : U -m> U -m> U := mon2 Umult.
```

```
Lemma Umult_eq_compat_left : ∀ x y z:U, x ≡ y → (x × z) ≡ (y × z).
Hint Resolve Umult_eq_compat_left.
Lemma Umult_eq_compat_right : ∀ x y z:U, x ≡ y → (z × x) ≡ (z × y).
Hint Resolve Umult_eq_compat_left Umult_eq_compat_right.
Definition UMult_simpl : ∀ x y, UMult x y = x×y.
Save.
Instance Umult_continuous2 : continuous2 (mon2 Umult).
Save.
Hint Resolve Umult_continuous2.
Lemma Umult_lub_eq : ∀ f g : nat -m> U,
      lub f × lub g ≡ lub ((UMult @² f) g).
```

4.3 U is a setoid

```
Add Morphism Umult with signature Oeq ⟹ Oeq ⟹ Oeq
  as Umult_eq_compat.
Qed.
Hint Immediate Umult_eq_compat.
Instance Uinv_mon : monotonic (o1:=lord U) Uinv.
Save.
Definition UInv : U -m> U := mon (o1:=lord U) Uinv.
Definition UInv_simpl : ∀ x, UInv x = [1-]x.
Save.
Lemma Ule_eq_compat :
∀ x1 x2 : U, x1 ≡ x2 → ∀ x3 x4 : U, x3 ≡ x4 → x1 ≤ x3 → x2 ≤ x4.
```

4.4 Definition and properties of $x < y$

```
Definition Ult (r1 r2:U) : Prop := ⌜(r2 ≤ r1)⌝.
Infix "<" := Ult : U_scope.
Hint Unfold Ult.
Add Morphism Ult with signature Oeq ⟹ Oeq ⟹ iff as Ult_eq_compat_iff.
Save.
Lemma Ult_eq_compat :
∀ x1 x2 : U, x1 ≡ x2 → ∀ x3 x4 : U, x3 ≡ x4 → x1 < x3 → x2 < x4.
Lemma Ult_class : ∀ x y, class (x<y).
Hint Resolve Ult_class.
```

4.4.1 Properties of $x \leq y$

```
Lemma Ule_zero_eq : ∀ x:U, x ≤ 0 → x ≡ 0.
Lemma Uge_one_eq : ∀ x:U, 1 ≤ x → x ≡ 1.
Hint Immediate Ule_zero_eq Uge_one_eq.
```

4.4.2 Properties of $x < y$

Lemma Ult_neq : $\forall\ x\ y{:}U,\ x < y \to x \not\equiv y$.

Lemma Ult_neq_rev : $\forall\ x\ y{:}U,\ x < y \to y \not\equiv x$.

Lemma Ult_trans : $\forall\ x\ y\ z,\ x < y \to y < z \to x < z$.

Lemma Ult_le : $\forall\ x\ y,\ x < y \to x \leq y$.

Lemma Ule_diff_lt : $\forall\ x\ y\ :\ U,\ x \leq y \to x \not\equiv y \to x < y$.

Hint Immediate Ult_neq Ult_neq_rev Ult_le.
Hint Resolve Ule_diff_lt.

Lemma Ult_neq_zero : $\forall\ x,\ 0 \not\equiv x \to 0 < x$.

Hint Resolve *Ule_total* Ult_neq_zero.

4.5 Properties of $+$ and $\times$

Lemma Udistr_plus_left : $\forall\ x\ y\ z,\ y \leq [1\text{-}]\ z \to x \times (y + z) \equiv x \times y + x \times z$.

Lemma Udistr_inv_left : $\forall\ x\ y,\ [1\text{-}](x \times y) \equiv (x \times ([1\text{-}]\ y)) + [1\text{-}]\ x$.

Hint Resolve Uinv_eq_compat Udistr_plus_left Udistr_inv_left.

Lemma Uplus_perm2 : $\forall\ x\ y\ z{:}U,\ x + (y + z) \equiv y + (x + z)$.

Lemma Umult_perm2 : $\forall\ x\ y\ z{:}U,\ x \times (y \times z) \equiv y \times (x \times z)$.

Lemma Uplus_perm3 : $\forall\ x\ y\ z\ :\ U,\ (x + (y + z)) \equiv z + (x + y)$.

Lemma Umult_perm3 : $\forall\ x\ y\ z\ :\ U,\ (x \times (y \times z)) \equiv z \times (x \times y)$.

Hint Resolve Uplus_perm2 Umult_perm2 Uplus_perm3 Umult_perm3.

Lemma Uinv_simpl : $\forall\ x\ y\ :\ U,\ [1\text{-}]\ x \equiv [1\text{-}]\ y \to x \equiv y$.

Hint Immediate Uinv_simpl.

Lemma Umult_decomp : $\forall\ x\ y,\ x \equiv x \times y + x \times [1\text{-}]y$.
Hint Resolve Umult_decomp.

4.6 More properties on $+$ and $\times$ and *Uinv*

Lemma Umult_one_right : $\forall\ x{:}U,\ x \times 1 \equiv x$.
Hint Resolve Umult_one_right.

Lemma Umult_one_right_eq : $\forall\ x\ y{:}U,\ y \equiv 1 \to x \times y \equiv x$.
Hint Resolve Umult_one_right_eq.

Lemma Umult_one_left_eq : $\forall\ x\ y{:}U,\ x \equiv 1 \to x \times y \equiv y$.
Hint Resolve Umult_one_left_eq.

Lemma Udistr_plus_left_le : $\forall\ x\ y\ z\ :\ U,\ x \times (y + z) \leq x \times y + x \times z$.

Lemma Uplus_eq_simpl_right :
$\forall\ x\ y\ z{:}U,\ z \leq [1\text{-}]\ x \to z \leq [1\text{-}]\ y \to (x + z) \equiv (y + z) \to x \equiv y$.

Lemma Ule_plus_right : $\forall\ x\ y,\ x \leq x + y$.

Lemma Ule_plus_left : $\forall\ x\ y,\ y \leq x + y$.
Hint Resolve Ule_plus_right Ule_plus_left.

Lemma Ule_mult_right : $\forall\ x\ y,\ x \times y \leq x$.

Lemma Ule_mult_left : $\forall\ x\ y,\ x \times y \leq y$.
Hint Resolve Ule_mult_right Ule_mult_left.

Lemma Uinv_le_perm_right : $\forall\ x\ y{:}U,\ x \leq [1\text{-}]\ y \to y \leq [1\text{-}]\ x$.

Hint Immediate Uinv_le_perm_right.

Lemma Uinv_le_perm_left : $\forall\ x\ y{:}U,\ [1\text{-}]\ x \le y \to [1\text{-}]\ y \le x$.
Hint Immediate Uinv_le_perm_left.

Lemma Uinv_le_simpl : $\forall\ x\ y{:}U,\ [1\text{-}]\ x \le [1\text{-}]\ y \to y \le x$.
Hint Immediate Uinv_le_simpl.

Lemma Uinv_double_le_simpl_right : $\forall\ x\ y,\ x \le y \to x \le [1\text{-}][1\text{-}]y$.
Hint Resolve Uinv_double_le_simpl_right.

Lemma Uinv_double_le_simpl_left : $\forall\ x\ y,\ x \le y \to [1\text{-}][1\text{-}]x \le y$.
Hint Resolve Uinv_double_le_simpl_left.

Lemma Uinv_eq_perm_left : $\forall\ x\ y{:}U,\ x \equiv [1\text{-}]\ y \to [1\text{-}]\ x \equiv y$.
Hint Immediate Uinv_eq_perm_left.

Lemma Uinv_eq_perm_right : $\forall\ x\ y{:}U,\ [1\text{-}]\ x \equiv y \to x \equiv [1\text{-}]\ y$.

Hint Immediate Uinv_eq_perm_right.

Lemma Uinv_eq_simpl : $\forall\ x\ y{:}U,\ [1\text{-}]\ x \equiv [1\text{-}]\ y \to x \equiv y$.
Hint Immediate Uinv_eq_simpl.

Lemma Uinv_double_eq_simpl_right : $\forall\ x\ y,\ x \equiv y \to x \equiv [1\text{-}][1\text{-}]y$.
Hint Resolve Uinv_double_eq_simpl_right.

Lemma Uinv_double_eq_simpl_left : $\forall\ x\ y,\ x \equiv y \to [1\text{-}][1\text{-}]x \equiv y$.
Hint Resolve Uinv_double_eq_simpl_left.

Lemma Uinv_plus_right : $\forall\ x\ y,\ y \le [1\text{-}]\ x \to [1\text{-}]\ (x + y) + y \equiv [1\text{-}]\ x$.
Hint Resolve Uinv_plus_right.

Lemma Uplus_eq_simpl_left :
$\forall\ x\ y\ z{:}U,\ x \le [1\text{-}]\ y \to x \le [1\text{-}]\ z \to (x + y) \equiv (x + z) \to y \equiv z$.

Lemma Uplus_eq_zero_left : $\forall\ x\ y{:}U,\ x \le [1\text{-}]\ y \to (x + y) \equiv y \to x \equiv 0$.

Lemma Uinv_le_trans : $\forall\ x\ y\ z\ t,\ x \le [1\text{-}]\ y \to z \le x \to t \le y \to z \le [1\text{-}]\ t$.

Lemma Uinv_plus_left_le : $\forall\ x\ y,\ [1\text{-}]y \le [1\text{-}](x{+}y)\ {+}x$.

Lemma Uinv_plus_right_le : $\forall\ x\ y,\ [1\text{-}]x \le [1\text{-}](x{+}y)\ {+}y$.

Hint Resolve Uinv_plus_left_le Uinv_plus_right_le.

4.7 Disequality

Lemma neq_sym : $\forall\ x\ y{:}U,\ x \not\equiv y \to y \not\equiv x$.
Hint Immediate neq_sym.

Lemma Uinv_neq_compat : $\forall\ x\ y,\ x \not\equiv y \to /[1\text{-}]\ x \equiv [1\text{-}]\ y$.

Lemma Uinv_neq_simpl : $\forall\ x\ y,\ /[1\text{-}]\ x \equiv [1\text{-}]\ y \to x \not\equiv y$.

Hint Resolve Uinv_neq_compat.
Hint Immediate Uinv_neq_simpl.

Lemma Uinv_neq_left : $\forall\ x\ y,\ x \not\equiv [1\text{-}]\ y \to /[1\text{-}]\ x \equiv y$.

Lemma Uinv_neq_right : $\forall\ x\ y,\ /[1\text{-}]\ x \equiv y \to x \not\equiv [1\text{-}]\ y$.

4.7.1 Properties of $<$

Lemma Ult_antirefl : $\forall\ x{:}U,\ x \not< x$.

Lemma Ult_0_1 : $(0 < 1)$.

Lemma Ule_lt_trans : $\forall\ x\ y\ z{:}U,\ x \le y \to y < z \to x < z$.

Lemma Ult_le_trans : $\forall\ x\ y\ z{:}U,\ x < y \to y \le z \to x < z$.

```
Hint Resolve Ult_0_1 Ult_antirefl.
Lemma Ule_neq_zero : ∀ (x y:U), 0̸ ≡ x → x ≤ y → 0̸ ≡ y.
Lemma Uplus_neq_zero_left : ∀ x y, 0̸ ≡ x → 0̸ ≡ x+y.
Lemma Uplus_neq_zero_right : ∀ x y, 0̸ ≡ y → 0̸ ≡ x+y.
Lemma not_Ult_le : ∀ x y, x̸ < y → y ≤ x.
Lemma Ule_not_lt : ∀ x y, x ≤ y → y̸ < x.
Hint Immediate not_Ult_le Ule_not_lt.
Theorem Uplus_le_simpl_left : ∀ x y z : U, z ≤ [1-] x → z + x ≤ z + y → x ≤ y.
Lemma Uplus_lt_compat_left : ∀ x y z:U, z ≤ [1-] y → x < y → (x + z) < (y + z).
Lemma Uplus_lt_compat_right : ∀ x y z:U, z ≤ [1-] y → x < y → (z + x) < (z + y).
Hint Resolve Uplus_lt_compat_right Uplus_lt_compat_left.
Lemma Uplus_lt_compat :
∀ x y z t:U, z ≤ [1-] x → t ≤ [1-] y → x < y → z < t → (x + z) < (y + t).
Hint Immediate Uplus_lt_compat.
Lemma Uplus_lt_simpl_left : ∀ x y z:U, z ≤ [1-] y → (z + x) < (z + y) → x < y.
Lemma Uplus_lt_simpl_right : ∀ x y z:U, z ≤ [1-] y → (x + z) < (y + z) → x < y.
Lemma Uplus_one_le : ∀ x y, x + y ≡ 1 → [1-] y ≤ x.
Hint Immediate Uplus_one_le.
Theorem Uplus_eq_zero : ∀ x, x ≤ [1-] x → (x + x) ≡ x → x ≡ 0.
Lemma Umult_zero_left : ∀ x, 0 × x ≡ 0.
Hint Resolve Umult_zero_left.
Lemma Umult_zero_right : ∀ x, (x × 0) ≡ 0.
Hint Resolve Uplus_eq_zero Umult_zero_right.
Lemma Umult_zero_left_eq : ∀ x y, x ≡ 0 → x × y ≡ 0.
Lemma Umult_zero_right_eq : ∀ x y, y ≡ 0 → x × y ≡ 0.
Lemma Umult_zero_eq : ∀ x y z, x ≡ 0 → x × y ≡ x × z.
```

4.7.2 Compatibility of operations with respect to order.

```
Lemma Umult_le_simpl_right : ∀ x y z, 0̸ ≡ z → (x × z) ≤ (y × z) → x ≤ y.
Hint Resolve Umult_le_simpl_right.
Lemma Umult_simpl_right : ∀ x y z, 0̸ ≡ z → (x × z) ≡ (y × z) → x ≡ y.
Lemma Umult_simpl_left : ∀ x y z, 0̸ ≡ x → (x × y) ≡ (x × z) → y ≡ z.
Lemma Umult_lt_compat_left : ∀ x y z, 0̸ ≡ z→ x < y → (x × z) < (y × z).
Lemma Umult_lt_compat_right : ∀ x y z, 0̸ ≡ z → x < y → (z × x) < (z × y).
Lemma Umult_lt_simpl_right : ∀ x y z, 0̸ ≡ z → (x × z) < (y × z) → x < y.
Lemma Umult_lt_simpl_left : ∀ x y z, 0̸ ≡ z → (z × x) < (z × y) → x < y.
Hint Resolve Umult_lt_compat_left Umult_lt_compat_right.
Lemma Umult_zero_simpl_right : ∀ x y, 0 ≡ x×y → 0̸ ≡ x → 0 ≡ y.
Lemma Umult_zero_simpl_left : ∀ x y, 0 ≡ x×y → 0̸ ≡ y → 0 ≡ x.
Lemma Umult_neq_zero : ∀ x y, 0̸ ≡ x → 0̸ ≡ y → 0̸ ≡ x×y.
Hint Resolve Umult_neq_zero.
Lemma Umult_lt_zero : ∀ x y, 0 < x → 0 < y → 0 < x×y.
Hint Resolve Umult_lt_zero.
Lemma Umult_lt_compat : ∀ x y z t, x < y → z < t → x × z < y × t.
```

4.7.3 More Properties

Lemma Uplus_one : $\forall\ x\ y$, [1-] $x \le y \to x + y \equiv 1$.
Hint Resolve Uplus_one.

Lemma Uplus_one_right : $\forall\ x$, $x + 1 \equiv 1$.

Lemma Uplus_one_left : $\forall\ x{:}U$, $1 + x \equiv 1$.
Hint Resolve Uplus_one_right Uplus_one_left.

Lemma Uinv_mult_simpl : $\forall\ x\ y\ z\ t$, $x \le$ [1-] $y \to (x \times z) \le$ [1-] $(y \times t)$.
Hint Resolve Uinv_mult_simpl.

Lemma Umult_inv_plus : $\forall\ x\ y$, $x \times$ [1-] $y + y \equiv x + y \times$ [1-] x.
Hint Resolve Umult_inv_plus.

Lemma Umult_inv_plus_le : $\forall\ x\ y\ z$, $y \le z \to x \times$ [1-] $y + y \le x \times$ [1-] $z + z$.
Hint Resolve Umult_inv_plus_le.

Lemma Uplus_lt_Uinv : $\forall\ x\ y$, $x+y < 1 \to x \le$ [1-] y.

Lemma Uinv_lt_perm_left: $\forall\ x\ y : U$, [1-] $x < y \to$ [1-] $y < x$.

Lemma Uinv_lt_perm_right: $\forall\ x\ y : U$, $x <$ [1-] $y \to y <$ [1-] x.

Hint Immediate Uinv_lt_perm_left Uinv_lt_perm_right.

Lemma Uinv_lt_one : $\forall\ x$, $0 < x \to$ [1-]$x < 1$.

Lemma Uinv_lt_zero : $\forall\ x$, $x < 1 \to 0 <$ [1-]x.

Hint Resolve Uinv_lt_one Uinv_lt_zero.

Lemma orc_inv_plus_one : $\forall\ x\ y$, orc $(x \le$[1-]$y)$ $(x+y \equiv 1)$.

Lemma Umult_lt_right : $\forall\ p\ q$, $p < 1 \to 0 < q \to p \times q < q$.

Lemma Umult_lt_left : $\forall\ p\ q$, $0 < p \to q < 1 \to p \times q < p$.

Hint Resolve Umult_lt_right Umult_lt_left.

4.8 Definition of x ^ n

Fixpoint Uexp $(x{:}U)$ $(n{:}$**nat**$)$ {struct n} : $U :=$
  match n with $0 \Rightarrow 1 \mid (S\ p) \Rightarrow x \times$ Uexp $x\ p$ end.

Infix "^" := Uexp : *U_scope*.

Lemma Uexp_1 : $\forall\ x$, $x\hat{}1 \equiv x$.

Lemma Uexp_0 : $\forall\ x$, $x\hat{}0 \equiv 1$.

Lemma Uexp_zero : $\forall\ n$, $(0<n)\%nat \to 0\hat{}n \equiv 0$.

Lemma Uexp_one : $\forall\ n$, $1\hat{}n \equiv 1$.

Lemma Uexp_le_compat_right :
  $\forall\ x\ n\ m$, $(n \le m)\%nat \to x\hat{}m \le x\hat{}n$.

Lemma Uexp_le_compat_left : $\forall\ x\ y\ n$, $x \le y \to x\hat{}n \le y\hat{}n$.
Hint Resolve Uexp_le_compat_left Uexp_le_compat_right.

Lemma Uexp_le_compat : $\forall\ x\ y\ (n\ m{:}$**nat**$)$,
  $x \le y \to n \le m \to x\hat{}m \le y\hat{}n$.

Instance Uexp_mon2 : **monotonic2** $(o1{:=}$lord $U)$ $(o3{:=}$lord $U)$ Uexp.
Save.

Definition UExp : U -m> (**nat** -m$\to$ U) := mon2 Uexp.

Add *Morphism* Uexp with *signature* Oeq $\Longrightarrow$ **eq** $\Longrightarrow$ Oeq as *Uexp_eq_compat*.
Save.

Lemma Uexp_inv_S : $\forall\ x\ n$, ([1-]$x\hat{}(S\ n)) \equiv x \times$ ([1-]$x\hat{}n)+$[1-]x.

Lemma Uexp_lt_compat : $\forall\ p\ q\ n$, $(0<n)\%nat \to p<q \to (p\hat{}n<q\hat{}n)$.

Hint Resolve Uexp_lt_compat.

Lemma Uexp_lt_zero : $\forall\ p\ n$, $(0<p) \to (0<p\hat{}n)$.
Hint Resolve Uexp_lt_zero.

Lemma Uexp_lt_one : $\forall\ p\ n$, $(0<n)\%nat \to p<1 \to (p\hat{}n<1)$.
Hint Resolve Uexp_lt_one.

Lemma Uexp_lt_antimon: $\forall\ p\ n\ m$,
$(n<m)\%nat \to 0 < p \to p < 1 \to p\hat{}m < p\hat{}n$.
Hint Resolve Uexp_lt_antimon.

4.9 Properties of division

Lemma Udiv_mult : $\forall\ x\ y$, $\not 0 \equiv y \to x \leq y \to (x/y) \times y \equiv x$.
Hint Resolve Udiv_mult.

Lemma Umult_div_le : $\forall\ x\ y$, $y \times (x\ /\ y) \leq x$.
Hint Resolve Umult_div_le.

Lemma Udiv_mult_le : $\forall\ x\ y$, $(x/y) \times y \leq x$.
Hint Resolve Udiv_mult_le.

Lemma Udiv_le_compat_left : $\forall\ x\ y\ z$, $x \leq y \to x/z \leq y/z$.
Hint Resolve Udiv_le_compat_left.

Lemma Udiv_eq_compat_left : $\forall\ x\ y\ z$, $x \equiv y \to x/z \equiv y/z$.
Hint Resolve Udiv_eq_compat_left.

Lemma Umult_div_le_left : $\forall\ x\ y\ z$, $\not 0 \equiv y \to x \times y \leq z \to x \leq z/y$.

Lemma Udiv_le_compat_right : $\forall\ x\ y\ z$, $\not 0 \equiv y \to y \leq z \to x/z \leq x/y$.
Hint Resolve Udiv_le_compat_right.

Lemma Udiv_eq_compat_right : $\forall\ x\ y\ z$, $y \equiv z \to x/z \equiv x/y$.
Hint Resolve Udiv_eq_compat_right.

Add *Morphism Udiv* with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *Udiv_eq_compat*.
Save.

Add *Morphism Udiv* with *signature* Ole ++> Oeq $\Longrightarrow$ Ole as *Udiv_le_compat*.
Save.

Lemma Umult_div_eq : $\forall\ x\ y\ z$, $\not 0 \equiv y \to x \times y \equiv z \to x \equiv z/y$.

Lemma Umult_div_le_right : $\forall\ x\ y\ z$, $x \leq y \times z \to x/z \leq y$.

Lemma Udiv_le : $\forall\ x\ y$, $\not 0 \equiv y \to x \leq x/y$.

Lemma Udiv_zero : $\forall\ x$, $0/x \equiv 0$.
Hint Resolve Udiv_zero.

Lemma Udiv_zero_eq : $\forall\ x\ y$, $0 \equiv x \to x/y \equiv 0$.
Hint Resolve Udiv_zero_eq.

Lemma Udiv_one : $\forall\ x$, $\not 0 \equiv x \to x/1 \equiv x$.
Hint Resolve Udiv_one.

Lemma Udiv_refl : $\forall\ x$, $\not 0 \equiv x \to x/x \equiv 1$.
Hint Resolve Udiv_refl.

Lemma Umult_div_assoc : $\forall\ x\ y\ z$, $y \leq z \to (x \times y)\ /\ z \equiv x \times (y/z)$.

Lemma Udiv_mult_assoc : $\forall\ x\ y\ z$, $x \leq y \times z \to x/(y \times z) \equiv (x/y)/z$.

Lemma Udiv_inv : $\forall\ x\ y$, $\not 0 \equiv y \to [1\text{-}](x/y) \leq ([1\text{-}]x)/y$.

Lemma Uplus_div_inv : $\forall\ x\ y\ z$, $x+y \leq z \to x \leq [1\text{-}]y \to x/z \leq [1\text{-}](y/z)$.
Hint Resolve Uplus_div_inv.

Lemma Udiv_plus_le : $\forall\ x\ y\ z,\ x/z + y/z \leq (x+y)/z$.
Hint Resolve Udiv_plus_le.

Lemma Udiv_plus : $\forall\ x\ y\ z,\ (x+y)/z \equiv x/z + y/z$.
Hint Resolve Udiv_plus.

Instance Udiv_mon : $\forall\ k$, **monotonic** (fun $x \Rightarrow (x/k)$).
Save.

Definition UDiv $(k{:}U) : U$ -m> $U :=$ mon (fun $x \Rightarrow (x/k)$).

Lemma UDiv_simpl : $\forall\ (k{:}U)\ x$, UDiv $k\ x = x/k$.

4.10 Definition and properties of x & y

A conjonction operation which coincides with min and mult on 0 and 1, see Morgan & McIver

Definition Uesp $(x\ y{:}U) :=$ [1-] ([1-] x + [1-] y).

Infix "&" := Uesp (*left associativity*, at *level* 40) : *U_scope*.

Lemma Uinv_plus_esp : $\forall\ x\ y$, [1-] $(x + y) \equiv$ [1-] x & [1-] y.
Hint Resolve Uinv_plus_esp.

Lemma Uinv_esp_plus : $\forall\ x\ y$, [1-] $(x$ & $y) \equiv$ [1-] x + [1-] y.
Hint Resolve Uinv_esp_plus.

Lemma Uesp_sym : $\forall\ x\ y : U,\ x$ & $y \equiv y$ & x.

Lemma Uesp_one_right : $\forall\ x : U,\ x$ & $1 \equiv x$.

Lemma Uesp_one_left : $\forall\ x : U,\ 1$ & $x \equiv x$.

Lemma Uesp_zero : $\forall\ x\ y,\ x \leq$ [1-] $y \to x$ & $y \equiv 0$.

Hint Resolve Uesp_sym Uesp_one_right Uesp_one_left Uesp_zero.

Lemma Uesp_zero_right : $\forall\ x : U,\ x$ & $0 \equiv 0$.

Lemma Uesp_zero_left : $\forall\ x : U,\ 0$ & $x \equiv 0$.

Hint Resolve Uesp_zero_right Uesp_zero_left.

Add *Morphism* Uesp with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *Uesp_eq_compat*.
Save.

Lemma Uesp_le_compat : $\forall\ x\ y\ z\ t,\ x \leq y \to z \leq t \to x\&z \leq y\&t$.

Hint Immediate Uesp_le_compat Uesp_eq_compat.

Lemma Uesp_zero_one_mult_left : $\forall\ x\ y$, orc $(x \equiv 0)\ (x \equiv 1) \to x$ & $y \equiv x \times y$.

Lemma Uesp_zero_one_mult_right : $\forall\ x\ y$, orc $(y \equiv 0)\ (y \equiv 1) \to x$ & $y \equiv x \times y$.

Hint Resolve Uesp_zero_one_mult_left Uesp_zero_one_mult_right.

Instance Uesp_mon : **monotonic2** Uesp.
Save.

Definition UEsp : U -m> U -m> $U :=$ mon2 Uesp.

Lemma UEsp_simpl : $\forall\ x\ y$, UEsp $x\ y = x$ & y.

Lemma Uesp_le_left : $\forall\ x\ y,\ x$ & $y \leq x$.

Lemma Uesp_le_right : $\forall\ x\ y,\ x$ & $y \leq y$.

Hint Resolve Uesp_le_left Uesp_le_right.

Lemma Uesp_plus_inv : $\forall\ x\ y$, [1-] $y \leq x \to x \equiv x$ & y + [1-] y.
Hint Resolve Uesp_plus_inv.

Lemma Uesp_le_plus_inv : $\forall\ x\ y,\ x \leq x$ & y + [1-] y.
Hint Resolve Uesp_le_plus_inv.

Lemma Uplus_inv_le_esp : $\forall\ x\ y\ z,\ x \leq y$ + ([1-] $z) \to x$ & $z \leq y$.
Hint Immediate Uplus_inv_le_esp.

4.11 Definition and properties of x - y

Definition Uminus (x y:U) := [1-] ([1-] x + y).

Infix "-" := Uminus : *U_scope*.

Lemma Uminus_le_compat_left : $\forall$ x y z, $x \leq y \to x$ - $z \leq y$ - z.

Lemma Uminus_le_compat_right : $\forall$ x y z, $y \leq z \to x$ - $z \leq x$ - y.

Hint Resolve Uminus_le_compat_left Uminus_le_compat_right.

Lemma Uminus_le_compat : $\forall$ x y z t, $x \leq y \to t \leq z \to x$ - $z \leq y$ - t.

Hint Immediate Uminus_le_compat.

Add *Morphism* Uminus with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *Uminus_eq_compat*.
Save.
Hint Immediate Uminus_eq_compat.

Lemma Uminus_zero_right : $\forall$ x, x - $0 \equiv x$.

Lemma Uminus_one_left : $\forall$ x, 1 - $x \equiv$ [1-] x.

Lemma Uminus_le_zero : $\forall$ x y, $x \leq y \to x$ - $y \equiv 0$.

Hint Resolve Uminus_zero_right Uminus_one_left Uminus_le_zero.

Lemma Uminus_zero_left : $\forall$ x, 0 - $x \equiv 0$.
Hint Resolve Uminus_zero_left.

Lemma Uminus_one_right : $\forall$ x, x - $1 \equiv 0$.
Hint Resolve Uminus_one_right.

Lemma Uminus_eq : $\forall$ x, x - $x \equiv 0$.
Hint Resolve Uminus_eq.

Lemma Uminus_le_left : $\forall$ x y, x - $y \leq x$.

Hint Resolve Uminus_le_left.

Lemma Uminus_le_inv : $\forall$ x y, x - $y \leq$ [1-]y.
Hint Resolve Uminus_le_inv.

Lemma Uminus_plus_simpl : $\forall$ x y, $y \leq x \to (x$ - $y)$ + $y \equiv x$.

Lemma Uminus_plus_zero : $\forall$ x y, $x \leq y \to (x$ - $y)$ + $y \equiv y$.

Hint Resolve Uminus_plus_simpl Uminus_plus_zero.

Lemma Uminus_plus_le : $\forall$ x y, $x \leq (x$ - $y)$ + y.

Hint Resolve Uminus_plus_le.

Lemma Uesp_minus_distr_left : $\forall$ x y z, $(x$ & $y)$ - $z \equiv (x$ - $z)$ & y.

Lemma Uesp_minus_distr_right : $\forall$ x y z, $(x$ & $y)$ - $z \equiv x$ & $(y$ - $z)$.

Hint Resolve Uesp_minus_distr_left Uesp_minus_distr_right.

Lemma Uesp_minus_distr : $\forall$ x y z t, $(x$ & $y)$ - $(z$ + $t) \equiv (x$ - $z)$ & $(y$ - $t)$.
Hint Resolve Uesp_minus_distr.

Lemma Uminus_esp_simpl_left : $\forall$ x y, [1-]$x \leq y \to x$ - $(x$ & $y) \equiv$ [1-]y.

Lemma Uplus_esp_simpl : $\forall$ x y, $(x$ - $(x$ & $y))$+$y \equiv x$+y.
Hint Resolve Uminus_esp_simpl_left Uplus_esp_simpl.

Lemma Uminus_esp_le_inv : $\forall$ x y, x - $(x$ & $y) \leq$ [1-]y.

Hint Resolve Uminus_esp_le_inv.

Lemma Uplus_esp_inv_simpl : $\forall$ x y, $x \leq$ [1-]$y \to (x$ + $y)$ & [1-]$y \equiv x$.
Hint Resolve Uplus_esp_inv_simpl.

Lemma Uplus_inv_esp_simpl : $\forall$ x y, $x \leq y \to (x$ + [1-]$y)$ & $y \equiv x$.
Hint Resolve Uplus_inv_esp_simpl.

4.12 Definition and properties of max

Definition max $(x\ y : U) : U := (x - y) + y$.

Lemma max_eq_right : $\forall\ x\ y : U,\ y \leq x \to \max\ x\ y \equiv x$.

Lemma max_eq_left : $\forall\ x\ y : U,\ x \leq y \to \max\ x\ y \equiv y$.

Hint Resolve max_eq_right max_eq_left.

Lemma max_eq_case : $\forall\ x\ y : U$, orc $(\max\ x\ y \equiv x)\ (\max\ x\ y \equiv y)$.

Add *Morphism* max with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *max_eq_compat*.
Save.

Lemma max_le_right : $\forall\ x\ y : U,\ x \leq \max\ x\ y$.

Lemma max_le_left : $\forall\ x\ y : U,\ y \leq \max\ x\ y$.

Hint Resolve max_le_right max_le_left.

Lemma max_le : $\forall\ x\ y\ z : U,\ x \leq z \to y \leq z \to \max\ x\ y \leq z$.

Lemma max_le_compat : $\forall\ x\ y\ z\ t\colon U,\ x \leq y \to z \leq t \to \max\ x\ z \leq \max\ y\ t$.
Hint Immediate max_le_compat.

Lemma max_idem : $\forall\ x$, $\max\ x\ x \equiv x$.
Hint Resolve max_idem.

Lemma max_sym_le : $\forall\ x\ y$, $\max\ x\ y \leq \max\ y\ x$.
Hint Resolve max_sym_le.

Lemma max_sym : $\forall\ x\ y$, $\max\ x\ y \equiv \max\ y\ x$.
Hint Resolve max_sym.

Lemma max_assoc : $\forall\ x\ y\ z$, $\max\ x\ (\max\ y\ z) \equiv \max\ (\max\ x\ y)\ z$.
Hint Resolve max_assoc.

Lemma max_0 : $\forall\ x$, $\max\ 0\ x \equiv x$.
Hint Resolve max_0.

Instance max_mon : **monotonic2** max.
Save.

Definition Max : U -m> U -m> $U := \text{mon2 max}$.

Lemma max_eq_mult : $\forall\ k\ x\ y$, $\max\ (k\times x)\ (k\times y) \equiv k \times \max\ x\ y$.

Lemma max_eq_plus_cte_right : $\forall\ x\ y\ k$, $\max\ (x+k)\ (y+k) \equiv (\max\ x\ y) + k$.

Hint Resolve max_eq_mult max_eq_plus_cte_right.

4.13 Definition and properties of min

Definition min $(x\ y : U) : U :=$ [1-] $((y - x) +$ [1-]$y)$.

Lemma min_eq_right : $\forall\ x\ y : U,\ x \leq y \to \min\ x\ y \equiv x$.

Lemma min_eq_left : $\forall\ x\ y : U,\ y \leq x \to \min\ x\ y \equiv y$.

Hint Resolve min_eq_right min_eq_left.

Lemma min_eq_case : $\forall\ x\ y : U$, orc $(\min\ x\ y \equiv x)\ (\min\ x\ y \equiv y)$.

Add *Morphism* min with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *min_eq_compat*.
Save.
Hint Immediate min_eq_compat.

Lemma min_le_right : $\forall\ x\ y : U$, $\min\ x\ y \leq x$.

Lemma min_le_left : $\forall\ x\ y : U$, $\min\ x\ y \leq y$.

Hint Resolve min_le_right min_le_left.

Lemma min_le : $\forall\ x\ y\ z : U,\ z \le x \to z \le y \to z \le \min\ x\ y$.

Lemma Uinv_min_max : $\forall\ x\ y,\ [1\text{-}](\min\ x\ y) \equiv \max\ ([1\text{-}]x)\ ([1\text{-}]y)$.

Lemma Uinv_max_min : $\forall\ x\ y,\ [1\text{-}](\max\ x\ y) \equiv \min\ ([1\text{-}]x)\ ([1\text{-}]y)$.

Lemma min_idem : $\forall\ x,\ \min\ x\ x \equiv x$.

Lemma min_mult : $\forall\ x\ y\ k,$
$\min\ (k \times x)\ (k \times y) \equiv k \times (\min\ x\ y)$.
Hint Resolve min_mult.

Lemma min_plus : $\forall\ x1\ x2\ y1\ y2,$
$(\min\ x1\ x2) + (\min\ y1\ y2) \le \min\ (x1{+}y1)\ (x2{+}y2)$.
Hint Resolve min_plus.

Lemma min_plus_cte : $\forall\ x\ y\ k,\ \min\ (x + k)\ (y + k) \equiv (\min\ x\ y) + k$.
Hint Resolve min_plus_cte.

Lemma min_le_compat : $\forall\ x1\ y1\ x2\ y2,$
$x1 \le y1 \to x2 \le y2 \to \min\ x1\ x2 \le \min\ y1\ y2$.
Hint Immediate min_le_compat.

Lemma min_sym_le : $\forall\ x\ y,\ \min\ x\ y \le \min\ y\ x$.
Hint Resolve min_sym_le.

Lemma min_sym : $\forall\ x\ y,\ \min\ x\ y \equiv \min\ y\ x$.
Hint Resolve min_sym.

Lemma min_assoc : $\forall\ x\ y\ z,\ \min\ x\ (\min\ y\ z) \equiv \min\ (\min\ x\ y)\ z$.
Hint Resolve min_assoc.

Lemma min_0 : $\forall\ x,\ \min\ 0\ x \equiv 0$.
Hint Resolve min_0.

Instance min_mon2 : **monotonic2** min.
Save.

Definition Min : U -m> U -m> $U := \mathrm{mon2}\ \min$.

Lemma Min_simpl : $\forall\ x\ y,\ \mathrm{Min}\ x\ y = \min\ x\ y$.

Lemma incr_decomp_aux : $\forall\ f\ g :$ **nat** -m> $U,$
$\forall\ n1\ n2,\ (\forall\ m,\ \not(n1 \le m)\%nat \wedge f\ n1 \le g\ m))$
$\to (\forall\ m,\ \sim((n2 \le m)\%nat \wedge g\ n2 \le f\ m)) \to (n1 \le n2)\%nat \to$ **False**.

Lemma incr_decomp : $\forall\ f\ g$: **nat** -m> $U,$
orc $(\forall\ n,$ exc $(\mathtt{fun}\ m \Rightarrow (n \le m)\%nat \wedge f\ n \le g\ m))$
$(\forall\ n,$ exc $(\mathtt{fun}\ m \Rightarrow (n \le m)\%nat \wedge g\ n \le f\ m))$.

4.14 Other properties

Lemma Uplus_minus_simpl_right : $\forall\ x\ y,\ y \le [1\text{-}]\ x \to (x + y) - y \equiv x$.
Hint Resolve Uplus_minus_simpl_right.

Lemma Uplus_minus_simpl_left : $\forall\ x\ y,\ y \le [1\text{-}]\ x \to (x + y) - x \equiv y$.

Lemma Uminus_assoc_left : $\forall\ x\ y\ z,\ (x - y) - z \equiv x - (y + z)$.

Hint Resolve Uminus_assoc_left.

Lemma Uminus_perm : $\forall\ x\ y\ z,\ (x - y) - z \equiv (x - z) - y$.
Hint Resolve Uminus_perm.

Lemma Uminus_le_perm_left : $\forall\ x\ y\ z,\ y \le x \to x - y \le z \to x \le z + y$.

Lemma Uplus_le_perm_left : $\forall\ x\ y\ z,\ x \le y + z \to x - y \le z$.

Lemma Uminus_eq_perm_left : $\forall\ x\ y\ z,\ y \le x \to x - y \equiv z \to x \equiv z + y$.

Lemma Uplus_eq_perm_left : $\forall\ x\ y\ z,\ y \le [1\text{-}]\ z \to x \equiv y + z \to x - y \equiv z$.

```
Hint Resolve Uminus_le_perm_left Uminus_eq_perm_left.
Hint Resolve Uplus_le_perm_left Uplus_eq_perm_left.
```

Lemma Uminus_le_perm_right : $\forall\ x\ y\ z,\ z \leq y \to x \leq y\ \texttt{-}\ z \to x\ \texttt{+}\ z \leq y$.

Lemma Uplus_le_perm_right : $\forall\ x\ y\ z,\ z \leq \texttt{[1-]}\ x \to x\ \texttt{+}\ z \leq y \to x \leq y\ \texttt{-}\ z$.

```
Hint Resolve Uminus_le_perm_right Uplus_le_perm_right.
```

Lemma Uminus_le_perm : $\forall\ x\ y\ z,\ z \leq y \to x \leq \texttt{[1-]}\ z \to x \leq y\ \texttt{-}\ z \to z \leq y\ \texttt{-}\ x$.

```
Hint Resolve Uminus_le_perm.
```

Lemma Uminus_eq_perm_right : $\forall\ x\ y\ z,\ z \leq y \to x \equiv y\ \texttt{-}\ z \to x\ \texttt{+}\ z \equiv y$.

```
Hint Resolve Uminus_eq_perm_right.
```

Lemma Uminus_plus_perm : $\forall\ x\ y\ z,\ y \leq x \to z \leq \texttt{[1-]}x \to (x\ \texttt{-}\ y)\ \texttt{+}\ z \equiv (x\ \texttt{+}\ z)\ \texttt{-}\ y$.

Lemma Uminus_zero_le : $\forall\ x\ y,\ x\ \texttt{-}\ y \equiv 0 \to x \leq y$.

Lemma Uminus_lt_non_zero : $\forall\ x\ y,\ x < y \to 0' \equiv y\ \texttt{-}\ x$.

```
Hint Immediate Uminus_zero_le Uminus_lt_non_zero.
```

Lemma Ult_le_nth_minus : $\forall\ x\ y,\ x < y \to$ exc $(\texttt{fun}\ n \Rightarrow x \leq y\ \texttt{-}\ \texttt{[1/]1+}n)$.

Lemma Ult_le_nth_plus : $\forall\ x\ y,\ x < y \to$ exc $(\texttt{fun}\ n : \textbf{nat} \Rightarrow x\ \texttt{+}\ \texttt{[1/]1+}n \leq y)$.

Lemma Uminus_distr_left : $\forall\ x\ y\ z,\ (x\ \texttt{-}\ y) \times z \equiv (x \times z)\ \texttt{-}\ (y \times z)$.

```
Hint Resolve Uminus_distr_left.
```

Lemma Uminus_distr_right : $\forall\ x\ y\ z,\ x \times (y\ \texttt{-}\ z) \equiv (x \times y)\ \texttt{-}\ (x \times z)$.

```
Hint Resolve Uminus_distr_right.
```

Lemma Uminus_assoc_right : $\forall\ x\ y\ z,\ y \leq x \to z \leq y \to x\ \texttt{-}\ (y\ \texttt{-}\ z) \equiv (x\ \texttt{-}\ y)\ \texttt{+}\ z$.

Lemma Uplus_minus_assoc_right : $\forall\ x\ y\ z,\ y \leq \texttt{[1-]}x \to z \leq y \to x\ \texttt{+}\ (y\ \texttt{-}\ z) \equiv (x\ \texttt{+}\ y)\ \texttt{-}\ z$.

Lemma Udiv_minus : $\forall\ x\ y\ z,\ 0' \equiv z \to x \leq z \to (x\ \texttt{-}\ y)\ \texttt{/}\ z \equiv x/z\ \texttt{-}\ y/z$.

Lemma Umult_inv_minus : $\forall\ x\ y,\ x \times \texttt{[1-]}y \equiv x\ \texttt{-}\ x \times y$.

```
Hint Resolve Umult_inv_minus.
```

Lemma Uinv_mult_minus : $\forall\ x\ y,\ (\texttt{[1-]}x) \times y \equiv y\ \texttt{-}\ x \times y$.

```
Hint Resolve Uinv_mult_minus.
```

Lemma Uminus_plus_perm_right : $\forall\ x\ y\ z,\ y \leq x \to y \leq z \to (x\ \texttt{-}\ y)\ \texttt{+}\ z \equiv x\ \texttt{+}\ (z\ \texttt{-}\ y)$.

```
Hint Resolve Uminus_plus_perm_right.
```

- triangular inequality

Lemma Uminus_triangular : $\forall\ x\ y\ z,\ x\ \texttt{-}\ y \leq (x\ \texttt{-}\ z)\ \texttt{+}\ (z\ \texttt{-}\ y)$.

```
Hint Resolve Uminus_triangular.
```

4.15 Definition and properties of generalized sums

```
Definition sigma : (nat → U) → nat -m> U.
Defined.
```

Lemma sigma_0 : $\forall\ (f : \textbf{nat} \to U),$ sigma f O $\equiv 0$.

Lemma sigma_S : $\forall\ (f : \textbf{nat} \to U)\ (n{:}\textbf{nat}),$ sigma f (S n) = $(f\ n)$ + (sigma f n).

Lemma sigma_1 : $\forall\ (f : \textbf{nat} \to U),$ sigma f (S 0) $\equiv f$ O.

Lemma sigma_incr : $\forall\ (f : \textbf{nat} \to U)\ (n\ m{:}\textbf{nat}),\ (n \leq m)\%nat \to$ sigma f $n \leq$ sigma f m.

```
Hint Resolve sigma_incr.
```

Lemma sigma_eq_compat : $\forall\ (f\ g{:}\ \textbf{nat} \to U)\ (n{:}\textbf{nat}),$
$(\forall\ k,\ (k < n)\%nat \to f\ k \equiv g\ k) \to$ sigma f $n \equiv$ sigma g n.

Lemma sigma_le_compat : $\forall\ (f\ g{:}\ \textbf{nat} \to U)\ (n{:}\textbf{nat}),$

$(\forall\ k,\ (k < n)\%nat \to f\ k \leq g\ k) \to$ sigma $f\ n \leq$ sigma $g\ n$.

Lemma sigma_S_lift : $\forall\ (f$:nat $\to U)\ (n$:nat),
sigma $f\ (S\ n) \equiv (f\ O)$ + (sigma (fun $k \Rightarrow f\ (S\ k))\ n$).

Lemma sigma_plus_lift : $\forall\ (f$:nat $\to U)\ (n\ m$:nat),
sigma $f\ (n+m)\%nat \equiv$ sigma $f\ n$ + sigma (fun $k \Rightarrow f\ (n+k)\%nat)\ m$.

Lemma sigma_zero : $\forall\ f\ n$,
$(\forall\ k,\ (k<n)\%nat \to f\ k \equiv 0) \to$ sigma $f\ n \equiv 0$.

Lemma sigma_not_zero : $\forall\ f\ n\ k,\ (k<n)\%nat \to 0 < f\ k \to 0 <$ sigma $f\ n$.

Lemma sigma_zero_elim : $\forall\ f\ n$,
(sigma $f\ n) \equiv 0 \to \forall\ k,\ (k<n)\%nat \to f\ k \equiv 0$.

Hint Resolve sigma_eq_compat sigma_le_compat sigma_zero.

Lemma sigma_le : $\forall\ f\ n\ k,\ (k<n)\%nat \to f\ k \leq$ sigma $f\ n$.
Hint Resolve sigma_le.

Lemma sigma_minus_decr : $\forall\ f\ n,\ (\forall\ k, f\ (S\ k) \leq f\ k) \to$
sigma (fun $k \Rightarrow f\ k$ - $f\ (S\ k))\ n \equiv f\ O$ - $f\ n$.

Lemma sigma_minus_incr : $\forall\ f\ n,\ (\forall\ k, f\ k \leq f\ (S\ k)) \to$
sigma (fun $k \Rightarrow f\ (S\ k)$ - $f\ k)\ n \equiv f\ n$ - $f\ O$.

4.16 Definition and properties of generalized products

Definition prod $(alpha$: nat $\to U)\ (n$:nat) := compn *Umult* 1 $alpha\ n$.

Lemma prod_0 : $\forall\ (f$: nat $\to U)$, prod $f\ 0 = 1$.

Lemma prod_S : $\forall\ (f$:nat $\to U)\ (n$:nat), prod $f\ (S\ n) = (f\ n) \times$ (prod $f\ n)$.

Lemma prod_1 : $\forall\ (f$: nat $\to U)$, prod $f\ (S\ 0) \equiv f\ O$.

Lemma prod_S_lift : $\forall\ (f$:nat $\to U)\ (n$:nat),
prod $f\ (S\ n) \equiv (f\ O) \times$ (prod (fun $k \Rightarrow f\ (S\ k))\ n$).

Lemma prod_decr : $\forall\ (f$: nat $\to U)\ (n\ m$:nat), $(n \leq m)\%nat \to$ prod $f\ m \leq$ prod $f\ n$.

Hint Resolve prod_decr.

Lemma prod_eq_compat : $\forall\ (f\ g$: nat $\to U)\ (n$:nat),
$(\forall\ k,\ (k < n)\%nat \to f\ k \equiv g\ k) \to$ (prod $f\ n) \equiv$ (prod $g\ n$).

Lemma prod_le_compat : $\forall\ (f\ g$: nat $\to U)\ (n$:nat),
$(\forall\ k,\ (k < n)\%nat \to f\ k \leq g\ k) \to$ prod $f\ n \leq$ prod $g\ n$.

Lemma prod_zero : $\forall\ f\ n\ k,\ (k<n)\%nat \to f\ k \equiv 0 \to$ prod $f\ n \equiv 0$.

Lemma prod_not_zero : $\forall\ f\ n$,
$(\forall\ k,\ (k<n)\%nat \to 0 < f\ k) \to 0 <$ prod $f\ n$.

Lemma prod_zero_elim : $\forall\ f\ n$,
prod $f\ n \equiv 0 \to$ exc (fun $k \Rightarrow (k<n)\%nat \wedge f\ k \equiv 0)$.

Hint Resolve prod_eq_compat prod_le_compat prod_not_zero.

Lemma prod_le : $\forall\ f\ n\ k,\ (k<n)\%nat \to$ prod $f\ n \leq f\ k$.

Lemma prod_minus : $\forall\ f\ n$, prod $f\ n$ - prod $f\ (S\ n) \equiv$ ([1-]$f\ n) \times$ prod $f\ n$.

Definition Prod : (nat $\to U) \to$ nat -m$\to U$.
Defined.

Lemma Prod_simpl : $\forall\ f\ n$, Prod $f\ n =$ prod $f\ n$.
Hint Resolve Prod_simpl.

4.17 Properties of *Unth*

Lemma Unth_zero : [1/]1+0 ≡ 1.

Notation "$\frac{1}{2}$" := (*Unth* 1).

Lemma Unth_one : $\frac{1}{2}$ ≡ [1-] $\frac{1}{2}$.

Hint Resolve Unth_zero Unth_one.

Lemma Unth_one_plus : $\frac{1}{2}$ + $\frac{1}{2}$ ≡ 1.
Hint Resolve Unth_one_plus.

Lemma Unth_one_refl : ∀ t, $\frac{1}{2}$ × t + $\frac{1}{2}$ × t ≡ t.

Lemma Unth_not_null : ∀ n, ∤(0 ≡ [1/]1+n).
Hint Resolve Unth_not_null.

Lemma Unth_lt_zero : ∀ n, 0 < [1/]1+n.
Hint Resolve Unth_lt_zero.

Lemma Unth_inv_lt_one : ∀ n, [1-][1/]1+n<1.
Hint Resolve Unth_inv_lt_one.

Lemma Unth_not_one : ∀ n, ∤(1 ≡ [1-][1/]1+n).
Hint Resolve Unth_not_one.

Lemma Unth_prop_sigma : ∀ n, [1/]1+n ≡ [1-] (sigma (fun k ⇒ [1/]1+n) n).
Hint Resolve Unth_prop_sigma.

Lemma Unth_sigma_n : ∀ n : **nat**, ∤(1 ≡ sigma (fun k ⇒ [1/]1+n) n).

Lemma Unth_sigma_Sn : ∀ n : **nat**, 1 ≡ sigma (fun k ⇒ [1/]1+n) (S n).

Hint Resolve Unth_sigma_n Unth_sigma_Sn.

Lemma Unth_decr : ∀ n, [1/]1+(S n) < [1/]1+n.
Hint Resolve Unth_decr.

Lemma Unth_anti_mon :
∀ n m, ($n \leq m$)%nat → [1/]1+m ≤ [1/]1+n.
Hint Resolve Unth_anti_mon.

Lemma Unth_le_half : ∀ n, [1/]1+(S n) ≤ $\frac{1}{2}$.
Hint Resolve Unth_le_half.

4.17.1 Mean of two numbers : $\frac{1}{2}$ x + $\frac{1}{2}$ y

Definition mean (x y:U) := $\frac{1}{2}$ × x + $\frac{1}{2}$ × y.

Lemma mean_eq : ∀ x:U, mean x x ≡x.

Lemma mean_le_compat_right : ∀ x y z, $y \leq z$ → mean x y ≤ mean x z.

Lemma mean_le_compat_left : ∀ x y z, $x \leq y$ → mean x z ≤ mean y z.

Hint Resolve mean_eq mean_le_compat_left mean_le_compat_right.

Lemma mean_lt_compat_right : ∀ x y z, $y < z$ → mean x y < mean x z.

Lemma mean_lt_compat_left : ∀ x y z, $x < y$ → mean x z < mean y z.

Hint Resolve mean_eq mean_le_compat_left mean_le_compat_right.
Hint Resolve mean_lt_compat_left mean_lt_compat_right.

Lemma mean_le_up : ∀ x y, $x \leq y$ → mean x y ≤ y.

Lemma mean_le_down : ∀ x y, $x \leq y$ → x ≤ mean x y.

Lemma mean_lt_up : ∀ x y, $x < y$ → mean x y < y.

Lemma mean_lt_down : ∀ x y, $x < y$ → x < mean x y.

Hint Resolve mean_le_up mean_le_down mean_lt_up mean_lt_down.

4.17.2 Properties of $\frac{1}{2}$

Lemma le_half_inv : $\forall\ x,\ x \leq \frac{1}{2} \to x \leq$ [1-] x.

Hint Immediate le_half_inv.

Lemma ge_half_inv : $\forall\ x,\ \frac{1}{2} \leq x \to$ [1-] $x \leq x$.

Hint Immediate ge_half_inv.

Lemma Uinv_le_half_left : $\forall\ x,\ x \leq \frac{1}{2} \to \frac{1}{2} \leq$ [1-] x.

Lemma Uinv_le_half_right : $\forall\ x,\ \frac{1}{2} \leq x \to$ [1-] $x \leq \frac{1}{2}$.

Hint Resolve Uinv_le_half_left Uinv_le_half_right.

Lemma half_twice : $\forall\ x,\ x \leq \frac{1}{2} \to \frac{1}{2} \times (x + x) \equiv x$.

Lemma half_twice_le : $\forall\ x,\ \frac{1}{2} \times (x + x) \leq x$.

Lemma Uinv_half : $\forall\ x,\ \frac{1}{2} \times$ ([1-] x) + $\frac{1}{2} \equiv$ [1-] ($\frac{1}{2} \times x$).

Lemma Uinv_half_plus : $\forall\ x,$ [1-]x + $\frac{1}{2} \times x \equiv$ [1-] ($\frac{1}{2} \times x$).

Lemma half_esp :
$\forall\ x,\ (\frac{1}{2} \leq x) \to (\frac{1}{2}) \times (x$ & $x)$ + $\frac{1}{2} \equiv x$.

Lemma half_esp_le : $\forall\ x,\ x \leq \frac{1}{2} \times (x$ & $x)$ + $\frac{1}{2}$.
Hint Resolve half_esp_le.

Lemma half_le : $\forall\ x\ y,\ y \leq$ [1-] $y \to x \leq y + y \to (\frac{1}{2}) \times x \leq y$.

Lemma half_Unth: $\forall\ n,\ (\frac{1}{2})$*([1/]1+$n$) $\leq$ [1/]1+(S n).
Hint Resolve half_le half_Unth.

Lemma half_exp : $\forall\ n,\ \frac{1}{2}$^$n \equiv \frac{1}{2}$^(S n) + $\frac{1}{2}$^(S n).

4.18 Diff function : | x - y |

Definition diff $(x\ y$:$U) :=$ $(x$ - $y)$ + $(y$ - $x)$.

Lemma diff_eq : $\forall\ x,$ diff $x\ x \equiv 0$.
Hint Resolve diff_eq.

Lemma diff_sym : $\forall\ x\ y,$ diff $x\ y \equiv$ diff $y\ x$.
Hint Resolve diff_sym.

Lemma diff_zero : $\forall\ x,$ diff $x\ 0 \equiv x$.
Hint Resolve diff_zero.

Add *Morphism* diff with *signature* Oeq $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *diff_eq_compat*.
Qed.
Hint Immediate diff_eq_compat.

Lemma diff_plus_ok : $\forall\ x\ y,\ x$ - $y \leq$ [1-]$(y$ - $x)$.
Hint Resolve diff_plus_ok.

Lemma diff_Uminus : $\forall\ x\ y,\ x \leq y \to$ diff $x\ y \equiv y$ - x.

Lemma diff_Uplus_le : $\forall\ x\ y,\ x \leq$ diff $x\ y$ + y.
Hint Resolve diff_Uplus_le.

Lemma diff_triangular : $\forall\ x\ y\ z,$ diff $x\ y \leq$ diff $x\ z$ + diff $y\ z$.
Hint Resolve diff_triangular.

4.19 Density

Lemma Ule_lt_lim : $\forall\ x\ y,\ (\forall\ t,\ t < x \to t \leq y) \to x \leq y$.

Lemma Ule_nth_lim : $\forall\ x\ y,\ (\forall\ p,\ x \leq y +$ [1/]1+$p) \to x \leq y$.

4.20 Properties of least upper bounds

Lemma lub_un : mlub (cte **nat** 1) ≡ 1.
Hint Resolve lub_un.

Lemma UPlusk_eq : ∀ k, UPlus k ≡ mon (*Uplus* k).

Lemma UMultk_eq : ∀ k, UMult k ≡ mon (*Umult* k).

Lemma UPlus_continuous_right : ∀ k, **continuous** (UPlus k).
Hint Resolve UPlus_continuous_right.

Lemma UPlus_continuous_left : **continuous** UPlus.
Hint Resolve UPlus_continuous_left.

Lemma UMult_continuous_right : ∀ k, **continuous** (UMult k).
Hint Resolve UMult_continuous_right.

Lemma UMult_continuous_left : **continuous** UMult.
Hint Resolve UMult_continuous_left.

Lemma lub_eq_plus_cte_left : ∀ (f:**nat** -m> U) (k:U), lub ((UPlus k) @ f) ≡ k + lub f.
Hint Resolve lub_eq_plus_cte_left.

Lemma lub_eq_mult : ∀ (k:U) (f:**nat** -m> U), lub ((UMult k) @ f) ≡ k × lub f.
Hint Resolve lub_eq_mult.

Lemma lub_eq_plus_cte_right : ∀ (f : **nat** -m> U) (k:U),
lub ((mshift UPlus k) @ f) ≡ lub f + k.
Hint Resolve lub_eq_plus_cte_right.

Lemma min_lub_le : ∀ f g : **nat** -m> U,
lub ((Min @^{2} f) g) ≤ min (lub f) (lub g).

Lemma min_lub_le_incr_aux : ∀ f g : **nat** -m> U,
(∀ n, exc (fun m ⇒ (n≤m)%nat ∧ f n ≤ g m))
→ min (lub f) (lub g) ≤ lub ((Min @^{2} f) g).

Lemma min_lub_le_incr : ∀ f g : **nat** -m> U,
min (lub f) (lub g) ≤ lub ((Min @^{2} f) g).

Lemma min_continuous2 : **continuous2** Min.
Hint Resolve min_continuous2.

Lemma lub_eq_esp_right :
∀ (f : **nat** -m> U) (k : U), lub ((mshift UEsp k) @ f) ≡ lub f & k.
Hint Resolve lub_eq_esp_right.

Lemma Udiv_continuous : ∀ (k:U), **continuous** (UDiv k).
Hint Resolve Udiv_continuous.

4.21 Greatest lower bounds

Definition glb (f:**nat** -m→ U) := [1-](lub (UInv @ f)).

Lemma glb_le: ∀ (f : **nat** -m→ U) (n : **nat**), glb f ≤ (f n).

Lemma le_glb: ∀ (f : **nat** -m→ U) (x:U),
(∀ n : **nat**, x ≤ f n) → x ≤ glb f.
Hint Resolve glb_le le_glb.

Definition Uopp : **cpo** (o:=lord U) U.
Defined.

Lemma Uopp_lub_simpl
: ∀ h : **nat** -m→ U, lub (cpo:=Uopp) h = glb h.

Lemma Uopp_mon_seq : ∀ f:**nat** -m→ U,

$\forall\ n\ m$:**nat**, $(n \leq m)\%nat \to f\ m \leq f\ n$.
Hint Resolve Uopp_mon_seq.

Infinite product: $\Pi_{i=0}^{\infty} f\ i$ Definition prod_inf (f : **nat** $\to$ U) : U := glb (Prod f).

Properties of *glb*

Lemma glb_le_compat:
$\forall\ f\ g$: **nat** -m$\to$ U, $(\forall\ x, f\ x \leq g\ x) \to$ glb $f \leq$ glb g.
Hint Resolve glb_le_compat.

Lemma glb_eq_compat:
$\forall\ f\ g$: **nat** -m$\to$ U, $f \equiv g \to$ glb $f \equiv$ glb g.
Hint Resolve glb_eq_compat.

Lemma glb_cte: $\forall\ c$: U, glb (mon (cte **nat** ($o1$:=(lord U)) c)) $\equiv c$.
Hint Resolve glb_cte.

Lemma glb_eq_plus_cte_right:
$\forall$ (f : **nat** -m$\to$ U) (k : U), glb (lmon (mshift UPlus k) @ f) $\equiv$ glb f + k.
Hint Resolve glb_eq_plus_cte_right.

Lemma glb_eq_plus_cte_left:
$\forall$ (f : **nat** -m$\to$ U) (k : U), glb (lmon (UPlus k) @ f) $\equiv k +$ glb f.
Hint Resolve glb_eq_plus_cte_left.

Lemma glb_eq_mult:
$\forall$ (k : U) (f : **nat** -m$\to$ U), glb (lmon (UMult k) @ f) $\equiv k \times$ glb f.

Lemma lmon2_plus_continuous
: **continuous2** ($c1$:=Uopp) ($c2$:=Uopp) ($c3$:=Uopp) (imon2 *Uplus*).

Hint Resolve lmon2_plus_continuous.

Lemma Uinv_continuous : **continuous** ($c1$:=Uopp) UInv.

Lemma Uinv_lub_eq : $\forall\ f$: **nat** -m$\to$ U, [1-](lub (cpo:=Uopp) f) $\equiv$ lub (UInv@f).

Lemma Uinvopp_mon : **monotonic** ($o2$:= lord U) *Uinv*.
Hint Resolve Uinvopp_mon.

Definition UInvopp : U -m$\to$ U
:= mon ($o2$:= lord U) *Uinv* ($fmonotonic$:=Uinvopp_mon).

Lemma UInvopp_simpl : $\forall\ x$, UInvopp x = [1-]x.

Lemma Uinvopp_continuous : **continuous** ($c2$:=Uopp) UInvopp.

Lemma Uinvopp_lub_eq
: $\forall\ f$: **nat** -m> U, [1-](lub f) $\equiv$ lub (cpo:=Uopp) (UInvopp@f).

Hint Resolve Uinv_continuous Uinvopp_continuous.

Instance Uminus_mon2 : **monotonic2** ($o2$:=lord U) Uminus.
Save.

Definition UMinus : U -m> U -m> U := mon2 Uminus.

Lemma UMinus_simpl : $\forall\ x\ y$, UMinus $x\ y$ = x - y.

Lemma Uminus_continuous2 : **continuous2** ($c2$:=Uopp) UMinus.
Hint Resolve Uminus_continuous2.

Lemma glb_le_esp : $\forall\ f\ g$:**nat** -m$\to$ U, (glb f) & (glb g) $\leq$ glb ((imon2 Uesp $@^2$ f) g).
Hint Resolve glb_le_esp.

Lemma Uesp_min : $\forall\ a1\ a2\ b1\ b2$, min $a1\ b1$ & min $a2\ b2 \leq$ min ($a1$ & $a2$) ($b1$ & $b2$).

Defining lubs of arbitrary sequences

Fixpoint seq_max (f:**nat** $\to$ U) (n:**nat**) : U := match n with
0 $\Rightarrow f$ 0 | S $p \Rightarrow$ max (seq_max f p) (f (S p)) end.

```
Lemma seq_max_incr : ∀ f n, seq_max f n ≤ seq_max f (S n).
Hint Resolve seq_max_incr.

Lemma seq_max_le : ∀ f n, f n ≤ seq_max f n.
Hint Resolve seq_max_le.

Instance seq_max_mon : ∀ (f:nat → U), monotonic (seq_max f).
Save.

Definition sMax (f:nat → U) : nat -m> U := mon (seq_max f).

Lemma sMax_mult : ∀ k (f:nat→U), sMax (fun n ⇒ k × f n) ≡ UMult k @ sMax f.

Lemma sMax_plus_cte_right : ∀ k (f:nat→ U),
    sMax (fun n ⇒ f n + k) ≡ mshift UPlus k @ sMax f.

Definition Ulub (f:nat → U) := lub (sMax f).

Lemma le_Ulub : ∀ f n, f n ≤ Ulub f.

Lemma Ulub_le : ∀ f x, (∀ n, f n ≤ x) → Ulub f ≤ x.

Hint Resolve le_Ulub Ulub_le.

Lemma Ulub_le_compat : ∀ f g : nat→U, f ≤ g → Ulub f ≤ Ulub g.
Hint Resolve Ulub_le_compat.

Add Morphism Ulub with signature Oeq ⟹ Oeq as Ulub_eq_compat.
Save.
Hint Resolve Ulub_eq_compat.

Lemma Ulub_eq_mult : ∀ k (f:nat→U), Ulub (fun n ⇒ k × f n)≡ k × Ulub f.

Lemma Ulub_eq_plus_cte_right : ∀ (f:nat→U) k, Ulub (fun n ⇒ f n + k)≡ Ulub f + k.

Hint Resolve Ulub_eq_mult Ulub_eq_plus_cte_right.

Lemma Ulub_eq_esp_right :
  ∀ (f : nat → U) (k : U), Ulub (fun n ⇒ f n & k) ≡ Ulub f & k.
Hint Resolve lub_eq_esp_right.

Lemma Ulub_le_plus : ∀ f g, Ulub (fun n ⇒ f n + g n) ≤ Ulub f + Ulub g.
Hint Resolve Ulub_le_plus.

Definition Uglb (f:nat → U) :U := [1-]Ulub (fun n ⇒ [1-](f n)).

Lemma Uglb_le: ∀ (f : nat → U) (n : nat), Uglb f ≤ f n.

Lemma le_Uglb: ∀ (f : nat → U) (x:U),
  (∀ n : nat, x ≤ f n) → x ≤ Uglb f.
Hint Resolve Uglb_le le_Uglb.

Lemma Uglb_le_compat : ∀ f g : nat → U, f ≤ g → Uglb f ≤ Uglb g.
Hint Resolve Uglb_le_compat.

Add Morphism Uglb with signature Oeq ⟹ Oeq as Uglb_eq_compat.
Save.
Hint Resolve Uglb_eq_compat.

Lemma Uglb_eq_plus_cte_right:
  ∀ (f : nat → U) (k : U), Uglb (fun n ⇒ f n + k) ≡ Uglb f + k.
Hint Resolve Uglb_eq_plus_cte_right.

Lemma Uglb_eq_mult:
  ∀ (k : U) (f : nat → U), Uglb (fun n ⇒ k × f n) ≡ k × Uglb f.
Hint Resolve Uglb_eq_mult Uglb_eq_plus_cte_right.

Lemma Uglb_le_plus : ∀ f g, Uglb f + Uglb g ≤ Uglb (fun n ⇒ f n + g n).
Hint Resolve Uglb_le_plus.

Lemma Ulub_lub : ∀ f:nat -m> U, Ulub f ≡ lub f.
```

```
Hint Resolve Ulub_lub.

Lemma Uglb_glb : ∀ f:nat -m→ U, Uglb f ≡ glb f.
Hint Resolve Uglb_glb.

Lemma lub_le_plus : ∀ (f g : nat -m> U), lub ((UPlus @² f) g) ≤ lub f + lub g.
Hint Resolve lub_le_plus.

Lemma glb_le_plus : ∀ (f g:nat -m→ U) , glb f + glb g ≤ glb ((Imon2 UPlus @² f) g).
Hint Resolve glb_le_plus.

Lemma lub_eq_plus : ∀ f g : nat -m> U, lub ((UPlus @² f) g) ≡ lub f + lub g.
Hint Resolve lub_eq_plus.

Lemma glb_mon : ∀ f : nat -m> U, Uglb f ≡ f 0.

Lemma lub_inv : ∀ (f g : nat -m> U), (∀ n, f n ≤ [1-] g n) → lub f ≤ [1-] (lub g).

Lemma glb_lift_left : ∀ (f:nat -m→ U) n,
      glb f ≡ glb (mon (seq_lift_left f n)).
Hint Resolve glb_lift_left.

Lemma Ulub_mon : ∀ f : nat -m→ U, Ulub f ≡ f 0.

Lemma lub_glb_le : ∀ (f:nat -m> U) (g:nat -m→ U),
       (∀ n, f n ≤ g n) → lub f ≤ glb g.

Lemma lub_lub_inv_le : ∀ f g :nat -m> U,
       (∀ n, f n ≤ [1-]g n) → lub f ≤ [1-] lub g.

Lemma Uplus_opp_continuous_right :
      ∀ k, continuous (c1:=Uopp) (c2:=Uopp) (Imon (UPlus k)).

Lemma Uplus_opp_continuous_left :
      continuous (c1:=Uopp) (c2:=fmon_cpo (o:=Iord U) (c:=Uopp))(Imon2 UPlus).

Hint Resolve Uplus_opp_continuous_right Uplus_opp_continuous_left.

Instance Uplusopp_continuous2 : continuous2 (c1:=Uopp) (c2:=Uopp) (c3:=Uopp) (Imon2 UPlus).
Save.

Lemma Uplusopp_lub_eq : ∀ (f g : nat -m→ U),
     lub (cpo:=Uopp) f + lub (cpo:=Uopp) g ≡ lub (cpo:=Uopp) ((Imon2 UPlus @² f) g).

Lemma glb_eq_plus : ∀ (f g : nat -m→ U), glb ((Imon2 UPlus @² f) g) ≡ glb f + glb g.
Hint Resolve glb_eq_plus.

Instance UEsp_continuous2 : continuous2 UEsp.
Save.

Lemma Uesp_lub_eq : ∀ f g : nat -m> U, lub f & lub g ≡ lub ((UEsp @² f) g).

Instance sigma_mon :monotonic sigma.
Save.

Definition Sigma : (nat → U) -m> nat-m> U
     := mon sigma (fmonotonic:=sigma_mon).

Lemma Sigma_simpl : ∀ f, Sigma f = sigma f.

Lemma sigma_continuous1 : continuous Sigma.

Lemma sigma_lub1 : ∀ (f : nat -m> (nat → U)) n,
        sigma (lub f) n ≡ lub ((mshift Sigma n) @ f).


Definition MF (A:Type) : Type := A → U.

Lemma MFcpo (A:Type) : cpo (MF A).

Definition MFopp (A:Type) : cpo (o:=Iord (A → U)) (MF A).
Defined.
```

```
Lemma MFopp_lub_eq : ∀ (A:Type) (h:nat-m→ MF A),
      lub (cpo:=MFopp A) h ≡ fun x ⇒ glb (lord_app x @ h).
Lemma fle_intro : ∀ (A:Type) (f g : MF A), (∀ x, f x ≤ g x) → f ≤ g.
Hint Resolve fle_intro.
Lemma feq_intro : ∀ (A:Type) (f g : MF A), (∀ x, f x ≡ g x) → f ≡ g.
Hint Resolve feq_intro.
Definition fplus (A:Type) (f g : MF A) : MF A :=
                fun x ⇒ f x + g x.
Definition fmult (A:Type) (k:U) (f : MF A) : MF A :=
                fun x ⇒ k × f x.
Definition finv (A:Type) (f : MF A) : MF A :=
                fun x ⇒ [1-] f x.
Definition fzero (A:Type) : MF A :=
                fun x ⇒ 0.
Definition fdiv (A:Type) (k:U) (f : MF A) : MF A :=
                fun x ⇒ (f x) / k.
Definition flub (A:Type) (f : nat -m> MF A) : MF A := lub f.
Lemma fplus_simpl : ∀ (A:Type)(f g : MF A) (x : A),
                                fplus f g x = f x + g x.
Lemma fplus_def : ∀ (A:Type)(f g : MF A),
                                fplus f g = fun x ⇒ f x + g x.
Lemma fmult_simpl : ∀ (A:Type)(k:U) (f : MF A) (x : A),
                                fmult k f x = k × f x.
Lemma fmult_def : ∀ (A:Type)(k:U) (f : MF A),
                                fmult k f = fun x ⇒ k × f x.
Lemma fdiv_simpl : ∀ (A:Type)(k:U) (f : MF A) (x : A),
                                fdiv k f x = f x / k.
Lemma fdiv_def : ∀ (A:Type)(k:U) (f : MF A),
                                fdiv k f = fun x ⇒ f x / k.
Implicit Arguments fzero [].
Lemma fzero_simpl : ∀ (A:Type)(x : A), fzero A x = 0.
Lemma fzero_def : ∀ (A:Type), fzero A = fun x:A ⇒ 0.
Lemma finv_simpl : ∀ (A:Type)(f : MF A) (x : A), finv f x = [1-]f x.
Lemma finv_def : ∀ (A:Type)(f : MF A), finv f = fun x ⇒ [1-](f x).
Lemma flub_simpl : ∀ (A:Type)(f:nat -m> MF A) (x:A),
                                (flub f) x = lub (f <o> x).
Lemma flub_def : ∀ (A:Type)(f:nat -m> MF A),
                                (flub f) = fun x ⇒ lub (f <o> x).
Hint Resolve fplus_simpl fmult_simpl fzero_simpl finv_simpl flub_simpl.
Definition fone (A:Type) : MF A := fun x ⇒ 1.
Implicit Arguments fone [].
Lemma fone_simpl : ∀ (A:Type) (x:A), fone A x = 1.
Lemma fone_def : ∀ (A:Type), fone A = fun (x:A) ⇒ 1.
Definition fcte (A:Type) (k:U): MF A := fun x ⇒ k.
Implicit Arguments fcte [].
Lemma fcte_simpl : ∀ (A:Type) (k:U) (x:A), fcte A k x = k.
```

```
Lemma fcte_def : ∀ (A:Type) (k:U), fcte A k = fun (x:A) ⇒ k.
Definition fminus (A:Type) (f g :MF A) : MF A := fun x ⇒ f x - g x.
Lemma fminus_simpl : ∀ (A:Type) (f g: MF A) (x:A), fminus f g x = f x - g x.
Lemma fminus_def : ∀ (A:Type) (f g: MF A), fminus f g = fun x ⇒ f x - g x.
Definition fesp (A:Type) (f g :MF A) : MF A := fun x ⇒ f x & g x.
Lemma fesp_simpl : ∀ (A:Type) (f g: MF A) (x:A), fesp f g x = f x & g x.
Lemma fesp_def : ∀ (A:Type) (f g: MF A) , fesp f g = fun x ⇒ f x & g x.
Definition fconj (A:Type)(f g:MF A) : MF A := fun x ⇒ f x × g x.
Lemma fconj_simpl : ∀ (A:Type) (f g: MF A) (x:A), fconj f g x = f x × g x.
Lemma fconj_def : ∀ (A:Type) (f g: MF A), fconj f g = fun x ⇒ f x × g x.

Lemma MF_lub_simpl : ∀ (A:Type) (f : nat -m> MF A) (x:A),
                lub f x = lub (f <o>x).
Hint Resolve MF_lub_simpl.
Lemma MF_lub_def : ∀ (A:Type) (f : nat -m> MF A),
                lub f = fun x ⇒ lub (f <o>x).
```

4.21.1 Defining morphisms

```
Lemma fplus_eq_compat : ∀ A (f1 f2 g1 g2:MF A),
            f1≡f2 → g1≡g2 → fplus f1 g1 ≡ fplus f2 g2.
Add Parametric Morphism (A:Type) : (@fplus A)
    with signature Oeq ⟹ Oeq ⟹ Oeq
    as fplus_feq_compat_morph.
Save.
Instance fplus_mon2 : ∀ A, monotonic2 (fplus (A:=A)).
Save.
Hint Resolve fplus_mon2.
Lemma fplus_le_compat : ∀ A (f1 f2 g1 g2:MF A),
            f1≤f2 → g1≤g2 → fplus f1 g1 ≤ fplus f2 g2.
Add Parametric Morphism A : (@fplus A) with signature Ole ++> Ole ++> Ole
    as fplus_fle_compat_morph.
Save.
Lemma finv_eq_compat : ∀ A (f g:MF A), f≡g → finv f ≡ finv g.
Add Parametric Morphism A : (@finv A) with signature Oeq ⟹ Oeq
    as finv_feq_compat_morph.
Save.
Instance finv_mon : ∀ A, monotonic (o2:=lord (MF A)) (finv (A:=A)).
Save.
Hint Resolve finv_mon.
Lemma finv_le_compat : ∀ A (f g:MF A), f ≤ g → finv g ≤ finv f.
Add Parametric Morphism A: (@finv A)
   with signature Ole -> Ole as finv_fle_compat_morph.
Save.
Lemma fmult_eq_compat : ∀ A k1 k2 (f1 f2:MF A),
            k1 ≡ k2 → f1 ≡ f2 → fmult k1 f1 ≡ fmult k2 f2.
Add Parametric Morphism A : (@fmult A)
   with signature Oeq ⟹ Oeq ⟹ Oeq as fmult_feq_compat_morph.
```

```
Save.
Instance fmult_mon2 : ∀ A, monotonic2 (fmult (A:=A)).
Save.
Hint Resolve fmult_mon2.
Lemma fmult_le_compat : ∀ A k1 k2 (f1 f2:MF A),
          k1 ≤ k2 → f1 ≤ f2 → fmult k1 f1 ≤ fmult k2 f2.
Add Parametric Morphism A : (@fmult A)
    with signature Ole ++> Ole ++> Ole as fmult_fle_compat_morph.
Save.
Lemma fminus_eq_compat : ∀ A (f1 f2 g1 g2:MF A),
          f1 ≡ f2 → g1 ≡ g2 → fminus f1 g1 ≡ fminus f2 g2.
Add Parametric Morphism A : (@fminus A)
    with signature Oeq ⟹ Oeq ⟹ Oeq as fminus_feq_compat_morph.
Save.
Instance fminus_mon2 : ∀ A, monotonic2 (o2:=lord (MF A)) (fminus (A:=A)).
Save.
Hint Resolve fminus_mon2.
Lemma fminus_le_compat : ∀ A (f1 f2 g1 g2:MF A),
          f1 ≤ f2 → g2 ≤ g1 → fminus f1 g1 ≤ fminus f2 g2.
Add Parametric Morphism A : (@fminus A)
    with signature Ole ++> Ole -> Ole as fminus_fle_compat_morph.
Save.
Lemma fesp_eq_compat : ∀ A (f1 f2 g1 g2:MF A),
          f1≡f2 → g1≡g2 → fesp f1 g1 ≡ fesp f2 g2.
Add Parametric Morphism A : (@fesp A) with signature Oeq ⟹ Oeq ⟹ Oeq as fesp_feq_compat_morph.
Save.
Instance fesp_mon2 : ∀ A, monotonic2 (fesp (A:=A)).
Save.
Hint Resolve fesp_mon2.
Lemma fesp_le_compat : ∀ A (f1 f2 g1 g2:MF A),
          f1≤f2 → g1≤g2 → fesp f1 g1 ≤ fesp f2 g2.
Add Parametric Morphism A : (@fesp A)
   with signature Ole ++> Ole ++> Ole as fesp_fle_compat_morph.
Save.
Lemma fconj_eq_compat : ∀ A (f1 f2 g1 g2:MF A),
            f1≡f2 → g1≡g2 → fconj f1 g1 ≡ fconj f2 g2.
 Add Parametric Morphism A : (@fconj A)
   with signature Oeq ⟹ Oeq ⟹ Oeq
 as fconj_feq_compat_morph.
 Save.
Instance fconj_mon2 : ∀ A, monotonic2 (fconj (A:=A)).
Save.
Hint Resolve fconj_mon2.
Lemma fconj_le_compat : ∀ A (f1 f2 g1 g2:MF A),
          f1 ≤ f2 → g1 ≤ g2 → fconj f1 g1 ≤ fconj f2 g2.
Add Parametric Morphism A : (@fconj A) with signature Ole ++> Ole ++> Ole
as fconj_fle_compat_morph.
Save.
Hint Immediate fplus_le_compat fplus_eq_compat fesp_le_compat fesp_eq_compat
```

fmult_le_compat fmult_eq_compat fminus_le_compat fminus_eq_compat
fconj_eq_compat.

Hint Resolve finv_eq_compat.

4.21.2 Elementary properties

Lemma fle_fplus_left : $\forall$ (A:Type) (f g : MF A), $f \leq$ fplus f g.

Lemma fle_fplus_right : $\forall$ (A:Type) (f g : MF A), $g \leq$ fplus f g.

Lemma fle_fmult : $\forall$ (A:Type) (k:U)(f : MF A), fmult k $f \leq f$.

Lemma fle_zero : $\forall$ (A:Type) (f : MF A), fzero $A \leq f$.

Lemma fle_one : $\forall$ (A:Type) (f : MF A), $f \leq$ fone A.

Lemma feq_finv_finv : $\forall$ (A:Type) (f : MF A), finv (finv f) $\equiv f$.

Lemma fle_fesp_left : $\forall$ (A:Type) (f g : MF A), fesp f $g \leq f$.

Lemma fle_fesp_right : $\forall$ (A:Type) (f g : MF A), fesp f $g \leq g$.

Lemma fle_fconj_left : $\forall$ (A:Type) (f g : MF A), fconj f $g \leq f$.

Lemma fle_fconj_right : $\forall$ (A:Type) (f g : MF A), fconj f $g \leq g$.

Lemma fconj_decomp : $\forall$ A (f g : MF A),
$f \equiv$ fplus (fconj f g) (fconj f (finv g)).
Hint Resolve fconj_decomp.

4.21.3 Compatibility of addition of two functions

Definition fplusok (A:Type) (f g : MF A) := $f \leq$ finv g.
Hint Unfold fplusok.

Lemma fplusok_sym : $\forall$ (A:Type) (f g : MF A) , fplusok f $g \rightarrow$ fplusok g f.
Hint Immediate fplusok_sym.

Lemma fplusok_inv : $\forall$ (A:Type) (f : MF A) , fplusok f (finv f).
Hint Resolve fplusok_inv.

Lemma fplusok_le_compat : $\forall$ (A:Type)($f1$ $f2$ $g1$ $g2$:MF A),
fplusok $f2$ $g2 \rightarrow f1 \leq f2 \rightarrow g1 \leq g2 \rightarrow$ fplusok $f1$ $g1$.

Hint Resolve fle_fplus_left fle_fplus_right fle_zero fle_one feq_finv_finv finv_le_compat
fle_fmult fle_fesp_left fle_fesp_right fle_fconj_left fle_fconj_right.

Lemma fconj_fplusok : $\forall$ (A:Type)(f g h:MF A),
fplusok g $h \rightarrow$ fplusok (fconj f g) (fconj f h).
Hint Resolve fconj_fplusok.

Definition Fconj A : MF A -m> MF A -m> MF A := mon2 (fconj (A:=A)).

Lemma Fconj_simpl : $\forall$ A f g, Fconj A f g = fconj f g.

Lemma fconj_sym : $\forall$ A (f g : MF A), fconj f $g \equiv$ fconj g f.
Hint Resolve fconj_sym.

Lemma Fconj_sym : $\forall$ A (f g : MF A), Fconj A f $g \equiv$ Fconj A g f.
Hint Resolve Fconj_sym.

Lemma lub_MF_simpl : $\forall$ A (h : **nat** -m> MF A) (x:A), lub h x = lub (h <o> x).

Instance fconj_continuous2 A : **continuous2** (Fconj A).
Save.

Definition Fmult A : U -m> MF A -m> MF A := mon2 (fmult (A:=A)).

Lemma Fmult_simpl : $\forall$ A k f, Fmult A k f = fmult k f.

Lemma fmult_continuous2 : $\forall\ A$, **continuous2** (Fmult A).

Lemma Umult_sym_cst:
$\forall\ A$: Type,
$\forall\ (k : U)\ (f : \mathsf{MF}\ A)$, (fun $x : A \Rightarrow f\ x \times k$) $\equiv$ (fun $x : A \Rightarrow k \times f\ x$).

4.22 Fixpoints of functions of type $A \to U$

Section FixDef.
Variable A :Type.

Variable F : MF A -m> MF A.

Definition mufix : MF $A :=$ fixp F.

Definition G : MF A -m$\to$ MF $A :=$ lmon F.

Definition nufix : MF $A :=$ fixp (c:=MFopp A) G.

Lemma mufix_inv : $\forall\ f :$ MF A, $F\ f \le f \to$ mufix $\le f$.
Hint Resolve mufix_inv.

Lemma nufix_inv : $\forall\ f$:MF A, $f \le F\ f \to f \le$ nufix.
Hint Resolve nufix_inv.

Lemma mufix_le : mufix $\le F$ mufix.
Hint Resolve mufix_le.

Lemma nufix_sup : F nufix $\le$ nufix.
Hint Resolve nufix_sup.

Lemma mufix_eq : **continuous** $F \to$ mufix $\equiv F$ mufix.
Hint Resolve mufix_eq.

Lemma nufix_eq : **continuous** ($c1$:=MFopp A) ($c2$:=MFopp A) G $\to$ nufix $\equiv F$ nufix.
Hint Resolve nufix_eq.

End FixDef.
Hint Resolve mufix_le mufix_eq nufix_sup nufix_eq.

Definition Fcte (A:Type) (f:MF A) : MF A -m> MF $A :=$ mon (cte (MF A) f).

Lemma mufix_cte : $\forall$ (A:Type) (f:MF A), mufix (Fcte f) $\equiv f$.

Lemma nufix_cte : $\forall$ (A:Type) (f:MF A), nufix (Fcte f) $\equiv f$.

Hint Resolve mufix_cte nufix_cte.

4.23 Properties of (pseudo-)barycenter of two points

Lemma Uinv_bary :
$\forall\ a\ b\ x\ y : U$, $a \le$ [1-]$b \to$
[1-] ($a \times x$ + $b \times y$) $\equiv a \times$ [1-] x + $b \times$ [1-] y + [1-] (a + b).
Hint Resolve Uinv_bary.

Lemma Uinv_bary_le :
$\forall\ a\ b\ x\ y : U$, $a \le$ [1-]$b \to a \times$ [1-] x + $b \times$ [1-] $y \le$ [1-] ($a \times x$ + $b \times y$).
Hint Resolve Uinv_bary_le.

Lemma Uinv_bary_eq : $\forall\ a\ b\ x\ y : U$, $a \equiv$ [1-]$b \to$
[1-] ($a \times x$ + $b \times y$) $\equiv a \times$ [1-] x + $b \times$ [1-] y.
Hint Resolve Uinv_bary_eq.

Lemma bary_refl_eq : $\forall\ a\ b\ x$, $a \equiv$ [1-]$b \to a \times x$ + $b \times x \equiv x$.
Hint Resolve bary_refl_eq.

Lemma bary_refl_feq : $\forall\ A\ a\ b$ (f:$A \to U$) ,
$a \equiv$ [1-]$b \to$ (fun $x \Rightarrow a \times f\ x$ + $b \times f\ x$) $\equiv f$.

```
Hint Resolve bary_refl_feq.
Lemma bary_le_left : ∀ a b x y, [1-] b ≤ a → x ≤ y → x ≤ a × x + b × y.
Lemma bary_le_right : ∀ a b x y, a ≤ [1-] b → x ≤ y → a × x + b × y ≤ y.
Hint Resolve bary_le_left bary_le_right.
Lemma bary_up_eq : ∀ a b x y : U, a ≡ [1-] b → x ≤ y → a × x + b × y ≡ x + b × (y - x).
Lemma bary_up_le : ∀ a b x y : U, a ≤ [1-] b → a × x + b × y ≤ x + b × (y - x).
Lemma bary_anti_mon : ∀ a b a' b' x y : U,
  a ≡ [1-] b → a' ≡ [1-] b' → a ≤ a' → x ≤ y → a' × x + b' × y ≤ a × x + b × y.
Hint Resolve bary_anti_mon.
Lemma bary_Uminus_left :
  ∀ a b x y : U, a ≤ [1-] b → (a × x + b × y) - x ≤ b × (y - x).
Lemma bary_Uminus_left_eq :
  ∀ a b x y : U, a ≡ [1-] b → x ≤ y → (a × x + b × y) - x ≡ b × (y - x).
Lemma Uminus_bary_left
  : ∀ a b x y : U, [1-] a ≤ b → x - (a × x + b × y) ≤ b × (x - y).
Lemma Uminus_bary_left_eq
  : ∀ a b x y : U, a ≡ [1-] b → y ≤ x → x - (a × x + b × y) ≡ b × (x - y).
Hint Resolve bary_up_eq bary_up_le bary_Uminus_left Uminus_bary_left bary_Uminus_left_eq Uminus_bary_left_eq.
Lemma bary_le_simpl_right
    : ∀ a b x y : U, a ≡ [1-] b → 0 ≢ a → a × x + b × y ≤ y → x ≤ y.
Lemma bary_le_simpl_left
    : ∀ a b x y : U, a ≡ [1-] b → 0 ≢ b → x ≤ a × x + b × y → x ≤ y.
Lemma diff_bary_left_eq
  : ∀ a b x y : U, a ≡ [1-] b → diff x (a × x + b × y) ≡ b × diff x y.
Hint Resolve diff_bary_left_eq.
Lemma Uinv_half_bary :
  ∀ x y : U, [1-] (½ × x + ½ × y) ≡ ½ × [1-] x + ½ × [1-] y.
Hint Resolve Uinv_half_bary.
```

4.24 Properties of generalized sums *sigma*

```
Lemma sigma_plus : ∀ (f g : nat → U) (n:nat),
  sigma (fun k ⇒ (f k) + (g k)) n ≡ sigma f n + sigma g n.
Definition retract (f : nat → U) (n : nat) := ∀ k, (k < n)%nat → f k ≤ [1-] (sigma f k).
Lemma retract0 : ∀ (f : nat → U), retract f 0.
Lemma retract_pred : ∀ (f : nat → U) (n : nat), retract f (S n) → retract f n.
Lemma retractS: ∀ (f : nat → U) (n : nat), retract f (S n) → f n ≤ [1-] (sigma f n).
Hint Immediate retract_pred retractS.
Lemma retractS_inv :
    ∀ (f : nat → U) (n : nat), retract f (S n) → sigma f n ≤ [1-] f n.
Hint Immediate retractS_inv.
Lemma retractS_intro: ∀ (f : nat → U) (n : nat),
  retract f n → f n ≤ [1-] (sigma f n) → retract f (S n).
Hint Resolve retract0 retractS_intro.
Lemma retract_lt : ∀ (f : nat → U) (n : nat), sigma f n < 1 → retract f n.
Lemma retract_unif :
```

```
  ∀ (f : nat → U) (n : nat),
        (∀ k, (k≤n)%nat → f k ≤ [1/]1+n) → retract f (S n).
Hint Resolve retract_unif.

Lemma sigma_mult :
  ∀ (f : nat → U) n c, retract f n → sigma (fun k ⇒ c × (f k)) n ≡ c × (sigma f n).
Hint Resolve sigma_mult.

Lemma sigma_prod_maj : ∀ (f g : nat → U) n,
   sigma (fun k ⇒ (f k) × (g k)) n ≤ sigma f n.

Hint Resolve sigma_prod_maj.

Lemma sigma_prod_le : ∀ (f g : nat → U) (c:U), (∀ k, (f k) ≤ c)
   → ∀ n, retract g n → sigma (fun k ⇒ (f k) × (g k)) n ≤ c × (sigma g n).

Lemma sigma_prod_ge : ∀ (f g : nat → U) (c:U), (∀ k, c ≤ (f k))
   → ∀ n, (retract g n) → c × (sigma g n) ≤ (sigma (fun k ⇒ (f k) × (g k)) n).

Hint Resolve sigma_prod_maj sigma_prod_le sigma_prod_ge.

Lemma sigma_inv : ∀ (f g : nat → U) (n:nat), (retract f n) →
  [1-] (sigma (fun k ⇒ f k × g k) n) ≡ (sigma (fun k ⇒ f k × [1-] (g k)) n) + [1-] (sigma f n).
```

4.25 Product by an integer

4.25.1 Definition of *Nmult n x* written *n */ x*

```
Fixpoint Nmult (n: nat) (x : U) {struct n} : U :=
   match n with O ⇒ 0 | (S O) ⇒ x | S p ⇒ x + (Nmult p x) end.
```

4.25.2 Condition for *n */ x* to be exact : $n = 0$ or $x \leq 1/n$

```
Definition Nmult_def (n: nat) (x : U) :=
   match n with O ⇒ True | S p ⇒ x ≤ [1/]1+p end.

Lemma Nmult_def_O : ∀ x, Nmult_def O x.
Hint Resolve Nmult_def_O.

Lemma Nmult_def_1 : ∀ x, Nmult_def (S O) x.
Hint Resolve Nmult_def_1.

Lemma Nmult_def_intro : ∀ n x , x ≤ [1/]1+n → Nmult_def (S n) x.
Hint Resolve Nmult_def_intro.

Lemma Nmult_def_Unth: ∀ n , Nmult_def (S n) ([1/]1+n).
Hint Resolve Nmult_def_Unth.

Lemma Nmult_def_pred : ∀ n x, Nmult_def (S n) x → Nmult_def n x.

Hint Immediate Nmult_def_pred.

Lemma Nmult_defS : ∀ n x, Nmult_def (S n) x → x ≤ [1/]1+n.
Hint Immediate Nmult_defS.

Lemma Nmult_def_class : ∀ n p, class (Nmult_def n p).
Hint Resolve Nmult_def_class.

Infix "*/" := Nmult (at level 60) : U_scope.

Add Morphism Nmult_def with signature eq ⟹ Oeq ⟹ iff as Nmult_def_eq_compat.
Save.

Lemma Nmult_def_zero : ∀ n, Nmult_def n 0.
Hint Resolve Nmult_def_zero.
```

4.25.3 Properties of n */ x

Lemma Nmult_0 : $\forall\ (x{:}U),\ 0*/x = 0$.

Lemma Nmult_1 : $\forall\ (x{:}U),\ (S\ 0)*/x = x$.

Lemma Nmult_zero : $\forall\ n,\ n\ */\ 0 \equiv 0$.

Lemma Nmult_SS : $\forall\ (n{:}\mathbf{nat})\ (x{:}U),\ S\ (S\ n)\ */x = x + (S\ n\ */\ x)$.

Lemma Nmult_2 : $\forall\ (x{:}U),\ 2*/x = x + x$.

Lemma Nmult_S : $\forall\ (n{:}\mathbf{nat})\ (x{:}U),\ S\ n\ */\ x \equiv x + (n*/x)$.

Hint Resolve Nmult_1 Nmult_SS Nmult_2 Nmult_S.

Add *Morphism* Nmult with *signature* **eq** $\Longrightarrow$ Oeq $\Longrightarrow$ Oeq as *Nmult_eq_compat*.
Save.
Hint Immediate Nmult_eq_compat.

Lemma Nmult_eq_compat_left : $\forall\ (n{:}\mathbf{nat})\ (x\ y{:}U),\ x \equiv y \to n\ */\ x \equiv n\ */\ y$.

Lemma Nmult_eq_compat_right : $\forall\ (n\ m{:}\mathbf{nat})\ (x{:}U),\ (n = m)\%nat \to n\ */\ x \equiv m\ */\ x$.
Hint Resolve Nmult_eq_compat_right.

Lemma Nmult_le_compat_right : $\forall\ n\ x\ y,\ x \leq y \to n\ */\ x \leq n\ */\ y$.

Lemma Nmult_le_compat_left : $\forall\ n\ m\ x,\ (n \leq m)\%nat \to n\ */\ x \leq m\ */\ x$.

Hint Resolve Nmult_eq_compat_right Nmult_le_compat_right Nmult_le_compat_left.

Lemma Nmult_le_compat : $\forall\ (n\ m{:}\mathbf{nat})\ x\ y,\ n \leq m \to x \leq y \to n\ */\ x \leq m\ */\ y$.
Hint Immediate Nmult_le_compat.

Instance Nmult_mon2 : **monotonic2** Nmult.
Save.

Definition NMult : **nat** -m> U -m> U :=mon2 Nmult.

Lemma Nmult_sigma : $\forall\ (n{:}\mathbf{nat})\ (x{:}U),\ n\ */\ x \equiv$ sigma (fun $k \Rightarrow x$) n.

Hint Resolve Nmult_sigma.

Lemma Nmult_Unth_prop : $\forall\ n{:}\mathbf{nat},\ [1/]1{+}n \equiv [1\text{-}]\ (n*/\ ([1/]1{+}n))$.
Hint Resolve Nmult_Unth_prop.

Lemma Nmult_n_Unth: $\forall\ n{:}\mathbf{nat},\ n\ */\ [1/]1{+}n \equiv [1\text{-}]\ ([1/]1{+}n)$.

Lemma Nmult_Sn_Unth: $\forall\ n{:}\mathbf{nat},\ S\ n\ */\ [1/]1{+}n \equiv 1$.

Hint Resolve Nmult_n_Unth Nmult_Sn_Unth.

Lemma Nmult_ge_Sn_Unth: $\forall\ n\ k,\ (S\ n \leq k)\%nat \to k\ */\ [1/]1{+}n \equiv 1$.

Lemma Nmult_le_n_Unth: $\forall\ n\ k,\ (k \leq n)\%nat \to k\ */\ [1/]1{+}n \leq [1\text{-}]\ ([1/]1{+}n)$.

Hint Resolve Nmult_ge_Sn_Unth Nmult_le_n_Unth.

Lemma Nmult_Umult_assoc_left : $\forall\ n\ x\ y,$ Nmult_def $n\ x \to n*/(x\times y) \equiv (n*/x)*y$.

Hint Resolve Nmult_Umult_assoc_left.

Lemma Nmult_Umult_assoc_right : $\forall\ n\ x\ y,$ Nmult_def $n\ y \to n*/(x\times y) \equiv x*(n*/y)$.

Hint Resolve Nmult_Umult_assoc_right.

Lemma plus_Nmult_distr : $\forall\ n\ m\ x,\ (n + m)\ */\ x \equiv (n\ */\ x) + (m\ */\ x)$.

Lemma Nmult_Uplus_distr : $\forall\ n\ x\ y,\ n\ */\ (x + y) \equiv (n\ */\ x) + (n\ */\ y)$.

Lemma Nmult_mult_assoc : $\forall\ n\ m\ x,\ (n \times m)\ */\ x \equiv n\ */\ (m\ */\ x)$.

Lemma Nmult_Unth_simpl_left : $\forall\ n\ x,\ (S\ n)\ */\ ([1/]1{+}n \times x) \equiv x$.

Lemma Nmult_Unth_simpl_right : $\forall\ n\ x,\ (S\ n)\ */\ (x \times [1/]1{+}n) \equiv x$.

Hint Resolve Nmult_Umult_assoc_right plus_Nmult_distr Nmult_Uplus_distr
Nmult_mult_assoc Nmult_Unth_simpl_left Nmult_Unth_simpl_right.

Lemma Uinv_Nmult : $\forall\ k\ n$, [1-] (k */ [1/]1+n) $\equiv$ ((S n) - k) */ [1/]1+n.

Lemma Nmult_neq_zero : $\forall\ n\ x$, 0$\not\equiv x \to 0 \not\equiv$S n */ x.
Hint Resolve Nmult_neq_zero.

Lemma Nmult_le_simpl : $\forall$ (n:**nat**) ($x\ y$:U),
Nmult_def (S n) $x \to$ Nmult_def (S n) $y \to$ (S n */ x) $\leq$ (S n */ y) $\to x \leq y$.

Lemma Nmult_Unth_le : $\forall$ ($n1\ n2\ m1\ m2$:**nat**),
($n2 \times$ S $n1 \leq m2 \times$ S $m1$)%$nat \to n2$ */ [1/]1+$m1 \leq m2$ */ [1/]1+$n1$.

Lemma Nmult_Unth_eq :
$\forall$ ($n1\ n2\ m1\ m2$:**nat**),
($n2 \times$ S $n1 = m2 \times$ S $m1$)%$nat \to n2$ */ [1/]1+$m1 \equiv m2$ */ [1/]1+$n1$.

Hint Resolve Nmult_Unth_le Nmult_Unth_eq.

Lemma Nmult_Unth_factor :
$\forall$ ($n\ m1\ m2$:**nat**),
($n \times$ S $m2 =$ S $m1$)%$nat \to n$ */ [1/]1+$m1 \equiv$ [1/]1+$m2$.
Hint Resolve Nmult_Unth_factor.

Lemma Nmult_def_lt : $\forall\ n\ x$, n */ x <1 $\to$ Nmult_def $n\ x$.

Hint Immediate Nmult_def_lt.

4.26 Conversion from booleans to U

Definition B2U :MF **bool** := fun (b:**bool**) $\Rightarrow$ if b then 1 else 0.

Definition NB2U :MF **bool** := fun (b:**bool**) $\Rightarrow$ if b then 0 else 1.

Lemma B2Uinv : NB2U $\equiv$ finv B2U.

Lemma NB2Uinv : B2U $\equiv$ finv NB2U.

Hint Resolve B2Uinv NB2Uinv.

4.27 Particular sequences

$pmin\ p\ n = p$ - $\frac{1}{2}$ ^ n

Definition pmin (p:U) (n:**nat**) := p - ($\frac{1}{2}$ ^ n).

Add *Morphism* pmin with *signature* Oeq $\Longrightarrow$ **eq** $\Longrightarrow$ Oeq as *pmin_eq_compat*.
Save.

4.27.1 Properties of *pmin*

Lemma pmin_esp_S : $\forall\ p\ n$, pmin (p & p) $n \equiv$ pmin p (S n) & pmin p (S n).

Lemma pmin_esp_le : $\forall\ p\ n$, pmin p (S n) $\leq \frac{1}{2} \times$ (pmin (p & p) n) + $\frac{1}{2}$.

Lemma pmin_plus_eq : $\forall\ p\ n$, $p \leq \frac{1}{2} \to$ pmin p (S n) $\equiv \frac{1}{2} \times$ (pmin (p + p) n).

Lemma pmin_0 : $\forall\ p$:U, pmin p O $\equiv 0$.

Lemma pmin_le : $\forall$ (p:U) (n:**nat**), p - ([1/]1+n) $\leq$ pmin $p\ n$.

Hint Resolve pmin_0 pmin_le.

Lemma pmin_le_compat : $\forall\ p$ ($n\ m$: **nat**), $n \leq m \to$ pmin $p\ n \leq$ pmin $p\ m$.
Hint Resolve pmin_le_compat.

Instance pmin_mon : $\forall\ p$, **monotonic** (pmin p).
Save.

Definition Pmin (p:U) :**nat** -m> U := mon (pmin p).

Lemma le_p_lim_pmin : $\forall\ p$, $p \leq$ lub (Pmin p).

```
Lemma le_lim_pmin_p : ∀ p, lub (Pmin p) ≤ p.
Hint Resolve le_p_lim_pmin le_lim_pmin_p.
Lemma eq_lim_pmin_p : ∀ p, lub (Pmin p) ≡ p.
Hint Resolve eq_lim_pmin_p.
```

Particular case where p = 1

```
Definition U1min := Pmin 1.
Lemma eq_lim_U1min : lub U1min ≡ 1.
Lemma U1min_S : ∀ n, U1min (S n) ≡ 1/2*(U1min n) + 1/2.
Lemma U1min_0 : U1min O ≡ 0.
Hint Resolve eq_lim_U1min U1min_S U1min_0.
Lemma glb_half_exp : glb (UExp 1/2) ≡ 0.
Hint Resolve glb_half_exp.
Lemma Ule_lt_half_exp : ∀ x y, (∀ p, x ≤ y + 1/2^p) → x ≤ y.
Lemma half_exp_le_half : ∀ p, 1/2^(S p) ≤ 1/2.
Hint Resolve half_exp_le_half.
Lemma twice_half_exp : ∀ p, 1/2^(S p) + 1/2^(S p) ≡ 1/2^p.
Hint Resolve twice_half_exp.
```

4.27.2 Dyadic numbers

```
Fixpoint exp2 (n:nat) : nat :=
  match n with O ⇒ (1%nat) | S p ⇒ (2 × (exp2 p))%nat end.
Lemma exp2_pos : ∀ n, (O < exp2 n)%nat.
Hint Resolve exp2_pos.
Lemma S_pred_exp2 : ∀ n, S (pred (exp2 n))=exp2 n.
Hint Resolve S_pred_exp2.
Notation "k /2^ p" := (k */ (1/2)^p) (at level 35, no associativity).
Lemma Unth_eq : ∀ n p, n*/ p ≡ [1-]p → p ≡ [1/]1+n.
Lemma Unth_half : ∀ n, (O<n)%nat → [1/]1+(pred (n+n)) ≡ 1/2 × [1/]1+pred n.
Lemma Unth_exp2 : ∀ p, 1/2^p ≡ [1/]1+pred (exp2 p).
Hint Resolve Unth_exp2.
Lemma Nmult_exp2 : ∀ p, (exp2 p)/2^p ≡ 1.
Hint Resolve Nmult_exp2.
Section Sequence.
Variable k : U.
Hypothesis kless1 : k < 1.
Lemma Ult_one_inv_zero : /0 ≡ [1-]k.
Hint Resolve Ult_one_inv_zero.
Lemma Umult_simpl_zero : ∀ x, x ≤ k × x → x ≡ 0.
Lemma Umult_simpl_one : ∀ x, k × x + [1-]k ≤ x → x ≡ 1.
Lemma bary_le_compat : ∀ k' x y, x ≤ y → k ≤ k' → k' × x + [1-]k' × y ≤ k × x + [1-]k × y.
Lemma bary_one_le_compat : ∀ k' x, k ≤ k' → k' × x + [1-]k' ≤ k × x + [1-]k.
Lemma glb_exp_0 : glb (UExp k) ≡ 0.
Instance Uinvexp_mon : monotonic (fun n ⇒ [1-]k ^ n).
Save.
```

```
Lemma lub_inv_exp_1 : mlub (fun n ⇒ [1-]k ^ n) ≡ 1.
```

```
End Sequence.
Hint Resolve glb_exp_0 lub_inv_exp_1 bary_one_le_compat bary_le_compat.
```

4.28 Tactic for simplification of goals

```
Ltac Usimpl := match goal with
    ⊢ context [(Uplus 0 ?x)] ⇒ setoid_rewrite (Uplus_zero_left x)
  | ⊢ context [(Uplus ?x 0)] ⇒ setoid_rewrite (Uplus_zero_right x)
  | ⊢ context [(Uplus 1 ?x)] ⇒ setoid_rewrite (Uplus_one_left x)
  | ⊢ context [(Uplus ?x 1)] ⇒ setoid_rewrite (Uplus_one_right x)
  | ⊢ context [(Umult 0 ?x)] ⇒ setoid_rewrite (Umult_zero_left x)
  | ⊢ context [(Umult ?x 0)] ⇒ setoid_rewrite (Umult_zero_right x)
  | ⊢ context [(Umult 1 ?x)] ⇒ setoid_rewrite (Umult_one_left x)
  | ⊢ context [(Umult ?x 1)] ⇒ setoid_rewrite (Umult_one_right x)
  | ⊢ context [(Uesp 0 ?x)] ⇒ setoid_rewrite (Uesp_zero_left x)
  | ⊢ context [(Uesp ?x 0)] ⇒ setoid_rewrite (Uesp_zero_right x)
  | ⊢ context [(Uesp 1 ?x)] ⇒ setoid_rewrite (Uesp_one_left x)
  | ⊢ context [(Uesp ?x 1)] ⇒ setoid_rewrite (Uesp_one_right x)
  | ⊢ context [(Uminus 0 ?x)] ⇒ setoid_rewrite (Uminus_zero_left x)
  | ⊢ context [(Uminus ?x 0)] ⇒ setoid_rewrite (Uminus_zero_right x)
  | ⊢ context [(Uminus ?x 1)] ⇒ setoid_rewrite (Uminus_one_right x)
  | ⊢ context [([1-] ([1-] ?x))] ⇒ setoid_rewrite (Uinv_inv x)
  | ⊢ context [ ?x + ([1-] ?x)] ⇒ setoid_rewrite (Uinv_opp_right x)
  | ⊢ context [ ([1-]?x) + ?x ] ⇒ setoid_rewrite (Uinv_opp_left x)
  | ⊢ context [([1-] 1)] ⇒ setoid_rewrite Uinv_one
  | ⊢ context [([1-] 0)] ⇒ setoid_rewrite Uinv_zero
  | ⊢ context [([1/]1+0)] ⇒ setoid_rewrite Unth_zero
  | ⊢ context [?x^0] ⇒ setoid_rewrite (Uexp_0 x)
  | ⊢ context [?x^(S 0)] ⇒ setoid_rewrite (Uexp_1 x)
  | ⊢ context [0^(?n)] ⇒ setoid_rewrite Uexp_zero; [idtac|omega]
  | ⊢ context [U1^(?n)] ⇒ setoid_rewrite Uexp_one
  | ⊢ context [(Nmult 0 ?x)] ⇒ setoid_rewrite (Nmult_0 x)
  | ⊢ context [(Nmult 1 ?x)] ⇒ setoid_rewrite (Nmult_1 x)
  | ⊢ context [(Nmult ?n 0)] ⇒ setoid_rewrite (Nmult_zero n)
  | ⊢ context [(sigma ?f 0)] ⇒ setoid_rewrite (sigma_0 f)
  | ⊢ context [(sigma ?f (S 0))] ⇒ setoid_rewrite (sigma_1 f)
  | ⊢ (Ole (Uplus ?x ?y) (Uplus ?x ?z)) ⇒ apply Uplus_le_compat_right
  | ⊢ (Ole (Uplus ?x ?z) (Uplus ?y ?z)) ⇒ apply Uplus_le_compat_left
  | ⊢ (Ole (Uplus ?x ?z) (Uplus ?z ?y)) ⇒ setoid_rewrite (Uplus_sym z y);
                                          apply Uplus_le_compat_left
  | ⊢ (Ole (Uplus ?x ?y) (Uplus ?z ?x)) ⇒ setoid_rewrite (Uplus_sym x y);
                                          apply Uplus_le_compat_left
  | ⊢ (Ole (Uinv ?y) (Uinv ?x)) ⇒ apply Uinv_le_compat
  | ⊢ (Ole (Uminus ?x ?y) (Uplus ?x ?z)) ⇒ apply Uminus_le_compat_right
  | ⊢ (Ole (Uminus ?x ?z) (Uplus ?y ?z)) ⇒ apply Uminus_le_compat_left
  | ⊢ ((Uinv ?x) ≡ (Uinv ?y)) ⇒ apply Uinv_eq_compat
  | ⊢ ((Uplus ?x ?y) ≡ (Uplus ?x ?z)) ⇒ apply Uplus_eq_compat_right
  | ⊢ ((Uplus ?x ?z) ≡ (Uplus ?y ?z)) ⇒ apply Uplus_eq_compat_left
  | ⊢ ((Uplus ?x ?z) ≡ (Uplus ?z ?y)) ⇒ setoid_rewrite (Uplus_sym z y);
                                          apply Uplus_eq_compat_left
  | ⊢ ((Uplus ?x ?y) ≡ (Uplus ?z ?x)) ⇒ setoid_rewrite (Uplus_sym x y);
                                          apply Uplus_eq_compat_left
  | ⊢ ((Uminus ?x ?y) ≡ (Uplus ?x ?z)) ⇒ apply Uminus_eq_compat;[apply Oeq_refl|idtac]
```

```
| ⊢ ((Uminus ?x ?z) ≡ (Uplus ?y ?z)) ⇒ apply Uminus_eq_compat;[idtac|apply Oeq_refl]
| ⊢ (Ole (Umult ?x ?y) (Umult ?x ?z)) ⇒ apply Umult_le_compat_right
| ⊢ (Ole (Umult ?x ?z) (Umult ?y ?z)) ⇒ apply Umult_le_compat_left
| ⊢ (Ole (Umult ?x ?z) (Umult ?z ?y)) ⇒ setoid_rewrite (Umult_sym z y);
                                        apply Umult_le_compat_left
| ⊢ (Ole (Umult ?x ?y) (Umult ?z ?x)) ⇒ setoid_rewrite (Umult_sym x y);
                                        apply Umult_le_compat_left
| ⊢ ((Umult ?x ?y) ≡ (Umult ?x ?z)) ⇒ apply Umult_eq_compat_right
| ⊢ ((Umult ?x ?z) ≡ (Umult ?y ?z)) ⇒ apply Umult_eq_compat_left
| ⊢ ((Umult ?x ?z) ≡ (Umult ?z ?y)) ⇒ setoid_rewrite (Umult_sym z y);
                                        apply Umult_eq_compat_left
| ⊢ ((Umult ?x ?y) ≡ (Umult ?z ?x)) ⇒ setoid_rewrite (Umult_sym x y);
                                        apply Umult_eq_compat_left
end.

Ltac Ucompute :=
 first [setoid_rewrite Uplus_zero_left |
        setoid_rewrite Uplus_zero_right |
        setoid_rewrite Uplus_one_left |
        setoid_rewrite Uplus_one_right |
        setoid_rewrite Umult_zero_left |
        setoid_rewrite Umult_zero_right |
        setoid_rewrite Umult_one_left |
        setoid_rewrite Umult_one_right |
        setoid_rewrite Uesp_zero_left |
        setoid_rewrite Uesp_zero_right |
        setoid_rewrite Uesp_one_left |
        setoid_rewrite Uesp_one_right |
        setoid_rewrite Uminus_zero_left |
        setoid_rewrite Uminus_zero_right |
        setoid_rewrite Uminus_one_right |
        setoid_rewrite Uinv_inv |
        setoid_rewrite Uinv_opp_right |
        setoid_rewrite Uinv_opp_left |
        setoid_rewrite Uinv_one |
        setoid_rewrite Uinv_zero |
        setoid_rewrite Unth_zero |
        setoid_rewrite Uexp_0 |
        setoid_rewrite Uexp_1 |
        (setoid_rewrite Uexp_zero; [idtac|omega]) |
        setoid_rewrite Uexp_one |
        setoid_rewrite Nmult_0 |
        setoid_rewrite Nmult_1 |
        setoid_rewrite Nmult_zero |
        setoid_rewrite sigma_0 |
        setoid_rewrite sigma_1
].
```

Properties of current values *Notation* "[1/3]" := (*Unth* 2%*nat*).
Notation "$\frac{1}{4}$" := (*Unth* 3%*nat*).
Notation "[1/8]" := (*Unth* 7).
Notation "$\frac{3}{4}$" := (*Uinv* $\frac{1}{4}$).

Lemma half_square : $\frac{1}{2}\times\frac{1}{2}\equiv\frac{1}{4}$.

Lemma half_cube : $\frac{1}{2}\times\frac{1}{2}\times\frac{1}{2}\equiv$[1/8].

Lemma three_quarter_decomp : $\frac{3}{4}\equiv\frac{1}{2}+\frac{1}{4}$.

Equality is not true, even for monotonic sequences fot instance n/m

```
Lemma Ulub_Uglb_exch_le : ∀ f : nat → nat → U,
    Ulub (fun n ⇒ Uglb (fun m ⇒ f n m)) ≤ Uglb (fun m ⇒ Ulub (fun n ⇒ f n m)).
```

4.29 Intervals

4.29.1 Definition

```
Record IU : Type := mk_IU {low:U; up:U; proper:low ≤ up}.
Hint Resolve proper.
```

the all set : [0,1] `Definition full := mk_IU 0 1 (Upos 1).`
singleton : [x] `Definition singl (x:U) := mk_IU x x (Ole_refl x).`
down segment : [0,x] `Definition inf (x:U) := mk_IU 0 x (Upos x).`
up segment : [x,1] `Definition sup (x:U) := mk_IU x 1 (Unit x).`

4.29.2 Relations

```
Definition lin (x:U) (I:IU) := low I ≤ x ∧ x ≤ up I.
Definition lincl I J := low J ≤ low I ∧ up I ≤ up J.
Definition leq I J := low I ≡ low J ∧ up I ≡ up J.
Hint Unfold lin lincl leq.
```

4.29.3 Properties

```
Lemma lin_low : ∀ I, lin (low I) I.
Lemma lin_up : ∀ I, lin (up I) I.
Hint Resolve lin_low lin_up.
Lemma lin_singl_elim : ∀ x y, lin x (singl y) → x ≡ y.
Lemma lin_inf_elim : ∀ x y, lin x (inf y) → x ≤ y.
Lemma lin_sup_elim : ∀ x y, lin x (sup y) → y ≤ x.
Lemma lin_singl_intro : ∀ x y, x ≡ y → lin x (singl y).
Lemma lin_inf_intro : ∀ x y, x ≤ y → lin x (inf y).
Lemma lin_sup_intro : ∀ x y, y ≤ x → lin x (sup y).
Hint Immediate lin_inf_elim lin_sup_elim lin_singl_elim.
Hint Resolve lin_inf_intro lin_sup_intro lin_singl_intro.
Lemma lin_class : ∀ I x, class (lin x I).
Lemma lincl_class : ∀ I J, class (lincl I J).
Lemma leq_class : ∀ I J, class (leq I J).
Hint Resolve lin_class lincl_class leq_class.
Lemma lincl_in : ∀ I J, lincl I J → ∀ x, lin x I → lin x J.
Lemma lincl_low : ∀ I J, lincl I J → low J ≤ low I.
Lemma lincl_up : ∀ I J, lincl I J → up I ≤ up J.
Hint Immediate lincl_low lincl_up.
Lemma lincl_refl : ∀ I, lincl I I.
Hint Resolve lincl_refl.
Lemma lincl_trans : ∀ I J K, lincl I J → lincl J K → lincl I K.
Instance IUord : ord IU := {Oeq := fun I J ⇒ leq I J; Ole := fun I J ⇒ lincl J I}.
```

```
Defined.

Lemma low_le_compat : ∀ I J:IU, I ≤ J → low I ≤ low J.

Lemma up_le_compat : ∀ I J : IU, I ≤ J → up J ≤ up I.

Instance low_mon : monotonic low.
Save.

Definition Low : IU -m> U := mon low.

Instance up_mon : monotonic (o2:=lord U) up.
Save.

Definition Up : IU -m→ U := mon (o2:=lord U) up.

Lemma leq_incl : ∀ I J, leq I J → lincl I J.

Lemma leq_incl_sym : ∀ I J, leq I J → lincl J I.
Hint Immediate leq_incl leq_incl_sym.

Lemma lincl_eq_compat : ∀ I J K L,
      leq I J → lincl J K → leq K L → lincl I L.

Lemma lincl_eq_trans : ∀ I J K,
      lincl I J → leq J K → lincl I K.

Lemma leq_incl_trans : ∀ I J K,
      leq I J → lincl J K → lincl I K.

Lemma lincl_antisym : ∀ I J, lincl I J → lincl J I → leq I J.
Hint Immediate lincl_antisym.

Lemma leq_refl : ∀ I, leq I I.
Hint Resolve leq_refl.

Lemma leq_sym : ∀ I J, leq I J → leq J I.
Hint Immediate leq_sym.

Lemma leq_trans : ∀ I J K, leq I J → leq J K → leq I K.

Lemma lsingl_eq : ∀ x y, lincl (singl x) (singl y) → x≡y.
Hint Immediate lsingl_eq.

Lemma lincl_full : ∀ I, lincl I full.
Hint Resolve lincl_full.
```

4.29.4 Operations on intervals

```
Definition lplus I J := mk_IU (low I + low J) (up I + up J)
                                                (Uplus_le_compat _ _ _ _ (proper I) (proper J)).

Lemma low_lplus : ∀ I J, low (lplus I J)=low I + low J.

Lemma up_lplus : ∀ I J, up (lplus I J)=up I + up J.

Lemma lplus_in : ∀ I J x y, lin x I → lin y J → lin (x+y) (lplus I J).

Lemma lplus_in_elim :
∀ I J z, low I ≤ [1-]up J → lin z (lplus I J)
                  → exc (fun x ⇒ lin x I ∧
                                                    exc (fun y ⇒ lin y J ∧ z≡x+y)).

Definition lmult I J := mk_IU (low I × low J) (up I × up J)
                                                (Umult_le_compat _ _ _ _ (proper I) (proper J)).

Lemma low_lmult : ∀ I J, low (lmult I J) = low I × low J.

Lemma up_lmult : ∀ I J, up (lmult I J) = up I × up J.

Definition lmultk p I := mk_IU (p × low I) (p × up I) (Umult_le_compat_right p _ _ (proper I)).
```

Lemma low_lmultk : $\forall\ p\ I$, low (lmultk $p\ I$) = $p \times$ low I.

Lemma up_lmultk : $\forall\ p\ I$, up (lmultk $p\ I$) = $p \times$ up I.

Lemma lmult_in : $\forall\ I\ J\ x\ y$, lin $x\ I \to$ lin $y\ J \to$ lin $(x \times y)$ (lmult $I\ J$).

Lemma lmultk_in : $\forall\ p\ I\ x$, lin $x\ I \to$ lin $(p \times x)$ (lmultk $p\ I$).

4.29.5 Limits of intervals

Definition llim : $\forall\ I$: **nat** -m> **IU**, **IU**.
Defined.

Lemma low_llim : $\forall$ (I:**nat** -m> **IU**), low (llim I) = lub (Low @ I).

Lemma up_llim : $\forall$ (I:**nat** -m> **IU**), up (llim I) = glb (Up @ I).

Lemma lim_lincl : $\forall$ (I:**nat** -m> **IU**) n, lincl (llim I) ($I\ n$).
Hint Resolve lim_lincl.

Lemma lincl_lim : $\forall\ J$ (I:**nat** -m>**IU**), ($\forall\ n$, lincl J ($I\ n$)) $\to$ lincl J (llim I).

Lemma llim_incl_stable : $\forall$ ($I\ J$:**nat** -m> **IU**), ($\forall\ n$, lincl ($I\ n$) ($J\ n$)) $\to$ lincl (llim I) (llim J).
Hint Resolve llim_incl_stable.

Instance IUcpo : **cpo IU** := {D0:=full; lub:=llim}.
Defined.

4.30 Limits inf and sup

Definition fsup (f:**nat**$\to U$) (n:**nat**) := Ulub (fun $k \Rightarrow f\ (n+k)\%nat$).

Definition finf (f:**nat**$\to U$) (n:**nat**) := Uglb (fun $k \Rightarrow f\ (n+k)\%nat$).

Lemma fsup_incr : $\forall$ (f:**nat**$\to U$) n, fsup f (S n) $\leq$ fsup $f\ n$.
Hint Resolve fsup_incr.

Lemma finf_incr : $\forall$ (f:**nat**$\to U$) n, finf $f\ n \leq$ finf f (S n).

Hint Resolve finf_incr.

Instance fsup_mon : $\forall\ f$, **monotonic** ($o2$:=lord U) (fsup f).
Save.

Instance finf_mon : $\forall\ f$, **monotonic** (finf f).
Save.

Definition Fsup (f:**nat**$\to U$) : **nat** -m$\to$ U := mon (fsup f).
Definition Finf (f:**nat**$\to U$) : **nat** -m> U := mon (finf f).

Lemma fn_fsup : $\forall\ f\ n, f\ n \leq$ fsup $f\ n$.
Hint Resolve fn_fsup.

Lemma finf_fn : $\forall\ f\ n$, finf $f\ n \leq f\ n$.
Hint Resolve finf_fn.

Definition limsup f := glb (Fsup f).
Definition liminf f := lub (Finf f).

Lemma le_liminf_sup : $\forall\ f$, liminf $f \leq$ limsup f.

Hint Resolve le_liminf_sup.

Definition has_lim f := limsup $f \leq$ liminf f.

Lemma eq_liminf_sup : $\forall\ f$, has_lim $f \to$ liminf $f \equiv$ limsup f.

Definition cauchy f := $\forall$ (p:**nat**), exc (fun M:**nat** $\Rightarrow \forall\ n\ m$,
$(M \leq n)\%nat \to (M \leq m)\%nat \to f\ n \leq f\ m + \frac{1}{2}\hat{}p$).

Definition is_limit f (l:U) := $\forall$ (p:**nat**), exc (fun M:**nat** $\Rightarrow \forall\ n$,

$(M \leq n)\%nat \to f\ n \leq l + \frac{1}{2}\hat{}p \wedge l \leq f\ n + \frac{1}{2}\hat{}p)$.

Lemma cauchy_lim : $\forall\ f$, cauchy $f \to$ is_limit f (limsup f).

Lemma has_limit_cauchy : $\forall\ f\ l$, is_limit $f\ l \to$ cauchy f.

Lemma limit_le_unique : $\forall\ f\ l1\ l2$, is_limit $f\ l1 \to$ is_limit $f\ l2 \to l1 \leq l2$.

Lemma limit_unique : $\forall\ f\ l1\ l2$, is_limit $f\ l1 \to$ is_limit $f\ l2 \to l1 \equiv l2$.
Hint Resolve limit_unique.

Lemma has_limit_compute : $\forall\ f\ l$, is_limit $f\ l \to$ is_limit f (limsup f).

Lemma limsup_eq_mult : $\forall\ k\ (f : \mathbf{nat} \to U)$,
limsup (fun $n \Rightarrow k \times f\ n$) $\equiv k \times$ limsup f.

Lemma liminf_eq_mult : $\forall\ k\ (f : \mathbf{nat} \to U)$,
liminf (fun $n \Rightarrow k \times f\ n$) $\equiv k \times$ liminf f.

Lemma limsup_eq_plus_cte_right : $\forall\ k\ (f : \mathbf{nat} \to U)$,
limsup (fun $n \Rightarrow (f\ n) + k$) $\equiv$ limsup $f + k$.

Lemma liminf_eq_plus_cte_right : $\forall\ k\ (f : \mathbf{nat} \to U)$,
liminf (fun $n \Rightarrow (f\ n) + k$) $\equiv$ liminf $f + k$.

Lemma limsup_le_plus : $\forall\ (f\ g\colon \mathbf{nat} \to U)$,
limsup (fun $x \Rightarrow f\ x + g\ x$) $\leq$ limsup $f +$ limsup g.

Lemma liminf_le_plus : $\forall\ (f\ g\colon \mathbf{nat} \to U)$,
liminf $f +$ liminf $g \leq$ liminf (fun $x \Rightarrow f\ x + g\ x$).

Hint Resolve liminf_le_plus limsup_le_plus.

Lemma limsup_le_compat : $\forall\ f\ g : \mathbf{nat} \to U, f \leq g \to$ limsup $f \leq$ limsup g.

Lemma liminf_le_compat : $\forall\ f\ g : \mathbf{nat} \to U, f \leq g \to$ liminf $f \leq$ liminf g.

Hint Resolve limsup_le_compat liminf_le_compat.

Lemma limsup_eq_compat : $\forall\ f\ g : \mathbf{nat} \to U, f \equiv g \to$ limsup $f \equiv$ limsup g.

Lemma liminf_eq_compat : $\forall\ f\ g : \mathbf{nat} \to U, f \equiv g \to$ liminf $f \equiv$ liminf g.

Hint Resolve liminf_eq_compat limsup_eq_compat.

Lemma limsup_inv : $\forall\ f : \mathbf{nat} \to U$, limsup (fun $x \Rightarrow$ [1-]$f\ x$) $\equiv$ [1-] liminf f.

Lemma liminf_inv : $\forall\ f : \mathbf{nat} \to U$, liminf (fun $x \Rightarrow$ [1-]$f\ x$) $\equiv$ [1-] limsup f.

Hint Resolve limsup_inv liminf_inv.

4.31 Limits of arbitrary sequences

Lemma liminf_incr : $\forall\ f$:**nat** -m> U, liminf $f \equiv$ lub f.

Lemma limsup_incr : $\forall\ f$:**nat** -m> U, limsup $f \equiv$ lub f.

Lemma has_limit_incr : $\forall\ f$:**nat** -m> U, has_lim f.

Lemma liminf_decr : $\forall\ f$:**nat** -m-> U, liminf $f \equiv$ glb f.

Lemma limsup_decr : $\forall\ f$:**nat**-m->U, limsup $f \equiv$ glb f.

Lemma has_limit_decr : $\forall\ f$:**nat** -m-> U, has_lim f.

Lemma has_limit_sum : $\forall\ f\ g\colon \mathbf{nat} \to U$, has_lim $f \to$ has_lim $g \to$ has_lim (fun $x \Rightarrow f\ x + g\ x$).

Lemma has_limit_inv : $\forall\ f : \mathbf{nat} \to U$, has_lim $f \to$ has_lim (fun $x \Rightarrow$ [1-]$f\ x$).

Lemma has_limit_cte : $\forall\ c$, has_lim (fun $n \Rightarrow c$).

4.32 Definition and properties of series : infinite sums

```
Definition serie (f : nat → U) : U := lub (sigma f).

Lemma serie_le_compat : ∀ (f g: nat → U),
  (∀ k, f k ≤ g k) → serie f ≤ serie g.

Lemma serie_eq_compat : ∀ (f g: nat → U),
  (∀ k, f k ≡ g k) → serie f ≡ serie g.

Lemma serie_sigma_lift : ∀ (f :nat → U) (n:nat),
            serie f ≡ sigma f n + serie (fun k ⇒ f (n + k)%nat).

Lemma serie_S_lift : ∀ (f :nat → U),
            serie f ≡ f 0 + serie (fun k ⇒ f (S k)).

Lemma serie_zero : ∀ f, (∀ k, f k ≡0) → serie f ≡0.

Lemma serie_not_zero : ∀ f k, 0 < f k → 0 < serie f.

Lemma serie_zero_elim : ∀ f, serie f ≡ 0 → ∀ k, f k ≡0.

Hint Resolve serie_eq_compat serie_le_compat serie_zero.

Lemma serie_le : ∀ f k, f k ≤ serie f.

Lemma serie_minus_incr : ∀ f :nat -m> U, serie (fun k ⇒ f (S k) - f k) ≡ lub f - f 0.

Lemma serie_minus_decr : ∀ f : nat -m→ U,
          serie (fun k ⇒ f k - f (S k)) ≡ f 0 - glb f.

Lemma serie_plus : ∀ (f g : nat → U),
    serie (fun k ⇒ (f k) + (g k)) ≡ serie f + serie g.

Definition wretract (f : nat → U) := ∀ k, f k ≤ [1-] (sigma f k).

Lemma retract_wretract : ∀ f, (∀ n, retract f n) → wretract f.

Lemma wretract_retract : ∀ f, wretract f → ∀ n, retract f n.

Hint Resolve wretract_retract.

Lemma wretract_lt : ∀ (f : nat → U), (∀ (n : nat), sigma f n < 1) → wretract f.

Lemma retract_zero_wretract :
        ∀ f n, retract f n → (∀ k, (n ≤ k)%nat → f k ≡ 0) → wretract f.

Lemma wretract_le : ∀ f g : nat→U, f ≤ g → wretract g → wretract f.

Lemma serie_mult :
  ∀ (f : nat → U) c, wretract f → serie (fun k ⇒ c × f k) ≡ c × serie f.
Hint Resolve serie_mult.

Lemma serie_prod_maj : ∀ (f g : nat → U),
    serie (fun k ⇒ f k × g k) ≤ serie f.

Hint Resolve serie_prod_maj.

Lemma serie_prod_le : ∀ (f g : nat → U) (c:U), (∀ k, f k ≤ c)
    → wretract g → serie (fun k ⇒ f k × g k) ≤ c × serie g.

Lemma serie_prod_ge : ∀ (f g : nat → U) (c:U), (∀ k, c ≤ (f k))
    → wretract g → c × serie g ≤ serie (fun k ⇒ f k × g k).

Hint Resolve serie_prod_le serie_prod_ge.

Lemma serie_inv_le : ∀ (f g : nat → U), wretract f →
  serie (fun k ⇒ f k × [1-] (g k)) ≤ [1-] (serie (fun k ⇒ f k × g k)).

Definition Serie : (nat → U) -m> U.
Defined.

Lemma Serie_simpl : ∀ f, Serie f = serie f.

Lemma serie_continuous : continuous Serie.
```

5 Monads.v: Monads for randomized constructions

```
Require Export Uprop.
```

5.1 Definition of monadic operators as the cpo of monotonic oerators

```
Definition M (A:Type) := MF A -m> U.

Instance app_mon (A:Type) (x:A) : monotonic (fun (f:MF A) ⇒ f x).
Save.

Definition unit (A:Type) (x:A) : M A := mon (fun (f:MF A) ⇒ f x).

Definition star : ∀ (A B:Type), M A → (A → M B) → M B.
Defined.

Lemma star_simpl : ∀ (A B:Type) (a:M A) (F:A → M B)(f:MF B),
      star a F f = a (fun x ⇒ F x f).
```

5.2 Properties of monadic operators

```
Lemma law1 : ∀ (A B:Type) (x:A) (F:A → M B) (f:MF B), star (unit x) F f ≡ F x f.

Lemma law2 :
 ∀ (A:Type) (a:M A) (f:MF A), star a (fun x:A ⇒ unit x) f ≡ a (fun x:A ⇒ f x).

Lemma law3 :
 ∀ (A B C:Type) (a:M A) (F:A → M B) (G:B → M C)
   (f:MF C), star (star a F) G f ≡ star a (fun x:A ⇒ star (F x) G) f.
```

5.3 Properties of distributions

5.3.1 Expected properties of measures

```
Definition stable_inv (A:Type) (m:M A) : Prop := ∀ f :MF A, m (finv f) ≤ [1-] (m f).

Definition stable_plus (A:Type) (m:M A) : Prop :=
  ∀ f g:MF A, fplusok f g → m (fplus f g) ≡ (m f) + (m g).

Definition le_plus (A:Type) (m:M A) : Prop :=
  ∀ f g:MF A, fplusok f g → (m f) + (m g) ≤ m (fplus f g).

Definition le_esp (A:Type) (m:M A) : Prop :=
  ∀ f g: MF A, (m f) & (m g) ≤ m (fesp f g).

Definition le_plus_cte (A:Type) (m:M A) : Prop :=
  ∀ (f : MF A) (k:U), m (fplus f (fcte A k)) ≤ m f + k.

Definition stable_mult (A:Type) (m:M A) : Prop :=
  ∀ (k:U) (f:MF A), m (fmult k f) ≡ k × (m f).
```

5.3.2 Stability for equality

```
Lemma stable_minus_distr : ∀ (A:Type) (m:M A),
      stable_plus m → stable_inv m →
      ∀ (f g : MF A), g ≤ f → m (fminus f g) ≡ m f - m g.

Hint Resolve stable_minus_distr.

Lemma inv_minus_distr : ∀ (A:Type) (m:M A),
      stable_plus m → stable_inv m →
      ∀ (f : MF A), m (finv f) ≡ m (fone A) - m f.
Hint Resolve inv_minus_distr.
```

Lemma le_minus_distr : $\forall$ (A : Type)(m:M A),
$\forall$ (f g:$A \to U$), m (fminus f g) $\leq$ m f.
Hint Resolve le_minus_distr.

Lemma le_plus_distr : $\forall$ (A : Type)(m:M A),
stable_plus m $\to$ stable_inv m $\to$ $\forall$ (f g:MF A), m (fplus f g) $\leq$ m f + m g.
Hint Resolve le_plus_distr.

Lemma le_esp_distr : $\forall$ (A : Type) (m:M A),
stable_plus m $\to$ stable_inv m $\to$ le_esp m.

Lemma unit_stable_eq : $\forall$ (A:Type) (x:A), **stable** (unit x).

Lemma star_stable_eq : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B), **stable** (star m F).

Lemma unit_monotonic : $\forall$ (A:Type) (x:A) (f g : MF A),
$f \leq g \to$ unit x $f \leq$ unit x g.

Lemma star_monotonic : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B) (f g : MF B),
$f \leq g \to$ star m F $f \leq$ star m F g.

Lemma star_le_compat : $\forall$ (A B:Type) ($m1$ $m2$:M A) ($F1$ $F2$:$A \to$ M B),
$m1 \leq m2 \to F1 \leq F2 \to$ star $m1$ $F1 \leq$ star $m2$ $F2$.
Hint Resolve star_le_compat.

5.3.3 Stability for inversion

Lemma unit_stable_inv : $\forall$ (A:Type) (x:A), stable_inv (unit x).

Lemma star_stable_inv : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B),
stable_inv m $\to$ ($\forall$ a:A, stable_inv (F a)) $\to$ stable_inv (star m F).

5.3.4 Stability for addition

Lemma unit_stable_plus : $\forall$ (A:Type) (x:A), stable_plus (unit x).

Lemma star_stable_plus : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B),
stable_plus m $\to$
($\forall$ a:A, $\forall$ f g, fplusok f g $\to$ (F a f) $\leq$ *Uinv* (F a g))
$\to$ ($\forall$ a:A, stable_plus (F a)) $\to$ stable_plus (star m F).

Lemma unit_le_plus : $\forall$ (A:Type) (x:A), le_plus (unit x).

Lemma star_le_plus : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B),
le_plus m $\to$
($\forall$ a:A, $\forall$ f g, fplusok f g $\to$ (F a f) $\leq$ *Uinv* (F a g))
$\to$ ($\forall$ a:A, le_plus (F a)) $\to$ le_plus (star m F).

5.3.5 Stability for product

Lemma unit_stable_mult : $\forall$ (A:Type) (x:A), stable_mult (unit x).

Lemma star_stable_mult : $\forall$ (A B:Type) (m:M A) (F:$A \to$ M B),
stable_mult m $\to$ ($\forall$ a:A, stable_mult (F a)) $\to$ stable_mult (star m F).

5.3.6 Continuity

Lemma unit_continuous : $\forall$ (A:Type) (x:A), **continuous** (unit x).

Lemma star_continuous : $\forall$ (A B : Type) (m : M A)(F: $A \to$ M B),
continuous m $\to$ ($\forall$ x, **continuous** (F x)) $\to$ **continuous** (star m F).

6 Probas.v: The monad for distributions

```
Require Export Monads.
```

6.1 Definition of distribution

Distributions are monotonic measure functions such that

- μ (1-f) $\leq$ 1 - μ f
- $f \leq 1 - g \Rightarrow \mu\ (f+g) \equiv \mu\ f\ +\ \mu\ g$
- $\mu\ (k \times f) = k \times \mu\ (f)$
- $\mu\ (lub\ f\_n) \leq lub\ \mu\ (f\_n)$

```
Record distr (A:Type) : Type :=
  {μ : M A;
   mu_stable_inv : stable_inv μ;
   mu_stable_plus : stable_plus μ;
   mu_stable_mult : stable_mult μ;
   mu_continuous : continuous μ}.
Hint Resolve mu_stable_plus mu_stable_inv mu_stable_mult mu_continuous.
```

6.2 Properties of measures

```
Lemma mu_monotonic : ∀ (A : Type)(m: distr A), monotonic (μ m).
Hint Resolve mu_monotonic.
Implicit Arguments mu_monotonic [A].
Lemma mu_stable_eq : ∀ (A : Type)(m: distr A), stable (μ m).
Hint Resolve mu_stable_eq.
Implicit Arguments mu_stable_eq [A].
Lemma mu_zero : ∀ (A : Type)(m: distr A), μ m (fzero A) ≡ 0.
Hint Resolve mu_zero.
Lemma mu_zero_eq : ∀ (A : Type)(m: distr A) f,
    (∀ x, f x ≡ 0) → μ m f ≡ 0.
Lemma mu_one_inv : ∀ (A : Type)(m:distr A),
   μ m (fone A) ≡ 1 → ∀ f, μ m (finv f) ≡ [1-] (μ m f).
Hint Resolve mu_one_inv.
Lemma mu_fplusok : ∀ (A : Type)(m:distr A) f g, fplusok f g →
             μ m f ≤ [1-] μ m g.
Hint Resolve mu_fplusok.
Lemma mu_le_minus : ∀ (A : Type)(m:distr A) (f g:MF A),
      μ m (fminus f g) ≤ μ m f.
Hint Resolve mu_le_minus.
Lemma mu_le_plus : ∀ (A : Type)(m:distr A) (f g:MF A),
      μ m (fplus f g) ≤ μ m f + μ m g.
Hint Resolve mu_le_plus.
Lemma mu_cte : ∀ (A : Type)(m:(distr A)) (c:U),
   μ m (fcte A c) ≡ c × μ m (fone A).
Hint Resolve mu_cte.
Lemma mu_cte_le : ∀ (A : Type)(m:distr A) (c:U), μ m (fcte A c) ≤ c.
```

```
Lemma mu_cte_eq : ∀ (A : Type)(m:distr A) (c:U),
    μ m (fone A) ≡ 1 → μ m (fcte A c) ≡ c.

Hint Resolve mu_cte_le mu_cte_eq.

Lemma mu_stable_mult_right : ∀ (A : Type)(m:distr A) (c:U) (f : MF A),
    μ m (fun x ⇒ (f x) × c) ≡ (μ m f) × c.

Lemma mu_stable_minus : ∀ (A:Type) (m:distr A)(f g : MF A),
  g ≤ f → μ m (fun x ⇒ f x - g x) ≡ μ m f - μ m g.

Lemma mu_inv_minus :
∀ (A:Type) (m:distr A)(f: MF A), μ m (finv f) ≡ μ m (fone A) - μ m f.

Lemma mu_stable_le_minus : ∀ (A:Type) (m:distr A)(f g : MF A),
  μ m f - μ m g ≤ μ m (fun x ⇒ f x - g x).

Lemma mu_inv_minus_inv : ∀ (A:Type) (m:distr A)(f: MF A),
      μ m (finv f) + [1-](μ m (fone A)) ≡ [1-](μ m f).

Lemma mu_le_esp_inv : ∀ (A:Type) (m:distr A)(f g : MF A),
  ([1-]μ m (finv f)) & μ m g ≤ μ m (fesp f g).
Hint Resolve mu_le_esp_inv.

Lemma mu_stable_inv_inv : ∀ (A:Type) (m:distr A)(f : MF A),
               μ m f ≤ [1-] μ m (finv f).
Hint Resolve mu_stable_inv_inv.

Lemma mu_stable_div : ∀ (A:Type) (m:distr A)(k:U)(f : MF A),
           0≢k → f ≤ fcte A k → μ m (fdiv k f) ≡ μ m f / k.

Lemma mu_stable_div_le : ∀ (A:Type) (m:distr A)(k:U)(f : MF A),
           0≢k → μ m (fdiv k f) ≤ μ m f / k.

Lemma mu_le_esp : ∀ (A:Type) (m:distr A)(f g : MF A),
  μ m f & μ m g ≤ μ m (fesp f g).
Hint Resolve mu_le_esp.

Lemma mu_esp_one : ∀ (A:Type)(m:distr A)(f g:MF A),
    1≤μ m f → μ m g ≡ μ m (fesp f g).

Lemma mu_esp_zero : ∀ (A:Type)(m:distr A)(f g:MF A),
    μ m (finv f) ≤ 0 → μ m g ≡ μ m (fesp f g).

Lemma mu_stable_mult2:
  ∀ (A : Type) (d : distr A), ∀ (k : U)
  (f : MF A), (μ d) (fun x ⇒ k × f x) ≡ k × (μ d) f.

Lemma mu_stable_plus2:
  ∀ (A : Type) (d : distr A) (f g: MF A),
    fplusok f g → (μ d) (fun x ⇒ f x + g x) ≡ (μ d) f + (μ d) g.

Lemma mu_fzero_eq : ∀ A m, @μ A m (fun x ⇒ 0) ≡ 0.

Instance Odistr (A:Type) : ord (distr A) :=
    {Ole := fun (f g : distr A) ⇒ μ f ≤ μ g;
     Oeq := fun (f g : distr A) ⇒ μ f ≡ μ g}.
Defined.
```

Probability of termination

```
Definition pone A (m:distr A) := μ m (fone A).

Add Parametric Morphism A : (pone (A:=A) )
    with signature Oeq ⟹ Oeq as pone_eq_compat.
Save.
Hint Resolve pone_eq_compat.
```

6.3 Monadic operators for distributions

```
Definition Munit : ∀ A:Type, A → distr A.
Defined.

Definition Mlet : ∀ A B:Type, distr A → (A → distr B) → distr B.
Defined.

Lemma Munit_simpl : ∀ (A:Type) (q:A → U) x, μ (Munit x) q = q x.

Lemma Mlet_simpl : ∀ (A B:Type) (m:distr A) (M:A → distr B) (f:B → U),
      μ (Mlet m M) f = μ m (fun x ⇒ (μ (M x) f)).
```

6.4 Operations on distributions

```
Lemma Munit_eq_compat : ∀ A (x y : A), x = y → Munit x ≡ Munit y.

Lemma Mlet_le_compat : ∀ (A B : Type) (m1 m2:distr A) (M1 M2 : A → distr B),
  m1 ≤ m2 → M1 ≤ M2 → Mlet m1 M1 ≤ Mlet m2 M2.
Hint Resolve Mlet_le_compat.

Add Parametric Morphism (A B : Type) : (Mlet (A:=A) (B:=B))
  with signature Ole ⟹ Ole ⟹ Ole
  as Mlet_le_morphism.
Save.

Add Parametric Morphism (A B : Type) : (Mlet (A:=A) (B:=B))
  with signature Ole ⟹ (@pointwise_relation A (distr B) (@Ole _ _)) ⟹ Ole
  as Mlet_le_pointwise_morphism.
Save.

Instance Mlet_mon2 : ∀ (A B : Type), monotonic2 (@Mlet A B).
Save.

Definition MLet (A B : Type) : distr A -m> (A → distr B) -m> distr B
                := mon2 (@Mlet A B).

Lemma MLet_simpl0 : ∀ (A B:Type) (m:distr A) (M:A → distr B),
      MLet A B m M = Mlet m M.

Lemma MLet_simpl : ∀ (A B:Type) (m:distr A) (M:A → distr B)(f:B → U),
      μ (MLet A B m M) f = μ m (fun x ⇒ μ (M x) f).

Lemma Mlet_eq_compat : ∀ (A B : Type) (m1 m2:distr A) (M1 M2 : A → distr B),
  m1 ≡ m2 → M1≡M2 → Mlet m1 M1 ≡ Mlet m2 M2.
Hint Resolve Mlet_eq_compat.

Add Parametric Morphism (A B : Type) : (Mlet (A:=A) (B:=B))
  with signature Oeq ⟹ Oeq ⟹ Oeq
  as Mlet_eq_morphism.
Save.

Add Parametric Morphism (A B : Type) : (Mlet (A:=A) (B:=B))
  with signature Oeq ⟹ (@pointwise_relation A (distr B) (@Oeq _ _)) ⟹ Oeq
  as Mlet_Oeq_pointwise_morphism.
Save.

Lemma mu_le_compat : ∀ (A:Type) (m1 m2:distr A),
  m1 ≤ m2 → ∀ f g : A → U, f ≤ g → μ m1 f ≤ μ m2 g.

Lemma mu_eq_compat : ∀ (A:Type) (m1 m2:distr A),
  m1 ≡ m2 → ∀ f g : A → U, f ≡ g → μ m1 f ≡ μ m2 g.
Hint Immediate mu_le_compat mu_eq_compat.

Add Parametric Morphism (A : Type) : (μ (A:=A))
  with signature Ole ⟹ Ole
```

as *mu_le_morphism*.
Save.

Add *Parametric Morphism* (*A* : Type) : (*μ* (*A*:=*A*))
with *signature* Oeq ⟹ Oeq
as *mu_eq_morphism*.
Save.

Add *Parametric Morphism* (*A*:Type) (*a*:**distr** *A*) : (@*μ* *A* *a*)
with *signature* (@pointwise_relation *A* *U* (@**eq** _) ⟹ Oeq) as *mu_distr_eq_morphism*.
Save.

Add *Parametric Morphism* (*A*:Type) (*a*:**distr** *A*) : (@*μ* *A* *a*)
with *signature* (@pointwise_relation *A* *U* (@Oeq _ _) ⟹ Oeq) as *mu_distr_Oeq_morphism*.
Save.

Add *Parametric Morphism* (*A*:Type) (*a*:**distr** *A*) : (@*μ* *A* *a*)
with *signature* (@pointwise_relation _ _ (@Ole _ _) ⟹ Ole) as *mu_distr_le_morphism*.
Save.

Add *Parametric Morphism* (*A B*:Type) : (@Mlet *A* *B*)
with *signature* (Ole ⟹ @pointwise_relation _ _ (@Ole _ _) ⟹ Ole) as *mlet_distr_le_morphism*.
Save.

Add *Parametric Morphism* (*A B*:Type) : (@Mlet *A* *B*)
with *signature* (Oeq ⟹ @pointwise_relation _ _ (@Oeq _ _) ⟹ Oeq) as *mlet_distr_eq_morphism*.
Save.

6.5 Properties of monadic operators

Lemma Mlet_unit : ∀ (*A B*:Type) (*x*:*A*) (*m*:*A* → **distr** *B*), Mlet (Munit *x*) *m* ≡ *m x*.

Lemma Mlet_ext : ∀ (*A*:Type) (*m*:**distr** *A*), Mlet *m* (fun *x* ⇒ Munit *x*) ≡ *m*.

Lemma Mlet_assoc : ∀ (*A B C*:Type) (*m1*: **distr** *A*) (*m2*:*A* → **distr** *B*) (*m3*:*B* → **distr** *C*),
Mlet (Mlet *m1* *m2*) *m3* ≡ Mlet *m1* (fun *x*:*A* ⇒ Mlet (*m2 x*) *m3*).

Lemma let_indep : ∀ (*A B*:Type) (*m1*:**distr** *A*) (*m2*: **distr** *B*) (*f*:MF *B*),
μ m1 (fun _ ⇒ *μ m2 f*) ≡ pone *m1* × (*μ m2 f*).

6.6 A specific distribution

Definition distr_null : ∀ *A* : Type, **distr** *A*.
Defined.

Lemma le_distr_null : ∀ (*A*:Type) (*m* : **distr** *A*), distr_null *A* ≤ *m*.
Hint Resolve le_distr_null.

6.7 Scaling a distribution

Definition Mmult *A* (*k*:MF *A*) (*m*:M *A*) : M *A*.
Defined.

Lemma Mmult_simpl : ∀ *A* (*k*:MF *A*) (*m*:M *A*) *f*, Mmult *k* *m* *f* = *m* (fun *x* ⇒ *k x* × *f x*).

Lemma Mmult_stable_inv : ∀ *A* (*k*:MF *A*) (*d*:**distr** *A*), stable_inv (Mmult *k* (*μ d*)).

Lemma Mmult_stable_plus : ∀ *A* (*k*:MF *A*) (*d*:**distr** *A*), stable_plus (Mmult *k* (*μ d*)).

Lemma Mmult_stable_mult : ∀ *A* (*k*:MF *A*) (*d*:**distr** *A*), stable_mult (Mmult *k* (*μ d*)).

Lemma Mmult_continuous : ∀ *A* (*k*:MF *A*) (*d*:**distr** *A*), **continuous** (Mmult *k* (*μ d*)).

Definition distr_mult *A* (*k*:MF *A*) (*d*:**distr** *A*) : **distr** *A*.
Defined.

```
Lemma distr_mult_assoc : ∀ A (k1 k2:MF A) (d:distr A),
              distr_mult k1 (distr_mult k2 d) ≡ distr_mult (fun x ⇒ k1 x × k2 x) d.
Add Parametric Morphism (A B : Type) : (distr_mult (A:=A))
     with signature Oeq ⟹ Oeq ⟹ Oeq
as distr_mult_eq_compat.
Save.
```

Scaling with a constant functions

```
Definition distr_scale A (k:U) (d:distr A) : distr A := distr_mult (fcte A k) d.
Lemma distr_scale_assoc : ∀ A (k1 k2:U) (d:distr A),
              distr_scale k1 (distr_scale k2 d) ≡ distr_scale (k1×k2) d.
Lemma distr_scale_simpl : ∀ A (k:U) (d:distr A)(f:MF A),
        μ (distr_scale k d) f ≡ k × μ d f.
Add Parametric Morphism A : (distr_scale (A:=A))
with signature Oeq ⟹ Oeq ⟹ Oeq
as distr_scale_eq_compat.
Save.
Hint Resolve distr_scale_eq_compat.
Lemma distr_scale_one : ∀ A (d:distr A), distr_scale 1 d ≡ d.
Lemma distr_scale_zero : ∀ A (d:distr A), distr_scale 0 d ≡ distr_null A.
Hint Resolve distr_scale_simpl distr_scale_assoc distr_scale_one distr_scale_zero.
Lemma let_indep_distr : ∀ (A B:Type) (m1:distr A) (m2: distr B),
        Mlet m1 (fun _ ⇒ m2) ≡ distr_scale (pone m1) m2.
Definition Mdiv A (k:U) (m:M A) : M A := UDiv k @ m.
Lemma Mdiv_simpl : ∀ A k (m:M A) f, Mdiv k m f = m f / k.
Lemma Mdiv_stable_inv : ∀ A (k:U) (d:distr A)(dk : μ d (fone A) ≤ k),
                  stable_inv (Mdiv k (μ d)).
Lemma Mdiv_stable_plus : ∀ A (k:U)(d:distr A), stable_plus (Mdiv k (μ d)).
Lemma Mdiv_stable_mult : ∀ A (k:U)(d:distr A)(dk : μ d (fone A) ≤ k),
                        stable_mult (Mdiv k (μ d)).
Lemma Mdiv_continuous : ∀ A (k:U)(d:distr A), continuous (Mdiv k (μ d)).
Definition distr_div A (k:U) (d:distr A) (dk : μ d (fone A) ≤ k)
               : distr A.
Defined.
```

6.8 Conditional probabilities

```
Definition mcond A (m:M A) (f:MF A) : M A.
Defined.
Lemma mcond_simpl : ∀ A (m:M A) (f g: MF A),
       mcond m f g = m (fconj f g) / m f.
Lemma mcond_stable_plus : ∀ A (m:distr A) (f: MF A), stable_plus (mcond (μ m) f).
Lemma mcond_stable_inv : ∀ A (m:distr A) (f: MF A), stable_inv (mcond (μ m) f).
Lemma mcond_stable_mult : ∀ A (m:distr A) (f: MF A), stable_mult (mcond (μ m) f).
Lemma mcond_continuous : ∀ A (m:distr A) (f: MF A), continuous (mcond (μ m) f).
Definition Mcond A (m:distr A) (f:MF A) : distr A :=
   Build_distr (mcond_stable_inv m f) (mcond_stable_plus m f)
               (mcond_stable_mult m f) (mcond_continuous m f).
```

```
Lemma Mcond_total : ∀ A (m:distr A) (f:MF A),
          0 ≡ μ m f → μ (Mcond m f) (fone A) ≡ 1.
Lemma Mcond_simpl : ∀ A (m:distr A) (f g:MF A),
      μ (Mcond m f) g = μ m (fconj f g) / μ m f.
Hint Resolve Mcond_simpl.
Lemma Mcond_zero_stable : ∀ A (m:distr A) (f g:MF A),
      μ m g ≡ 0 → μ (Mcond m f) g ≡ 0.
Lemma Mcond_null : ∀ A (m:distr A) (f g:MF A),
      μ m f ≡ 0 → μ (Mcond m f) g ≡ 0.
Lemma Mcond_conj : ∀ A (m:distr A) (f g:MF A),
          μ m (fconj f g) ≡ μ (Mcond m f) g × μ m f.
Lemma Mcond_decomp :
    ∀ A (m:distr A) (f g:MF A),
          μ m g ≡ μ (Mcond m f) g × μ m f + μ (Mcond m (finv f)) g × μ m (finv f).
Lemma Mcond_bayes : ∀ A (m:distr A) (f g:MF A),
          μ (Mcond m f) g ≡ (μ (Mcond m g) f × μ m g) / (μ m f).
```

6.9 Least upper bound of increasing sequences of distributions

```
Lemma M_lub_simpl : ∀ A (h: nat -m> M A) (f:MF A),
     lub h f = lub (mshift h f).
Section Lubs.
Variable A : Type.
Definition Mu : distr A -m> M A.
Defined.
Lemma Mu_simpl : ∀ d f, Mu d f = μ d f.
Variable muf : nat -m> distr A.
Definition mu_lub: distr A.

Defined.
Lemma mu_lub_le : ∀ n:nat, muf n ≤ mu_lub.
Lemma mu_lub_sup : ∀ m: distr A, (∀ n:nat, muf n ≤ m) → mu_lub ≤ m.
End Lubs.
Hint Resolve mu_lub_le mu_lub_sup.
```

6.9.1 distributions seen as a Ccpo

```
Instance cdistr (A:Type) : cpo (distr A) :=
   {D0 := distr_null A; lub:=mu_lub (A:=A)}.
Defined.
Lemma distr_lub_simpl : ∀ A (h : nat -m> distr A) (f:MF A),
      μ (lub h) f = lub (mshift (Mu A @ h) f).
Hint Resolve distr_lub_simpl.
```

6.10 Fixpoints

```
Definition Mfix (A B:Type) (F: (A → distr B) -m> (A → distr B))
    : A → distr B := fixp F.
Definition MFix (A B:Type) : ((A → distr B) -m> (A → distr B)) -m> (A → distr B)
```

$:=$ Fixp $(A \to$ **distr** $B)$.

Lemma Mfix_le : $\forall$ $(A\ B$:Type$)$ $(F$: $(A \to$ **distr** $B)$ -m> $(A \to$ **distr** $B))$ $(x$:$A)$,
Mfix $F\ x \leq F$ (Mfix F) x.

Lemma Mfix_eq : $\forall$ $(A\ B$:Type$)$ $(F$: $(A \to$ **distr** $B)$ -m> $(A \to$ **distr** $B))$,
continuous $F \to \forall$ $(x$:$A)$, Mfix $F\ x \equiv F$ (Mfix F) x.

Hint Resolve Mfix_le Mfix_eq.

Lemma Mfix_le_compat : $\forall$ $(A\ B$:Type$)$ $(F\ G$: $(A \to$ **distr** $B)$-m> $(A \to$ **distr** $B))$,
$F \leq G \to$ Mfix $F \leq$ Mfix G.

Definition Miter $(A\ B$:Type$)$:= Ccpo.iter $(D$:=$A \to$ **distr** $B)$.

Lemma Mfix_le_iter : $\forall$ $(A\ B$:Type$)$ $(F$:$(A \to$ **distr** $B)$ -m> $(A \to$ **distr** $B))$ $(n$:**nat**$)$,
Miter $F\ n \leq$ Mfix F.

6.11 Continuity

Section Continuity.

Variables $A\ B$:Type.

Instance Mlet_continuous_right
: $\forall$ a:**distr** A, **continuous** $(D1$:= $A \to$ **distr** $B)$ $(D2$:=**distr** $B)$ (MLet $A\ B\ a$).
Save.

Lemma Mlet_continuous_left
: **continuous** $(D1$:=**distr** $A)$ $(D2$:=$(A \to$ **distr** $B)$ -m> **distr** $B)$ (MLet $A\ B$).

Hint Resolve Mlet_continuous_right Mlet_continuous_left.

Lemma Mlet_continuous2 : **continuous2** $(D1$:=**distr** $A)$ $(D2$:= $A \to$**distr** $B)$ $(D3$:=**distr** $B)$ (MLet $A\ B$).
Hint Resolve Mlet_continuous2.

Lemma Mlet_lub_le : $\forall$ $(mun$:**nat** -m> **distr** $A)$ $(Mn$: **nat** -m> $(A \to$ **distr** $B))$,
Mlet (lub mun) (lub Mn) $\leq$ lub ((MLet $A\ B\ @^2\ mun$) Mn).

Lemma Mlet_lub_le_left : $\forall$ $(mun$:**nat** -m> **distr** $A)$
$(M$: $A \to$ **distr** $B)$,
Mlet (lub mun) $M \leq$ lub (mshift (MLet $A\ B$ @ mun) M).

Lemma Mlet_lub_le_right : $\forall$ $(m$:**distr** $A)$
$(Mun$: **nat** -m> $(A \to$ **distr** $B))$,
Mlet m (lub Mun) $\leq$ lub ((MLet $A\ B\ m$)@Mun).

Lemma Mlet_lub_fun_le_right : $\forall$ $(m$:**distr** $A)$
$(Mun$: $A \to$ **nat** -m> **distr** $B)$,
Mlet m (fun $x \Rightarrow$ lub $(Mun\ x)) \leq$ lub ((MLet $A\ B\ m$)@(ishift Mun)).

Lemma Mfix_continuous :
$\forall$ $(Fn$: **nat** -m> $(A \to$ **distr** $B)$ -m> $(A \to$ **distr** $B))$,
$(\forall\ n$, **continuous** $(Fn\ n)) \to$
Mfix (lub Fn) $\leq$ lub (MFix $A\ B$ @ Fn).

End Continuity.

6.12 distribution for *flip*

The distribution associated to *flip* () is f -> $\frac{1}{2}$ $(f\ true)$ + $\frac{1}{2}$ $(f\ false)$

Definition flip : M **bool** := mon (fun $(f$: **bool** $\to U) \Rightarrow \frac{1}{2} \times (f$ true) + $\frac{1}{2} \times (f$ false)).

Lemma flip_stable_inv : stable_inv flip.

Lemma flip_stable_plus : stable_plus flip.

Lemma flip_stable_mult : stable_mult flip.

Lemma flip_continuous : **continuous** flip.

Definition ctrue : MF **bool** := fun (b:**bool**) ⇒ if b then 1 else 0.
Definition cfalse : MF **bool** := fun (b:**bool**) ⇒ if b then 0 else 1.

Lemma flip_ctrue : flip ctrue ≡ $\frac{1}{2}$.

Lemma flip_cfalse : flip cfalse ≡ $\frac{1}{2}$.

Hint Resolve flip_ctrue flip_cfalse.

Definition Flip : **distr** **bool**.
Defined.

Lemma Flip_simpl : ∀ f, μ Flip f = $\frac{1}{2}$ × (f true) + $\frac{1}{2}$ × (f false).

6.13 Uniform distribution beween 0 and n

Require Arith.

6.13.1 Definition of *fnth*

fnth n k is defined as [1/]1+n

Definition fnth (n:**nat**) : **nat** → U := fun k ⇒ [1/]1+n.

6.13.2 Basic properties of *fnth*

Lemma Unth_eq : ∀ n, *Unth* n ≡ [1-] (sigma (fnth n) n).
Hint Resolve Unth_eq.

Lemma sigma_fnth_one : ∀ n, sigma (fnth n) (S n) ≡ 1.
Hint Resolve sigma_fnth_one.

Lemma Unth_inv_eq : ∀ n, [1-] ([1/]1+n) ≡ sigma (fnth n) n.

Lemma sigma_fnth_sup : ∀ n m, (m > n) → sigma (fnth n) m ≡ sigma (fnth n) (S n).

Lemma sigma_fnth_le : ∀ n m, (sigma (fnth n) m) ≤ (sigma (fnth n) (S n)).

Hint Resolve sigma_fnth_le.

fnth is a retract Lemma fnth_retract : ∀ n:**nat**,(retract (fnth n) (S n)).

Implicit Arguments fnth_retract [].

6.14 distributions and general summations

Definition sigma_fun A (f:**nat** → MF A) (n:**nat**) : MF A := fun x ⇒ sigma (fun k ⇒ f k x) n.
Definition serie_fun A (f:**nat** → MF A) : MF A := fun x ⇒ serie (fun k ⇒ f k x).

Definition Sigma_fun A (f:**nat** → MF A) : **nat** -m> MF A :=
ishift (fun x ⇒ Sigma (fun k ⇒ f k x)).

Lemma Sigma_fun_simpl : ∀ A (f:**nat** → MF A) (n:**nat**),
Sigma_fun f n = sigma_fun f n.

Lemma serie_fun_lub_sigma_fun : ∀ A (f:**nat** → MF A),
serie_fun f ≡ lub (Sigma_fun f).
Hint Resolve serie_fun_lub_sigma_fun.

Lemma sigma_fun_0 : ∀ A (f:**nat** → MF A), sigma_fun f 0 ≡ fzero A.

Lemma sigma_fun_S : ∀ A (f:**nat**→MF A) (n:**nat**),
sigma_fun f (S n) ≡ fplus (f n) (sigma_fun f n).

Lemma mu_sigma_le : ∀ A (d:**distr** A) (f:**nat** → MF A) (n:**nat**),
μ d (sigma_fun f n) ≤ sigma (fun k ⇒ μ d (f k)) n.

```
Lemma retract_fplusok : ∀ A (f:nat → MF A) (n:nat),
      (∀ x, retract (fun k ⇒ f k x) n) →
      ∀ k, (k < n)%nat → fplusok (f k) (sigma_fun f k).

Lemma mu_sigma_eq : ∀ A (d:distr A) (f:nat → MF A) (n:nat),
      (∀ x, retract (fun k ⇒ f k x) n) →
       μ d (sigma_fun f n) ≡ sigma (fun k ⇒ μ d (f k)) n.

Lemma mu_serie_le : ∀ A (d:distr A) (f:nat → MF A),
       μ d (serie_fun f) ≤ serie (fun k ⇒ μ d (f k)).

Lemma mu_serie_eq : ∀ A (d:distr A) (f:nat → MF A),
      (∀ x, wretract (fun k ⇒ f k x)) →
       μ d (serie_fun f) ≡ serie (fun k ⇒ μ d (f k)).

Lemma wretract_fplusok : ∀ A (f:nat → MF A),
      (∀ x, wretract (fun k ⇒ f k x)) →
      ∀ k, fplusok (f k) (sigma_fun f k).
```

6.15 Discrete distributions

```
Instance discrete_mon : ∀ A (c : nat → U) (p : nat → A),
     monotonic (fun f : A→U ⇒ serie (fun k ⇒ c k × f (p k))).
Save.

Definition discrete A (c : nat → U) (p : nat → A) : M A :=
       mon (fun f : A→U ⇒ serie (fun k ⇒ c k × f (p k))).

Lemma discrete_simpl : ∀ A (c : nat → U) (p : nat → A) f,
     discrete c p f = serie (fun k ⇒ c k × f (p k)).

Lemma discrete_stable_inv : ∀ A (c : nat → U) (p : nat → A),
    wretract c → stable_inv (discrete c p).

Lemma discrete_stable_plus : ∀ A (c : nat → U) (p : nat → A),
    stable_plus (discrete c p).

Lemma discrete_stable_mult : ∀ A (c : nat → U) (p : nat → A),
    wretract c → stable_mult (discrete c p).

Lemma discrete_continuous : ∀ A (c : nat → U) (p : nat → A),
    continuous (discrete c p).

Record discr (A:Type) : Type :=
     {coeff : nat → U; coeff_retr : wretract coeff; points : nat → A}.
Hint Resolve coeff_retr.

Definition Discrete : ∀ A, discr A → distr A.
Defined.

Lemma Discrete_simpl : ∀ A (d:discr A),
     μ (Discrete d) = discrete (coeff d) (points d).

Definition is_discrete (A:Type) (m: distr A) :=
      ∃ d : discr A, m ≡ Discrete d.
```

6.15.1 distribution for *random n*

The distribution associated to *random n* is *f -> sigma (i=0..n) [1]1+n (f i)* we cannot factorize [1/]1+n because of possible overflow

```
Instance random_mon : ∀ n, monotonic (fun (f:MF nat) ⇒ sigma (fun k ⇒ Unth n × f k) (S n)).
Save.

Definition random (n:nat):M nat := mon (fun (f:MF nat) ⇒ sigma (fun k ⇒ Unth n × f k) (S n)).
```

```
Lemma random_simpl : ∀ n (f : MF nat),
                random n f = sigma (fun k ⇒ Unth n × f k) (S n).
```

6.15.2 Properties of *random*

```
Lemma random_stable_inv : ∀ n, stable_inv (random n).
Lemma random_stable_plus : ∀ n, stable_plus (random n).
Lemma random_stable_mult : ∀ n, stable_mult (random n).
Lemma random_continuous : ∀ n, continuous (random n).
Definition Random (n:nat) : distr nat.
Defined.
Lemma Random_simpl : ∀ (n:nat), μ (Random n) = random n.
Lemma Random_total : ∀ n : nat, μ (Random n) (fone nat) ≡ 1.
Hint Resolve Random_total.
Lemma Random_inv : ∀ f n, μ (Random n) (finv f) ≡ [1-] (μ (Random n) f).
Hint Resolve Random_inv.
```

6.16 Tacticals

```
Ltac mu_plus d :=
  match goal with
 | ⊢ context [fmont (μ d) (fun x ⇒ (Uplus (@?f x) (@?g x)))] ⇒
      rewrite (mu_stable_plus d (f:=f) (g:=g))
  end.
Ltac mu_mult d :=
  match goal with
 | ⊢ context [fmont (μ d) (fun x ⇒ (Umult ?k (@?f x)))] ⇒
      rewrite (mu_stable_mult d k f)
  end.
Lemma fplusok_mu_fone : ∀ (A B:Type) (d:distr B) (f f':A → MF B),
  (∀ x:A, fplusok (f x) (f' x)) →
  μ d (fone _) ≡ 1 →
  fplusok (fun x : A ⇒ μ d (f x)) (fun x : A ⇒ μ d (f' x)).
```

7 SProbas.v: Definition of the monad for sub-distributions

```
Require Export Probas.
```

7.1 Definition of (sub)distribution

Subdistributions are measure functions μ such that

- $\mu\ (1\text{-}f) \leq 1$ - $\mu\ f$
- $f \leq 1\text{-}g \rightarrow \mu\ f\ +\ \mu\ g \leq \mu\ (f+g)$
- $\mu\ f\ \&\ \mu\ g \leq \mu\ (f\ \&\ g)$ - [$\mu\ (f+k) \leq \mu\ f\ +\ k$] - [$\mu\ (k \times f) = k \times \mu\ (f)$] - [$\mu\ (lub\ f\_n) \leq lub\ \mu\ (f\_n)$

```
Record] sdistr (A:Type) : Type :=
  {smu : M A;
   smu_stable_inv : stable_inv smu;
```

smu_le_plus : le_plus smu;
smu_le_esp : le_esp smu;
smu_le_plus_cte : le_plus_cte smu;
smu_stable_mult : stable_mult smu;
smu_continuous : **continuous** smu}.

```
Hint Resolve smu_le_plus smu_stable_inv smu_le_esp smu_stable_mult
smu_continuous.
```

7.2 Properties of sub-measures

```
Lemma smu_monotonic : ∀ (A : Type)(m: sdistr A), monotonic (smu m).
Hint Resolve smu_monotonic.
Implicit Arguments smu_monotonic [A].

Lemma smu_stable : ∀ (A : Type)(m: sdistr A), stable (smu m).
Hint Resolve smu_stable.
Implicit Arguments smu_stable [A].

Lemma smu_zero : ∀ (A : Type)(m: sdistr A), smu m (fzero A) ≡ 0.
Hint Resolve smu_zero.

Lemma smu_stable_mult_right : ∀ (A : Type)(m:(sdistr A)) (c:U) (f : A → U),
   smu m (fun x ⇒ (f x) × c) ≡ (smu m f) × c.

Lemma smu_le_minus_left : ∀ (A : Type)(m:sdistr A) (f g:A → U),
     smu m (fminus f g) ≤ smu m f.
Hint Resolve smu_le_minus_left.

Lemma smu_le_minus : ∀ (A:Type) (m:sdistr A)(f g: A → U),
 g ≤ f → smu m (fminus f g) ≤ smu m f - smu m g.
Hint Resolve smu_le_minus.

Lemma smu_cte : ∀ (A : Type)(m:(sdistr A)) (c:U),
   smu m (fcte A c) ≡ c × smu m (fone A).
Hint Resolve smu_cte.

Lemma smu_cte_le : ∀ (A : Type)(m:(sdistr A)) (c:U),
   smu m (fcte A c) ≤ c.

Lemma smu_cte_eq : ∀ (A : Type)(m:(sdistr A)) (c:U),
   smu m (fone A) ≡ 1 → smu m (fcte A c) ≡ c.

Hint Resolve smu_cte_le smu_cte_eq.

Lemma smu_le_minus_cte : ∀ (A:Type) (m:sdistr A)(f: A → U) (k:U),
     smu m f - k ≤ smu m (fminus f (fcte A k)).

Lemma smu_inv_le_minus :
∀ (A:Type) (m:sdistr A)(f: A → U), smu m (finv f) ≤ smu m (fone A) - smu m f.

Lemma smu_inv_minus_inv : ∀ (A:Type) (m:sdistr A)(f: A → U),
     smu m (finv f) + [1-](smu m (fone A)) ≤ [1-](smu m f).

Definition stable_plus_sdistr : ∀ A (m:M A),
  stable_plus m → stable_inv m → stable_mult m → continuous m → sdistr A.
Defined.

Definition distr_sdistr : ∀ A, distr A → sdistr A.
Defined.

Definition Sunit A (x:A) : sdistr A := distr_sdistr (Munit x).

Lemma Sunit_unit : ∀ A (x:A), smu (Sunit x) = unit x.

Lemma Sunit_simpl : ∀ A (x:A) (f : MF A), smu (Sunit x) f = f x.
```

```
Definition Slet : ∀ A B:Type, (sdistr A) → (A → sdistr B) → sdistr B.
Defined.

Lemma Slet_star : ∀ (A B:Type) (m:sdistr A) (M : A → sdistr B),
      smu (Slet m M) = star (smu m) (fun x ⇒ smu (M x)).

Lemma Slet_simpl : ∀ A B (m:sdistr A) (M : A → sdistr B) (f:MF B),
                   smu (Slet m M) f = smu m (fun x ⇒ smu (M x) f).
```

Non deterministic choice

```
Definition Smin (A:Type)(m1 m2 : sdistr A) : sdistr A.
Save.
```

7.3 Operations on sub-distributions

```
Instance Osdistr (A : Type) : ord (sdistr A) :=
    { Ole := fun f g ⇒ smu f ≤ smu g;
      Oeq := fun f g ⇒ smu f ≡ smu g}.
Defined.

Lemma Sunit_compat : ∀ A (x y : A), x = y → Sunit x ≡ Sunit y.

Lemma Slet_compat : ∀ (A B : Type) (m1 m2:sdistr A) (M1 M2 : A→ sdistr B),
  m1 ≡ m2 → M1 ≡ M2 → Slet m1 M1 ≡ Slet m2 M2.

Lemma le_sdistr_gen : ∀ (A:Type) (m1 m2:sdistr A),
  m1 ≤ m2 → ∀ f g, f ≤ g → smu m1 f ≤ smu m2 g.
```

7.4 Properties of monadic operators

```
Lemma Slet_unit : ∀ (A B:Type) (x:A) (m:A → sdistr B), Slet (Sunit x) m ≡ m x.

Lemma M_ext : ∀ (A:Type) (m:sdistr A), Slet m (fun x ⇒ Sunit x) ≡ m.

Lemma Mcomp : ∀ (A B C:Type) (m1:(sdistr A)) (m2:A → sdistr B) (m3:B → sdistr C),
      Slet (Slet m1 m2) m3 ≡ Slet m1 (fun x:A ⇒ (Slet (m2 x) m3)).

Lemma Slet_le_compat : ∀ (A B:Type) (m1 m2: sdistr A) (f1 f2 : A → sdistr B),
  m1 ≤ m2 → f1 ≤ f2 → Slet m1 f1 ≤ Slet m2 f2.
```

7.5 A specific subdistribution

```
Definition sdistr_null : ∀ A : Type, sdistr A.
Defined.

Lemma le_sdistr_null : ∀ (A:Type) (m : sdistr A), sdistr_null A ≤ m.
Hint Resolve le_sdistr_null.
```

7.6 Least upper bound of increasing sequences of sdistributions

```
Section Lubs.
Variable A : Type.

Definition Smu : sdistr A -m> M A.
Defined.

Lemma Smu_simpl : ∀ d f, Smu d f = smu d f.

Variable smuf : nat -m> sdistr A.

Definition smu_lub: sdistr A.


Defined.
```

Lemma smu_lub_simpl : smu smu_lub = lub (Smu @ $smuf$).

Lemma smu_lub_le : $\forall$ n:**nat**, $smuf\ n \leq$ smu_lub.

Lemma smu_lub_sup : $\forall$ m:**sdistr** A, ($\forall$ n:**nat**, $smuf\ n \leq m$) $\rightarrow$ smu_lub $\leq m$.

End Lubs.

7.7 Sub-distribution for *flip*

The distribution associated to *flip* () is $f \mapsto \frac{1}{2}f(true)+\frac{1}{2}f(false)$ Definition Sflip : **sdistr bool** := distr_sdistr Flip.

Lemma Sflip_simpl : smu Sflip = flip.

7.8 Uniform sub-distribution beween 0 and n

Require Arith.

7.8.1 Distribution for *Srandom n*

The sdistribution associated to *Srandom n* is $f \mapsto \Sigma_{i=0}^{n} \frac{f(i)}{n+1}$ we cannot factorize $\frac{1}{n+1}$ because of possible overflow

Definition Srandom (n:**nat**): **sdistr nat**:= distr_sdistr (Random n).

Lemma Srandom_simpl : $\forall$ n, smu (Srandom n) = random n.

8 Prog.v: Composition of distributions

Require Export Probas.

8.1 Conditional

Definition Mif (A:Type) (b:**distr bool**) ($m1\ m2$: **distr** A)
  := Mlet b (fun x:**bool** $\Rightarrow$ if x then $m1$ else $m2$).

Lemma Mif_le_compat : $\forall$ (A:Type) ($b1\ b2$:**distr bool**) ($m1\ m2\ n1\ n2$: **distr** A),
  $b1 \leq b2 \rightarrow m1 \leq m2 \rightarrow n1 \leq n2 \rightarrow$ Mif $b1\ m1\ n1 \leq$ Mif $b2\ m2\ n2$.

Hint Resolve Mif_le_compat.

Instance Mif_mon2 : $\forall$ (A:Type) b, **monotonic2** (Mif (A:=A) b).
Save.

Definition Mlf : $\forall$ (A:Type), **distr bool** -m> **distr** A -m> **distr** A -m> **distr** A.
Defined.

Lemma Mlf_simpl : $\forall$ $A\ b\ d1\ d2$, Mlf $A\ b\ d1\ d2$ = Mif $b\ d1\ d2$.

Instance if_mon : $\forall$ '{o:**ord** A} (b:**bool**), **monotonic2** (fun ($x\ y$:A) $\Rightarrow$ if b then x else y).
Save.

Definition If '{o:**ord** A} (b:**bool**) : A -m> A -m> A := mon2 (fun ($x\ y$:A) $\Rightarrow$ if b then x else y).

Instance Mif_continuous2 : $\forall$ (A:Type) b, **continuous2** (Mlf $A\ b$).
Save.

Hint Resolve Mif_continuous2.

Instance Mif_cond_continuous : $\forall$ (A:Type), **continuous** (Mlf A).
Save.

Hint Resolve Mif_cond_continuous.

8.2 Probabilistic choice

8.2.1 The distribution associated to *pchoice p m1 m2* is

f -> p ($m1$ f) + (1-p) ($m2$ f)

```
Definition pchoice : ∀ A, U → M A → M A → M A.
Defined.

Lemma pchoice_simpl : ∀ A p (m1 m2:M A) f,
      pchoice p m1 m2 f = p × m1 f + [1-]p × m2 f.

Definition Mchoice (A:Type) (p:U) (m1 m2: distr A) : distr A.
Defined.

Lemma Mchoice_simpl : ∀ A p (m1 m2:distr A) f,
      μ (Mchoice p m1 m2) f = p × μ m1 f + [1-]p × μ m2 f.

Lemma Mchoice_le_compat : ∀ (A:Type) (p:U) (m1 m2 n1 n2: distr A),
        m1≤m2 → n1≤n2 → Mchoice p m1 n1 ≤ Mchoice p m2 n2.
Hint Resolve Mchoice_le_compat.

Add Parametric Morphism (A:Type) : (Mchoice (A:=A))
  with signature Oeq ⟹ Oeq ⟹ Oeq ⟹ Oeq
as Mchoice_eq_compat.
Save.
Hint Immediate Mchoice_eq_compat.

Instance Mchoice_mon2 : ∀ (A:Type) (p:U), monotonic2 (Mchoice (A:=A) p).
Save.

Definition MChoice A (p:U) : distr A -m> distr A -m> distr A :=
                  mon2 (Mchoice (A:=A) p).

Lemma MChoice_simpl : ∀ A (p:U) (m1 m2 : distr A),
MChoice A p m1 m2 = Mchoice p m1 m2.

Lemma Mchoice_sym_le : ∀ (A:Type) (p:U) (m1 m2: distr A),
             Mchoice p m1 m2 ≤ Mchoice ([1-]p) m2 m1.
Hint Resolve Mchoice_sym_le.

Lemma Mchoice_sym : ∀ (A:Type) (p:U) (m1 m2: distr A),
             Mchoice p m1 m2 ≡ Mchoice ([1-]p) m2 m1.

Lemma Mchoice_continuous_right
    : ∀ (A:Type) (p:U) (m: distr A), continuous (D1:=distr A) (D2:=distr A) (MChoice A p m).
Hint Resolve Mchoice_continuous_right.

Lemma Mchoice_continuous_left : ∀ (A:Type) (p:U),
    continuous (D1:=distr A) (D2:=distr A -m> distr A) (MChoice A p).

Lemma Mchoice_continuous :
∀ (A:Type) (p:U), continuous2 (D1:=distr A) (D2:=distr A) (D3:=distr A) (MChoice A p).
```

8.3 Image distribution

```
Definition im_distr (A B : Type) (f:A → B) (m:distr A) : distr B :=
    Mlet m (fun a ⇒ Munit (f a)).

Lemma im_distr_simpl : ∀ A B (f:A → B) (m:distr A)(h:B → U),
     μ (im_distr f m) h = μ m (fun a ⇒ h (f a)).

Add Parametric Morphism (A B : Type) : (im_distr (A:=A) (B:=B))
  with signature (feq (A:=A) (B:=B)) ⟹ Oeq ⟹ Oeq
  as im_distr_eq_compat.
Save.
```

```
Lemma im_distr_comp : ∀ A B C (f:A → B) (g:B → C) (m:distr A),
              im_distr g (im_distr f m) ≡ im_distr (fun a ⇒ g (f a)) m.
Lemma im_distr_id : ∀ A (f:A → A) (m:distr A), (∀ x, f x = x) →
              im_distr f m ≡ m.
```

8.4 Product distribution

```
Definition prod_distr (A B : Type)(d1:distr A)(d2:distr B) : distr (A×B) :=
                  Mlet d1 (fun x ⇒ Mlet d2 (fun y ⇒ Munit (x,y))).
Add Parametric Morphism (A B : Type) : (prod_distr (A:=A) (B:=B))
with signature Ole ++> Ole ++> Ole
as prod_distr_le_compat.
Save.
Hint Resolve prod_distr_le_compat.
Add Parametric Morphism (A B : Type) : (prod_distr (A:=A) (B:=B))
with signature Oeq ⟹ Oeq ⟹ Oeq
as prod_distr_eq_compat.
Save.
Hint Immediate prod_distr_eq_compat.
Instance prod_distr_mon2 : ∀ (A B :Type), monotonic2 (prod_distr (A:=A) (B:=B)).
Save.
Definition Prod_distr (A B :Type): distr A -m> distr B -m> distr (A×B) :=
      mon2 (prod_distr (A:=A) (B:=B)).
Lemma Prod_distr_simpl : ∀ (A B:Type)(d1: distr A) (d2:distr B),
      Prod_distr A B d1 d2 = prod_distr d1 d2.
Lemma prod_distr_rect : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f:A → U)(g:B → U),
          μ (prod_distr d1 d2) (fun xy ⇒ f (fst xy) × g (snd xy)) ≡ μ d1 f × μ d2 g.
Lemma prod_distr_fst : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f:A → U),
                μ (prod_distr d1 d2) (fun xy ⇒ f (fst xy)) ≡ pone d2 × μ d1 f.
Lemma prod_distr_snd : ∀ (A B : Type) (d1: distr A) (d2:distr B) (g:B → U),
                μ (prod_distr d1 d2) (fun xy ⇒ g (snd xy)) ≡ pone d1 × μ d2 g.
Lemma prod_distr_fst_eq : ∀ (A B : Type) (d1: distr A) (d2:distr B),
      pone d2 ≡ 1 → im_distr (fst (A:=A) (B:=B)) (prod_distr d1 d2) ≡ d1.
Lemma prod_distr_snd_eq : ∀ (A B : Type) (d1: distr A) (d2:distr B),
        pone d1 ≡ 1 → im_distr (snd (A:=A) (B:=B)) (prod_distr d1 d2) ≡ d2.
Definition swap A B (x:A×B) : B × A := (snd x,fst x).
Definition arg_swap A B (f : MF (A×B)) : MF (B×A) := fun z ⇒ f (swap z).
Definition Arg_swap A B : MF (A×B) -m> MF (B×A).
Defined.
Lemma Arg_swap_simpl : ∀ A B f, Arg_swap A B f = arg_swap f.
Definition prod_distr_com A B (d1: distr A) (d2:distr B) (f : MF (A × B)) :=
       μ (prod_distr d1 d2) f ≡ μ (prod_distr d2 d1) (arg_swap f).
Lemma prod_distr_com_eq_compat : ∀ A B (d1: distr A) (d2:distr B) (f g: MF (A × B)),
  f ≡ g → prod_distr_com d1 d2 f → prod_distr_com d1 d2 g.
Lemma prod_distr_com_rect : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f:A → U)(g:B → U),
          prod_distr_com d1 d2 (fun xy ⇒ f (fst xy) × g (snd xy)).
Lemma prod_distr_com_cte : ∀ (A B : Type) (d1: distr A) (d2:distr B) (c:U),
          prod_distr_com d1 d2 (fcte (A×B) c).
```

```
Lemma prod_distr_com_one : ∀ (A B : Type) (d1: distr A) (d2:distr B),
          prod_distr_com d1 d2 (fone (A×B)).

Lemma prod_distr_com_plus : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f g:MF (A×B)),
          fplusok f g →
          prod_distr_com d1 d2 f → prod_distr_com d1 d2 g →
          prod_distr_com d1 d2 (fplus f g).

Lemma prod_distr_com_mult : ∀ (A B : Type) (d1: distr A) (d2:distr B) (k:U)(f:MF (A×B)),
          prod_distr_com d1 d2 f → prod_distr_com d1 d2 (fmult k f).

Lemma prod_distr_com_inv : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f:MF (A×B)),
          prod_distr_com d1 d2 f → prod_distr_com d1 d2 (finv f).

Lemma prod_distr_com_lub : ∀ (A B : Type) (d1: distr A) (d2:distr B) (f:nat -m> MF (A×B)),
          (∀ n, prod_distr_com d1 d2 (f n)) → prod_distr_com d1 d2 (lub f).

Lemma prod_distr_com_sym : ∀ A B (d1:distr A) (d2:distr B) (f:MF (A×B)),
      prod_distr_com d1 d2 f → prod_distr_com d2 d1 (arg_swap f).

Lemma discrete_commute : ∀ A B (d1:distr A) (d2:distr B) (f:MF (A×B)),
     is_discrete d1 → prod_distr_com d1 d2 f.

Lemma is_discrete_swap: ∀ A B C (d1:distr A) (d2:distr B) (f:A → B → distr C),
    is_discrete d1 →
    Mlet d1 (fun x ⇒ Mlet d2 (fun y ⇒ f x y)) ≡ Mlet d2 (fun y ⇒ Mlet d1 (fun x ⇒ f x y)).

Definition fst_distr A B (m : distr (A×B)) : distr A := im_distr (fst (B:=B)) m.
Definition snd_distr A B (m : distr (A×B)) : distr B := im_distr (snd (B:=B)) m.

Add Parametric Morphism (A B : Type) : (fst_distr (A:=A) (B:=B))
   with signature Oeq ⟹ Oeq as fst_distr_eq_compat.
Save.

Add Parametric Morphism (A B : Type) : (snd_distr (A:=A) (B:=B))
   with signature Oeq ⟹ Oeq as snd_distr_eq_compat.
Save.

Lemma fst_prod_distr : ∀ A B (m1:distr A) (m2:distr B),
            fst_distr (prod_distr m1 m2) ≡ distr_scale (pone m2) m1.

Lemma snd_prod_distr : ∀ A B (m1:distr A) (m2:distr B),
           snd_distr (prod_distr m1 m2) ≡ distr_scale (pone m1) m2.

Lemma pone_prod : ∀ A B (m1:distr A) (m2:distr B),
           pone (prod_distr m1 m2) ≡ pone m1 × pone m2.
```

8.5 Independance of distribution

```
Definition prod_indep A B (m:distr (A×B)) :=
        distr_scale (pone m) m ≡ prod_distr (fst_distr m) (snd_distr m).

Lemma prod_distr_indep : ∀ A B (m1:distr A) (m2:distr B), prod_indep (prod_distr m1 m2).

Add Parametric Morphism A B : (prod_indep (A:=A) (B:=B))
   with signature Oeq ⟹ Basics.impl
   as prod_indep_eq_compat.
Save.
Hint Resolve prod_indep_eq_compat.

Lemma distr_indep_mult
   : ∀ A B (m:distr (A×B)), prod_indep m →
                        ∀ (f1 : MF A) (f2:MF B),
                        pone m × μ m (fun p ⇒ f1 (fst p) × f2 (snd p)) ≡
                        μ (fst_distr m) f1 × μ (snd_distr m) f2.
```

8.6 Range of a distribution

Definition range A (P:$A \to$ Prop) (d: **distr** A) :=
  $\forall f$, ($\forall x$, $P\ x \to 0 \equiv f\ x$) $\to 0 \equiv \mu\ d\ f$.

Lemma range_le : $\forall$ A (P:$A \to$ Prop) (d:**distr** A), range $P\ d \to$
  $\forall f\ g$, ($\forall a$, $P\ a \to f\ a \leq g\ a$) $\to \mu\ d\ f \leq \mu\ d\ g$.

Lemma range_eq : $\forall$ A (P:$A \to$ Prop) (d:**distr** A), range $P\ d \to$
  $\forall f\ g$, ($\forall a$, $P\ a \to f\ a \equiv g\ a$) $\to \mu\ d\ f \equiv \mu\ d\ g$.

9 Prog.v: Axiomatic semantics

9.1 Definition of correctness judgements

$ok\ p\ e\ q$ is defined as $p \leq \mu\ e\ q$ $up\ p\ e\ q$ is defined as $\mu\ e\ q \leq p$

Definition ok (A:Type) (p:U) (e:**distr** A) (q:$A \to U$) := $p \leq \mu\ e\ q$.

Definition okfun ($A\ B$:Type)(p:$A \to U$)(e:$A \to$ **distr** B)(q:$A \to B \to U$)
  := $\forall\ x$:A, ok ($p\ x$) ($e\ x$) ($q\ x$).

Definition okup (A:Type) (p:U) (e:**distr** A) (q:$A \to U$) := $\mu\ e\ q \leq p$.

Definition upfun ($A\ B$:Type)(p:$A \to U$)(e:$A \to$ **distr** B)(q:$A \to B \to U$)
  := $\forall\ x$:A, okup ($p\ x$) ($e\ x$) ($q\ x$).

9.2 Stability properties

Lemma ok_le_compat : $\forall$ (A:Type) ($p\ p'$:U) (e:**distr** A) ($q\ q'$:$A \to U$),
    $p' \leq p \to q \leq q' \to$ ok $p\ e\ q \to$ ok $p'\ e\ q'$.

Lemma ok_eq_compat : $\forall$ (A:Type) ($p\ p'$:U) ($e\ e'$:**distr** A) ($q\ q'$:$A \to U$),
    $p' \equiv p \to q \equiv q' \to e \equiv e' \to$ ok $p\ e\ q \to$ ok $p'\ e'\ q'$.

Add *Parametric Morphism* (A:Type) : (@ok A)
  with *signature* Ole -> Oeq ⟹ Ole ⟹ Basics.impl
  as *ok_le_morphism*.
Save.

Add *Parametric Morphism* (A:Type) : (@ok A)
  with *signature* Oeq -> Oeq ⟹ Oeq ⟹ iff
  as *ok_eq_morphism*.
Save.

Lemma okfun_le_compat :
$\forall$ ($A\ B$:Type) ($p\ p'$:$A \to U$) (e:$A \to$ **distr** B) ($q\ q'$:$A \to B \to U$),
    $p' \leq p \to q \leq q' \to$ okfun $p\ e\ q \to$ okfun $p'\ e\ q'$.

Lemma okfun_eq_compat :
$\forall$ ($A\ B$:Type) ($p\ p'$:$A \to U$) ($e\ e'$:$A \to$ **distr** B) ($q\ q'$:$A \to B \to U$),
    $p' \equiv p \to q \equiv q' \to e \equiv e' \to$ okfun $p\ e\ q \to$ okfun $p'\ e'\ q'$.

Add *Parametric Morphism* ($A\ B$:Type) : (@okfun $A\ B$)
  with *signature* Ole -> Oeq ⟹ Ole ⟹ Basics.impl
  as *okfun_le_morphism*.
Save.

Add *Parametric Morphism* ($A\ B$:Type) : (@okfun $A\ B$)
  with *signature* Oeq -> Oeq ⟹ Oeq ⟹ iff
  as *okfun_eq_morphism*.
Save.

Lemma ok_mult : $\forall$ (A:Type)(k p:U)(e:**distr** A)(f : $A \to U$),

ok p e f $\rightarrow$ ok $(k \times p)$ e (fmult k f).

Lemma ok_inv : $\forall$ (A:Type)(p:U)(e:**distr** A)(f : $A \rightarrow U$),
ok p e f $\rightarrow$ μ e (finv f) $\leq$ [1-]p.

Lemma okup_le_compat : $\forall$ (A:Type) (p p':U) (e:**distr** A) (q q':$A \rightarrow U$),
$p \leq p' \rightarrow q' \leq q \rightarrow$ okup p e $q \rightarrow$ okup p' e q'.

Lemma okup_eq_compat : $\forall$ (A:Type) (p p':U) (e e':**distr** A) (q q':$A \rightarrow U$),
$p \equiv p' \rightarrow q \equiv q' \rightarrow e \equiv e' \rightarrow$ okup p e $q \rightarrow$ okup p' e' q'.

Lemma upfun_le_compat : $\forall$ (A B:Type) (p p':$A \rightarrow U$) (e:$A \rightarrow$ **distr** B)
(q q':$A \rightarrow B \rightarrow U$),
$p \leq p' \rightarrow q' \leq q \rightarrow$ upfun p e $q \rightarrow$ upfun p' e q'.

Lemma okup_mult : $\forall$ (A:Type)(k p:U)(e:**distr** A)(f : $A \rightarrow U$), okup p e $f \rightarrow$ okup $(k \times p)$ e (fmult k f).

9.3 Basic rules

9.3.1 Rules for application:

- $ok\ r\ a\ p$ and $\forall\ x$, $ok\ (p\ x)\ (f\ x)\ q$ implies $ok\ r\ (f\ a)\ q$
- $up\ r\ a\ p$ and $\forall\ x$, $up\ (p\ x)\ (f\ x)\ q$ implies $up\ r\ (f\ a)\ q$

Lemma apply_rule : $\forall$ (A B:Type)(a:(**distr** A))(f:$A \rightarrow$ **distr** B)(r:U)(p:$A \rightarrow U$)(q:$B \rightarrow U$),
ok r a $p \rightarrow$ okfun p f (fun $x \Rightarrow q$) $\rightarrow$ ok r (Mlet a f) q.

Lemma okup_apply_rule : $\forall$ (A B:Type)(a:**distr** A)(f:$A \rightarrow$ **distr** B)(r:U)(p:$A \rightarrow U$)(q:$B \rightarrow U$),
okup r a $p \rightarrow$ upfun p f (fun $x \Rightarrow q$) $\rightarrow$ okup r (Mlet a f) q.

9.3.2 Rules for abstraction

Lemma lambda_rule : $\forall$ (A B:Type)(f:$A \rightarrow$ **distr** B)(p:$A \rightarrow U$)(q:$A \rightarrow B \rightarrow U$),
($\forall$ x:A, ok $(p\ x)$ $(f\ x)$ $(q\ x)$) $\rightarrow$ okfun p f q.

Lemma okup_lambda_rule : $\forall$ (A B:Type)(f:$A \rightarrow$ **distr** B)(p:$A \rightarrow U$)(q:$A \rightarrow B \rightarrow U$),
($\forall$ x:A, okup $(p\ x)$ $(f\ x)$ $(q\ x)$) $\rightarrow$ upfun p f q.

9.3.3 Rules for conditional

- $ok\ p1\ e1\ q$ and $ok\ p2\ e2\ q$ implies $ok\ (p1 \times \mu\ b\ (\chi\ true) + p2 \times \mu\ b\ (\chi\ false)$ if(b then $e1$ else $e2$) q
- $up\ p1\ e1\ q$ and $up\ p2\ e2\ q$ implies $up\ (p1 \times \mu\ b\ (\chi\ true) + p2 \times \mu\ b\ (\chi\ false)$ if(b then $e1$ else $e2$) q

Lemma combiok : $\forall$ (A:Type) p q ($f1$ $f2$: $A \rightarrow U$), $p \leq$ [1-]$q \rightarrow$ fplusok (fmult p $f1$) (fmult q $f2$).
Hint Extern 1 $\Rightarrow$ apply combiok.

Lemma fmult_fplusok : $\forall$ (A:Type) p q ($f1$ $f2$: $A \rightarrow U$), fplusok $f1$ $f2$ $\rightarrow$ fplusok (fmult p $f1$) (fmult q $f2$).
Hint Resolve fmult_fplusok.

Lemma ifok : $\forall$ $f1$ $f2$, fplusok (fmult $f1$ ctrue) (fmult $f2$ cfalse).
Hint Resolve ifok.

Lemma Mif_eq : $\forall$ (A:Type)(b:(**distr bool**))($f1$ $f2$:**distr** A)(q:MF A),
μ (Mif b $f1$ $f2$) $q \equiv$ (μ $f1$ q) $\times$ (μ b ctrue) + (μ $f2$ q) $\times$ (μ b cfalse).

Lemma Mif_eq2 : $\forall$ (A : Type) (b : **distr bool**) ($f1$ $f2$: **distr** A) (q : MF A),
μ (Mif b $f1$ $f2$) $q \equiv \mu$ b ctrue $\times$ μ $f1$ q + μ b cfalse $\times$ μ $f2$ q.

Lemma ifrule :
$\forall$ (A:Type)(b:(**distr bool**))($f1$ $f2$:**distr** A)($p1$ $p2$:U)(q:$A \rightarrow U$),
ok $p1$ $f1$ $q \rightarrow$ ok $p2$ $f2$ q
$\rightarrow$ ok ($p1$ $\times$ (μ b ctrue) + $p2$ $\times$ (μ b cfalse)) (Mif b $f1$ $f2$) q.

```
Lemma okup_ifrule :
  ∀ (A:Type)(b:(distr bool))(f1 f2:distr A)(p1 p2:U)(q:A → U),
       okup p1 f1 q → okup p2 f2 q
       → okup (p1 × (μ b ctrue) + p2 × (μ b cfalse)) (Mif b f1 f2) q.
```

9.3.4 Rule for fixpoints

with $\phi\ x = F\ \phi\ x$, p an increasing sequence of functions starting from 0
$\forall\ f\ i, \forall(\ x,\ ok\ (p\ i\ x)\ f\ q \Rightarrow \forall\ x,\ ok\ p\ (i{+}1)\ x\ (F\ f\ x)\ q)$ implies $\forall\ x,\ ok\ (lub\ p\ x)\ (\phi\ x)\ q$ `Section` Fixrule.

```
Variables A B : Type.

Variable F : (A → distr B) -m> (A → distr B).

Section Ruleseq.
Variable q : A → B → U.

Lemma fixrule_Ulub : ∀ (p : A → nat → U),
    (∀ x:A, p x O ≡ 0)->
    (∀ (i:nat) (f:A → distr B),
        (okfun (fun x ⇒ p x i) f q) → okfun (fun x ⇒ p x (S i)) (fun x ⇒ F f x) q)
    → okfun (fun x ⇒ Ulub (p x)) (Mfix F) q.

Lemma fixrule : ∀ (p : A → nat -m> U),
    (∀ x:A, p x O ≡ 0)->
    (∀ (i:nat) (f:A → distr B),
        (okfun (fun x ⇒ p x i) f q) → okfun (fun x ⇒ p x (S i)) (fun x ⇒ F f x) q)
    → okfun (fun x ⇒ lub (p x)) (Mfix F) q.

Lemma fixrule_up_Ulub : ∀ (p : A → nat → U),
    (∀ (i:nat) (f:A → distr B),
        (upfun (fun x ⇒ p x i) f q) → upfun (fun x ⇒ p x (S i)) (fun x ⇒ F f x) q)
    → upfun (fun x ⇒ Ulub (p x)) (Mfix F) q.

Lemma fixrule_up_lub : ∀ (p : A → nat -m> U),
    (∀ (i:nat) (f:A → distr B),
        (upfun (fun x ⇒ p x i) f q) → upfun (fun x ⇒ p x (S i)) (fun x ⇒ F f x) q)
    → upfun (fun x ⇒ lub (p x)) (Mfix F) q.

Lemma okup_fixrule_glb :
    ∀ p : A → nat -m→ U,
    (∀ (i:nat) (f:A → distr B),
         (upfun (fun x ⇒ p x i) f q) → upfun (fun x ⇒ p x (S i)) (fun x ⇒ F f x) q)
    → upfun (fun x ⇒ glb (p x)) (Mfix F) q.
End Ruleseq.

Lemma okup_fixrule_inv : ∀ (p : A → U) (q : A → B → U),
    (∀ (f:A → distr B), upfun p f q → upfun p (fun x ⇒ F f x) q)
            → upfun p (Mfix F) q.
```

9.3.5 Rules using commutation properties

```
Section TransformFix.

Section Fix_muF.
Variable q : A → B → U.
Variable muF : MF A -m> MF A.

Definition admissible (P:(A → distr B) → Prop) := P 0 ∧ ∀ f, P f → P (F f).

Lemma admissible_true : admissible (fun f ⇒ True).

Lemma admissible_le_fix :
```

continuous ($D1$:=$A \to$ **distr** B) ($D2$:=$A \to$ **distr** B) $F \to$ admissible (fun $f \Rightarrow f \leq$ Mfix F).
BUG: rewrite fails

```
Lemma muF_stable : stable muF.

Definition mu_muF_commute_le :=
  ∀ f x, f ≤ Mfix F → μ (F f x) (q x) ≤ muF (fun y ⇒ μ (f y) (q y)) x.
Hint Unfold mu_muF_commute_le.

Section F_muF_results.
Hypothesis F_muF_le : mu_muF_commute_le.

Lemma mu_mufix_le : ∀ x, μ (Mfix F x) (q x) ≤ mufix muF x.
Hint Resolve mu_mufix_le.

Lemma muF_le : ∀ f, muF f ≤ f
      → ∀ x, μ (Mfix F x) (q x) ≤ f x.

Hypothesis muF_F_le :
    ∀ f x, f ≤ Mfix F → muF (fun y ⇒ μ (f y) (q y)) x ≤ μ (F f x) (q x).

Lemma mufix_mu_le : ∀ x, mufix muF x ≤ μ (Mfix F x) (q x).

End F_muF_results.
Hint Resolve mu_mufix_le mufix_mu_le.

Lemma mufix_mu :
   (∀ f x, f ≤ Mfix F → μ (F f x) (q x) ≡ muF (fun y ⇒ μ (f y) (q y)) x)
   → ∀ x, mufix muF x ≡ μ (Mfix F x) (q x).
Hint Resolve mufix_mu.
End Fix_muF.

Section Fix_Term.

Definition pterm : MF A := fun (x:A) ⇒ μ (Mfix F x) (fone B).
Variable muFone : MF A -m> MF A.

Hypothesis F_muF_eq_one :
  ∀ f x, f ≤ Mfix F → μ (F f x) (fone B) ≡ muFone (fun y ⇒ μ (f y) (fone B)) x.

Hypothesis muF_cont : continuous muFone.

Lemma muF_pterm : pterm ≡ muFone pterm.
Hint Resolve muF_pterm.
End Fix_Term.

Section Fix_muF_Term.

Variable q : A → B → U.
Definition qinv x y := [1-] q x y.

Variable muFqinv : MF A -m> MF A.

Hypothesis F_muF_le_inv : mu_muF_commute_le qinv muFqinv.

Lemma muF_le_term : ∀ f, muFqinv (finv f) ≤ finv f →
    ∀ x, f x & pterm x ≤ μ (Mfix F x) (q x).

Lemma muF_le_term_minus :
∀ f, f ≤ pterm → muFqinv (fminus pterm f) ≤ fminus pterm f →
    ∀ x, f x ≤ μ (Mfix F x) (q x).

Variable muFq : MF A -m> MF A.

Hypothesis F_muF_le : mu_muF_commute_le q muFq.

Lemma muF_eq : ∀ f, muFq f ≤ f → muFqinv (finv f) ≤ finv f →
    ∀ x, pterm x ≡ 1 → μ (Mfix F x) (q x) ≡ f x.

End Fix_muF_Term.
End TransformFix.
```

```
Section LoopRule.
Variable q : A → B → U.
Variable stop : A → distr bool.
Variable step : A → distr A.
Variable a : U.

Definition Loop : MF A -m> MF A.
Defined.

Lemma Loop_eq :
    ∀ f x, Loop f x = μ (stop x) (fun b ⇒ if b then a else μ (step x) f).

Definition loop := mufix Loop.

Lemma Mfixvar :
  (∀ (f:A → distr B),
      okfun (fun x ⇒ Loop (fun y ⇒ μ (f y) (q y)) x) (fun x ⇒ F f x) q)
 → okfun loop (Mfix F) q.

Definition up_loop : MF A := nufix Loop.

Lemma Mfixvar_up :
  (∀ (f:A → distr B),
      upfun (fun x ⇒ Loop (fun y ⇒ μ (f y) (q y)) x) (fun x ⇒ F f x) q)
 → upfun up_loop (Mfix F) q.

End LoopRule.

End Fixrule.
```

9.4 Rules for intervals

Distributions operates on intervals

```
Definition lmu : ∀ A:Type, distr A → (A → IU) → IU.
Defined.

Lemma low_lmu : ∀ (A:Type) (e:distr A) (F: A → IU),
                low (lmu e F) = μ e (fun x ⇒ low (F x)).

Lemma up_lmu : ∀ (A:Type) (e:distr A) (F: A → IU),
               up (lmu e F) = μ e (fun x ⇒ up (F x)).

Lemma lmu_monotonic : ∀ (A:Type) (e:distr A) (F G : A → IU),
              (∀ x, lincl (F x) (G x)) → lincl (lmu e F) (lmu e G).

Lemma lmu_stable_eq : ∀ (A:Type) (e:distr A) (F G : A → IU),
              (∀ x, leq (F x) (G x)) → leq (lmu e F) (lmu e G).
Hint Resolve lmu_monotonic lmu_stable_eq.

Lemma lmu_singl : ∀ (A:Type) (e:distr A) (f:A → U),
            leq (lmu e (fun x ⇒ singl (f x))) (singl (μ e f)).

Lemma lmu_inf : ∀ (A:Type) (e:distr A) (f:A → U),
            leq (lmu e (fun x ⇒ inf (f x))) (inf (μ e f)).

Lemma lmu_sup : ∀ (A:Type) (e:distr A) (f:A → U),
            lincl (lmu e (fun x ⇒ sup (f x))) (sup (μ e f)).

Lemma lin_mu_lmu :
   ∀ (A:Type) (e:distr A) (F:A → IU) (f:A → U),
            (∀ x, lin (f x) (F x)) → lin (μ e f) (lmu e F).
Hint Resolve lin_mu_lmu.

Definition lok (A:Type) (I:IU) (e:distr A) (F:A → IU) := lincl (lmu e F) I.
Definition lokfun (A B:Type)(I:A → IU) (e:A → distr B) (F:A → B → IU)
                  := ∀ x, lok (I x) (e x) (F x).
```

```
Lemma lin_mu_lok :
  ∀ (A:Type) (I:IU) (e:distr A) (F:A → IU) (f:A → U),
        (∀ x, lin (f x) (F x)) → lok I e F → lin (μ e f) I.
```

9.4.1 Stability

```
Lemma lok_le_compat : ∀ (A:Type) (I J:IU) (e:distr A) (F G:A → IU),
  lincl I J → (∀ x, lincl (G x) (F x)) → lok I e F → lok J e G.

Lemma lokfun_le_compat : ∀ (A B:Type) (I J:A → IU) (e:A → distr B) (F G:A → B → IU),
  (∀ x, lincl (I x) (J x)) → (∀ x y, lincl (G x y) (F x y)) → lokfun I e F → lokfun J e G.
```

9.4.2 Rule for values

```
Lemma lunit_eq : ∀ (A: Type) (a:A) (F:A → IU), leq (lmu (Munit a) F) (F a).
```

9.4.3 Rule for application

```
Lemma llet_eq : ∀ (A B : Type) (a:distr A) (f:A → distr B)(I:IU)(G:B → IU),
  leq (lmu (Mlet a f) G) (lmu a (fun x ⇒ lmu (f x) G)).
Hint Resolve llet_eq.

Lemma lapply_rule : ∀ (A B : Type) (a:distr A) (f:A → distr B)(I:IU)(F:A → IU)(G:B → IU),
    lok I a F → lokfun F f (fun x ⇒ G) → lok I (Mlet a f) G.
```

9.4.4 Rule for abstraction

```
Lemma llambda_rule : ∀ (A B:Type)(f:A → distr B)(F:A → IU)(G:A → B → IU),
  (∀ x:A, lok (F x) (f x) (G x)) → lokfun F f G.
```

9.4.5 Rule for conditional

```
Lemma lmu_Mif_eq : ∀ (A:Type)(b:distr bool)(f1 f2:distr A)(F:A → IU),
 leq (lmu (Mif b f1 f2) F) (lplus (lmultk (μ b ctrue) (lmu f1 F)) (lmultk (μ b cfalse) (lmu f2 F))).

Lemma lifrule :
  ∀ (A:Type)(b:(distr bool))(f1 f2:distr A)(I1 I2:IU)(F:A → IU),
      lok I1 f1 F → lok I2 f2 F
      → lok (lplus (lmultk (μ b ctrue) I1) (lmultk (μ b cfalse) I2)) (Mif b f1 f2) F.
```

9.4.6 Rule for fixpoints

with $\phi\ x = F\ \phi\ x$, p a decreasing sequence of intervals functions ($p\ (i{+}1)\ x$ is a bubset of $(p\ i\ x)$ such that $(p\ 0\ x)$ contains 0 for all x.

$\forall\ f\ i, \forall(\ x, iok\ (p\ i\ x)\ f\ (q\ x)) \Rightarrow \forall\ x, iok\ (p\ (i{+}1)\ x)\ (F\ f\ x)\ (q\ x)$ implies $\forall\ x, iok\ (lub\ p\ x)\ (\phi\ x)\ (q\ x)$

```
Section lFixrule.
Variables A B : Type.

Variable F : (A → distr B) -m> (A → distr B).

Section lRuleseq.
Variable Q : A → B → IU.

Variable I : A → nat -m> IU.

Lemma lfixrule :
  (∀ x:A, lin 0 (I x 0)) →
  (∀ (i:nat) (f:A → distr B),
      (lokfun (fun x ⇒ I x i) f Q) → lokfun (fun x ⇒ I x (S i)) (fun x ⇒ F f x) Q)
  → lokfun (fun x ⇒ lim (I x)) (Mfix F) Q.
```

```
End IRuleseq.

Section ITransformFix.

Section IFix_muF.
Variable Q : A → B → IU.
Variable ImuF : (A → IU) -m> (A → IU).

Lemma ImuF_stable : ∀ I J, I≡J → ImuF I ≡ ImuF J.

Section IF_muF_results.
Hypothesis Iincl_F_ImuF :
    ∀ f x, f ≤ Mfix F →
                        Iincl (Imu (F f x) (Q x)) (ImuF (fun y ⇒ Imu (f y) (Q y)) x).

Lemma Iincl_fix_ifix : ∀ x, Iincl (Imu (Mfix F x) (Q x)) (fixp (D:=A → IU) ImuF x).
Hint Resolve Iincl_fix_ifix.

End IF_muF_results.

End IFix_muF.
End ITransformFix.
End IFixrule.
```

9.5 Rules for *Flip*

```
Lemma Flip_ctrue : μ Flip ctrue ≡ 1/2.

Lemma Flip_cfalse : μ Flip cfalse ≡ 1/2.

Lemma ok_Flip : ∀ q : bool → U, ok (1/2 × q true + 1/2 × q false) Flip q.

Lemma okup_Flip : ∀ q : bool → U, okup (1/2 × q true + 1/2 × q false) Flip q.

Hint Resolve ok_Flip okup_Flip Flip_ctrue Flip_cfalse.

Lemma Flip_eq : ∀ q : bool → U, μ Flip q ≡ 1/2 × q true + 1/2 × q false.
Hint Resolve Flip_eq.

Lemma IFlip_eq : ∀ Q : bool → IU, Ieq (Imu Flip Q) (Iplus (Imultk 1/2 (Q true)) (Imultk 1/2 (Q false))).
Hint Resolve IFlip_eq.
```

9.6 Rules for total (well-founded) fixpoints

```
Section Wellfounded.
Variables A B : Type.
Variable R : A → A → Prop.
Hypothesis Rwf : well_founded R.
Variable F : ∀ x, (∀ y, R y x → distr B) → distr B.

Definition WfFix : A → distr B := Fix Rwf (fun _ ⇒ distr B) F.

Hypothesis Fext : ∀ x f g, (∀ y (p:R y x), f y p ≡ g y p) → F f ≡ F g.

Lemma Acc_iter_distr :
    ∀ x, ∀ r s : Acc R x, Acc_iter (fun _⇒ distr B) F r ≡ Acc_iter (fun _⇒ distr B) F s.

Lemma WfFix_eq : ∀ x, WfFix x ≡ F (fun (y:A) (p:R y x) ⇒ WfFix y).

Variable P : distr B → Prop.
Hypothesis Pext : ∀ m1 m2, m1 ≡ m2 → P m1 → P m2.

Lemma WfFix_ind :
      (∀ x f, (∀ y (p:R y x), P (f y p)) → P (F f))
  → ∀ x, P (WfFix x).

End Wellfounded.
```

```
Ltac distrsimpl := match goal with
 | ⊢ (Ole (fmont (μ ?d1) ?f) (fmont (μ ?d2) ?g)) ⇒ apply (mu_le_compat (m1:=d1) (m2:=d2) (Ole_refl d1)
(f:=f) (g:=g)); intro
 | ⊢ (Oeq (fmont (μ ?d1) ?f) (fmont (μ ?d2) ?g)) ⇒ apply (mu_eq_compat (m1:=d1) (m2:=d2) (Oeq_refl d1)
(f:=f) (g:=g)); intro
 | ⊢ (Oeq (Munit ?x) (Munit ?y)) ⇒ apply (Munit_eq_compat x y)
 | ⊢ (Oeq (Mlet ?x1 ?f) (Mlet ?x2 ?g))
                ⇒ apply (Mlet_eq_compat (m1:=x1) (m2:=x2) (M1:=f) (M2:=g) (Oeq_refl x1)); intro
 | ⊢ (Ole (Mlet ?x1 ?f) (Mlet ?x2 ?g))
                ⇒ apply (Mlet_le_compat (m1:=x1) (m2:=x2) (M1:=f) (M2:=g) (Ole_refl x1)); intro
 | ⊢ context [(fmont (μ (Mlet ?m ?M)) ?f)] ⇒ rewrite (Mlet_simpl m M f)
 | ⊢ context [(fmont (μ (Munit ?x)) ?f)] ⇒ rewrite (Munit_simpl f x)
 | ⊢ context [(Mlet (Mlet ?m ?M) ?f)] ⇒ rewrite (Mlet_assoc m M f)
 | ⊢ context [(Mlet (Munit ?x) ?f)] ⇒ rewrite (Mlet_unit x f)
 | ⊢ context [(fmont (μ Flip) ?f)] ⇒ rewrite (Flip_simpl f)
 | ⊢ context [(fmont (μ (Discrete ?d)) ?f)] ⇒ rewrite (Discrete_simpl d);
                                                    rewrite (discrete_simpl (coeff d) (points d)
f)
 | ⊢ context [(fmont (μ (Random ?n)) ?f)] ⇒ rewrite (Random_simpl n);
                                                    rewrite (random_simpl n f)
 | ⊢ context [(fmont (μ (Mif ?b ?f ?g)) ?h)] ⇒ unfold Mif
 | ⊢ context [(fmont (μ (Mchoice ?p ?m1 ?m2)) ?f)] ⇒ rewrite (Mchoice_simpl p m1 m2 f)
 | ⊢ context [(fmont (μ (im_distr ?f ?m)) ?h)] ⇒ rewrite (im_distr_simpl f m h)
 | ⊢ context [(fmont (μ (prod_distr ?m1 ?m2)) ?h)] ⇒ unfold prod_distr
 | ⊢ context [((mon ?f (fmonotonic:=?mf)) ?x)] ⇒ rewrite (mon_simpl f (mf:=mf) x)
end.

Require Export Setoid.
Require Omega.
```

10 Sets.v: Definition of sets as predicates over a type A

```
Section sets.
Variable A : Type.
Variable decA : ∀ x y :A, {x=y}+{x≠y}.

Definition set := A→Prop.
Definition full : set := fun (x:A) ⇒ True.
Definition empty : set := fun (x:A) ⇒ False.
Definition add (a:A) (P:set) : set := fun (x:A) ⇒ x=a ∨ (P x).
Definition singl (a:A) :set := fun (x:A) ⇒ x=a.
Definition union (P Q:set) :set := fun (x:A) ⇒ (P x) ∨ (Q x).
Definition compl (P:set) :set := fun (x:A) ⇒ ∉ P x.
Definition inter (P Q:set) :set := fun (x:A) ⇒ (P x) ∧ (Q x).
Definition rem (a:A) (P:set) :set := fun (x:A) ⇒ x≠a ∧ (P x).
```

10.1 Equivalence

```
Definition eqset (P Q:set) := ∀ (x:A), P x ↔ Q x.

Implicit Arguments full [].
Implicit Arguments empty [].

Lemma eqset_refl : ∀ P:set, eqset P P.

Lemma eqset_sym : ∀ P Q:set, eqset P Q → eqset Q P.

Lemma eqset_trans : ∀ P Q R:set,
```

eqset P Q → eqset Q R → eqset P R.

```
Hint Resolve eqset_refl.
Hint Immediate eqset_sym.
```

10.2 Setoid structure

```
Lemma set_setoid : Setoid_Theory set eqset.

Add Setoid set eqset set_setoid as Set_setoid.

Add Morphism add : eqset_add.
Save.

Add Morphism rem : eqset_rem.
Save.
Hint Resolve eqset_add eqset_rem.

Add Morphism union : eqset_union.
Save.
Hint Immediate eqset_union.

Lemma eqset_union_left :
  ∀ P1 Q P2,
    eqset P1 P2 → eqset (union P1 Q) (union P2 Q).

Lemma eqset_union_right :
  ∀ P Q1 Q2 ,
    eqset Q1 Q2 → eqset (union P Q1) (union P Q2).

Hint Resolve eqset_union_left eqset_union_right.

Add Morphism inter : eqset_inter.
Save.
Hint Immediate eqset_inter.

Add Morphism compl : eqset_compl.
Save.
Hint Resolve eqset_compl.

Lemma eqset_add_empty : ∀ (a:A) (P:set), ~eqset (add a P) empty.
```

10.3 Finite sets given as an enumeration of elements

```
Inductive finite (P: set) : Type :=
    fin_eq_empty : eqset P empty → finite P
  | fin_eq_add : ∀ (x:A)(Q:set),
                 ~Q x→ finite Q → eqset P (add x Q) → finite P.
Hint Constructors finite.

Lemma fin_empty : (finite empty).

Lemma fin_add : ∀ (x:A)(P:set),
                 ~P x → finite P → finite (add x P).

Lemma fin_eqset: ∀ (P Q : set), (eqset P Q)->(finite P)->(finite Q).

Hint Resolve fin_empty fin_add.
```

10.3.1 Emptyness is decidable for finite sets

```
Definition isempty (P:set) := eqset P empty.
Definition notempty (P:set) := not (eqset P empty).

Lemma isempty_dec : ∀ P, finite P → {isempty P}+{notempty P}.
```

10.3.2 Size of a finite set

```
Fixpoint size (P:set) (f:finite P) {struct f}: nat :=
  match f with fin_eq_empty _ ⇒ 0%nat
             | fin_eq_add _ Q _ f' _ ⇒ S (size f')
  end.

Lemma size_eqset : ∀ P Q (f:finite P) (e:eqset P Q),
    (size (fin_eqset e f)) = (size f).
```

10.4 Inclusion

```
Definition incl (P Q:set) := ∀ x, P x → Q x.

Lemma incl_refl : ∀ (P:set), incl P P.

Lemma incl_trans : ∀ (P Q R:set),
incl P Q → incl Q R → incl P R.

Lemma eqset_incl : ∀ (P Q : set), eqset P Q → incl P Q.

Lemma eqset_incl_sym : ∀ (P Q : set), eqset P Q → incl Q P.

Lemma eqset_incl_intro :
∀ (P Q : set), incl P Q → incl Q P → eqset P Q.

Hint Resolve incl_refl incl_trans eqset_incl_intro.
Hint Immediate eqset_incl eqset_incl_sym.
```

10.5 Properties of operations on sets

```
Lemma incl_empty : ∀ P, incl empty P.

Lemma incl_empty_false : ∀ P a, incl P empty → ⁄P a.

Lemma incl_add_empty : ∀ (a:A) (P:set), ⁄incl (add a P) empty.

Lemma eqset_empty_false : ∀ P a, eqset P empty → P a → False.

Hint Immediate incl_empty_false eqset_empty_false incl_add_empty.

Lemma incl_rem_stable : ∀ a P Q, incl P Q → incl (rem a P) (rem a Q).

Lemma incl_add_stable : ∀ a P Q, incl P Q → incl (add a P) (add a Q).

Lemma incl_rem_add_iff :
  ∀ a P Q, incl (rem a P) Q ↔ incl P (add a Q).

Lemma incl_rem_add:
  ∀ (a:A) (P Q:set),
     (P a) → incl Q (rem a P) → incl (add a Q) P.

Lemma incl_add_rem :
  ∀ (a:A) (P Q:set),
     ⁄Q a → incl (add a Q) P → incl Q (rem a P) .

Hint Immediate incl_rem_add incl_add_rem.

Lemma eqset_rem_add :
 ∀ (a:A) (P Q:set),
     (P a) → eqset Q (rem a P) → eqset (add a Q) P.

Lemma eqset_add_rem :
 ∀ (a:A) (P Q:set),
     ⁄Q a → eqset (add a Q) P → eqset Q (rem a P).

Hint Immediate eqset_rem_add eqset_add_rem.

Lemma add_rem_eq_eqset :
```

∀ x (P:set), eqset (add x (rem x P)) (add x P).

```
Lemma add_rem_diff_eqset :
  ∀ x y (P:set),
  x≠y → eqset (add x (rem y P)) (rem y (add x P)).

Lemma add_eqset_in :
  ∀ x (P:set), P x → eqset (add x P) P.

Hint Resolve add_rem_eq_eqset add_rem_diff_eqset add_eqset_in.

Lemma add_rem_eqset_in :
  ∀ x (P:set), P x → eqset (add x (rem x P)) P.

Hint Resolve add_rem_eqset_in.

Lemma rem_add_eq_eqset :
  ∀ x (P:set), eqset (rem x (add x P)) (rem x P).

Lemma rem_add_diff_eqset :
  ∀ x y (P:set),
  x≠y → eqset (rem x (add y P)) (add y (rem x P)).

Lemma rem_eqset_notin :
  ∀ x (P:set), ~P x → eqset (rem x P) P.

Hint Resolve rem_add_eq_eqset rem_add_diff_eqset rem_eqset_notin.

Lemma rem_add_eqset_notin :
  ∀ x (P:set), ~P x → eqset (rem x (add x P)) P.

Hint Resolve rem_add_eqset_notin.

Lemma rem_not_in : ∀ x (P:set), ~rem x P x.

Lemma add_in : ∀ x (P:set), add x P x.

Lemma add_in_eq : ∀ x y P, x=y → add x P y.

Lemma add_intro : ∀ x (P:set) y, P y → add x P y.

Lemma add_incl : ∀ x (P:set), incl P (add x P).

Lemma add_incl_intro : ∀ x (P Q:set), (Q x) → (incl P Q) → (incl (add x P) Q).

Lemma rem_incl : ∀ x (P:set), incl (rem x P) P.

Hint Resolve rem_not_in add_in rem_incl add_incl.

Lemma union_sym : ∀ P Q : set,
      eqset (union P Q) (union Q P).

Lemma union_empty_left : ∀ P : set,
      eqset P (union P empty).

Lemma union_empty_right : ∀ P : set,
      eqset P (union empty P).

Lemma union_add_left : ∀ (a:A) (P Q: set),
      eqset (add a (union P Q)) (union P (add a Q)).

Lemma union_add_right : ∀ (a:A) (P Q: set),
      eqset (add a (union P Q)) (union (add a P) Q).

Hint Resolve union_sym union_empty_left union_empty_right
union_add_left union_add_right.

Lemma union_incl_left : ∀ P Q, incl P (union P Q).

Lemma union_incl_right : ∀ P Q, incl Q (union P Q).

Lemma union_incl_intro : ∀ P Q R, incl P R → incl Q R → incl (union P Q) R.

Hint Resolve union_incl_left union_incl_right union_incl_intro.
```

```
Lemma incl_union_stable : ∀ P1 P2 Q1 Q2,
        incl P1 P2 → incl Q1 Q2 → incl (union P1 Q1) (union P2 Q2).
Hint Immediate incl_union_stable.

Lemma inter_sym : ∀ P Q : set,
      eqset (inter P Q) (inter Q P).

Lemma inter_empty_left : ∀ P : set,
      eqset empty (inter P empty).

Lemma inter_empty_right : ∀ P : set,
      eqset empty (inter empty P).

Lemma inter_add_left_in : ∀ (a:A) (P Q: set),
      (P a) → eqset (add a (inter P Q)) (inter P (add a Q)).

Lemma inter_add_left_out : ∀ (a:A) (P Q: set),
      ∉P a → eqset (inter P Q) (inter P (add a Q)).

Lemma inter_add_right_in : ∀ (a:A) (P Q: set),
      Q a → eqset (add a (inter P Q)) (inter (add a P) Q).

Lemma inter_add_right_out : ∀ (a:A) (P Q: set),
      ∉Q a → eqset (inter P Q) (inter (add a P) Q).

Hint Resolve inter_sym inter_empty_left inter_empty_right
inter_add_left_in inter_add_left_out inter_add_right_in inter_add_right_out.
```

10.6 Generalized union

```
Definition gunion (I:Type)(F:I→set) : set := fun z ⇒ ∃ i, F i z.

Lemma gunion_intro : ∀ I (F:I→set) i, incl (F i) (gunion F).

Lemma gunion_elim : ∀ I (F:I→set) (P:set), (∀ i, incl (F i) P) → incl (gunion F) P.

Lemma gunion_monotonic : ∀ I (F G : I → set),
      (∀ i, incl (F i) (G i))-> incl (gunion F) (gunion G).
```

10.7 Removing an element from a finite set

```
Lemma finite_rem : ∀ (P:set) (a:A),
   finite P → finite (rem a P).

Lemma size_finite_rem:
   ∀ (P:set) (a:A) (f:finite P),
     (P a) → size f = S (size (finite_rem a f)).

Require Import Arith.

Lemma size_incl :
  ∀ (P:set)(f:finite P) (Q:set)(g:finite Q),
  (incl P Q)-> size f ≤ size g.

Lemma size_unique :
  ∀ (P:set)(f:finite P) (Q:set)(g:finite Q),
  (eqset P Q)-> size f = size g.
```

10.8 Decidable sets

```
Definition dec (P:set) := ∀ x, {P x}+{∉P x}.

Lemma finite_incl : ∀ P:set,
   finite P → ∀ Q:set, dec Q → incl Q P → finite Q.

Lemma finite_dec : ∀ P:set, finite P → dec P.
```

```
Lemma fin_add_in : ∀ (a:A) (P:set), finite P → finite (add a P).
Lemma finite_union :
      ∀ P Q, finite P → finite Q → finite (union P Q).
Lemma finite_full_dec : ∀ P:set, finite full → dec P → finite P.
Require Import Lt.
```

10.8.1 Filter operation

```
Lemma finite_inter : ∀ P Q, dec P → finite Q → finite (inter P Q).
Lemma size_inter_empty : ∀ P Q (decP:dec P) (e:eqset Q empty),
   size (finite_inter decP (fin_eq_empty e))=O.
Lemma size_inter_add_in :
      ∀ P Q R (decP:dec P)(x:A)(nq:∉ x)(FQ:finite Q)(e:eqset R (add x Q)),
        P x →size (finite_inter decP (fin_eq_add nq FQ e))=S (size (finite_inter decP FQ)).
Lemma size_inter_add_notin :
      ∀ P Q R (decP:dec P)(x:A)(nq:∉ x)(FQ:finite Q)(e:eqset R (add x Q)),
    ∉P x → size (finite_inter decP (fin_eq_add nq FQ e))=size (finite_inter decP FQ).
Lemma size_inter_incl : ∀ P Q (decP:dec P)(FP:finite P)(FQ:finite Q),
     (incl P Q) → size (finite_inter decP FQ)=size FP.
```

10.8.2 Selecting elements in a finite set

```
Fixpoint nth_finite (P:set) (k:nat) (PF : finite P) {struct PF}: (k < size PF) → A :=
  match PF as F return (k < size F) → A with
        fin_eq_empty H ⇒ (fun (e : k<0) ⇒ match lt_n_O k e with end)
      | fin_eq_add x Q nqx fq eqq ⇒
            match k as k0 return k0<S (size fq)->A with
                  O ⇒ fun e ⇒ x
          | (S k1) ⇒ fun (e:S k1<S (size fq)) ⇒ nth_finite fq (lt_S_n k1 (size fq) e)
            end
  end.
```

A set with size > 1 contains at least 2 different elements

```
Lemma select_non_empty : ∀ (P:set), finite P → notempty P → sigT P.
Lemma select_diff : ∀ (P:set) (FP:finite P),
      (1 < size FP)%nat → sigT (fun x ⇒ sigT (fun y ⇒ P x ∧ P y ∧ x≠y)).
End sets.
Hint Resolve eqset_refl.
Hint Resolve eqset_add eqset_rem.
Hint Immediate eqset_sym finite_dec finite_full_dec eqset_incl eqset_incl_sym eqset_incl_intro.
Hint Resolve incl_refl.
Hint Immediate incl_union_stable.
Hint Resolve union_incl_left union_incl_right union_incl_intro incl_empty rem_incl
incl_rem_stable incl_add_stable.
Hint Constructors finite.
Hint Resolve add_in add_in_eq add_intro add_incl add_incl_intro union_sym union_empty_left union_empty_right
union_add_left union_add_right finite_union eqset_union_left
eqset_union_right.
Implicit Arguments full [].
Implicit Arguments empty [].
Add Parametric Relation (A:Type) : (set A) (eqset (A:=A))
```

```
      reflexivity proved by (eqset_refl (A:=A))
      symmetry proved by (eqset_sym (A:=A))
      transitivity proved by (eqset_trans (A:=A))
as eqset_rel.

Add Parametric Relation (A:Type) : (set A) (incl (A:=A))
      reflexivity proved by (incl_refl (A:=A))
      transitivity proved by (incl_trans (A:=A))
as incl_rel.
```

11 Cover.v: Characteristic functions

```
Require Export Prog.
Require Export Sets.
Require Export Arith.
```

Properties of zero_one functions

```
Definition zero_one (A:Type)(f:MF A) := ∀ x, orc (f x ≡ 0) (f x ≡ 1).
Hint Unfold zero_one.

Lemma zero_one_not_one :
∀ (A:Type)(f:MF A) x, zero_one f → 1 ≤ f x → f x ≡ 0.

Lemma zero_one_not_zero :
∀ (A:Type)(f:MF A) x, zero_one f → f x ≤ 0 → f x ≡ 1.

Hint Resolve zero_one_not_one zero_one_not_zero.

Lemma ctrue_zero_one: zero_one ctrue.

Lemma cfalse_zero_one: zero_one cfalse.

Lemma ctrue_zero_one2: ∀ b:bool,
  orc ((if b then 1 else 0) ≡ 0) ((if b then 1 else 0) ≡ 1).

Lemma cfalse_zero_one2: ∀ b:bool,
  orc ((if b then 0 else 1) ≡ 0) ((if b then 0 else 1) ≡ 1).

Hint Immediate ctrue_zero_one cfalse_zero_one ctrue_zero_one2 cfalse_zero_one2.

Definition fesp_zero_one : ∀ (A:Type)(f g:MF A),
      zero_one f → zero_one g → zero_one (fesp f g).
Save.

Lemma fesp_conj_zero_one : ∀ (A:Type)(f g:MF A),
      zero_one f → fesp f g ≡ fconj f g.

Lemma fconj_zero_one : ∀ (A:Type)(f g:MF A),
      zero_one f → zero_one g → zero_one (fconj f g).

Lemma fplus_zero_one : ∀ (A:Type)(f g:MF A),
      zero_one f → zero_one g → zero_one (fplus f g).

Lemma finv_zero_one : ∀ (A:Type)(f :MF A),
      zero_one f → zero_one (finv f).

Lemma fesp_zero_one_mult_left : ∀ (A:Type)(f:MF A)(p:U),
      zero_one f → ∀ x, f x & p ≡ f x × p.

Lemma fesp_zero_one_mult_right : ∀ (A:Type)(p:U)(f:MF A),
     zero_one f → ∀ x, p & f x ≡ p × f x.
Hint Resolve fesp_zero_one_mult_left fesp_zero_one_mult_right.
```

11.1 Covering functions

```
Definition cover (A:Type)(P:set A)(f:MF A) :=
      ∀ x, (P x → 1 ≤ f x) ∧ (/P x → f x ≤ 0).

Lemma cover_eq_one : ∀ (A:Type)(P:set A)(f:MF A) (z:A),
       cover P f → P z → f z ≡ 1.

Lemma cover_eq_zero : ∀ (A:Type)(P:set A)(f:MF A) (z:A),
       cover P f → /P z → f z ≡ 0.

Lemma cover_orc_0_1 : ∀ (A:Type)(P:set A)(f:MF A),
   cover P f → ∀ x, orc (f x ≡ 0) (f x ≡ 1).

Lemma cover_zero_one : ∀ (A:Type)(P:set A)(f:MF A),
   cover P f → zero_one f.

Lemma zero_one_cover : ∀ (A:Type)(f:MF A),
   zero_one f → cover (fun x ⇒ 1 ≤ f x) f.

Lemma cover_esp_mult_left : ∀ (A:Type)(P:set A)(f:MF A)(p:U),
       cover P f → ∀ x, f x & p ≡ f x × p.

Lemma cover_esp_mult_right : ∀ (A:Type)(P:set A)(p:U)(f:MF A),
     cover P f → ∀ x, p & f x ≡ p × f x.
Hint Immediate cover_esp_mult_left cover_esp_mult_right.

Lemma cover_elim : ∀ (A:Type)(P:set A)(f:MF A),
cover P f → ∀ x, orc (/P x ∧ f x ≡ 0) (P x ∧ f x ≡ 1).

Lemma cover_eq_one_elim_class : ∀ (A:Type)(P Q:set A)(f:MF A),
       cover P f → ∀ z, f z ≡ 1 → class (Q z) → incl P Q → Q z.

Lemma cover_eq_one_elim : ∀ (A:Type)(P:set A)(f:MF A),
       cover P f → ∀ z, f z ≡ 1 → //P z.

Lemma cover_eq_zero_elim : ∀ (A:Type)(P:set A)(f:MF A) (z:A),
       cover P f → f z ≡ 0 → /P z.

Lemma cover_unit : ∀ (A:Type)(P:set A)(f:MF A)(a:A),
              cover P f → P a → 1 ≤ μ (Munit a) f.

Lemma cover_let : ∀ (A B:Type)(m1: distr A)(m2: A→distr B) (P:set A)(cP:MF A)(f:MF B)(p:U),
     cover P cP → 1 ≤ μ m1 cP → (∀ x:A, P x → p ≤ μ (m2 x) f) → p ≤ μ (Mlet m1 m2) f.

Lemma cover_incl_fle : ∀ (A:Type)(P Q:set A)(f g:MF A),
       cover P f → cover Q g → incl P Q → f ≤ g.

Lemma cover_incl_eq: ∀ (A:Type)(P:set A)(f g:MF A),
       cover P f → cover P g → f ≡ g.

Lemma cover_eqset_stable : ∀ (A:Type)(P Q:set A)(EQ:eqset P Q)(f:MF A),
       cover P f → cover Q f.

Lemma cover_eq_stable : ∀ (A:Type)(P:set A)(f g:MF A),
       cover P f → f ≡ g → cover P g.

Lemma cover_eqset_eq_stable : ∀ (A:Type)(P Q:set A)(f g:MF A),
       cover P f → eqset P Q → f ≡ g → cover Q g.

Add Parametric Morphism (A:Type) : (cover (A:=A))
with signature eqset (A:=A) ⟹ Oeq ⟹ iff as cover_eqset_compat.
Save.

Lemma cover_union : ∀ (A:Type)(P Q:set A)(f g : MF A),
     cover P f → cover Q g → cover (union P Q) (fplus f g).

Lemma cover_inter_esp : ∀ (A:Type)(P Q:set A)(f g : MF A),
       cover P f → cover Q g → cover (inter P Q) (fesp f g).
```

Lemma cover_inter_mult : $\forall$ (A:Type)(P Q:set A)(f g : MF A),
cover P f $\rightarrow$ cover Q g $\rightarrow$ cover (inter P Q) (fun $x \Rightarrow f\ x \times g\ x$).

Lemma cover_compl : $\forall$ (A:Type)(P:set A)(f:MF A),
cover P f $\rightarrow$ cover (compl P) (finv f).

Lemma cover_empty : $\forall$ (A:Type), cover (empty A) (fzero A).

Lemma cover_comp : $\forall$ (A B:Type)(h:$A\rightarrow B$)(P:set B)(f:MF B),
cover P f $\rightarrow$ cover (fun $a \Rightarrow P\ (h\ a)$) (fun $a \Rightarrow f\ (h\ a)$).

11.2 Caracteristic functions for decidable predicates

Definition carac (A:Type)(P:set A)($Pdec$: dec P) : MF A
:= fun $z \Rightarrow$ if $Pdec\ z$ then 1 else 0.

Lemma carac_incl: $\forall$ (A:Type)(P Q:$A \rightarrow$ Prop)($Pdec$: dec P)($Qdec$: dec Q),
incl P Q $\rightarrow$ carac $Pdec \leq$ carac $Qdec$.

Lemma carac_monotonic : $\forall$ (A B:Type)(P:$A \rightarrow$ Prop)(Q:$B\rightarrow$Prop)($Pdec$: dec P)($Qdec$: dec Q) x y,
$(P\ x \rightarrow Q\ y) \rightarrow$ carac $Pdec\ x \leq$ carac $Qdec\ y$.
Hint Resolve carac_monotonic.

Lemma carac_eq_compat : $\forall$ (A B:Type)(P:$A \rightarrow$ Prop)(Q:$B\rightarrow$Prop)($Pdec$: dec P)($Qdec$: dec Q) x y,
$(P\ x \leftrightarrow Q\ y) \rightarrow$ carac $Pdec\ x \equiv$ carac $Qdec\ y$.
Hint Resolve carac_eq_compat.

Lemma carac_one : $\forall$ (A:Type)(P:$A \rightarrow$ Prop)($Pdec$:dec P)(z:A),
$P\ z \rightarrow$ carac $Pdec\ z \equiv 1$.

Lemma carac_zero : $\forall$ (A:Type)(P:$A \rightarrow$ Prop)($Pdec$:dec P)(z:A),
$\not P\ z \rightarrow$ carac $Pdec\ z \equiv 0$.

Lemma cover_dec : $\forall$ (A:Type)(P:set A)($Pdec$: dec P), cover P (carac $Pdec$).

Lemma cover_mult_fun : $\forall$ (A:Type)(P:set A)(cP : MF A)(f g:$A\rightarrow U$),
$(\forall\ x,\ P\ x \rightarrow f\ x \equiv g\ x) \rightarrow$ cover P $cP \rightarrow \forall\ x,\ cP\ x \times f\ x \equiv cP\ x \times g\ x$.

Lemma cover_esp_fun : $\forall$ (A:Type)(P:set A)(cP : MF A)(f g:$A\rightarrow U$),
$(\forall\ x,\ P\ x \rightarrow f\ x \equiv g\ x) \rightarrow$ cover P $cP \rightarrow \forall\ x,\ cP\ x$ & $f\ x \equiv cP\ x$ & $g\ x$.

Lemma cover_esp_fun_le : $\forall$ (A:Type)(P:set A)(cP : MF A)(f g:$A\rightarrow U$),
$(\forall\ x,\ P\ x \rightarrow f\ x \leq g\ x) \rightarrow$ cover P $cP \rightarrow \forall\ x,\ cP\ x$ & $f\ x \leq cP\ x$ & $g\ x$.
Hint Resolve cover_esp_fun_le.

Lemma cover_ok : $\forall$ (A:Type)(P Q:set A)(f g : MF A),
$(\forall\ x,\ P\ x \rightarrow \not Q\ x) \rightarrow$ cover P $f \rightarrow$ cover Q $g \rightarrow$ fplusok f g.
Hint Resolve cover_ok.

11.3 Assuming m is a distribution under assumption P and cP is 0 or 1, builds

a distribution which is m if cP is 1 and 0 otherwise

Definition Mrestr A (cp:U) (m:M A) : M A := UMult cp @ m.

Lemma Mrestr_simpl : $\forall$ A cp (m:M A) f, Mrestr cp m f = $cp \times (m\ f)$.

Lemma Mrestr0 : $\forall$ A cP (m:M A), $cP \leq 0 \rightarrow \forall$ f, Mrestr cP m $f \equiv 0$.

Lemma Mrestr1 : $\forall$ A cP (m:M A), $1 \leq cP \rightarrow \forall$ f, Mrestr cP m $f \equiv m$ f.

Definition distr_restr : $\forall$ A (P:Prop) (cp:U) (m:M A),
$((P \rightarrow 1 \leq cp) \wedge (\not P \rightarrow cp \leq 0)) \rightarrow (P \rightarrow$ stable_inv $m) \rightarrow$
$(P \rightarrow$ stable_plus $m) \rightarrow (P \rightarrow$ stable_mult $m) \rightarrow (P \rightarrow$ **continuous** $m)$
$\rightarrow$ **distr** A.
Defined.

```
Lemma distr_restr_simpl : ∀ A (P:Prop) (cp:U) (m:M A)
      (Hp: (P → 1 ≤ cp) ∧ (¬P → cp ≤ 0)) (Hinv:P → stable_inv m)
      (Hplus:P → stable_plus m)(Hmult:P → stable_mult m)(Hcont:P → continuous m) f,
      μ (distr_restr cp Hp Hinv Hplus Hmult Hcont) f = cp × m f.
```

11.4 Modular reasoning on programs

```
Lemma range_cover : ∀ A (P:A → Prop) d cP, range P d → cover P cP →
   ∀ f, μ d f ≡ μ d (fun x ⇒ cP x × f x).

Lemma mu_cut : ∀ (A:Type)(m:distr A)(P:set A)(cP f g:MF A),
  cover P cP → (∀ x, P x → f x ≡ g x) → 1≤μ m cP → μ m f ≡ μ m g.
```

11.5 Uniform measure on finite sets

```
Section SigmaFinite.
Variable A:Type.
Variable decA : ∀ x y:A, {x=y}+{x≠y}.

Section RandomFinite.
```

11.5.1 Distribution for *random_fin P* over $\{k{:}nat \mid k \leq n\}$

The distribution associated to *random_fin P* is *f -> sigma (a in P) [1/]1+n (f a) with [n+1] the size of [P] we cannot factorize [[1/]1+n] because of possible overflow*

```
Fixpoint sigma_fin (f:A->U)(P:A->Prop)(FP:finite P){struct FP}:U :=
match FP with
  | (fin_eq_empty eq) ⇒ 0
  | (fin_eq_add x Q nQx FQ eq) ⇒ (f x) + sigma_fin f FQ
end.

Definition retract_fin (P:A->Prop) (f:A->U) :=
     ∀ Q (FQ: finite Q), incl Q P → ∀ x, (Q x) → (P x) → f x ≤ [1-](sigma_fin f FQ).

Lemma retract_fin_inv :
     ∀ (P:A->Prop) (f:A->U),
              retract_fin P f → ∀ Q (FQ: finite Q), incl Q P → ∀ x, (Q x) → (P x) → sigma_fin f FQ
<=[1-]f x.

Hint Immediate retract_fin_inv.

Lemma retract_fin_incl : ∀ P Q f, retract_fin P f → incl Q P → retract_fin Q f.

Lemma sigma_fin_monotonic : ∀ (f g : A → U)(P:A->Prop)(FP:finite P),
       (∀ x, P x → (f x) ≤ (g x))→ sigma_fin f FP ≤ sigma_fin g FP.

Lemma sigma_fin_eq_compat :
∀ (f g : A → U)(P:A->Prop)(FP:finite P),
       (∀ x, P x → (f x) ≡ (g x))→ sigma_fin f FP ≡ sigma_fin g FP.

Lemma retract_fin_le : ∀ (P:A->Prop) (f g:A->U),
         (∀ x, P x → f x ≤ g x) → retract_fin P g → retract_fin P f.

Lemma sigma_fin_mult : ∀ (f : A → U) c (P:A->Prop)(FP:finite P),
        retract_fin P f → sigma_fin (fun k ⇒ c × f k) FP ≡ c × sigma_fin f FP.

Lemma sigma_fin_plus : ∀ (f g: A → U) (P:A->Prop)(FP:finite P),
        sigma_fin (fun k ⇒ f k + g k) FP ≡ sigma_fin f FP + sigma_fin g FP.

Lemma sigma_fin_prod_maj :
∀ (f g : A → U)(P:A->Prop)(FP:finite P),
        sigma_fin (fun k ⇒ f k × g k) FP ≤ sigma_fin f FP.
```

```
Lemma sigma_fin_prod_le :
∀ (f g : A → U) (c:U) , (∀ k, f k ≤ c) → ∀ (P:A->Prop)(FP:finite P),
   retract_fin P g → sigma_fin (fun k ⇒ f k × g k) FP ≤ c × sigma_fin g FP.

Lemma sigma_fin_prod_ge :
∀ (f g : A → U) (c:U) , (∀ k, c ≤ f k) → ∀ (P:A->Prop)(FP:finite P),
   retract_fin P g → c × sigma_fin g FP ≤ sigma_fin (fun k ⇒ f k × g k) FP.
Hint Resolve sigma_fin_prod_maj sigma_fin_prod_ge sigma_fin_prod_le.

Lemma sigma_fin_inv : ∀ (f g : A → U)(P:A->Prop)(FP:finite P),
       retract_fin P f →
       [1-] sigma_fin (fun k ⇒ f k × g k) FP ≡
       sigma_fin (fun k ⇒ f k × [1-] g k) FP + [1-] sigma_fin f FP.

Lemma sigma_fin_eqset : ∀ f P Q (FP:finite P) (e:eqset P Q),
    (sigma_fin f (fin_eqset e FP)) = (sigma_fin f FP).

Lemma sigma_fin_rem : ∀ f P (FP:finite P) a,
               P a → sigma_fin f FP ≡ f a + sigma_fin f (finite_rem decA a FP).

Lemma sigma_fin_incl : ∀ f P (FP:finite P) Q (FQ:finite Q),
                (incl P Q) → sigma_fin f FP ≤ sigma_fin f FQ.

Lemma sigma_fin_unique : ∀ f P Q (FP:finite P) (FQ:finite Q), (eqset P Q)
                          → sigma_fin f FP ≡ sigma_fin f FQ.

Lemma sigma_fin_cte : ∀ c P (FP:finite P),
       sigma_fin (fun _ ⇒ c) FP ≡ (size FP) */ c.

End RandomFinite.

End SigmaFinite.
```

11.6 Properties of the Random distribution

```
Definition le_dec (n:nat) : dec (fun x ⇒ (x ≤ n)%nat).
Defined.

Definition lt_dec (n:nat) : dec (fun x ⇒ (x < n)%nat).
Defined.

Definition gt_dec : ∀ x, dec (lt x).
Defined.

Definition ge_dec : ∀ x, dec (le x).
Defined.

Definition carac_eq n := carac (eq_nat_dec n).
Definition carac_le n := carac (le_dec n).
Definition carac_lt n := carac (lt_dec n).
Definition carac_gt n := carac (gt_dec n).
Definition carac_ge n := carac (ge_dec n).

Definition is_eq (n:nat) : cover (fun x ⇒ n = x) (carac_eq n) := cover_dec (eq_nat_dec n).
Definition is_le (n:nat) : cover (fun x ⇒ (x ≤n)%nat) (carac_le n) := cover_dec (le_dec n).
Definition is_lt (n:nat) : cover (fun x ⇒ (x < n)%nat) (carac_lt n) := cover_dec (lt_dec n).
Definition is_gt (n:nat) : cover (fun x ⇒ (n < x)%nat) (carac_gt n):= cover_dec (gt_dec n).
Definition is_ge (n:nat) : cover (fun x ⇒ (n ≤ x)%nat) (carac_ge n) := cover_dec (ge_dec n).

Lemma carac_gt_S :
  ∀ x y, carac_gt (S y) (S x) ≡ carac_gt y x.

Lemma carac_lt_S : ∀ x y, carac_lt (S x) (S y) ≡ carac_lt x y.

Lemma carac_le_S : ∀ x y, carac_le (S x) (S y) ≡ carac_le x y.

Lemma carac_ge_S : ∀ x y, carac_ge (S x) (S y) ≡ carac_ge x y.
```

```
Lemma carac_eq_S : ∀ x y, carac_eq (S x) (S y) ≡ carac_eq x y.
Lemma carac_lt_0 : ∀ y, carac_lt 0 y ≡ 0.
Lemma carac_lt_zero : carac_lt 0 ≡ fzero _.
Lemma carac_lt_if_compat : ∀ x (t:bool) u v,
  (carac_lt x (if t then u else v))
  ≡
  (if t
    then (carac_lt x u)
    else (carac_lt x v)).
Hint Resolve carac_le_S carac_eq_S carac_lt_S carac_ge_S carac_gt_S carac_lt_0 carac_lt_zero.
```

Count the number of elements between 0 and n-1 which satisfy P

```
Fixpoint nb_elts (P:nat → Prop)(Pdec : dec P)(n:nat) {struct n} : nat :=
match n with
  0 ⇒ 0%nat
| S n ⇒ if Pdec n then (S (nb_elts Pdec n)) else (nb_elts Pdec n)
end.
Lemma nb_elts_true : ∀ (P:nat → Prop)(Pdec : dec P)(n:nat),
        (∀ k, (k < n)%nat → P k) → nb_elts Pdec n =n.
Hint Resolve nb_elts_true.
Lemma nb_elts_false : ∀ P,∀ Pdec:dec P,∀ n,
  (∀ x, (x<n)%nat → ~P x) → nb_elts Pdec n = 0%nat.
```

- the probability for a random number between 0 and n to satisfy P is equal to the number of elements below n which satisfy P divided by n+1

```
Lemma Random_carac : ∀ (P:nat → Prop)(Pdec : dec P)(n:nat),
    μ (Random n) (carac Pdec) ≡ (nb_elts Pdec (S n)) */ [1/]1+n.
Lemma nb_elts_lt_le : ∀ k n, (k ≤ n)%nat → nb_elts (lt_dec k) n = k.
Lemma nb_elts_lt_ge : ∀ k n, (n ≤ k)%nat → nb_elts (lt_dec k) n = n.
Lemma nb_elts_eq_nat_ge :∀ n k,
  (n ≤ k)%nat → nb_elts (eq_nat_dec k) n = 0%nat.
Lemma beq_nat_neq: ∀ x y : nat, x ≠ y → false = beq_nat x y.
Lemma nb_elt_eq :∀ n k,
  (k < n)%nat → nb_elts (eq_nat_dec k) n = 1%nat.
Hint Resolve nb_elts_lt_ge nb_elts_lt_le nb_elts_eq_nat_ge nb_elt_eq.
Lemma Random_lt : ∀ n k, μ (Random n) (carac_lt k) ≡ k */ [1/]1+n.
Hint Resolve Random_lt.
Lemma Random_le : ∀ n k, μ (Random n) (carac_le k) ≡ (S k) */ [1/]1+n.
Hint Resolve Random_le.
Lemma Random_eq : ∀ n k, (k ≤ n)%nat → μ (Random n) (carac_eq k) ≡ 1 */ [1/]1+n.
Hint Resolve Random_eq.
```

11.7 Properties of distributions and set

```
Section PickElemts.
Variable A : Type.
Variable P : A → Prop.
Variable cP : A → U.
```

```
Hypothesis coverP : cover P cP.
Variable ceq : A → A → U.
Hypothesis covereq : ∀ x, cover (eq x) (ceq x).
Variable d : distr A.
Variable k : U.
Hypothesis deqP : ∀ x, P x → k ≤ μ d (ceq x).
Lemma d_coverP : ∀ x, P x → k ≤ μ d cP.
Lemma d_coverP_exists : (∃ x, P x) → k ≤ μ d cP.
Lemma d_coverP_not_empty : ¬(∀ x, ¬P x) → k ≤ μ d cP.
End PickElemts.
```

12 Bernoulli.v: Simulating Bernoulli and Binomial distributions

```
Require Export Cover.
Require Export Misc.
```

12.1 Program for computing a Bernoulli distribution

bernoulli p gives true with probability p and false with probability (1-p)

```
let rec bernoulli p =
        if flip
        then (if p < 1/2 then false else bernoulli (2 p - 1))
        else  (if p < 1/2 then bernoulli (2 p) else true)
```

```
Hypothesis dec_demi : ∀ x : U, {x < 1/2}+{1/2 ≤ x}.
Instance Fbern_mon : monotonic
    (fun (f:U → distr bool) p ⇒ Mif Flip
       (if dec_demi p then Munit false else f (p & p))
       (if dec_demi p then f (p + p) else Munit true)).
Save.
Definition Fbern : (U → distr bool) -m> (U → distr bool)
    := mon (fun f p ⇒ Mif Flip
       (if dec_demi p then Munit false else f (p & p))
       (if dec_demi p then f (p + p) else Munit true)).
Lemma Fbern_simpl : ∀ f p,
Fbern f p = Mif Flip
       (if dec_demi p then Munit false else f (p & p))
       (if dec_demi p then f (p + p) else Munit true).
Definition bernoulli : U → distr bool := Mfix Fbern.
```

12.2 Properties of the Bernoulli program

12.2.1 Proofs using fixpoint rules

```
Definition Mubern (q: bool → U) : MF U -m> MF U.
Defined.
Lemma Mubern_simpl : ∀ (q: bool → U) f p,
  Mubern q f p = if dec_demi p then 1/2*(q false)+1/2*(f (p+p))
                               else 1/2*(f (p&p)) + 1/2*(q true).
```

```
Lemma Mubern_eq : ∀ (q: bool → U) (f:U → distr bool) (p:U),
                μ (Fbern f p) q ≡ Mubern q (fun y ⇒ μ (f y) q) p.
Hint Resolve Mubern_eq.

Lemma Bern_eq :
    ∀ q : bool → U, ∀ p, μ (bernoulli p) q ≡ mufix (Mubern q) p.
Hint Resolve Bern_eq.

Lemma Bern_commute : ∀ q : bool → U,
   mu_muF_commute_le Fbern (fun (x:U)=>q) (Mubern q).
Hint Resolve Bern_commute.

Lemma Bern_term : ∀ p, μ (bernoulli p) (fone bool) ≡ 1.
Hint Resolve Bern_term.
```

12.2.2 p is an invariant of Mubern qtrue

```
Lemma MuBern_true : ∀ p, Mubern B2U (fun q ⇒ q) p ≡ p.
Hint Resolve MuBern_true.

Lemma MuBern_false : ∀ p, Mubern (finv B2U) (finv (fun q ⇒ q)) p ≡ [1-]p.
Hint Resolve MuBern_false.

Lemma Bern_true : ∀ p, μ (bernoulli p) B2U ≡ p.

Lemma Bern_false : ∀ p, μ (bernoulli p) NB2U ≡ [1-]p.

Lemma Mubern_inv : ∀ (q: bool → U) (f:U → U) (p:U),
       Mubern (finv q) (finv f) p ≡ [1-] Mubern q f p.
```

12.2.3 Proofs using lubs

Invariant *pmin p pmin p n* = *p* - $\frac{1}{2}$ ^ *n*

Property : ∀ *p*, *ok p* (*bernoulli p*) χ (.=*true*)

```
Definition qtrue (p:U) := B2U.
Definition qfalse (p:U) := NB2U.

Lemma bernoulli_true : okfun (fun p ⇒ p) bernoulli qtrue.
```

Property : ∀ *p*, *ok* (1-*p*) (*bernoulli p*) (χ (.=*false*))

```
Lemma bernoulli_false : okfun (fun p ⇒ [1-] p) bernoulli qfalse.
```

Probability for the result of (*bernoulli p*) to be true is exactly *p*

```
Lemma qtrue_qfalse_inv : ∀ (b:bool) (x:U), qtrue x b ≡ [1-] (qfalse x b).

Lemma bernoulli_eq_true : ∀ p, μ (bernoulli p) (qtrue p) ≡ p.

Lemma bernoulli_eq_false : ∀ p, μ (bernoulli p) (qfalse p)≡ [1-]p.

Lemma bernoulli_eq : ∀ p f, μ (bernoulli p) f ≡ p × f true + ([1-]p) × f false.

Lemma bernoulli_total : ∀ p , μ (bernoulli p) (fone bool)≡1.
```

12.3 Binomial distribution

(*binomial p n*) gives *k* with probability *C*(*k*,*n*) *p*^*k*(1-*p*)^(*n*-*k*)

12.3.1 Definition and properties of binomial coefficients

```
Fixpoint comb (k n:nat) {struct n} : nat :=
         match n with O ⇒ match k with O ⇒ (1%nat) | (S l) ⇒ O end
                | (S m) ⇒ match k with O ⇒ (1%nat)
                                     | (S l) ⇒ ((comb l m) + (comb k m))%nat end
```

```
    end.
Lemma comb_0_n : ∀ n, comb 0 n = 1%nat.
Lemma comb_not_le : ∀ n k, le (S n) k → comb k n=0%nat.
Lemma comb_Sn_n : ∀ n, comb (S n) n =0%nat.
Lemma comb_n_n : ∀ n, comb n n = (1%nat).
Lemma comb_1_Sn : ∀ n, comb 1 (S n) = (S n).
Lemma comb_inv : ∀ n k, (k≤n)%nat → comb k n = comb (n-k) n.
Lemma comb_n_Sn : ∀ n, comb n (S n) = (S n).
Definition fc (p:U)(n k:nat) := (comb k n) */ (p^k × ([1-]p)^(n-k)).
Lemma fcp_0 : ∀ p n, fc p n O ≡ ([1-]p)^n.
Lemma fcp_n : ∀ p n, fc p n n ≡ p^n.
Lemma fcp_not_le : ∀ p n k, (S n ≤ k)%nat → fc p n k ≡ 0.
Lemma fc0 : ∀ n k, fc 0 n (S k) ≡ 0.
Hint Resolve fc0.
Add Morphism fc with signature Oeq ⟹ eq ⟹ eq ⟹ Oeq
as fc_eq_compat.
Save.
Hint Resolve fc_eq_compat.
Lemma sigma_fc0 : ∀ n k, sigma (fc 0 n) (S k) ≡1.
Lemma retract_class : ∀ f n, class (retract f n).
Hint Resolve retract_class.
Lemma fc_retract :
      ∀ p n, ([1-]p^n ≡ sigma (fc p n) n) → retract (fc p n) (S n).
Hint Resolve fc_retract.
Lemma fc_Nmult_def :
      ∀ p n k, ([1-]p^n ≡ sigma (fc p n) n) → Nmult_def (comb k n) (p^k × ([1-]p) ^(n-k)).
Hint Resolve fc_Nmult_def.
Lemma fcp_S :
     ∀ p n k, ([1-]p^n ≡ sigma (fc p n) n) → fc p (S n) (S k) ≡ p × (fc p n k) + ([1-]p) × (fc p n (S k)).
Lemma sigma_fc_1 : ∀ p n, ([1-]p^n ≡ sigma (fc p n) n) ->1≡sigma (fc p n) (S n).
Hint Resolve sigma_fc_1.
Lemma Uinv_exp : ∀ p n, [1-](p^n)≡sigma (fc p n) n.
Hint Resolve Uinv_exp.
Lemma Nmult_comb : ∀ p n k, Nmult_def (comb k n) (p ^ k × ([1-] p) ^ (n - k)).
Hint Resolve Nmult_comb.
```

12.3.2 Definition of binomial distribution

```
Fixpoint binomial (p:U)(n:nat) {struct n}: distr nat :=
    match n with O ⇒ (Munit O)
                 | S m ⇒ Mlet (binomial p m)
                              (fun x ⇒ Mif (bernoulli p) (Munit (S x)) (Munit x))
    end.
```

12.3.3 Properties of binomial distribution

```
Lemma binomial_eq_k :
   ∀ p n k, μ (binomial p n) (carac_eq k) ≡ fc p n k.
```

Lemma binomial_le_n :
  $\forall\ p\ n,\ 1 \leq \mu\ (\text{binomial}\ p\ n)\ (\text{carac\_le}\ n)$.

Lemma binomial_le_S : $\forall\ p\ n\ k$,
  $\mu\ (\text{binomial}\ p\ (\text{S}\ n))\ (\text{carac\_le}\ (\text{S}\ k)) \equiv$
  $p \times (\mu\ (\text{binomial}\ p\ n)\ (\text{carac\_le}\ k))$ + ([1-]p) $\times$ ($\mu\ (\text{binomial}\ p\ n)\ (\text{carac\_le}\ (\text{S}\ k))$).

Lemma binomial_lt_S : $\forall\ p\ n\ k$,
  $\mu\ (\text{binomial}\ p\ (\text{S}\ n))\ (\text{carac\_lt}\ (\text{S}\ k)) \equiv$
  $p \times (\mu\ (\text{binomial}\ p\ n)\ (\text{carac\_lt}\ k))$ + ([1-]p) $\times$ ($\mu\ (\text{binomial}\ p\ n)\ (\text{carac\_lt}\ (\text{S}\ k))$).

13 DistrTactic.v: tactics for reasoning on distributions.

```
Require Export List.
Require Export Bool.
Require Export Streams.
Require Export Decidable.
Require Export Arith.
Require Export Prog.
Require Export Bernoulli.
Require Import Omega.
Require Import Compare.
Require Export Misc.
Open Local Scope U_scope.
Open Local Scope O_scope.
```

The tactic to use is *simplmu* for one step simplification, *rsimplmu* for repeated simplifications. These two tactics can be cloned and extended using *simplmu_arg*.

```
Hint Extern 2 ⇒ Usimpl.

Ltac simpl_mu_rewrite tacsubgoals := first [
progress setoid_rewrite if_beq_nat_nat_eq_dec|rewrite if_beq_nat_nat_eq_dec|
progress setoid_rewrite Umult_sym_cst|rewrite Umult_sym_cst|
progress setoid_rewrite Mif_eq2|rewrite Mif_eq2|
progress setoid_rewrite Bern_true|rewrite Bern_true|
progress setoid_rewrite Bern_false|rewrite Bern_false|
progress setoid_rewrite Mlet_simpl|rewrite Mlet_simpl|
progress setoid_rewrite Munit_simpl|rewrite Munit_simpl|

progress setoid_rewrite bary_refl_feq;[|complete auto]|rewrite bary_refl_feq;[|complete auto]|

progress setoid_rewrite Uinv_inv|rewrite Uinv_inv|
progress setoid_rewrite bernoulli_eq|rewrite bernoulli_eq|
progress setoid_rewrite binomial_lt_S|rewrite binomial_lt_S|
progress setoid_rewrite carac_lt_S|rewrite carac_lt_S|


progress setoid_rewrite mu_stable_mult2|rewrite mu_stable_mult2|
progress setoid_rewrite mon_simpl|rewrite mon_simpl|

progress setoid_rewrite im_distr_simpl|rewrite im_distr_simpl|
progress setoid_rewrite Mchoice_simpl|rewrite Mchoice_simpl|
progress setoid_rewrite Random_total|rewrite Random_total|
progress setoid_rewrite discrete_simpl|rewrite discrete_simpl|
progress setoid_rewrite Discrete_simpl|rewrite Discrete_simpl|
progress setoid_rewrite Flip_simpl|rewrite Flip_simpl|
```

```
progress setoid_rewrite (@mu_fzero_eq _ _) | rewrite (@mu_fzero_eq _ _) |
progress setoid_rewrite mu_fzero_eq |rewrite mu_fzero_eq |
progress setoid_rewrite Mlet_unit|rewrite Mlet_unit|
progress setoid_rewrite Mlet_assoc|rewrite Mlet_assoc|

progress setoid_rewrite mu_stable_plus2;[|complete tacsubgoals ] | rewrite mu_stable_plus2;[|complete tac-
subgoals ]|

progress setoid_rewrite carac_lt_if_compat | rewrite carac_lt_if_compat
].
```

Try simplification of Oeq and Ole at top level.

```
Ltac simplmu_aux :=
  match goal with
    | ⊢ (Ole (fmont (μ ?d1) ?f) (fmont (μ ?d2) ?g)) ⇒ apply (mu_le_compat (m1:=d1) (m2:=d2) (Ole_refl
d1) (f:=f) (g:=g)); intro
    | ⊢ (Oeq (fmont (μ ?d1) ?f) (fmont (μ ?d2) ?g)) ⇒ apply (mu_eq_compat (m1:=d1) (m2:=d2) (Oeq_refl
d1) (f:=f) (g:=g)); unfold Oeq;intro
    | ⊢ (Oeq (Munit ?x) (Munit ?y)) ⇒ apply (Munit_eq_compat x y)
    | ⊢ (Oeq (Mlet ?x1 ?f) (Mlet ?x2 ?g))
      ⇒ apply (Mlet_eq_compat (m1:=x1) (m2:=x2) (M1:=f) (M2:=g) (Oeq_refl x1)); intro
    | ⊢ (Ole (Mlet ?x1 ?f) (Mlet ?x2 ?g))
      ⇒ apply (Mlet_le_compat (m1:=x1) (m2:=x2) (M1:=f) (M2:=g) (Ole_refl x1)); intro
  end.
Ltac simplmu_arg tacsidecond :=
  Usimpl || simplmu_aux || simpl_mu_rewrite ltac:tacsidecond.
Ltac simplmu := simplmu_arg idtac.
Ltac rsimplmu := (repeat progress (simplmu;simpl)).
```

FOLLOWING TACTIC ARE EXPERIMENTS ONLY. USE rsimplmu instead.

```
Ltac declare_dummyvar dummy_value ty nme :=
  let dummy := dummy_value ty in
    pose (nme := dummy); clearbody nme.
Ltac dummyvar_names dummy_value lty :=
  match lty with
    | nil ⇒ idtac
    | ?ty::?lty' ⇒
        let nme := fresh "dummy" in
        declare_dummyvar dummy_value ty nme;
        dummyvar_names dummy_value lty'
  end.
Ltac find_var_type ty :=
  match goal with
     X : ty ⊢ _ ⇒ X
  end.
Ltac generate_var_type dummy_value ty :=
   let nme := fresh "dummy" in
   declare_dummyvar dummy_value ty nme;
   constr:nme.
Ltac clear_dummys lty :=
  match lty with
    | nil ⇒ idtac
    | ?ty::?lty' ⇒
        let nme := find_var_type ty in
```

```
          clear nme; clear_dummys lty'
  end.
Ltac do_under_lambdas t tac :=
  match t with

    | (Ole ?t ?u) ⇒ let res := do_under_lambdas t tac in res
    | (Ole ?t ?u) ⇒ let res := do_under_lambdas u tac in res
    | (Oeq ?t ?u) ⇒ do_under_lambdas t tac
    | (Oeq ?t ?u) ⇒ do_under_lambdas u tac
    | ?x ⇒ tac t

    | context C [ (fmont (μ ?x) (fun (y: ?ty) ⇒ (@?fct y))) ] ⇒

      let dummyvar := find_var_type ty in
        let appliedf := (eval lazy beta in (fct dummyvar)) in
          let res := do_under_lambdas appliedf tac in
            match res with (?t',?res') ⇒

              let res'' := eval pattern dummyvar in res' in
                let res''' :=
                  match res'' with
                    | ?fres _ ⇒
                      let res := eval lazy beta in (fmont (μ x) fres) in
                        context C [res]
                  end in
                  constr:(t,res''')
            end
  end.
Ltac deepsimplmu_arg ldummy dummy_value tacrec tacfin :=

  dummyvar_names dummy_value ldummy;
  match goal with | ⊢ ?u ⇒

    let res := do_under_lambdas u tacrec in

      match res with (?t,?rewt) ⇒
        let h := fresh "__heq" in
          (assert (h:t ≡ rewt);

            clear_dummys ldummy);

          [ tacfin | rewrite h ; clear h]
      end
  end.
```

14 IterFlip.v: An example of probabilistic termination

```
Require Export Prog.
```

14.1 Definition of a random walk

We interpret the probabilistic program

```
let rec iter x = if flip() then iter (x+1) else x
```

```
Require Import ZArith.
```

```
Instance Fiter_mon :
    monotonic (fun (f:Z → distr Z) (x:Z) ⇒ Mif Flip (f (Zsucc x)) (Munit x)).
Save.
```

```
Definition Fiter : (Z → distr Z) -m> (Z → distr Z)
:= mon (fun f (x:Z) ⇒ Mif Flip (f (Zsucc x)) (Munit x)).
```

```
Lemma Fiter_simpl : ∀ f x, Fiter f x = Mif Flip (f (Zsucc x)) (Munit x).
```

```
Definition iterflip : Z → distr Z := Mfix Fiter.
```

14.2 Main result

Probability for *iter* to terminate is 1

14.2.1 Auxiliary function *p*

Definition $p\_n = 1 - \frac{1}{2}\hat{}\ n$

```
Fixpoint p_ (n : nat) : U := match n with O ⇒ 0 | (S n) ⇒ 1/2 × p_ n + 1/2 end.
```

```
Lemma p_incr : ∀ n, p_ n ≤ p_ (S n).
```

```
Hint Resolve p_incr.
```

```
Definition p : nat -m> U := fnatO_intro p_ p_incr.
```

```
Lemma pS_simpl : ∀ n, p (S n) = 1/2 × p n + 1/2.
```

```
Lemma p_eq : ∀ n:nat, p n ≡ [1-](1/2^n).
Hint Resolve p_eq.
```

```
Lemma p_le : ∀ n:nat, [1-]([1/]1+n) ≤ p n.
```

```
Hint Resolve p_le.
```

```
Lemma lim_p_one : 1 ≤ lub p.
```

```
Hint Resolve lim_p_one.
```

14.2.2 Proof of probabilistic termination

```
Definition q1 (z1 z2:Z) := 1.
```

```
Lemma iterflip_term : okfun (fun k ⇒ 1) iterflip q1.
```

15 Choice.v: An example of probabilistic choice

```
Require Export Prog.
```

15.1 Definition of a probabilistic choice

We interpret the probabilistic program *p* which executes two probabilistic programs *p1* and *p2* and then make a choice between the two computed results.

```
let rec p () = let x = p1 () in let y = p2 () in choice x y
```

```
Section CHOICE.
Variable A : Type.
Variables p1 p2 : distr A.
Variable choice : A → A → A.
Definition p : distr A := Mlet p1 (fun x ⇒ Mlet p2 (fun y ⇒ Munit (choice x y))).
```

15.2 Main result

We estimate the probability for p to satisfy Q given estimations for both *p1* and *p2*.

15.2.1 Assumptions

We need extra properties on *p1*, *p2* and *choice*.

- *p1* and *p2* terminate with probability 1
- Q value on *choice* is not less than the sum of values of Q on separate elements.

If Q is a boolean function it means than if one of x or y satisfies Q then (*choicexy*) will also satisfy Q
`Hypothesis` *p1_terminates* : (*μ p1* (fone *A*))==1.
`Hypothesis` *p2_terminates* : (*μ p2* (fone *A*))==1.

`Variable` Q : MF A.
`Hypothesis` *choiceok* : ∀ x y, Q x + Q y ≤ Q (*choice* x y).

15.2.2 Proof of estimation:

ok k1 p1 Q and *ok k2 p2 Q* implies *ok* (*k1*(1-*k2*)+*k2*) *p Q*

`Lemma` choicerule : ∀ *k1 k2*,
k1 ≤ *μ p1 Q* → *k2* ≤ *μ p2 Q* → (*k1* × ([1-] *k2*) + *k2*) ≤ *μ* p *Q*.

`End` CHOICE.
`Require Export` Cover.
`Require` Arith.

16 Ycart.v: An exemple of partial termination

`Section` YcartExample.

16.1 Program giving an example of partiality

Given a function K : nat ->nat, N:nat

```
let rec ycart p x = if random p < K x then x else ycart (x+1)
```

The probability of termination is $1-\prod_{k=x}^{\infty}(1-K(k)/p+1)$ `Variable` K : **nat** → **nat**.
`Variable` N : **nat**.

`Instance` FYcart_mon : **monotonic**
(fun (*f*:**nat** → **distr nat**) *n* ⇒
Mlet (Random *N*) (fun *x* ⇒ if lt_dec (*K n*) *x* then Munit *n* else *f* (S *n*))).
`Save`.

`Definition` FYcart : (**nat** → **distr nat**) -m> (**nat** → **distr nat**)
:= mon (fun (*f*:**nat** → **distr nat**) *n* ⇒
Mlet (Random *N*) (fun *x* ⇒ if lt_dec (*K n*) *x* then Munit *n* else *f* (S *n*))).

`Lemma` FYcart_simpl : ∀ *f n*,
FYcart *f n* = Mlet (Random *N*) (fun *x* ⇒ if lt_dec (*K n*) *x* then Munit *n* else *f* (S *n*)).

`Definition` Ycart : **nat** → **distr nat** := Mfix FYcart.

`Definition` F *x* := (*K x*) */ [1/]1+*N*.

16.2 Properties of Ycart

```
Lemma FYcart_val : ∀ (q: MF nat) f x,
      μ (FYcart f x) q ≡ F x × q x + [1-](F x) × μ (f (S x)) q.
Definition P (x : nat) : nat -m→ U := Prod (fun i ⇒ [1-]F (x+i)).
Definition p (x:nat) : nat -m> U := sigma (fun k ⇒ F (x+k) × P x k).
Lemma P_prod : ∀ x k, F (x+k) × P x k ≡ P x k - P x (S k).
Hint Resolve P_prod.
Lemma p_diff : ∀ x n, (p x n : U) ≡ [1-] P x n.
Hint Resolve p_diff.
Lemma p_lub : ∀ x, lub (p x) ≡ [1-] prod_inf (fun i ⇒ [1-]F (x+i)).
Hint Resolve p_lub.
Lemma p_equation : ∀ x n, p x (S n) ≡ F x + [1-](F x) × p (S x) n.
Hint Resolve p_equation.
Lemma Ycart_term1 : ∀ x, μ (Ycart x) (fone nat) ≡ [1-] prod_inf (fun i ⇒ [1-]F (x+i)).
```

A shorter proof using mu (Ycart x) (f_one nat) = mu h. muYcart h x

```
Instance Ycart_one_mon
    : monotonic (fun (p:nat → U) (y:nat) ⇒ F y + [1-](F y) × p (S y)).
Save.
Definition Ycart_one : (nat → U) -m> (nat → U) :=
    mon (fun (p:nat→U) (y:nat) ⇒ F y + [1-](F y) × p (S y)).
Lemma Ycart_one_simpl : ∀ p y, Ycart_one p y = F y + [1-](F y) × p (S y).
Lemma Ycart_term2 : ∀ x, μ (Ycart x) (fone nat) ≡ [1-] prod_inf (fun i ⇒ [1-]F (x+i)).
Instance FYcart_lt_mon : monotonic (fun (p: nat → U) (y:nat) ⇒ [1-](F y) × p (S y)).
Save.
Definition FYcart_lt : (nat → U) -m> (nat → U) :=
    mon (fun (p:nat→U) (y:nat) ⇒ [1-](F y) × p (S y)).
Lemma FYcart_lt_simpl : ∀ (p:nat → U) (y:nat), FYcart_lt p y = [1-](F y) × p (S y).
Lemma Ycart_ltx : ∀ x, μ (Ycart x) (carac_lt x) ≤ 0.
Lemma Ycart_eqx : ∀ x, μ (Ycart x) (carac (eq_nat_dec x)) ≡ F x.
End YcartExample.
```