Kahn Networks in Coq

Christine Paulin-Mohring

July 19, 2007

Abstract

This document is extracted from the development of a Coq [5, 1] library for the representation of the semantics of Kahn networks. It mainly follows the original paper [2]. A high-level description of this library is available in a joint paper [4].

Contents

1 Cpo.v: Specification and properties of a cpo

Require Export *Setoid*.
Require Export *Arith*.
Require Export *Omega*.
Open Scope nat_scope.

1.1 Ordered type

Record *ord* : Type := *mk_ord*
{*tord*:>Type; *Ole*:*tord*→*tord*→Prop; *Ole_refl* : ∀ *x* :*tord*, *Ole x x*;
Ole_trans : ∀ *x y z*:*tord*, *Ole x y* → *Ole y z* → *Ole x z*}.

Hint *Resolve Ole_refl Ole_trans*.

Hint *Extern* 2 (*Ole* (*o*:=?*X1*) ?*X2* ?*X3*) ⇒ *simpl Ole*.

Delimit Scope O_scope with *tord*.

Infix "≤" := *Ole* : *O_scope*.
Open Scope O_scope.

1.1.1 Associated equality

Definition *Oeq* (*O*:*ord*) (*x y* : *O*) := *x* ≤ *y* ∧ *y* ≤ *x*.
Infix "==" := *Oeq* (*at level* 70) : *O_scope*.

Lemma *Ole_refl_eq* : ∀ (*O*:*ord*) (*x y*:*O*), *x*=*y* → *x* ≤ *y*.

Hint *Resolve Ole_refl_eq*.

Lemma *Ole_antisym* : ∀ (*O*:*ord*) (*x y*:*O*), *x*≤*y* → *y* ≤*x* → *x*==*y*.
Hint Immediate *Ole_antisym*.

Lemma *Oeq_refl* : ∀ (*O*:*ord*) (*x*:*O*), *x* == *x*.
Hint *Resolve Oeq_refl*.

Lemma *Oeq_refl_eq* : ∀ (*O*:*ord*) (*x y*:*O*), *x*=*y* → *x* == *y*.
Hint *Resolve Oeq_refl_eq*.

Lemma *Oeq_sym* : ∀ (*O*:*ord*) (*x y*:*O*), *x* == *y* → *y* == *x*.

Lemma *Oeq_le* : ∀ (*O*:*ord*) (*x y*:*O*), *x* == *y* → *x* ≤ *y*.

Lemma *Oeq_le_sym* : ∀ (*O*:*ord*) (*x y*:*O*), *x* == *y* → *y* ≤ *x*.

Hint *Resolve Oeq_le*.
Hint Immediate *Oeq_sym Oeq_le_sym*.

Lemma *Oeq_trans* : ∀ (*O*:*ord*) (*x y z*:*O*), *x* == *y* → *y* == *z* → *x* == *z*.
Hint *Resolve Oeq_trans*.

1.1.2 Setoid relations

Add Relation tord Oeq
reflexivity proved by Oeq_refl symmetry proved by Oeq_sym
transitivity proved by Oeq_trans as *Oeq_Relation*.

Add Relation tord Ole
reflexivity proved by Ole_refl
transitivity proved by Ole_trans as *Ole_Relation*.

Add Morphism Ole with *signature Oeq* ==> *Oeq* ==> *iff* as *Ole_eq_compat_iff*.

Lemma *Ole_eq_compat* :
∀ (*O* : *ord*) (*x1 x2* : *O*),
x1 == *x2* → ∀ *x3 x4* : *O*, *x3* == *x4* → *x1* ≤ *x3* → *x2* ≤ *x4*.

Lemma *Ole_eq_right* : ∀ (*O* : *ord*) (*x y z*: *O*),
x ≤ *y* → *y*==*z* → *x*≤*z*.

Lemma *Ole_eq_left* : ∀ (*O* : *ord*) (*x y z*: *O*),
x == *y* → *y*≤*z* → *x*≤*z*.

1.1.3 Dual order

Definition *Iord* (*O*:*ord*):*ord*.

1.1.4 Order on functions

Definition *ford* (*A*:Type) (*O*:*ord*) : *ord*.

Infix "-*o*>" := *ford* (*right associativity, at level* 30) : *O_scope* .

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

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

Lemma *ford_eq_elim* : ∀ *A* (*O*:*ord*) (*f g*:*A* -*o*> *O*), *f* == *g* →∀ *n*, *f n* == *g n*.
Hint Immediate *ford_eq_elim*.

Lemma *ford_eq_intro* : ∀ *A* (*O*:*ord*) (*f g*:*A* -*o*> *O*), (∀ *n*, *f n* == *g n*) → *f* == *g*.
Hint *Resolve ford_eq_intro*.

Hint *Extern* 2 (*Ole* (*o*:=*ford* ?*X1* ?*X2*) ?*X3* ?*X4*) ⇒ *intro*.

1.2 Monotonicity

1.2.1 Definition and properties

Definition *monotonic* (*O1 O2*:*ord*) (*f* : *O1* → *O2*) := ∀ *x y*, *x* ≤ *y* → *f x* ≤ *f y*.
Hint *Unfold monotonic*.

Definition *stable* (*O1 O2*:*ord*) (*f* : *O1* → *O2*) := ∀ *x y*, *x* == *y* → *f x* == *f y*.
Hint *Unfold stable*.

Lemma *monotonic_stable* : ∀ (*O1 O2* : *ord*) (*f*:*O1* → *O2*),
monotonic f → *stable f*.
Hint *Resolve monotonic_stable*.

1.2.2 Type of monotonic functions

Record *fmono* (*O1 O2*:*ord*) : Type := *mk_fmono*
{*fmonot* :> *O1*→*O2*; *fmonotonic*: *monotonic fmonot*}.
Hint *Resolve fmonotonic*.

Definition *fmon* (*O1 O2*:*ord*) : *ord*.

Infix "-*m*>" := *fmon* (*at level* 30, *right associativity*) : *O_scope*.

Lemma *fmon_stable* : ∀ (*O1 O2*:*ord*) (*f*:*O1*-*m*>*O2*), *stable f*.
Hint *Resolve fmon_stable*.

Lemma *fmon_le_elim* : ∀ (*O1 O2*:*ord*) (*f g*:*O1* -*m*> *O2*), *f* ≤ *g* →∀ *n*, *f n* ≤ *g n*.
Hint Immediate *fmon_le_elim*.

Lemma *fmon_le_intro* : ∀ (*O1 O2*:*ord*) (*f g*:*O1* -*m*> *O2*), (∀ *n*, *f n* ≤ *g n*) → *f* ≤ *g*.
Hint *Resolve fmon_le_intro*.

Lemma *fmon_eq_elim* : ∀ (*O1 O2*:*ord*) (*f g*:*O1* -*m*> *O2*), *f* == *g* →∀ *n*, *f n* == *g n*.
Hint Immediate *fmon_eq_elim*.

Lemma *fmon_eq_intro* : ∀ (*O1 O2*:*ord*) (*f g*:*O1* -*m*> *O2*), (∀ *n*, *f n* == *g n*) → *f* == *g*.
Hint *Resolve fmon_eq_intro*.

Hint *Extern* 2 (*Ole* (*o*:=*fmon* ?*X1* ?*X2*) ?*X3* ?*X4*) ⇒ *intro*.

1.2.3 Monotonicity and dual order

Definition *Imon* : ∀ *O1 O2*, (*O1*-*m*>*O2*) → *Iord O1* -*m*> *Iord O2*.

Definition *Imon2* : ∀ *O1 O2 O3*, (*O1*-*m*>*O2*-*m*>*O3*) → *Iord O1* -*m*> *Iord O2* -*m*> *Iord O3*.

1.2.4 Monotonic functions with 2 arguments

Definition *le_compat2_mon* : ∀ (*O1 O2 O3*:*ord*)(*f*:*O1* → *O2* → *O3*),
(∀ (*x y*:*O1*) (*z t*:*O2*), *x*≤*y* → *z* ≤ *t* → *f x z* ≤ *f y t*) → (*O1*-*m*>*O2*-*m*>*O3*).

1.3 Sequences

1.3.1 Order on natural numbers

Definition *natO* : *ord*.

Definition *fnatO_intro* : ∀ (*O*:*ord*) (*f*:*nat* → *O*), (∀ *n*, *f n* ≤ *f* (*S n*)) → *natO*-*m*>*O*.

Lemma *fnatO_elim* : ∀ (*O*:*ord*) (*f*:*natO* -*m*> *O*) (*n*:*nat*), *f n* ≤ *f* (*S n*).
Hint *Resolve fnatO_elim*.

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

Definition *mseq_lift_left* : ∀ (*O*:*ord*) (*f*:*natO -m> O*) (*n*:*nat*), *natO -m> O*.

Lemma *mseq_lift_left_le_compat* : ∀ (*O*:*ord*) (*f g*:*natO -m> O*) (*n*:*nat*),
f ≤ *g* → *mseq_lift_left f n* ≤ *mseq_lift_left g n*.
Hint *Resolve mseq_lift_left_le_compat*.

Add Morphism mseq_lift_left with *signature Oeq ==> eq ==> Oeq*
as *mseq_lift_left_eq_compat*.
Hint *Resolve mseq_lift_left_eq_compat*.

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

Definition *mseq_lift_right* : ∀ (*O*:*ord*) (*f*:*natO -m> O*) (*n*:*nat*), *natO -m> O*.

Lemma *mseq_lift_right_le_compat* : ∀ (*O*:*ord*) (*f g*:*natO -m> O*) (*n*:*nat*),
f ≤ *g* → *mseq_lift_right f n* ≤ *mseq_lift_right g n*.
Hint *Resolve mseq_lift_right_le_compat*.

Add Morphism mseq_lift_right with *signature Oeq ==> eq ==> Oeq*
as *mseq_lift_right_eq_compat*.

Lemma *mseq_lift_right_left* : ∀ (*O*:*ord*) (*f*:*natO -m> O*) *n*,
mseq_lift_left f n == mseq_lift_right f n.

1.3.2 Monotonicity and functions

- (ford_app f x) n = f n x

Definition *ford_app* : ∀ (*A*:Type)(*O1 O2*:*ord*)(*f*:*O1 -m>* (*A -o> O2*))(*x*:*A*), *O1 -m> O2*.

Infix "<o>" := *ford_app* (*at level* 30, *no associativity*) : *O_scope*.

Lemma *ford_app_simpl* : ∀ (*A*:Type)(*O1 O2*:*ord*) (*f* : *O1 -m> A -o> O2*) (*x*:*A*)(*y*:*O1*),
(*f* <o> *x*) *y* = *f y x*.

Lemma *ford_app_le_compat* : ∀ (*A*:Type)(*O1 O2*:*ord*) (*f g*:*O1 -m> A -o> O2*) (*x*:*A*),
f ≤ *g* → *f* <o> *x* ≤ *g* <o> *x*.
Hint *Resolve ford_app_le_compat*.

Add Morphism ford_app with *signature Oeq ==> eq ==> Oeq*
as *ford_app_eq_compat*.

- ford_shift f x y == f y x

Definition *ford_shift* : ∀ (*A*:Type)(*O1 O2*:*ord*)(*f*:*A -o>* (*O1 -m> O2*)), *O1 -m>* (*A-o>O2*).

Lemma *ford_shift_le_compat* : ∀ (*A*:Type)(*O1 O2*:*ord*) (*f g*: *A -o>* (*O1 -m> O2*)),
f ≤ *g* → *ford_shift f* ≤ *ford_shift g*.
Hint *Resolve ford_shift_le_compat*.

Add Morphism ford_shift with *signature Oeq ==> Oeq*
as *ford_shift_eq_compat*.

- (fmon_app f x) n = f n x

Definition *fmon_app* : ∀ (*O1 O2 O3*:*ord*)(*f*:*O1 -m> O2 -m> O3*)(*x*:*O2*), *O1 -m> O3*.

Infix "<_>" := *fmon_app* (*at level* 35, *no associativity*) : *O_scope*.

Lemma *fmon_app_simpl* : ∀ (*O1 O2 O3*:*ord*)(*f*:*O1 -m> O2 -m> O3*)(*x*:*O2*)(*y*:*O1*),
(*f* <_> *x*) *y* = *f y x*.

Lemma *fmon_app_le_compat* : ∀ (*O1 O2 O3*:*ord*) (*f g*:*O1 -m>* (*O2 -m> O3*)) (*x y*:*O2*),
f ≤ *g* → *x* ≤ *y* → *f* <_> *x* ≤ *g* <_> *y*.
Hint *Resolve fmon_app_le_compat*.

Add Morphism fmon_app with *signature Oeq ==> Oeq ==> Oeq*
as *fmon_app_eq_compat*.

- fmon_id c = c

Definition *fmon_id* : ∀ (*O*:*ord*), *O* -*m*> *O*.

Lemma *fmon_id_simpl* : ∀ (*O*:*ord*) (*x*:*O*), *fmon_id O x* = *x*.

- (fmon_cte c) n = c

Definition *fmon_cte* : ∀ (*O1 O2*:*ord*)(*c*:*O2*), *O1* -*m*> *O2*.

Lemma *fmon_cte_simpl* : ∀ (*O1 O2*:*ord*)(*c*:*O2*)(*c*:*O2*) (*x*:*O1*), *fmon_cte O1 c x* = *c*.

Definition *mseq_cte* : ∀ *O*:*ord*, *O* → *natO*-*m*>*O* := *fmon_cte natO*.

Lemma *fmon_cte_le_compat* : ∀ (*O1 O2*:*ord*) (*c1 c2*:*O2*),
c1 ≤ *c2* → *fmon_cte O1 c1* ≤ *fmon_cte O1 c2*.

Add Morphism fmon_cte with *signature Oeq* ==> *Oeq*
as *fmon_cte_eq_compat*.

- (fmon_diag h) n = h n n

Definition *fmon_diag* : ∀ (*O1 O2*:*ord*)(*h*:*O1* -*m*> (*O1* -*m*> *O2*)), *O1* -*m*> *O2*.

Lemma *fmon_diag_le_compat* : ∀ (*O1 O2*:*ord*) (*f g*:*O1* -*m*> (*O1* -*m*> *O2*)),
f ≤ *g* → *fmon_diag f* ≤ *fmon_diag g*.
Hint *Resolve fmon_diag_le_compat*.

Lemma *fmon_diag_simpl* : ∀ (*O1 O2*:*ord*) (*f*:*O1* -*m*> (*O1* -*m*> *O2*)) (*x*:*O1*),
fmon_diag f x = *f x x*.

Add Morphism fmon_diag with *signature Oeq* ==> *Oeq*
as *fmon_diag_eq_compat*.

- (fmon_shift h) n m = h m n

Definition *fmon_shift* : ∀ (*O1 O2 O3*:*ord*)(*h*:*O1* -*m*> *O2* -*m*> *O3*), *O2* -*m*> *O1* -*m*> *O3*.

Lemma *fmon_shift_simpl* : ∀ (*O1 O2 O3*:*ord*)(*h*:*O1* -*m*> *O2* -*m*> *O3*) (*x* : *O2*) (*y*:*O1*),
fmon_shift h x y = *h y x*.

Lemma *fmon_shift_le_compat* : ∀ (*O1 O2 O3*:*ord*) (*f g*:*O1* -*m*> *O2* -*m*> *O3*),
f ≤ *g* → *fmon_shift f* ≤ *fmon_shift g*.
Hint *Resolve fmon_shift_le_compat*.

Add Morphism fmon_shift with *signature Oeq* ==> *Oeq*
as *fmon_shift_eq_compat*.

Lemma *fmon_shift_shift_eq* : ∀ (*O1 O2 O3*:*ord*) (*h* : *O1* -*m*> *O2* -*m*> *O3*),
fmon_shift (*fmon_shift h*) == *h*.

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

Definition *fmon_comp* : ∀ *O1 O2 O3*:*ord*, (*O2* -*m*> *O3*) → (*O1* -*m*> *O2*) → *O1* -*m*> *O3*.

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

Lemma *fmon_comp_simpl* : ∀ (*O1 O2 O3*:*ord*) (*f* :*O2* -*m*> *O3*) (*g*:*O1* -*m*> *O2*) (*x*:*O1*),
(*f* @ *g*) *x* = *f* (*g x*).

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

Definition *fmon_comp2* :
∀ *O1 O2 O3 O4*:*ord*, (*O2* -*m*> *O3* -*m*> *O4*) → (*O1* -*m*> *O2*) → (*O1* -*m*> *O3*) → *O1*-*m*>*O4*.

Infix "@2" := *fmon_comp2* (*at level* 70) : *O_scope*.

Lemma *fmon_comp2_simpl* :

∀ (*O1 O2 O3 O4*:*ord*) (*f*:*O2* -*m*> *O3* -*m*> *O4*) (*g*:*O1* -*m*> *O2*) (*h*:*O1* -*m*> *O3*) (*x*:*O1*),
(*f* @2 *g*) *h x* = *f* (*g x*) (*h x*).

Add Morphism fmon_comp with *signature Ole* ++> *Ole* ++> *Ole* as *fmon_comp_le_compat_morph*.

Lemma *fmon_comp_le_compat* :
∀ (*O1 O2 O3*:*ord*) (*f1 f2*: *O2* -*m*> *O3*) (*g1 g2*:*O1* -*m*> *O2*),
f1 ≤ *f2* → *g1*≤ *g2* → *f1* @ *g1* ≤ *f2* @ *g2*.
Hint Immediate *fmon_comp_le_compat*.

Add Morphism fmon_comp with *signature Oeq* ==> *Oeq* ==> *Oeq* as *fmon_comp_eq_compat*.
Hint Immediate *fmon_comp_eq_compat*.

Lemma *fmon_comp_monotonic2* :
∀ (*O1 O2 O3*:*ord*) (*f*: *O2* -*m*> *O3*) (*g1 g2*:*O1* -*m*> *O2*),
g1≤ *g2* → *f* @ *g1* ≤ *f* @ *g2*.
Hint *Resolve fmon_comp_monotonic2*.

Lemma *fmon_comp_monotonic1* :
∀ (*O1 O2 O3*:*ord*) (*f1 f2*: *O2* -*m*> *O3*) (*g*:*O1* -*m*> *O2*),
f1≤ *f2* → *f1* @ *g* ≤ *f2* @ *g*.
Hint *Resolve fmon_comp_monotonic1*.

Definition *fcomp* : ∀ *O1 O2 O3*:*ord*, (*O2* -*m*> *O3*) -*m*> (*O1* -*m*> *O2*) -*m*> (*O1* -*m*> *O3*).

Lemma *fmon_le_compat* : ∀ (*O1 O2*:*ord*) (*f*: *O1* -*m*> *O2*) (*x y*:*O1*), *x*≤*y* → *f x* ≤ *f y*.
Hint *Resolve fmon_le_compat*.

Lemma *fmon_le_compat2* : ∀ (*O1 O2 O3*:*ord*) (*f*: *O1* -*m*> *O2* -*m*> *O3*) (*x y*:*O1*) (*z t*:*O2*),
x≤*y* → *z* ≤*t* → *f x z* ≤ *f y t*.
Hint *Resolve fmon_le_compat2*.

Lemma *fmon_cte_comp* : ∀ (*O1 O2 O3*:*ord*)(*c*:*O3*)(*f*:*O1*-*m*>*O2*),
fmon_cte O2 c @ *f* == *fmon_cte O1 c*.

1.4 Basic operators of omega-cpos

- Constant : 0
- lub : limit of monotonic sequences

1.4.1 Definition of cpos

Record *cpo* : Type := *mk_cpo*
{*tcpo*:>*ord*; *D0* : *tcpo*; *lub*: (*natO* -*m*> *tcpo*) → *tcpo*;
Dbot : ∀ *x*:*tcpo*, *D0* ≤ *x*;
le_lub : ∀ (*f* : *natO* -*m*> *tcpo*) (*n*:*nat*), *f n* ≤ *lub f*;
lub_le : ∀ (*f* : *natO* -*m*> *tcpo*) (*x*:*tcpo*), (∀ *n*, *f n* ≤ *x*) → *lub f* ≤ *x*}.

Implicit *Arguments D0* [c].
Notation "0" := *D0* : *O_scope*.

Hint *Resolve Dbot le_lub lub_le*.

1.4.2 Least upper bounds

Add Morphism lub with *signature Ole* ++> *Ole* as *lub_le_compat_morph*.
Hint *Resolve lub_le_compat_morph*.

Lemma *lub_le_compat* : ∀ (*D*:*cpo*) (*f g*:*natO* -*m*> *D*), *f* ≤ *g* → *lub f* ≤ *lub g*.
Hint *Resolve lub_le_compat*.

Definition *Lub* : ∀ (*D*:*cpo*), (*natO* -*m*> *D*) -*m*> *D*.

Add Morphism lub with *signature Oeq ==> Oeq* as *lub_eq_compat.*
Hint *Resolve lub_eq_compat.*

Lemma *lub_cte* : ∀ (*D*:*cpo*) (*c*:*D*), *lub* (*fmon_cte natO c*) == *c*.

Hint *Resolve lub_cte.*

Lemma *lub_lift_right* : ∀ (*D*:*cpo*) (*f*:*natO* -*m*> *D*) *n*, *lub f* == *lub* (*mseq_lift_right f n*).
Hint *Resolve lub_lift_right.*

Lemma *lub_lift_left* : ∀ (*D*:*cpo*) (*f*:*natO* -*m*> *D*) *n*, *lub f* == *lub* (*mseq_lift_left f n*).
Hint *Resolve lub_lift_left.*

Lemma *lub_le_lift* : ∀ (*D*:*cpo*) (*f g*:*natO* -*m*> *D*) (*n*:*natO*),
(∀ *k*, *n* ≤ *k* → *f k* ≤ *g k*) → *lub f* ≤ *lub g*.

Lemma *lub_eq_lift* : ∀ (*D*:*cpo*) (*f g*:*natO* -*m*> *D*) (*n*:*natO*),
(∀ *k*, *n* ≤ *k* → *f k* == *g k*) → *lub f* == *lub g*.

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

Definition *lub_fun* : ∀ (*O*:*ord*) (*D*:*cpo*) (*h* : *natO* -*m*> *O* -*m*> *D*), *O* -*m*> *D*.

Lemma *lub_fun_eq* : ∀ (*O*:*ord*) (*D*:*cpo*) (*h* : *natO* -*m*> *O* -*m*> *D*) (*x*:*O*),
lub_fun h x == *lub* (*h* <_> *x*).

Lemma *lub_fun_shift* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> (*natO* -*m*> *D*)),
lub_fun h == *Lub D* @ (*fmon_shift h*).

Lemma *double_lub_simpl* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> *natO* -*m*> *D*),
lub (*Lub D* @ *h*) == *lub* (*fmon_diag h*).

Lemma *lub_exch_le* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> (*natO* -*m*> *D*)),
lub (*Lub D* @ *h*) ≤ *lub* (*lub_fun h*).
Hint *Resolve lub_exch_le.*

Lemma *lub_exch_eq* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> (*natO* -*m*> *D*)),
lub (*Lub D* @ *h*) == *lub* (*lub_fun h*).

Hint *Resolve lub_exch_eq.*

1.4.3 Functional cpos

Definition *fcpo* : Type → *cpo* → *cpo*.

Infix "-*O*→" := *fcpo* (*right associativity*, *at level* 30) : *O_scope*.

Lemma *fcpo_lub_simpl* : ∀ *A* (*D*:*cpo*) (*h*:*natO*-*m*> *A*-*O*→*D*)(*x*:*A*),
(*lub h*) *x* = *lub*(*c*:=*D*) (*h*<o> *x*).

1.5 Continuity

Lemma *lub_comp_le* :
∀ (*D1 D2* : *cpo*) (*f*:*D1* -*m*> *D2*) (*h* : *natO* -*m*> *D1*), *lub* (*f* @ *h*) ≤ *f* (*lub h*).
Hint *Resolve lub_comp_le.*

Lemma *lub_comp2_le* : ∀ (*D1 D2 D3*: *cpo*) (*F*:*D1* -*m*> *D2*-*m*>*D3*) (*f* : *natO* -*m*> *D1*) (*g*: *natO* -*m*> *D2*),
lub ((*F* @2 *f*) *g*) ≤ *F* (*lub f*) (*lub g*).
Hint *Resolve lub_comp2_le.*

Definition *continuous* (*D1 D2* : *cpo*) (*f*:*D1* -*m*> *D2*)
:= ∀ *h* : *natO* -*m*> *D1*, *f* (*lub h*) ≤ *lub* (*f* @ *h*).

Lemma *continuous_eq_compat* : ∀ (*D1 D2* : *cpo*) (*f g*:*D1* -*m*> *D2*),
f==*g* → *continuous f* → *continuous g*.

Add Morphism continuous with *signature Oeq ==> iff* as *continuous_eq_compat_iff.*

Lemma *lub_comp_eq* :
∀ (*D1 D2* : *cpo*) (*f*:*D1* -*m*> *D2*) (*h* : *natO* -*m*> *D1*), *continuous f* → *f* (*lub h*) == *lub* (*f* @ *h*).
Hint *Resolve lub_comp_eq*.

- mon0 x == 0

Definition *mon0* (*O1*:*ord*) (*D2* : *cpo*) : *O1* -*m*> *D2* := *fmon_cte O1* (0:*D2*).

Lemma *cont0* : ∀ (*D1 D2* : *cpo*), *continuous* (*mon0 D1 D2*).
Implicit *Arguments cont0* [].

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

Definition *double_app* (*O1 O2 O3 O4*: *ord*) (*f*:*O1* -*m*> *O3* -*m*> *O4*) (*g*:*O2* -*m*> *O3*)
: *O2* -*m*> (*O1* -*m*> *O4*) := (*fmon_shift f*) @ *g*.

1.6 Cpo of monotonic functions

Definition *fmon_cpo* : ∀ (*O*:*ord*) (*D*:*cpo*), *cpo*.

Infix "-*M*→" := *fmon_cpo* (*at level* 30, *right associativity*) : *O_scope*.

Lemma *fmon_lub_simpl* : ∀ (*O*:*ord*) (*D*:*cpo*) (*h*:*natO*-*m*>*O*-*M*→*D*) (*x*:*O*),
(*lub h*) *x* = *lub* (*h* <_> *x*).

Lemma *double_lub_diag* : ∀ (*D*:*cpo*) (*h*:*natO*-*m*>*natO*-*M*→*D*),
lub (*lub h*) == *lub* (*fmon_diag h*).

1.6.1 Continuity

Definition *continuous2* (*D1 D2 D3*: *cpo*) (*F*:*D1* -*m*> *D2* -*m*> *D3*)
:= ∀ (*f* : *natO*-*m*>*D1*) (*g* :*natO*-*m*>*D2*), *F* (*lub f*) (*lub g*) ≤ *lub* ((*F* @2 *f*) *g*).

Lemma *continuous2_app* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1*-*m*>*D2*-*m*>*D3*),
continuous2 F → ∀ *k*, *continuous* (*F k*).

Lemma *continuous2_continuous* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1*-*m*>*D2*-*M*→*D3*),
continuous2 F → *continuous F*.

Lemma *continuous2_left* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1*-*m*>*D2*-*M*→*D3*) (*h*:*natO*-*m*>*D1*) (*x*:*D2*),
continuous2 F → *F* (*lub h*) *x* ≤ *lub* ((*F* <_> *x*) @*h*).

Lemma *continuous2_right* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1*-*m*>*D2*-*M*→*D3*) (*x*:*D1*)(*h*:*natO*-*m*>*D2*),
continuous2 F → *F x* (*lub h*) ≤ *lub* (*F x* @*h*).

Lemma *continuous_continuous2* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1*-*m*>*D2*-*M*→*D3*),
(∀ *k*:*D1*, *continuous* (*F k*)) → *continuous F* → *continuous2 F*.

Hint *Resolve continuous2_app continuous2_continuous continuous_continuous2*.

Lemma *lub_comp2_eq* : ∀ (*D1 D2 D3*:*cpo*) (*F* : *D1* -*m*> *D2* -*M*→ *D3*),
(∀ *k*:*D1*, *continuous* (*F k*)) → *continuous F* →
∀ (*f* : *natO*-*m*>*D1*) (*g* :*natO*-*m*>*D2*),
F (*lub f*) (*lub g*) == *lub* ((*F*@2 *f*) *g*).

Lemma *continuous_sym* : ∀ (*D1 D2*:*cpo*) (*F* : *D1*-*m*> *D1* -*M*→ *D2*),
(∀ *x y*, *F x y* == *F y x*) → (∀ *k*:*D1*, *continuous* (*F k*)) → *continuous F*.

Lemma *continuous2_sym* : ∀ (*D1 D2*:*cpo*) (*F* : *D1*-*m*>*D1*-*m*>*D2*),
(∀ *x y*, *F x y* == *F y x*) → (∀ *k*, *continuous* (*F k*)) → *continuous2 F*.
Hint *Resolve continuous2_sym*.

- continuity is preserved by composition

Lemma *continuous_comp* : ∀ (*D1 D2 D3*:*cpo*) (*f*:*D2*-*m*>*D3*)(*g*:*D1*-*m*>*D2*),
continuous f → *continuous g* → *continuous* (*f*@*g*).
Hint *Resolve continuous_comp*.

1.7 Cpo of continuous functions

Lemma *cont_lub* : ∀ (*D1 D2* : *cpo*) (*f*:*natO -m> (D1 -m> D2)*),
(∀ *n*, *continuous* (*f n*)) →
continuous (*lub* (*c*:=*D1-M→D2*) *f*).

Record *fconti* (*D1 D2*:*cpo*): Type
:= *mk_fconti* {*fcontit* : *D1 -m> D2*; *fcontinuous* : *continuous fcontit*}.
Hint *Resolve fcontinuous.*

Definition *fconti_fun* (*D1 D2* :*cpo*) (*f*:*fconti D1 D2*) : *D1*→ *D2* :=fun *x* ⇒ *fcontit f x.*
Coercion fconti_fun : *fconti* >-> *Funclass.*

Definition *fcont_ord* : *cpo* → *cpo* → *ord.*

Infix "-*c*>" := *fcont_ord* (*at level* 30, *right associativity*) : *O_scope.*

Lemma *fcont_le_intro* : ∀ (*D1 D2*:*cpo*) (*f g* : *D1 -c> D2*), (∀ *x*, *f x* ≤ *g x*) → *f* ≤ *g.*

Lemma *fcont_le_elim* : ∀ (*D1 D2*:*cpo*) (*f g* : *D1 -c> D2*), *f* ≤ *g* → ∀ *x*, *f x* ≤ *g x.*

Lemma *fcont_eq_intro* : ∀ (*D1 D2*:*cpo*) (*f g* : *D1 -c> D2*), (∀ *x*, *f x* == *g x*) → *f* == *g.*

Lemma *fcont_eq_elim* : ∀ (*D1 D2*:*cpo*) (*f g* : *D1 -c> D2*), *f* == *g* → ∀ *x*, *f x* == *g x.*

Lemma *fcont_monotonic* : ∀ (*D1 D2*:*cpo*) (*f* : *D1 -c> D2*) (*x y* : *D1*),
x ≤ *y* → *f x* ≤ *f y.*
Hint *Resolve fcont_monotonic.*

Lemma *fcont_stable* : ∀ (*D1 D2*:*cpo*) (*f* : *D1 -c> D2*) (*x y* : *D1*),
x == *y* → *f x* == *f y.*
Hint *Resolve fcont_monotonic.*

Definition *fcont0* (*D1 D2*:*cpo*) : *D1 -c> D2* := *mk_fconti* (*cont0 D1 D2*).

Definition *Fcontit* (*D1 D2*:*cpo*) : (*D1 -c> D2*) *-m> D1-m> D2.*

Definition *fcont_lub* (*D1 D2*:*cpo*) : (*natO -m> D1 -c> D2*) → *D1 -c> D2.*

Definition *fcont_cpo* : *cpo* → *cpo* → *cpo.*

Infix "-*C*→" := *fcont_cpo* (*at level* 30, *right associativity*) : *O_scope.*

Definition *fcont_app* (*O*:*ord*) (*D1 D2*:*cpo*) (*f*: *O -m> D1-c> D2*) (*x*:*D1*) : *O -m> D2*
:= *Fcontit D1 D2* @ *f* <_> *x.*

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

Lemma *fcont_app_simpl* : ∀ (*O*:*ord*) (*D1 D2*:*cpo*) (*f*: *O -m> D1-c> D2*) (*x*:*D1*)(*y*:*O*),
(*f* <__> *x*) *y* = *f y x.*

Definition *ford_fcont_shift* (*A*:Type) (*D1 D2*:*cpo*) (*f*: *A -o> (D1-c> D2)*) : *D1 -c> A -O→ D2.*

Definition *fmon_fcont_shift* (*O*:*ord*) (*D1 D2*:*cpo*) (*f*: *O -m> D1-c> D2*) : *D1 -c> O -M→ D2.*

Lemma *fcont_app_continuous* :
∀ (*O*:*ord*) (*D1 D2*:*cpo*) (*f*: *O -m> D1-c> D2*) (*h*:*natO-m>D1*),
f <__> (*lub h*) ≤ *lub* (*c*:=*O-M→D2*) (*fcontit* (*fmon_fcont_shift f*) @ *h*).

Lemma *fcont_lub_simpl* : ∀ (*D1 D2*:*cpo*) (*h*:*natO-m>D1-C→D2*)(*x*:*D1*),
lub h x = *lub* (*h* <__> *x*).

Definition *continuous2_cont_app* : ∀ (*D1 D2 D3* :*cpo*) (*f*:*D1-m> D2 -M→ D3*),
(∀ *k*, *continuous* (*f k*)) → *D1 -m>* (*D2 -C→ D3*).

Lemma *continuous2_cont_app_simpl* :
∀ (*D1 D2 D3* :*cpo*) (*f*:*D1-m> D2 -M→ D3*)(*H*:∀ *k*, *continuous* (*f k*))
(*k*:*D1*), *continuous2_cont_app H k* = *mk_fconti* (*H k*).

Lemma *continuous2_cont* : ∀ (*D1 D2 D3* :*cpo*) (*f*:*D1-m> D2 -M→ D3*),
continuous2 f → *D1 -c>* (*D2 -C→ D3*).

Lemma *Fcontit_cont* : ∀ *D1 D2*, *continuous* (*D1*:=*D1-C→D2*) (*D2*:=*D1-M→D2*) (*Fcontit D1 D2*).

Hint *Resolve Fcontit_cont.*

Definition *fcont_comp* : ∀ (*D1 D2 D3*:*cpo*), (*D2 -c> D3*) → (*D1-c> D2*) → *D1 -c> D3.*

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

Lemma *fcont_comp_simpl* : ∀ (*D1 D2 D3*:*cpo*)(*f*:*D2 -c> D3*)(*g*:*D1-c> D2*) (*x*:*D1*),
(*f* @_ *g*) *x* = *f* (*g x*).

Lemma *fcontit_comp_simpl* : ∀ (*D1 D2 D3*:*cpo*)(*f*:*D2 -c> D3*)(*g*:*D1-c> D2*) (*x*:*D1*),
fcontit (*f* @_ *g*) = *fcontit f* @ *fcontit g.*

Lemma *fcont_comp_le_compat* : ∀ (*D1 D2 D3*:*cpo*) (*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 Morphism fcont_comp with *signature Ole ++> Ole ++> Ole* as *fcont_comp_le_morph.*

Add Morphism fcont_comp with *signature Oeq ==> Oeq ==> Oeq* as *fcont_comp_eq_compat.*

Definition *fcont_Comp* (*D1 D2 D3*:*cpo*) : (*D2 -C→ D3*) *-m>* (*D1-C→ D2*) *-m> D1 -C→ D3*
:= *le_compat2_mon* (*fcont_comp_le_compat* (*D1*:=*D1*) (*D2*:=*D2*) (*D3*:=*D3*)).

Lemma *fcont_Comp_simpl* : ∀ (*D1 D2 D3*:*cpo*) (*f*:*D2 -c> D3*) (*g*:*D1-c> D2*),
fcont_Comp D1 D2 D3 f g = *f* @_ *g.*

Lemma *fcont_Comp_continuous2* : ∀ (*D1 D2 D3*:*cpo*), *continuous2* (*fcont_Comp D1 D2 D3*).

Definition *fcont_COMP* (*D1 D2 D3*:*cpo*) : (*D2 -C→ D3*) *-c>* (*D1-C→ D2*) *-C→ D1 -C→ D3*
:= *continuous2_cont* (*fcont_Comp_continuous2* (*D1*:=*D1*) (*D2*:=*D2*) (*D3*:=*D3*)).

Lemma *fcont_COMP_simpl* : ∀ (*D1 D2 D3*:*cpo*) (*f*: *D2 -C→ D3*) (*g*:*D1-C→ D2*),
fcont_COMP D1 D2 D3 f g = *f* @_ *g.*

Definition *fcont2_COMP* (*D1 D2 D3 D4*:*cpo*) : (*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* (*D1 D2 D3 D4*:*cpo*) (*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* : ∀ (*D1 D2 D3 D4*:*cpo*) (*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* : ∀ (*D1 D2 D3*:*cpo*) (*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* : ∀ (*D1 D2 D3*:*cpo*) (*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* : ∀ (*D1 D2* : *cpo*) (*f*:*D1 -c> D2*)(*h*:*natO-m>D1*),
f (*lub h*) ≤ *lub* (*fcontit f* @ *h*).
Hint *Resolve fcont_continuous.*

Lemma *fcont_continuous2* : ∀ (*D1 D2 D3*:*cpo*) (*f*:*D1-c>*(*D2-C→D3*)),
continuous2 (*Fcontit D2 D3* @ *fcontit f*).
Hint *Resolve fcont_continuous2.*

Definition *fcont_shift* (*D1 D2 D3* : *cpo*) (*f*:*D1-c>D2-C→D3*) : *D2-c>D1-C→D3.*

Lemma *fcont_shift_simpl* : ∀ (*D1 D2 D3* : *cpo*) (*f*:*D1-c>D2-C→D3*) (*x*:*D2*) (*y*:*D1*),
fcont_shift f x y = *f y x.*

Definition *fcont_SEQ* (*D1 D2 D3*:*cpo*) : (*D1-C→ D2*) *-C→* (*D2 -C→ D3*) *-C→ D1 -C→ D3*
:= *fcont_shift* (*fcont_COMP D1 D2 D3*).

Lemma *fcont_SEQ_simpl* : ∀ (*D1 D2 D3*:*cpo*) (*f*: *D1 -C→ D2*) (*g*:*D2-C→ D3*),
fcont_SEQ D1 D2 D3 f g = *g* @_ *f.*

Definition fcont_comp2 : ∀ (D1 D2 D3 D4:cpo),
(D2 -c> D3 -C→D4) → (D1-c> D2) → (D1 -c> D3) → D1-c>D4.

Infix "@2_" := fcont_comp2 (at level 35, right associativity) : O_scope.

Lemma fcont_comp2_simpl : ∀ (D1 D2 D3 D4:cpo)
(F:D2 -c> D3 -C→D4) (f:D1-c> D2) (g:D1 -c> D3) (x:D1), (F@2_ f) g x = F (f x) (g x).

Add Morphism fcont_comp2 with signature Ole++>Ole ++> Ole ++> Ole
as fcont_comp2_le_morph.

Add Morphism fcont_comp2 with signature Oeq ==> Oeq ==> Oeq ==> Oeq as fcont_comp2_eq_compat.

- Identity function is continuous

Definition Id : ∀ O:ord, O-m>O.

Definition ID : ∀ D:cpo, D-c>D.

Lemma Id_simpl : ∀ O x, Id O x = x.

Lemma ID_simpl : ∀ D x, ID D x = Id D x.

Definition AP (D1 D2:cpo) : (D1-C→D2)-c>D1-C→D2:=ID (D1-C→D2).

Lemma AP_simpl : ∀ (D1 D2:cpo) (f : D1-C→D2) (x:D1), AP D1 D2 f x = f x.

Definition fcont_comp3 (D1 D2 D3 D4 D5:cpo)
(F:D2 -c> D3 -C→D4-C→D5)(f:D1-c> D2)(g:D1 -c> D3)(h:D1-c>D4): D1-c>D5
:= (AP D4 D5 @2_ ((F @2_ f) g)) h.

Infix "@3_" := fcont_comp3 (at level 35, right associativity) : O_scope.

Lemma fcont_comp3_simpl : ∀ (D1 D2 D3 D4 D5:cpo)
(F:D2 -c> D3 -C→D4-C→D5) (f:D1-c> D2) (g:D1 -c> D3) (h:D1-c>D4) (x:D1),
(F@3_ f) g h x = F (f x) (g x) (h x).

1.8 Product of two cpos

Definition Oprod : ord → ord → ord.

Definition Fst (O1 O2 : ord) : Oprod O1 O2 -m> O1.

Definition Snd (O1 O2 : ord) : Oprod O1 O2 -m> O2.

Definition Pairr (O1 O2 : ord) : O1 → O2 -m> Oprod O1 O2.

Definition Pair (O1 O2 : ord) : O1 -m> O2 -m> Oprod O1 O2.

Lemma Fst_simpl : ∀ (O1 O2 : ord) (p:Oprod O1 O2), Fst O1 O2 p = fst p.

Lemma Snd_simpl : ∀ (O1 O2 : ord) (p:Oprod O1 O2), Snd O1 O2 p = snd p.

Lemma Pair_simpl : ∀ (O1 O2 : ord) (x:O1)(y:O2), Pair O1 O2 x y = (x,y).

Definition prod0 (D1 D2:cpo) : Oprod D1 D2 := (0: D1,0: D2).
Definition prod_lub (D1 D2:cpo) (f : natO -m> Oprod D1 D2) := (lub (Fst D1 D2@f), lub (Snd D1 D2@f)).

Definition Dprod : cpo → cpo → cpo.

Lemma Dprod_eq_intro : ∀ (D1 D2:cpo) (p1 p2: Dprod D1 D2),
fst p1 == fst p2 → snd p1 == snd p2 → p1 == p2.
Hint Resolve Dprod_eq_intro.

Lemma Dprod_eq_pair : ∀ (D1 D2:cpo) (x1 y1:D1) (x2 y2:D2),
x1==y1 → x2==y2 → ((x1,x2):Dprod D1 D2) == (y1,y2).
Hint Resolve Dprod_eq_pair.

Lemma Dprod_eq_elim_fst : ∀ (D1 D2:cpo) (p1 p2: Dprod D1 D2),

p1==p2 → fst p1 == fst p2.
Hint Immediate *Dprod_eq_elim_fst.*

Lemma *Dprod_eq_elim_snd* : ∀ (*D1 D2*:*cpo*) (*p1 p2*: *Dprod D1 D2*),
p1==p2 → snd p1 == snd p2.
Hint Immediate *Dprod_eq_elim_snd.*

Definition *FST* (*D1 D2*:*cpo*) : *Dprod D1 D2 -c> D1.*

Definition *SND* (*D1 D2*:*cpo*) : *Dprod D1 D2 -c> D2.*

Lemma *Pair_continuous2* : ∀ (*D1 D2*:*cpo*), *continuous2* (*D3*:=*Dprod D1 D2*) (*Pair D1 D2*).

Definition *PAIR* (*D1 D2*:*cpo*) : *D1 -c> D2 -C→ Dprod D1 D2*
:= *continuous2_cont* (*Pair_continuous2* (*D1*:=*D1*) (*D2*:=*D2*)).

Lemma *FST_simpl* : ∀ (*D1 D2* :*cpo*) (*p*:*Dprod D1 D2*), *FST D1 D2 p* = *Fst D1 D2 p.*

Lemma *SND_simpl* : ∀ (*D1 D2* :*cpo*) (*p*:*Dprod D1 D2*), *SND D1 D2 p* = *Snd D1 D2 p.*

Lemma *PAIR_simpl* : ∀ (*D1 D2* :*cpo*) (*p1*:*D1*) (*p2*:*D2*), *PAIR D1 D2 p1 p2* = *Pair D1 D2 p1 p2.*

Lemma *FST_PAIR_simpl* : ∀ (*D1 D2* :*cpo*) (*p1*:*D1*) (*p2*:*D2*),
FST D1 D2 (*PAIR D1 D2 p1 p2*) = *p1.*

Lemma *SND_PAIR_simpl* : ∀ (*D1 D2* :*cpo*) (*p1*:*D1*) (*p2*:*D2*),
SND D1 D2 (*PAIR D1 D2 p1 p2*) = *p2.*

Definition *Prod_map* : ∀ (*D1 D2 D3 D4*:*cpo*)(*f*:*D1-m>D3*)(*g*:*D2-m>D4*) ,
Dprod D1 D2 -m> Dprod D3 D4.

Lemma *Prod_map_simpl* : ∀ (*D1 D2 D3 D4*:*cpo*)(*f*:*D1-m>D3*)(*g*:*D2-m>D4*) (*p*:*Dprod D1 D2*),
Prod_map f g p = *pair* (*f* (*fst p*)) (*g* (*snd p*)).

Definition *PROD_map* : ∀ (*D1 D2 D3 D4*:*cpo*)(*f*:*D1-c>D3*)(*g*:*D2-c>D4*) ,
Dprod D1 D2 -c> Dprod D3 D4.

Lemma *PROD_map_simpl* : ∀ (*D1 D2 D3 D4*:*cpo*)(*f*:*D1-c>D3*)(*g*:*D2-c>D4*)(*p*:*Dprod D1 D2*),
PROD_map f g p = *pair* (*f* (*fst p*)) (*g* (*snd p*)).

Definition *curry* (*D1 D2 D3* : *cpo*) (*f*:*Dprod D1 D2 -c> D3*) : *D1 -c>* (*D2-C→D3*) :=
fcont_COMP D1 (*D2-C→Dprod D1 D2*) (*D2-C→D3*)
(*fcont_COMP D2* (*Dprod D1 D2*) *D3 f*) (*PAIR D1 D2*).

Definition *Curry* : ∀ (*D1 D2 D3* : *cpo*), (*Dprod D1 D2 -c> D3*) *-m> D1 -c>* (*D2-C→D3*).

Lemma *Curry_simpl* : ∀ (*D1 D2 D3* : *cpo*) (*f*:*Dprod D1 D2 -C→ D3*) (*x*:*D1*) (*y*:*D2*),
Curry D1 D2 D3 f x y = *f* (*x*,*y*).

Definition *CURRY* : ∀ (*D1 D2 D3* : *cpo*), (*Dprod D1 D2 -C→ D3*) *-c> D1 -C→* (*D2-C→D3*).

Lemma *CURRY_simpl* : ∀ (*D1 D2 D3* : *cpo*) (*f*:*Dprod D1 D2 -C→ D3*),
CURRY D1 D2 D3 f = *Curry D1 D2 D3 f.*

Definition *uncurry* (*D1 D2 D3* : *cpo*) (*f*:*D1 -c>* (*D2-C→D3*)) : *Dprod D1 D2 -c> D3*
:= (*f* @2_ (*FST D1 D2*)) (*SND D1 D2*).

Definition *Uncurry* : ∀ (*D1 D2 D3* : *cpo*), (*D1 -c>* (*D2-C→D3*)) *-m> Dprod D1 D2 -c> D3.*

Lemma *Uncurry_simpl* : ∀ (*D1 D2 D3* : *cpo*) (*f*:*D1 -c>* (*D2-C→D3*)) (*p*:*Dprod D1 D2*),
Uncurry D1 D2 D3 f p = *f* (*fst p*) (*snd p*).

Definition *UNCURRY* : ∀ (*D1 D2 D3* : *cpo*), (*D1 -C→* (*D2-C→D3*)) *-c> Dprod D1 D2 -C→ D3.*

Lemma *UNCURRY_simpl* : ∀ (*D1 D2 D3* : *cpo*) (*f*:*D1 -c>* (*D2-C→D3*)),
UNCURRY D1 D2 D3 f = *Uncurry D1 D2 D3 f.*

1.9 Indexed product of cpo's

Definition *Oprodi* (I:Type)(O:$I \rightarrow ord$) : *ord*.

Lemma *Oprodi_eq_intro* : ∀ (I:Type)(O:$I \rightarrow ord$) (p q : *Oprodi* O), (∀ i, p i == q i) → p==q.

Lemma *Oprodi_eq_elim* : ∀ (I:Type)(O:$I \rightarrow ord$) (p q : *Oprodi* O), p==q → ∀ i, p i == q i.

Definition *Proj* (I:Type)(O:$I \rightarrow ord$) (i:I) : *Oprodi* O -m> O i.

Lemma *Proj_simpl* : ∀ (I:Type)(O:$I \rightarrow ord$) (i:I) (x:*Oprodi* O),
Proj O i x = x i.

Definition *Dprodi* (I:Type)(D:$I \rightarrow cpo$) : *cpo*.

Lemma *Dprodi_lub_simpl* : ∀ (I:Type)(Di:$I \rightarrow cpo$)(h:*natO*-m>*Dprodi* Di)(i:I),
lub h i = *lub* (c:=Di i) (*Proj* Di i @ h).

Lemma *Dprodi_continuous* : ∀ (D:*cpo*)(I:Type)(Di:$I \rightarrow cpo$)
(f:D -m> *Dprodi* Di), (∀ i, *continuous* (*Proj* Di i @ f)) →
continuous f.

Definition *Dprodi_lift* : ∀ (I J:Type)(Di:$I \rightarrow cpo$)(f:$J \rightarrow I$),
Dprodi Di -m> *Dprodi* (fun j ⇒ Di (f j)).

Lemma *Dprodi_lift_simpl* : ∀ (I J:Type)(Di:$I \rightarrow cpo$)(f:$J \rightarrow I$)(p:*Dprodi* Di),
Dprodi_lift Di f p = fun j ⇒ p (f j).

Lemma *Dprodi_lift_cont* : ∀ (I J:Type)(Di:$I \rightarrow cpo$)(f:$J \rightarrow I$),
continuous (*Dprodi_lift* Di f).

Definition *DLIFTi* (I J:Type)(Di:$I \rightarrow cpo$)(f:$J \rightarrow I$) : *Dprodi* Di -c> *Dprodi* (fun j ⇒ Di (f j))
:= *mk_fconti* (*Dprodi_lift_cont* (Di:=Di) f).

Definition *Dmapi* : ∀ (I:Type)(Di Dj:$I \rightarrow cpo$)(f:∀ i, Di i -m> Dj i),
Dprodi Di -m> *Dprodi* Dj.

Lemma *Dmapi_simpl* : ∀ (I:Type)(Di Dj:$I \rightarrow cpo$)(f:∀ i, Di i -m> Dj i) (p:*Dprodi* Di) (i:I),
Dmapi f p i = f i (p i).

Lemma *DMAPi* : ∀ (I:Type)(Di Dj:$I \rightarrow cpo$)(f:∀ i, Di i -c> Dj i),
Dprodi Di -c> *Dprodi* Dj.

Lemma *DMAPi_simpl* : ∀ (I:Type)(Di Dj:$I \rightarrow cpo$)(f:∀ i, Di i -c> Dj i) (p:*Dprodi* Di) (i:I),
DMAPi f p i = f i (p i).

Lemma *Proj_cont* : ∀ (I:Type)(Di:$I \rightarrow cpo$) (i:I),
continuous ($D1$:=*Dprodi* Di) ($D2$:=Di i) (*Proj* Di i).

Definition *PROJ* (I:Type)(Di:$I \rightarrow cpo$) (i:I) : *Dprodi* Di -c> Di i :=
mk_fconti (*Proj_cont* (Di:=Di) i).

Lemma *PROJ_simpl* : ∀ (I:Type)(Di:$I \rightarrow cpo$) (i:I)(d:*Dprodi* Di),
PROJ Di i d = d i.

1.9.1 Particular cases with one or two elements

Section *Product2*.

Definition *I2* := *bool*.
Variable *DI2* : *bool* → *cpo*.

Definition *DP1* := *DI2* *true*.
Definition *DP2* := *DI2* *false*.

Definition *PI1* : *Dprodi* *DI2* -c> *DP1* := *PROJ* *DI2* *true*.
Definition *pi1* (d:*Dprodi* *DI2*) := *PI1* d.

Definition *PI2* : *Dprodi* *DI2* -c> *DP2* := *PROJ* *DI2* *false*.
Definition *pi2* (d:*Dprodi* *DI2*) := *PI2* d.

Definition *pair2* (*d1*:*DP1*) (*d2*:*DP2*) : *Dprodi DI2* := *bool_rect DI2 d1 d2*.

Lemma *pair2_le_compat* : ∀ (*d1 d'1*:*DP1*) (*d2 d'2*:*DP2*), *d1* ≤ *d'1* → *d2* ≤ *d'2*
→ *pair2 d1 d2* ≤ *pair2 d'1 d'2*.

Definition *Pair2* : *DP1 -m> DP2 -m> Dprodi DI2* := *le_compat2_mon pair2_le_compat*.

Definition *PAIR2* : *DP1 -c> DP2 -C→ Dprodi DI2*.

Lemma *PAIR2_simpl* : ∀ (*d1*:*DP1*) (*d2*:*DP2*), *PAIR2 d1 d2* = *Pair2 d1 d2*.

Lemma *Pair2_simpl* : ∀ (*d1*:*DP1*) (*d2*:*DP2*), *Pair2 d1 d2* = *pair2 d1 d2*.

Lemma *pi1_simpl* : ∀ (*d1*: *DP1*) (*d2*:*DP2*), *pi1* (*pair2 d1 d2*) = *d1*.

Lemma *pi2_simpl* : ∀ (*d1*: *DP1*) (*d2*:*DP2*), *pi2* (*pair2 d1 d2*) = *d2*.

Definition *DI2_map* (*f1* : *DP1 -c> DP1*) (*f2*:*DP2 -c> DP2*)
: *Dprodi DI2 -c> Dprodi DI2* :=
DMAPi (*bool_rect* (fun *b*:*bool* ⇒ *DI2 b -c>DI2 b*) *f1 f2*).

Lemma *Dl2_map_eq* : ∀ (*f1* : *DP1 -c> DP1*) (*f2*:*DP2 -c> DP2*) (*d*:*Dprodi DI2*),
DI2_map f1 f2 d == *pair2* (*f1* (*pi1 d*)) (*f2* (*pi2 d*)).
End *Product2*.
Hint *Resolve Dl2_map_eq*.

Section *Product1*.
Definition *I1* := *unit*.
Variable *D* : *cpo*.

Definition *DI1* (_:*unit*) := *D*.
Definition *PI* : *Dprodi DI1 -c> D* := *PROJ DI1 tt*.
Definition *pi* (*d*:*Dprodi DI1*) := *PI d*.

Definition *pair1* (*d*:*D*) : *Dprodi DI1* := *unit_rect DI1 d*.

Definition *pair1_simpl* : ∀ (*d*:*D*) (*x*:*unit*), *pair1 d x* = *d*.

Definition *Pair1* : *D -m> Dprodi DI1*.

Lemma *Pair1_simpl* : ∀ (*d*:*D*), *Pair1 d* = *pair1 d*.

Definition *PAIR1* : *D -c> Dprodi DI1*.

Lemma *pi_simpl* : ∀ (*d*:*D*), *pi* (*pair1 d*) = *d*.

Definition *DI1_map* (*f* : *D -c> D*)
: *Dprodi DI1 -c> Dprodi DI1* :=
DMAPi (fun *t*:*unit* ⇒ *f*).

Lemma *DI1_map_eq* : ∀ (*f* : *D -c> D*) (*d*:*Dprodi DI1*),
DI1_map f d == *pair1* (*f* (*pi d*)).
End *Product1*.

Hint *Resolve DI1_map_eq*.

1.10 Fixpoints

Section *Fixpoints*.
Variable *D*: *cpo*.
Variable *f* : *D -m>D*.

Hypothesis *fcont* : *continuous f*.

Fixpoint *iter_ n* : *D* := match *n* with *O* ⇒ 0 | *S m* ⇒ *f* (*iter_ m*) end.

Lemma *iter_incr* : ∀ *n*, *iter_ n* ≤ *f* (*iter_ n*).
Hint *Resolve iter_incr*.

Definition *iter* : *natO -m> D*.

Definition *fixp* : *D* := *lub iter*.

Lemma *fixp_le* : *fixp* ≤ *f fixp*.
Hint *Resolve fixp_le*.

Lemma *fixp_eq* : *fixp* == *f fixp*.

Lemma *fixp_inv* : ∀ *g*, *f g* ≤ *g* → *fixp* ≤ *g*.

End *Fixpoints*.
Hint *Resolve fixp_le fixp_eq fixp_inv*.

Definition *fixp_cte* : ∀ (*D*:*cpo*) (*d*:*D*), *fixp* (*fmon_cte D d*) == *d*.
Hint *Resolve fixp_cte*.

Lemma *fixp_le_compat* : ∀ (*D*:*cpo*) (*f g* : *D*-*m*>*D*), *f*≤*g* → *fixp f* ≤ *fixp g*.
Hint *Resolve fixp_le_compat*.

Add Morphism fixp with *signature Oeq* ==> *Oeq* as *fixp_eq_compat*.
Hint *Resolve fixp_eq_compat*.

Definition *Fixp* : ∀ (*D*:*cpo*), (*D*-*m*>*D*) -*m*> *D*.

Lemma *Fixp_simpl* : ∀ (*D*:*cpo*) (*f*:*D*-*m*>*D*), *Fixp D f* = *fixp f*.

Definition *Iter* : ∀ *D*:*cpo*, (*D*-*M*→*D*) -*m*> (*natO* -*M*→*D*).

Lemma *IterS_simpl* : ∀ (*D*:*cpo*) *f n*, *Iter D f* (*S n*) = *f* (*Iter D f n*).

Lemma *iterS_simpl* : ∀ (*D*:*cpo*) *f n*, *iter f* (*S n*) = *f* (*iter* (*D*:=*D*) *f n*).

Lemma *iter_continuous* : ∀ (*D*:*cpo*),
  ∀ *h* : *natO* -*m*> (*D*-*M*→*D*), (∀ *n*, *continuous* (*h n*)) →
    iter (*lub h*) ≤ *lub* (*Iter D* @ *h*).

Hint *Resolve iter_continuous*.

Lemma *iter_continuous_eq* : ∀ (*D*:*cpo*),
  ∀ *h* : *natO* -*m*> (*D*-*M*→*D*), (∀ *n*, *continuous* (*h n*)) →
    iter (*lub h*) == *lub* (*Iter D* @ *h*).

Lemma *fixp_continuous* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> (*D*-*M*→*D*)),
  (∀ *n*, *continuous* (*h n*)) → *fixp* (*lub h*) ≤ *lub* (*Fixp D* @ *h*).
Hint *Resolve fixp_continuous*.

Lemma *fixp_continuous_eq* : ∀ (*D*:*cpo*) (*h* : *natO* -*m*> (*D* -*M*→ *D*)),
  (∀ *n*, *continuous* (*h n*)) → *fixp* (*lub h*) == *lub* (*Fixp D* @ *h*).

Definition *FIXP* : ∀ (*D*:*cpo*), (*D*-*C*→*D*) -*c*> *D*.

Lemma *FIXP_simpl* : ∀ (*D*:*cpo*) (*f*:*D*-*c*>*D*), *FIXP D f* = *Fixp D* (*fcontit f*).

Lemma *FIXP_le_compat* : ∀ (*D*:*cpo*) (*f g* : *D*-*C*→*D*),
  f ≤ *g* → *FIXP D f* ≤ *FIXP D g*.
Hint *Resolve FIXP_le_compat*.

Lemma *FIXP_eq_compat* : ∀ (*D*:*cpo*) (*f g* : *D*-*C*→*D*),
  f == *g* → *FIXP D f* == *FIXP D g*.
Hint *Resolve FIXP_eq_compat*.

Lemma *FIXP_eq* : ∀ (*D*:*cpo*) (*f*:*D*-*c*>*D*), *FIXP D f* == *f* (*FIXP D f*).
Hint *Resolve FIXP_eq*.

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

1.10.1 Iteration of functional

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

Lemma *FIXP_comp* : ∀ (*D*:*cpo*) (*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*:*cpo*)(*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_com* : ∀ (*D*:*cpo*)(*f*:*D*-*c*>*D*) (*n*:*nat*),
f @_ (*fcont_compn f n*) ≤ *fcont_compn f n* @_ *f*.

Lemma *FIXP_compn* :
∀ (*D*:*cpo*) (*f*:*D*-*c*>*D*) (*n*:*nat*), *FIXP D* (*fcont_compn f n*) == *FIXP D f*.

Lemma *fixp_double* : ∀ (*D*:*cpo*) (*f*:*D*-*c*>*D*), *FIXP D* (*f* @_ *f*) == *FIXP D f*.

Lemma *FIXP_proj* : ∀ (*I*:Type)(*DI*: *I* → *cpo*) (*F*:*Dprodi DI* -*c*>*Dprodi DI*) (*i*:*I*) (*fi* : *DI i* -*c*> *DI i*),
(∀ *X* : *Dprodi DI*, *F X i* == *fi* (*X i*)) → *FIXP* (*Dprodi DI*) *F i* == *FIXP* (*DI i*) *fi*.

1.10.2 Induction principle

Definition *admissible* (*D*:*cpo*)(*P*:*D*→Type) :=
∀ *f* : *natO* -*m*> *D*, (∀ *n*, *P* (*f n*)) → *P* (*lub f*).

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

1.11 Directed complete partial orders without minimal element

Record *dcpo* : Type := *mk_dcpo*
{*tdcpo*:>*ord*; *dlub*: (*natO* -*m*> *tdcpo*) → *tdcpo*;
le_dlub : ∀ (*f* : *natO* -*m*> *tdcpo*) (*n*:*nat*), *f n* ≤ *dlub f*;
dlub_le : ∀ (*f* : *natO* -*m*> *tdcpo*) (*x*:*tdcpo*), (∀ *n*, *f n* ≤ *x*) → *dlub f* ≤ *x*}.

Hint *Resolve le_dlub dlub_le*.

Lemma *dlub_le_compat* : ∀ (*D*:*dcpo*)(*f1 f2* : *natO* -*m*> *D*), *f1* ≤ *f2* → *dlub f1* ≤ *dlub f2*.
Hint *Resolve dlub_le_compat*.

Lemma *dlub_eq_compat* : ∀ (*D*:*dcpo*)(*f1 f2* : *natO* -*m*> *D*), *f1* == *f2* → *dlub f1* == *dlub f2*.
Hint *Resolve dlub_eq_compat*.

Lemma *dlub_lift_right* : ∀ (*D*:*dcpo*) (*f*:*natO*-*m*>*D*) *n*, *dlub f* == *dlub* (*mseq_lift_right f n*).
Hint *Resolve dlub_lift_right*.

Lemma *dlub_cte* : ∀ (*D*:*dcpo*) (*c*:*D*), *dlub* (*mseq_cte c*) == *c*.

1.11.1 A cpo is a dcpo

Definition *cpo_dcpo* : *cpo* → *dcpo*.

1.12 Setoid type

Record *setoid* : Type := *mk_setoid*
{*tset*:>Type; *Seq*:*tset*→*tset*→Prop; *Seq_refl* : ∀ *x* :*tset*, *Seq x x*;
Seq_sym : ∀ *x y*:*tset*, *Seq x y* → *Seq y x*;
Seq_trans : ∀ *x y z*:*tset*, *Seq x y* → *Seq y z* → *Seq x z*}.

Hint *Resolve Seq_refl*.
Hint Immediate *Seq_sym*.

1.12.1 A setoid is an ordered set

Definition *setoid_ord* : *setoid* → *ord*.

Definition *ord_setoid* : *ord* → *setoid*.

1.12.2 A Type is an ordered set and a setoid with Leibniz equality

Definition *type_ord* (X:Type) : *ord*.

Definition *type_setoid* (X:Type) : *setoid*.

1.12.3 A setoid is a dcpo

Definition *lub_eq* (S:*setoid*) (f:*natO-m>setoid_ord S*) := f O.

Lemma *le_lub_eq* : ∀ (S:*setoid*) (f:*natO-m>setoid_ord S*) (n:*nat*), f n ≤ *lub_eq f*.

Lemma *lub_eq_le* : ∀ (S:*setoid*) (f:*natO-m>setoid_ord S*)(x:*setoid_ord S*),
(∀ (n:*nat*), f n ≤ x) → *lub_eq f* ≤ x.

Hint *Resolve le_lub_eq lub_eq_le*.

Definition *setoid_dcpo* : *setoid* → *dcpo*.

Cpo of arrays seen as functions from nat to D with a bound n

Definition *lek* (O:*ord*) (k:*nat*) (f g : *nat* → O) := ∀ n, n < k → f n ≤ g n.
Hint *Unfold lek*.

Lemma *lek_refl* : ∀ (O:*ord*) k (f:*nat* → O), *lek k f f*.
Hint *Resolve lek_refl*.

Lemma *lek_trans* : ∀ (O:*ord*) (k:*nat*) (f g h: *nat* → O), *lek k f g* → *lek k g h* → *lek k f h*.

Definition *natk_ord* : *ord* → *nat* → *ord*.

Definition *norm* (O:*ord*) (x:O) (k:*nat*) (f: *natk_ord O k*) : *natk_ord O k* :=
fun n ⇒ if *le_lt_dec k n* then x else f n.

Lemma *norm_simpl_lt* : ∀ (O:*ord*) (x:O) (k:*nat*) (f: *natk_ord O k*) (n:*nat*),
n < k → *norm x f n* = f n.

Lemma *norm_simpl_le* : ∀ (O:*ord*) (x:O) (k:*nat*) (f: *natk_ord O k*) (n:*nat*),
(k ≤ n)%*nat* → *norm x f n* = x.

Definition *natk_mon_shift* : ∀ (*O1 O2* : *ord*)(x:*O2*) (k:*nat*),
(*O1 -m> natk_ord O2 k*) → *natk_ord* (*O1 -m> O2*) k.

Lemma *natk_mon_shift_simpl*
: ∀ (*O1 O2* : *ord*)(x:*O2*) (k:*nat*)(f:*O1 -m> natk_ord O2 k*) (n:*nat*) (y:*O1*),
natk_mon_shift x f n y = *norm x* (f y) n.

Definition *natk_shift_mon* : ∀ (*O1 O2* : *ord*)(k:*nat*),
(*natk_ord* (*O1 -m> O2*) k) → *O1 -m> natk_ord O2 k*.

Lemma *natk_shift_mon_simpl*
: ∀ (*O1 O2* : *ord*)(k:*nat*)(f:*natk_ord* (*O1 -m> O2*) k) (x:*O1*)(n:*nat*),
natk_shift_mon f x n = f n x.

Definition *natk0* (D:*cpo*) (k:*nat*) : *natk_ord D k* := fun n : *nat* ⇒ (0:D).

Definition *natklub* (D:*cpo*) (k:*nat*) (h:*natO-m>natk_ord D k*) : *natk_ord D k* :=
fun n ⇒ *lub* (*natk_mon_shift* (0:D) h n).

Lemma *natklub_less* : ∀ (D:*cpo*) (k:*nat*) (h:*natO-m>natk_ord D k*) (n:*nat*),
h n ≤ *natklub h*.

Lemma *natklub_least* : ∀ (D:*cpo*) (k:*nat*) (h:*natO-m>natk_ord D k*) (p:*natk_ord D k*),
(∀ n:*nat*, h n ≤ p) → *natklub h* ≤ p.

Definition *Dnatk* : ∀ (D:*cpo*) (k:*nat*), *cpo*.

Notation "k –> D" := (*Dnatk D k*) (*at level* 30, *right associativity*) : *O_scope*.

Definition *natk_shift_cont* : ∀ (*D1 D2* : *cpo*)(k:*nat*),
(k –> (*D1-C→D2*)) → *D1 -c>* (k –> *D2*).

Lemma *natk_shift_cont_simpl*
 : ∀ (*D1 D2*:*cpo*)(*k*:*nat*)(*f*:*k* -> (*D1*-*C*→*D2*)) (*n*:*nat*) (*x*:*D1*),
 natk_shift_cont f x n = *f n x*.

Lemma *natklub_simpl* : ∀ (*D*:*cpo*) (*k*:*nat*) (*h*:*natO* -*m*> *k* -> *D*) (*n*:*nat*),
 lub h n = *lub* (*natk_mon_shift* (0:*D*) *h n*).

Require Export *Arith*.
Require Export *Omega*.

2 Equations.v: Decision of equations between schemes

2.1 Markov rule

Definition *dec* (*P*:*nat*→Prop) := ∀ *n*, {*P n*} + {˜ *P n*}.

Record *Dec* : Type := *mk_Dec* {*prop* :> *nat* → Prop; *is_dec* : *dec prop*}.

Definition *PS* : *Dec* → *Dec*.

Definition *ord* (*P Q*:*Dec*) := ∀ *n*, *Q n* → ∃ *m*, *m* < *n* ∧ *P m*.

Lemma *ord_eq_compat* : ∀ (*P1 P2 Q1 Q2*:*Dec*),
 (∀ *n*, *P1 n* → *P2 n*) → (∀ *n*, *Q2 n* → *Q1 n*)
 → *ord P1 Q1* → *ord P2 Q2*.

Lemma *ord_not_0* : ∀ *P Q* : *Dec*, *ord P Q* → ¬*Q* 0.

Lemma *ord_0* : ∀ *P Q* : *Dec*, *P* 0 → ¬*Q* 0 → *ord P Q*.

- first elt of P then Q

Definition *PP* : *Dec* → *Dec* → *Dec*.

Lemma *PP_PS* : ∀ (*P*:*Dec*) *n*, *PP P* (*PS P*) *n* ↔ *P n*.

Lemma *PS_PP* : ∀ (*P Q*:*Dec*) *n*, *PS* (*PP P Q*) *n* ↔ *Q n*.

Lemma *ord_PS* : ∀ *P* : *Dec*, ¬ *P* 0 → *ord* (*PS P*) *P*.

Lemma *ord_PP* : ∀ (*P Q*: *Dec*), ¬ *P* 0 → *ord Q* (*PP P Q*).

Lemma *ord_PS_PS* : ∀ *P Q* : *Dec*, *ord P Q* → ¬*P* 0 → *ord* (*PS P*) (*PS Q*).

Lemma *Acc_ord_equiv* : ∀ *P Q* : *Dec*, (∀ *n*, *P n* ↔ *Q n*) → *Acc ord P* → *Acc ord Q*.

Lemma *Acc_ord_0* : ∀ *P* : *Dec*, *P* 0 → *Acc ord P*.
Hint Immediate *Acc_ord_0*.

Lemma *Acc_ord_PP* : ∀ (*P Q*:*Dec*), *Acc ord Q* → *Acc ord* (*PP P Q*).

Lemma *Acc_ord_PS* : ∀ (*P*:*Dec*), *Acc ord* (*PS P*) → *Acc ord P*.

Lemma *Acc_ord* : ∀ (*P*:*Dec*), (∃ *n*,*P n*) → *Acc ord P*.

Fixpoint *min_acc* (*P*:*Dec*) (*a*:*Acc ord P*) {*struct a*} : *nat* :=
 match *is_dec P* 0 with
 left _ ⇒ 0 | *right H* ⇒ *S* (*min_acc* (*Acc_inv a* (*PS P*) (*ord_PS P H*))) end.

Definition *minimize* (*P*:*Dec*) (*e*:∃ *n*, *P n*) : *nat* := *min_acc* (*Acc_ord P e*).

Lemma *minimize_P* : ∀ (*P*:*Dec*) (*e*:∃ *n*, *P n*), *P* (*minimize P e*).

Lemma *minimize_min* : ∀ (*P*:*Dec*) (*e*:∃ *n*, *P n*) (*m*:*nat*), *m* < *minimize P e* → ¬ *P m*.

Lemma *minimize_incr* : ∀ (*P Q*:*Dec*)(*e*:∃ *n*, *P n*)(*f*:∃ *n*, *Q n*),
 (∀ *n*, *P n* → *Q n*) → *minimize Q f* ≤ *minimize P e*.

Require Export *Cpo*.

2.2 Definition of terms

Section *Terms*.

Variables F : Type.
Hypothesis $decF$: $\forall\ f\ g : F, \{f{=}g\}{+}\{\tilde{}f{=}g\}$.

Variable $Ar : F \rightarrow nat$.

Record $ind\ (f{:}F)$: Type := $mk\_ind\ \{val :> nat\ ;\ val\_less : val < Ar\ f\}$.

Inductive $term$: Type := $X \mid Ap : F \rightarrow (nat \rightarrow term) \rightarrow term$.

Implicit *Arguments* Ap [].

Inductive $le\_term : term \rightarrow term \rightarrow$ Prop :=
$le\_X : \forall\ t : term,\ le\_term\ X\ t$
$\mid le\_Ap : \forall\ (f{:}F)\ (st1\ st2{:}\ nat \rightarrow term),$
$(\forall\ (i{:}nat), (i < Ar\ f) \rightarrow le\_term\ (st1\ i)\ (st2\ i))$
$\rightarrow le\_term\ (Ap\ f\ st1)\ (Ap\ f\ st2)$.
Hint *Constructors le_term*.

Lemma $le\_term\_refl : \forall\ t : term,\ le\_term\ t\ t$.

Lemma $le\_term\_trans : \forall\ t1\ t2\ t3 : term,\ le\_term\ t1\ t2 \rightarrow le\_term\ t2\ t3 \rightarrow le\_term\ t1\ t3$.

Lemma $not\_le\_term\_Ap\_X : \forall\ f\ st, \neg\ le\_term\ (Ap\ f\ st)\ X$.
Hint *Resolve not_le_term_Ap_X*.

Lemma $not\_le\_term\_Ap\_diff : \forall\ f\ g\ st1\ st2, \neg\ f{=}g \rightarrow \neg\ le\_term\ (Ap\ f\ st1)\ (Ap\ g\ st2)$.
Hint *Resolve not_le_term_Ap_diff*.

Lemma $not\_le\_term\_Ap\_st : \forall\ f\ st1\ st2\ (n{:}nat),$
$n < Ar\ f \rightarrow \neg\ le\_term\ (st1\ n)\ (st2\ n) \rightarrow \neg\ le\_term\ (Ap\ f\ st1)\ (Ap\ f\ st2)$.

Lemma $dec_finite : \forall\ P{:}nat{\rightarrow}$Prop$,\ dec\ P \rightarrow \forall\ n,$
$\{\forall\ i,\ i < n \rightarrow P\ i\} + \{\exists\ i,\ i < n \wedge \neg P\ i\}$.

Definition $le\_term\_dec : \forall\ t\ u, \{le\_term\ t\ u\}{+}\{\tilde{}\ le\_term\ t\ u\}$.

Definition $term\_ord : ord$.

Fixpoint $substX\ (t\ u{:}term\_ord)\ \{struct\ t\} : term\_ord$:=
match t with $X \Rightarrow u \mid Ap\ f\ st \Rightarrow Ap\ f$ (fun $i \Rightarrow substX\ (st\ i)\ u$) end.

Lemma $substX\_le : \forall\ (t\ u{:}term\_ord),\ t \leq substX\ t\ u$.

2.3 Interpretation of a term in cpo

Section *InterpTerm*.
Variable $D : cpo$.

Variable $Finterp : \forall\ f{:}F, (Ar\ f\ {-}{>}\ D)\ {-}c{>}\ D$.

Fixpoint $interp\_term\ (t{:}term) : D\ {-}c{>}\ D$:=
match t with $X \Rightarrow ID\ D$
$\mid Ap\ f\ st \Rightarrow Finterp\ f\ @\_$
$natk\_shift\_cont$ (fun $i \Rightarrow interp\_term\ (st\ i)$)
end.

Lemma $interp\_term\_X : \forall\ x{:}D,\ interp\_term\ X\ x{=}x$.

Lemma $interp\_term\_Ap : \forall\ (f{:}F)\ (st : nat \rightarrow term)\ (x{:}D),$
$interp\_term\ (Ap\ f\ st)\ x = Finterp\ f$ (fun $i \Rightarrow interp\_term\ (st\ i)\ x$).

Definition $interp\_equation\ (t{:}term) : D := FIXP\ D\ (interp\_term\ t)$.

Lemma $interp\_equa\_eq : \forall\ (t{:}term),\ interp\_equation\ t == interp\_term\ t\ (interp\_equation\ t)$.

End *InterpTerm*.
Hint *Resolve interp_term_X interp_term_Ap interp_equa_eq*.

2.4 Construction of the universal domain for terms

Definition *TU* := *natO -m> term_ord*.

2.4.1 Order the universal domain

Definition *TUle* (*T T' : TU*) := ∀ *n*, ∃ *m*, *n<m* ∧ *T n* ≤ *T' m*.

Lemma *TUle_refl* : ∀ *T* : *TU*, *TUle T T*.

Lemma *TUle_trans* : ∀ *T1 T2 T3* : *TU*, *TUle T1 T2* → *TUle T2 T3* → *TUle T1 T3*.

Definition *TU_ord* : *ord*.

2.4.2 Cpo structure for the universal domain

Definition *TU0* : *TU_ord* := *mseq_cte* (*X*:*term_ord*).

Lemma *TU0_less* : ∀ *T* : *TU_ord*, *TU0* ≤ *T*.

- find the smallest m greater than n such that T n <= T' m

Definition *le_term_next* : ∀ (*T T'* : *TU_ord*) (*n*:*nat*), *Dec*.

Definition *TUle_next* (*T T'* : *TU_ord*) (*n*:*nat*) (*p*: *T* ≤ *T'*):= *minimize* (*le_term_next T T' n*) (*p n*).

Lemma *TUle_next_le_term* : ∀ (*T T'* : *TU_ord*) (*p*: *T* ≤ *T'*) (*n*:*nat*),
T n ≤ *T'* (*TUle_next n p*).

Lemma *TUle_next_le* : ∀ (*T T'* : *TU_ord*) (*p*: *T* ≤ *T'*) (*n*:*nat*),
(*n* < *TUle_next n p*)%*nat*.

Lemma *TUle_next_incr* : ∀ (*T T'* : *TU_ord*) (*p q*: *T* ≤ *T'*) (*n m*:*nat*),
(*n* ≤ *m*)%*nat* → (*TUle_next n p* ≤ *TUle_next m q*)%*nat*.

2.4.3 Definition of lubs in the universal domain

- lub T 0 = T 0 0,
- lub T i = T i j with T k l <= lub T i for k <= i, l <= i,
- i <= j, lub T i <= lub T (i+1)

- find the apropriate index in T n starting from T 0 k

Fixpoint *lub_index* (*T* : *natO-m>TU_ord*) (*k*:*nat*) (*n*:*nat*) {*struct n*} : *nat* :=
match *n* with *O* ⇒ *k*
| *S p* ⇒ *TUle_next* (*lub_index T k p*) (*fnatO_elim T p*)
end.

Lemma *lub_index_S* : ∀ (*T* : *natO-m>TU_ord*) (*k*:*nat*) (*n*:*nat*),
lub_index T k (*S n*) = *TUle_next* (*lub_index T k n*) (*fnatO_elim T n*).

Lemma *lub_index_incr* : ∀ (*T* : *natO-m>TU_ord*) (*k l*:*nat*) (*n*:*nat*),
(*k* ≤ *l*) % *nat* → (*lub_index T k n* ≤ *lub_index T l n*)%*nat*.
Hint *Resolve lub_index_incr*.

Lemma *lub_index_le_term_S* : ∀ (*T* : *natO-m>TU_ord*) (*k*:*nat*) (*n*:*nat*),
T n (*lub_index T k n*) ≤ *T* (*S n*) (*lub_index T k* (*S n*)).
Hint *Resolve lub_index_le_term_S*.

Lemma *lub_index_le_term* : ∀ (*T* : *natO-m>TU_ord*) (*k*:*nat*) (*n m*:*nat*),
(*n* ≤ *m*)%*nat* → *T n* (*lub_index T k n*) ≤ *T m* (*lub_index T k m*).

Hint *Resolve lub_index_le_term.*

Lemma *lub_index_le* : ∀ (*T* : *natO-m>TU_ord*) (*k*:*nat*) (*n*:*nat*),
(*n*+*k* ≤ *lub_index T k n*)%*nat.*
Hint *Resolve lub_index_le.*

Definition *TUlub* : (*natO-m>TU_ord*) → *TU_ord.*

Lemma *TUlub_simpl* : ∀ *T n*, *TUlub T n* = *T n* (*lub_index T n n*).

Lemma *TUlub_le_term* : ∀ (*T* : *natO-m>TU_ord*) (*k l n* : *nat*),
(*k* ≤ *n*)%*nat* → (*l*≤*n*)%*nat* → *T k l* ≤ *TUlub T n.*
Hint *Resolve TUlub_le_term.*

Lemma *TUlub_less* : ∀ *T* : *natO-m>TU_ord*, ∀ *n*, *T n* ≤ *TUlub T.*

Lemma *TUlub_least* : ∀ (*T* : *natO-m>TU_ord*) (*T'*:*TU_ord*),
(∀ *n*, *T n* ≤ *T'*) → *TUlub T* ≤ *T'.*

2.4.4 Declaration of the cpo structure

Definition *DTU* : *cpo.*

2.5 Interpretation of terms in the universal domain

Fixpoint *maxk* (*f*:*nat* → *nat*) (*k*:*nat*) (*def*:*nat*) {*struct k*}: *nat* :=
match *k* with *O* ⇒ *def* | *S p* ⇒ let *m*:=*maxk f p def* in
let *a*:= *f p* in
if *le_lt_dec m a* then *a* else *m*
end.

Lemma *maxk_le* : ∀ (*f*:*nat* → *nat*) (*k*:*nat*) (*def*:*nat*),
∀ *p*, *p* < *k* → (*f p* ≤ *maxk f k def*)%*nat.*

Lemma *maxk_le_def* : ∀ (*f*:*nat* → *nat*) (*k*:*nat*) (*def*:*nat*),
(*def*≤ *maxk f k def*)%*nat.*

Definition *TUcte* (*t*:*term*) : *DTU* := *mseq_cte* (*O*:=*term_ord*) *t.*

Definition *DTUAp* : ∀ (*f*:*F*) (*ST*: *Ar f* –> *DTU*), *DTU.*

Lemma *DTUAp_simpl*
: ∀ (*f*:*F*) (*ST*: *Ar f* –> *DTU*)(*n*:*nat*), *DTUAp ST n* = *Ap f* (fun *i* ⇒ *ST i n*).

Definition *DTUAp_mon* : ∀ (*f*:*F*), (*Ar f* –> *DTU*) -*m*> *DTU.*

Lemma *DTUAp_mon_simpl* :
∀ (*f*:*F*) (*ST*: *Ar f* –> *DTU*)(*n*:*nat*), *DTUAp_mon f ST n* = *Ap f* (fun *i* ⇒ *ST i n*).

Definition *TUAp* : ∀ (*f*:*F*), (*Ar f* –> *DTU*) -*c*> *DTU.*

Fixpoint *DTUfix* (*T*:*term*) (*n*:*nat*) {*struct n*}: *term_ord*
:= match *n* with *O* ⇒ *X* | *S p* ⇒ *substX* (*DTUfix T p*) *T* end.

Definition *TUfix* (*T*:*term*) : *DTU.*

Lemma *TUfix_simplS* : ∀ (*T*:*term*) *n*, *TUfix T* (*S n*) = *substX* (*TUfix T n*) *T.*

Lemma *TUfix_simpl0* : ∀ (*T*:*term*) , *TUfix T O* = *X.*

End *Terms.*

Require Export *Cpo.*
Require Export *Arith.*
Require Export *ZArith.*

3 Cpo_flat.v : Flat cpo over a type D

Section *Flat_cpo*.
Variable D : Type.

3.1 Definition

CoInductive *Dflat* : Type := *Eps* : *Dflat* → *Dflat* | *Val* : D → *Dflat*.

Lemma *DF_inv* : ∀ d, d = match d with *Eps* x ⇒ *Eps* x | *Val* d ⇒ *Val* d end.
Hint *Resolve DF_inv*.

3.2 Removing Eps steps

Definition *pred* d : *Dflat* := match d with *Eps* x ⇒ x | *Val* _ ⇒ d end.

```
Fixpoint pred_nth (d:Dflat) (n:nat) {struct n} : Dflat :=
   match n with 0 ⇒ d
              |S m ⇒ match d with Eps x ⇒ pred_nth x m
                                  | Val _ ⇒ d
                     end
   end.
```

Lemma *pred_nth_val* : ∀ x n, *pred_nth* (*Val* x) n = *Val* x.
Hint *Resolve pred_nth_val*.

Lemma *pred_nth_Sn_acc* : ∀ n d, *pred_nth* d (S n) = *pred_nth* (*pred* d) n.

Lemma *pred_nth_Sn* : ∀ n d, *pred_nth* d (S n) = *pred* (*pred_nth* d n).

3.3 Order

```
CoInductive DFle : Dflat → Dflat → Prop :=
     DFleEps : ∀ x y, DFle x y → DFle (Eps x) (Eps y)
   | DFleEpsVal : ∀ x d, DFle x (Val d) → DFle (Eps x) (Val d)
   | DFleVal : ∀ d n y, pred_nth y n = Val d → DFle (Val d) y.
```

Hint *Constructors DFle*.

```
Lemma DFle_rec : ∀ R : Dflat → Dflat → Prop,
  (∀ x y, R (Eps x) (Eps y) → R x y) →
  (∀ x d, R (Eps x) (Val d) → R x (Val d)) →
  (∀ d y, R (Val d) y → ∃ n, pred_nth y n = Val d)
  → ∀ x y, R x y → DFle x y.
```

3.3.1 Properties of the order

Lemma *DFle_refl* : ∀ x, *DFle* x x.
Hint *Resolve DFle_refl*.

Lemma *DFleEps_right* : ∀ x y, *DFle* x y → *DFle* x (*Eps* y).
Hint *Resolve DFleEps_right*.

Lemma *DFleEps_left* : ∀ x y, *DFle* x y → *DFle* (*Eps* x) y.

Hint *Resolve DFleEps_left*.

Lemma *DFle_pred_left* : ∀ x y, *DFle* x y → *DFle* (*pred* x) y.

Lemma *DFle_pred_right* : ∀ x y, *DFle* x y → *DFle* x (*pred* y).

Hint *Resolve DFle_pred_left DFle_pred_right*.

Lemma *DFle_pred* : ∀ x y, *DFle* x y → *DFle* (*pred* x) (*pred* y).

Hint *Resolve DFle_pred.*

Lemma *DFle_pred_nth_left* : ∀ *n x y*, *DFle x y* → *DFle* (*pred_nth x n*) *y*.

Lemma *DFle_pred_nth_right* : ∀ *n x y*,
DFle x y → *DFle x* (*pred_nth y n*).

Hint *Resolve DFle_pred_nth_left DFle_pred_nth_right.*

Lemma *DFleVal_eq* : ∀ *x y*, *DFle* (*Val x*) (*Val y*) → *x*=*y*.
Hint Immediate *DFleVal_eq.*

Lemma *DFleVal_sym* : ∀ *x y*, *DFle* (*Val x*) *y* → *DFle y* (*Val x*).

Lemma *DFle_trans* : ∀ *x y z*, *DFle x y* → *DFle y z* → *DFle x z*.

3.3.2 Declaration of the ordered set

Definition *DF_ord* : *ord.*

3.4 Definition of the cpo structure

Lemma *eq_Eps* : ∀ *x*:*DF_ord*, *x* == *Eps x*.
Hint *Resolve eq_Eps.*

3.4.1 Bottom is given by an infinite chain of Eps

CoFixpoint *DF_bot* : *DF_ord* := *Eps DF_bot.*

Lemma *DF_bot_eq* : *DF_bot* = *Eps DF_bot.*

Lemma *DF_bot_least* : ∀ *x*:*DF_ord*, *DF_bot* ≤ *x*.

3.4.2 More properties of elements in the flat domain

Lemma *DFle_eq* : ∀ *x* (*y*:*DF_ord*), (*Val x*:*DF_ord*) ≤ *y* → (*Val x*:*DF_ord*) == *y*.

Lemma *DFle_Val_exists_pred* :
∀ (*x*:*DF_ord*) *d*, (*Val d*:*DF_ord*) ≤ *x* → ∃ *k*, *pred_nth x k* = *Val d*.

Lemma *Val_exists_pred_le* :
∀ (*x*:*DF_ord*) *d*, (∃ *k*, *pred_nth x k* = *Val d*) → (*Val d*:*DF_ord*) ≤ *x*.
Hint Immediate *DFle_Val_exists_pred Val_exists_pred_le.*

Lemma *Val_exists_pred_eq* :
∀ (*x*:*DF_ord*) *d*, (∃ *k*, *pred_nth x k* = *Val d*) → (*Val d*:*DF_ord*) == *x*.

3.4.3 Construction of least upper bounds

Definition *isEps* (*x*:*DF_ord*) := match *x* with *Eps* _ ⇒ *True* | _ ⇒ *False* end.

Lemma *isEps_Eps* : ∀ *x*:*DF_ord*, *isEps* (*Eps x*).

Lemma *not_isEpsVal* : ∀ *d*, ¬ (*isEps* (*Val d*)).
Hint *Resolve isEps_Eps not_isEpsVal.*

Lemma *isEps_dec* : ∀ (*x*:*DF_ord*) , {*d*:*D*|*x*=*Val d*}+{*isEps x*}.

Lemma *fVal* : ∀ (*c*:*natO -m> DF_ord*) (*n*:*nat*),
{*d*:*D* | ∃ *k*, *k*<*n* ∧ *c k* = *Val d*} + {∀ *k*, *k*<*n* → *isEps* (*c k*)}.

3.4.4 Flat lubs

Definition *cpred* (*c*:*natO -m> DF_ord*) : *natO-m>DF_ord*.

CoFixpoint *DF_lubn* (*c*:*natO-m> DF_ord*) (*n*:*nat*) : *DF_ord* :=
match *fVal c n* with *inleft* (*exist d _*) ⇒ *Val d*
| *inright _* ⇒ *Eps* (*DF_lubn* (*cpred c*) (*S n*))
end.

Lemma *DF_lubn_inv* : ∀ (*c*:*natO-m> DF_ord*) (*n*:*nat*), *DF_lubn c n* =
match *fVal c n* with *inleft* (*exist d _*) ⇒ *Val d*
| *inright _* ⇒ *Eps* (*DF_lubn* (*cpred c*) (*S n*))
end.

Lemma *chain_Val_eq* : ∀ (*c*:*natO-m> DF_ord*) (*n n'*:*nat*) *d d'*,
(*Val d* : *DF_ord*) ≤ *c n* → (*Val d'* : *DF_ord*) ≤ *c n'* → *d*=*d'*.

Lemma *pred_lubn_Val* : ∀ (*d*:*D*)(*n k p*:*nat*) (*c*:*natO-m> DF_ord*),
(*n* <*k*+*p*)%*nat* → *pred_nth* (*c n*) *k* = *Val d*
→ *pred_nth* (*DF_lubn c p*) *k* = *Val d*.

Lemma *pred_lubn_Val_inv* : ∀ (*d*:*D*)(*k p*:*nat*) (*c*:*natO-m> DF_ord*),
pred_nth (*DF_lubn c p*) *k* = *Val d*
→ ∃ *n*, (*n* <*k*+*p*)%*nat* ∧ *pred_nth* (*c n*) *k* = *Val d*.

Definition *DF_lub* (*c*:*natO-m> DF_ord*) := *DF_lubn c* 1.

Lemma *pred_lub_Val* : ∀ (*d*:*D*)(*n*:*nat*) (*c*:*natO-m> DF_ord*),
(*Val d*:*DF_ord*) ≤ (*c n*) → (*Val d*:*DF_ord*) ≤ *DF_lub c*.

Lemma *pred_lub_Val_inv* : ∀ (*d*:*D*)(*c*:*natO-m> DF_ord*),
(*Val d*:*DF_ord*) ≤ *DF_lub c* → ∃ *n*, (*Val d*:*DF_ord*) ≤ (*c n*).

Lemma *DF_lub_upper* : ∀ *c*:*natO-m> DF_ord*, ∀ *n*, *c n* ≤ *DF_lub c*.

Lemma *DF_lub_least* : ∀ (*c*:*natO-m> DF_ord*) *a*,
(∀ *n*, *c n* ≤ *a*) → *DF_lub c* ≤ *a*.

3.4.5 Declaration of the flat cpo

Definition *DF* : *cpo*.

End *Flat_cpo*.

3.5 Trivial cpo with only the bottom element

Inductive *DTriv* : Type := *DTbot* : *DTriv*.

Definition *DT_ord* : *ord*.

Definition *DT* : *cpo*.

Lemma *DT_eqBot* : ∀ *x* : *DT*, *x* = *DTbot*.

Require *List*.
Require Export *Cpo*.

4 Cpo_streams_type.v: Domain of possibly infinite streams on a type

CoInductive *DStr* (*D*:Type) : Type
:= *Eps* : *DStr D* → *DStr D* | *Con* : *D* → *DStr D* → *DStr D*.

Lemma *DS_inv* : ∀ (*D*:Type) (*d*:*DStr D*),
d = match *d* with *Eps x* ⇒ *Eps x* | *Con a s* ⇒ *Con a s* end.
Hint *Resolve DS_inv*.

- Extraction of a finite list from the n first constructors of a stream

```
Fixpoint DS_to_list (D:Type)(d:DStr D) (n : nat) {struct n}: List.list D :=
 match n with O ⇒ List.nil
          | S p ⇒ match d with Eps d' ⇒ DS_to_list d' p
                             | Con a d' ⇒ List.cons a (DS_to_list d' p)
                  end
 end.
```

4.1 Removing Eps steps

```
Definition pred (D:Type) d : DStr D := match d with Eps x ⇒ x | Con _ _ ⇒ d end.

Inductive isCon (D:Type) : DStr D → Prop :=
      isConEps : ∀ x, isCon x → isCon (Eps x)
    | isConCon : ∀ a s, isCon (Con a s).
Hint Constructors isCon.

Lemma isCon_pred : ∀ D (x:DStr D), isCon x → isCon (pred x).
Hint Resolve isCon_pred.

Definition isEps (D:Type) (x:DStr D) := match x with Eps _ ⇒ True | _ ⇒ False end.
```

Less general than isCon_pred but the result is a subterm of the argument (isCon x), used in uncons

```
Lemma isConEps_inv : ∀ D (x:DStr D), isCon x → isEps x → isCon (pred x).

Lemma isCon_intro : ∀ D (x:DStr D), isCon (pred x) → isCon x.
Hint Resolve isCon_intro.

Fixpoint pred_nth D (x:DStr D) (n:nat) {struct n} : DStr D :=
        match n with 0 ⇒ x
                   | S m ⇒ pred_nth (pred x) m
        end.

Lemma pred_nth_switch : ∀ D k (x:DStr D), pred_nth (pred x) k = pred (pred_nth x k).
Hint Resolve pred_nth_switch .

Lemma pred_nthS : ∀ D k (x:DStr D), pred_nth x (S k) = pred (pred_nth x k).
Hint Resolve pred_nthS.

Lemma pred_nthCon : ∀ D a (s:DStr D) n, pred_nth (Con a s) n = (Con a s).
Hint Resolve pred_nthCon.

Definition decomp D (a:D) (s x:DStr D) : Prop := ∃ k, pred_nth x k = Con a s.
Hint Unfold decomp.

Lemma decomp_isCon : ∀ D a (s x:DStr D), decomp a s x → isCon x.

Lemma decompCon : ∀ D a (s:DStr D), decomp a s (Con a s).
Hint Resolve decompCon.

Lemma decompCon_eq :
   ∀ D a b (s t:DStr D), decomp a s (Con b t) → Con a s = Con b t.
Hint Immediate decompCon_eq.

Lemma decompEps : ∀ D a (s x:DStr D), decomp a s x → decomp a s (Eps x).
Hint Resolve decompEps.

Lemma decompEps_pred : ∀ D a (s x:DStr D), decomp a s x → decomp a s (pred x).

Lemma decompEps_pred_sym : ∀ D a (s x:DStr D), decomp a s (pred x) → decomp a s x.

Hint Immediate decompEps_pred_sym decompEps_pred.

Lemma decomp_ind : ∀ D a (s:DStr D) (P : DStr D→ Prop),
  (∀ x, P x → decomp a s x → P (Eps x))
 → P (Con a s) → ∀ x, decomp a s x → P x.

Lemma DStr_match : ∀ D (x:DStr D), {a:D & {s:DStr D | x = Con a s}}+{isEps x}.

Lemma uncons : ∀ D (x:DStr D), isCon x ->{a:D & {s:DStr D| decomp a s x}}.
```

4.2 Definition of the order

CoInductive $DSle$ (D:Type) : $DStr\ D \to DStr\ D \to$ Prop :=
$DSleEps : \forall\ x\ y,\ DSle\ x\ y \to DSle\ (Eps\ x)\ y$
| $DSleCon : \forall\ a\ s\ t\ y,\ decomp\ a\ t\ y \to DSle\ s\ t \to DSle\ (Con\ a\ s)\ y.$

Hint *Constructors DSle.*

4.3 Properties of the order

Lemma $DSle\_pred\_eq : \forall\ D\ (x\ y{:}DStr\ D),\ \forall\ n,\ x{=}pred\_nth\ y\ n \to DSle\ x\ y.$

Lemma $DSle\_refl : \forall\ D\ (x{:}DStr\ D),\ DSle\ x\ x.$
Hint *Resolve DSle_refl.*

Lemma $DSle\_pred\_right : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ x\ y \to DSle\ x\ (pred\ y).$

Lemma $DSleEps\_right\_elim : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ x\ (Eps\ y) \to DSle\ x\ y.$

Lemma $DSle\_pred\_right\_elim : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ x\ (pred\ y) \to DSle\ x\ y.$

Lemma $DSle\_pred\_left : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ x\ y \to DSle\ (pred\ x)\ y.$
Hint *Resolve DSle_pred_left DSle_pred_right.*

Lemma $DSle\_pred : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ x\ y \to DSle\ (pred\ x)\ (pred\ y).$
Hint *Resolve DSle_pred.*

Lemma $DSle\_pred\_left\_elim : \forall\ D\ (x\ y{:}DStr\ D),\ DSle\ (pred\ x)\ y \to DSle\ x\ y.$

Lemma $DSle\_decomp : \forall\ D\ a\ (s\ x\ y{:}DStr\ D),$
$decomp\ a\ s\ x \to DSle\ x\ y \to \exists\ t,\ decomp\ a\ t\ y \land DSle\ s\ t.$

Lemma $DSle\_trans : \forall\ D\ (x\ y\ z{:}DStr\ D),\ DSle\ x\ y \to DSle\ y\ z \to DSle\ x\ z.$

4.3.1 Defintion of the ordered set

Definition DS_ord (D:Type) : $ord := mk\_ord\ (DSle\_refl\ (D{:=}D))\ (DSle\_trans\ (D{:=}D)).$

4.3.2 more Properties

Lemma $DSleEps\_right : \forall\ (D{:}\text{Type})\ (x\ y : DS\_ord\ D),\ x \le y \to x \le Eps\ y.$
Hint *Resolve DSleEps_right.*

Lemma $DSleEps\_left : \forall\ D\ (x\ y : DS\_ord\ D),\ x \le y \to (Eps\ x{:}DS\_ord\ D) \le y.$
Hint *Resolve DSleEps_left.*

Lemma $DSeq\_pred : \forall\ D\ (x{:}DS\_ord\ D),\ x == pred\ x.$
Hint *Resolve DSeq_pred.*

Lemma $pred\_nth\_eq : \forall\ D\ n\ (x{:}DS\_ord\ D),\ x == pred\_nth\ x\ n.$
Hint *Resolve pred_nth_eq.*

Lemma $DSleCon0$:
$\forall\ D\ a\ (s\ t{:}DS\_ord\ D),\ s \le t \to (Con\ a\ s{:}DS\_ord\ D) \le Con\ a\ t.$
Hint *Resolve DSleCon0.*

Lemma Con_compat :
$\forall\ D\ a\ (s\ t{:}DS\_ord\ D),\ s == t \to (Con\ a\ s{:}DS\_ord\ D) == Con\ a\ t.$
Hint *Resolve Con_compat.*

Lemma $DSleCon\_hd : \forall\ (D{:}\text{Type})\ a\ b\ (s\ t{:}DS\_ord\ D),$
$(Con\ a\ s{:}DS\_ord\ D) \le Con\ b\ t \to a = b.$

Lemma $Con\_hd\_simpl : \forall\ D\ a\ b\ (s\ t : DS\_ord\ D),\ (Con\ a\ s{:}DS\_ord\ D) == Con\ b\ t \to a = b.$

Lemma $DSleCon\_tl : \forall\ D\ a\ b\ (s\ t{:}DS\_ord\ D),\ (Con\ a\ s{:}DS\_ord\ D) \le Con\ b\ t \to (s{:}DS\_ord\ D) \le t.$

Lemma $Con\_tl\_simpl : \forall\ D\ a\ b\ (s\ t{:}DS\_ord\ D),\ (Con\ a\ s{:}DS\_ord\ D) == Con\ b\ t \to (s{:}DS\_ord\ D) == t.$

Lemma *eqEps* : ∀ D (x:DS_ord D), x == Eps x.
Hint Resolve eqEps.

Lemma *decomp_eqCon* : ∀ D a s (x:DS_ord D), decomp a s x → x == Con a s.
Hint Immediate decomp_eqCon.

Lemma *decomp_DSleCon* : ∀ D a s (x:DS_ord D), decomp a s x → x ≤ Con a s.

Lemma *decomp_DSleCon_sym* :
  ∀ D a s (x:DS_ord D), decomp a s x → (Con a s:DS_ord D)<=x.
Hint Immediate decomp_DSleCon decomp_DSleCon_sym.

Lemma *DSleCon_exists_decomp* :
    ∀ D (x:DS_ord D) a (s:DS_ord D), (Con a s:DS_ord D) ≤ x
        → ∃ b, ∃ t, decomp b t x ∧ a = b ∧ s ≤ t.

Lemma *Con_exists_decompDSle* :
    ∀ D (x:DS_ord D) a (s:DS_ord D),
    (∃ t, decomp a t x ∧ s ≤ t) → (Con a s:DS_ord D) ≤ x.
Hint Immediate DSleCon_exists_decomp Con_exists_decompDSle.

Lemma *DSle_isCon* : ∀ D a (s x : DS_ord D), (Con a s : DS_ord D) ≤ x → isCon x.

Lemma *DSle_uncons* :
  ∀ D (x:DS_ord D) a (s:DS_ord D), (Con a s:DS_ord D) ≤ x
        → { t : DS_ord D | decomp a t x ∧ s ≤ t}.

Lemma *DSle_rec* : ∀ D (R : DStr D → DStr D → Prop),
  (∀ x y, R (Eps x) y → R x y) →
  (∀ a s y, R (Con a s) y → ∃ t, decomp a t y ∧ R s t)
  → ∀ x y : DS_ord D, R x y → x ≤ y.

Lemma *isEps_Eps* : ∀ D (x:DS_ord D), isEps (Eps x).

Lemma *not_isEpsCon* : ∀ D a (s:DS_ord D), ¬ isEps (Con a s).
Hint Resolve isEps_Eps not_isEpsCon.

Lemma *isCon_le* : ∀ D (x y : DS_ord D), isCon x → x ≤ y → isCon y.

Lemma *decomp_eq* : ∀ D a (s x:DS_ord D),
    x == Con a s → ∃ t, decomp a t x ∧ s==t.

Lemma *DSle_rec_eq* : ∀ D (R : DStr D → DStr D → Prop),
  (∀ x1 x2 y1 y2:DS_ord D, R x1 y1 → x1==x2 → y1==y2 → R x2 y2) →
  (∀ a s (y:DS_ord D), R (Con a s) y → ∃ t, y==Con a t ∧ R s t)
  → ∀ x y : DS_ord D, R x y → x ≤ y.

Lemma *DSeq_rec* : ∀ D (R : DStr D → DStr D → Prop),
  (∀ x1 x2 y1 y2:DS_ord D, R x1 y1 → x1==x2 → y1==y2 → R x2 y2) →
  (∀ a s (y:DS_ord D), R (Con a s) y → ∃ t, y==Con a t ∧ R s t)->
  (∀ a s (x:DS_ord D), R x (Con a s) → ∃ t, x==Con a t ∧ R t s)
  → ∀ x y : DS_ord D, R x y → x == y.

4.4 Bottom is given by an infinite chain of Eps

CoFixpoint *DS_bot* (D:Type) : DS_ord D := Eps (DS_bot D).

Lemma *DS_bot_eq* (D:Type) : DS_bot D = Eps (DS_bot D).

Lemma *DS_bot_least* : ∀ D (x:DS_ord D), DS_bot D ≤ x.
Hint Resolve DS_bot_least.

4.5 Construction of least upper bounds

Lemma *chain_tl* : ∀ D (c:natO-m> DS_ord D), isCon (c O) → natO -m> DS_ord D.

Lemma *chain_uncons* :

```
  ∀ D (c:natO -m> DS_ord D), isCon (c O) →
     {hd:D & {ctl : natO -m> DS_ord D | ∀ n, c n == Con hd (ctl n)}}.
```

```
Lemma fCon : ∀ D (c:natO -m> DS_ord D) (n:nat),
             {hd: D &
             {tlc:natO -m> DS_ord D|
                    ∃ m, m < n ∧ ∀ k, c (k+m) == Con hd (tlc k)}}
             + {∀ k, k<n → isEps (c k)}.
```

4.6 Lubs on streams

```
Definition cpred D (c:natO -m> DS_ord D) : natO -m> DS_ord D.
```

```
CoFixpoint DS_lubn D (c:natO -m> DS_ord D) (n:nat) : DS_ord D :=
    match fCon c n with
      inleft (existS hd (exist tlc _)) ⇒ Con hd (DS_lubn tlc 1)
   | inright _ ⇒ Eps (DS_lubn (cpred c) (S n))
    end.
```

```
Definition DS_lub (D:Type) (c:natO -m> DS_ord D) := DS_lubn c 1.
```

```
Lemma DS_lubn_inv : ∀ D (c:natO -m> DS_ord D) (n:nat), DS_lubn c n =
     match fCon c n with
        inleft (existS hd (exist tlc _)) ⇒ Con hd (DS_lub tlc)
     | inright _ ⇒ Eps (DS_lubn (cpred c) (S n))
    end.
```

```
Lemma DS_lubn_pred_nth : ∀ D a (s:DS_ord D) n k p (c:natO -m> DS_ord D),
   (n<k+p)%nat → pred_nth (c n) k = Con a s →
   ∃ d:natO -m> DS_ord D,
                     DS_lubn c p == Con a (DS_lub d) ∧ (s:DS_ord D) ≤ d n.
```

```
Lemma DS_lubn_pred_nth_inv : ∀ D a (s:DS_ord D) k p (c:natO -m> DS_ord D),
   pred_nth (DS_lubn c p) k = Con a s →
   ∃ tlc :natO -m> DS_ord D, s= DS_lub tlc ∧ ∃ m, ∀ l, c (l+m) == Con a (tlc l).
```

```
Lemma DS_lubCon_inv : ∀ D a (s:DS_ord D) (c:natO -m> DS_ord D),
   (DS_lub c == Con a s) →
   ∃ tlc :natO -m> DS_ord D,
            s==DS_lub tlc ∧ ∃ m, ∀ l, c (l+m) == Con a (tlc l).
```

```
Lemma DS_lubCon : ∀ D a s n (c:natO -m> DS_ord D),
   (Con a s :DS_ord D) ≤ c n →
   ∃ d:natO -m> DS_ord D,
              DS_lub c == Con a (DS_lub d) ∧ (s:DS_ord D) ≤ d n.
```

```
Lemma DS_lub_upper : ∀ D (c:natO -m> DS_ord D), ∀ n, c n ≤ DS_lub c.
```

```
Lemma DS_lub_least : ∀ D (c:natO -m> DS_ord D) x,
                           (∀ n, c n ≤ x) → DS_lub c ≤ x.
```

4.7 Definition of the cpo of streams

```
Definition DS : Type → cpo.
```

```
Lemma DS_lub_inv : ∀ D (c:natO -m> DS D), lub c =
     match fCon c 1 with
        inleft (existS hd (exist tlc _)) ⇒ Con hd (lub (c:=DS D) tlc)
     | inright _ ⇒ Eps (DS_lubn (cpred c) 2)
    end.
```

```
Definition cons D (a : D) (s: DS D) : DS D := Con a s.
```

```
Lemma cons_le_compat :
     ∀ D a b (s t:DS D), a = b → s ≤ t → cons a s ≤ cons b t.
```

Hint *Resolve cons_le_compat.*

Lemma *cons_eq_compat* :
$\forall$ *D a b* (*s t*:*DS D*), *a* = *b* $\rightarrow$ *s* == *t* $\rightarrow$ *cons a s* == *cons b t*.
Hint *Resolve cons_eq_compat.*

Add Morphism cons with *signature eq* ==> *Oeq* ==> *Oeq* as *cons_eq_compat_morph.*

Lemma *not_le_consBot*: $\forall$ *D a* (*s*:*DS D*), $\neg$ *cons a s* $\leq$ 0.
Hint *Resolve not_le_consBot.*

Lemma *DSle_intro_cons* :
$\forall$ *D* (*x y*:*DS D*), ($\forall$ *a s*, *x*==*cons a s* $\rightarrow$ *cons a s* $\leq$ *y*) $\rightarrow$ *x* $\leq$ *y*.

Definition *is_cons D* (*x*:*DS D*) := *isCon x*.

Lemma *is_cons_intro* : $\forall$ *D* (*a*:*D*) (*s*:*DS D*), *is_cons* (*cons a s*).
Hint *Resolve is_cons_intro.*

Lemma *is_cons_elim* : $\forall$ *D* (*x*:*DS D*), *is_cons x* $\rightarrow$ $\exists$ *a*, $\exists$ *s* : *DS D*, *x* == *cons a s*.

Lemma *not_is_consBot* : $\forall$ *D*, $\neg$ *is_cons* (0:*DS D*).
Hint *Resolve not_is_consBot.*

Lemma *is_cons_le_compat* : $\forall$ *D* (*x y*:*DS D*), *x* $\leq$ *y* $\rightarrow$ *is_cons x* $\rightarrow$ *is_cons y*.

Lemma *is_cons_eq_compat* : $\forall$ *D* (*x y*:*DS D*), *x* == *y* $\rightarrow$ *is_cons x* $\rightarrow$ *is_cons y*.

Lemma *DSle_intro_is_cons* : $\forall$ *D* (*x y*:*DS D*), (*is_cons x* $\rightarrow$ *x* $\leq$ *y*) $\rightarrow$ *x* $\leq$ *y*.

Lemma *DSeq_intro_is_cons* : $\forall$ *D* (*x y*:*DS D*),
(*is_cons x* $\rightarrow$ *x* $\leq$ *y*) $\rightarrow$ (*is_cons y* $\rightarrow$ *y* $\leq$ *x*) $\rightarrow$ *x* == *y*.

Add Morphism is_cons with *signature Oeq* ==> *iff* as *is_cons_eq_iff.*

Add Morphism cons with *signature eq* ==> *Ole* ++> *Ole* as *cons_le_morph.*

Hint *Resolve cons_le_morph.*

4.8 Basic functions

Section *Simple_functions.*

4.8.1 Build a function F such that F (Con a s) = f a s and F (Eps x) = Eps (F x)

Variable *D D'*: Type.
Variable *f* : *D* $\rightarrow$ *DS D -m> DS D'*.

CoFixpoint *DScase* (*s*:*DS D*) : *DS D'*:=
match *s* with *Eps x* $\Rightarrow$ *Eps* (*DScase x*) | *Con a l* $\Rightarrow$ *f a l* end.

Lemma *DScase_inv* :
$\forall$ (*s*:*DS D*), *DScase s* = match *s* with *Eps l* $\Rightarrow$ *Eps* (*DScase l*) | *Con a l* $\Rightarrow$ *f a l* end.

Lemma *DScaseEps* : $\forall$ (*s*:*DS D*), *DScase* (*Eps s*) = *Eps* (*DScase s*).

Lemma *DScase_cons* : $\forall$ *a* (*s*:*DS D*), *DScase* (*cons a s*) = *f a s*.
Hint *Resolve DScaseEps DScase_cons.*

Lemma *DScase_decomp* : $\forall$ *a* (*s x*:*DS D*), *decomp a s x* $\rightarrow$ *DScase x* == *f a s*.

Lemma *DScase_eq_cons* : $\forall$ *a* (*s x*:*DS D*), *x* == *cons a s* $\rightarrow$ *DScase x* == *f a s*.
Hint *Resolve DScase_eq_cons.*

Lemma *DScase_bot* : *DScase* 0 $\leq$ 0.

Lemma *DScase_le_cons* : $\forall$ *a* (*s x*:*DS D*), *cons a s* $\leq$ *x* $\rightarrow$ *f a s* $\leq$ *DScase x*.

Lemma *DScase_le_compat* : $\forall$ (*s t*:*DS D*), *s* $\leq$ *t* $\rightarrow$ *DScase s* $\leq$ *DScase t*.
Hint *Resolve DScase_le_compat.*

Lemma *DScase_eq_compat* : ∀ (*s t*:*DS D*), *s* == *t* → *DScase s* == *DScase t*.
Hint *Resolve DScase_eq_compat*.

Add Morphism DScase with *signature Oeq* ==> *Oeq* as *DScase_eq_compat_morph*.

Definition *DSCase* : *DS D -m> DS D'*.

Lemma *DSCase_simpl* : ∀ (*s*:*DS D*), *DSCase s* = *DScase s*.

Lemma *DScase_decomp_elim* : ∀ *a* (*s*:*DS D'*) (*x*:*DS D*),
decomp a s (*DScase x*) → ∃ *b*, ∃ *t*, *x*==*cons b t* ∧ *f b t* == *Con a s*.

Lemma *DScase_eq_cons_elim* : ∀ *a* (*s* : *DS D'*) (*x*:*DS D*),
DScase x == *cons a s* → ∃ *b*, ∃ *t*, *x*==*cons b t* ∧ *f b t* == *cons a s*.

Lemma *DScase_is_cons* : ∀ (*x*:*DS D*), *is_cons* (*DScase x*) → *is_cons x*.

Lemma *is_cons_DScase* : (∀ *a* (*s*:*DS D*), *is_cons* (*f a s*)) → ∀ (*x*:*DS D*), *is_cons x* → *is_cons* (*DScase x*).

Hypothesis *fcont* : ∀ *c*, *continuous* (*f c*).

Lemma *DScase_cont* : *continuous DSCase*.
Hint *Resolve DScase_cont*.

Lemma *DScase_cont_eq* : ∀ (*c*:*natO-m>DS D*), *DScase* (*lub c*) == *lub* (*DSCase* @ *c*).

End *Simple_functions*.

Hint *Resolve DScaseEps DScase_cons DScase_le_compat DScase_eq_compat DScase_bot DScase_cont*.

Definition *DSCASE_mon* : ∀ *D D'*, (*D-O->*(*DS D -M→ DS D'*)) *-M→ DS D -M→ DS D'*.

Lemma *DSCASE_mon_simpl* : ∀ *D D' f s*, *DSCASE_mon D D' f s* = *DScase f s*.

Lemma *DSCASE_mon_cont* : ∀ *D D'*, *continuous* (*DSCASE_mon D D'*).

Definition *DSCASE_cont* : ∀ *D D'*, (*D-O->*(*DS D -C→ DS D'*)) *-m>* (*DS D -C→ DS D'*).

Lemma *DSCASE_cont_simpl* : ∀ *D D' f s*,
DSCASE_cont D D' f s = *DScase* (fun *a* ⇒ *fcontit* (*f a*)) *s*.

Definition *DSCASE* : ∀ *D D'*, (*D-O→DS D -C→DS D'*)*-c> DS D -C→DS D'*.

Lemma *DSCASE_simpl* : ∀ *D D' f s*, *DSCASE D D' f s* = *DScase* (fun *a* ⇒ *fcontit* (*f a*)) *s*.

4.9 Basic functions on streams

- Cons is continuous

Definition *Cons* (*D*:Type): *D -o>* (*DS D -m> DS D*).

Lemma *Cons_simpl* : ∀ *D* (*a* : *D*) (*s* : *DS D*), *Cons a s* = *cons a s*.

Lemma *Cons_cont* : ∀ *D* (*a* : *D*), *continuous* (*Cons a*).
Hint *Resolve Cons_cont*.

Definition *CONS D* (*a* : *D*) : *DS D -c> DS D* := *mk_fconti* (*Cons_cont a*).

Lemma *CONS_simpl* : ∀ *D* (*a* : *D*) (*s* : *DS D*), *CONS a s* = *cons a s*.

- first takes a stream and return the stream with only the first element f a s = cons a nil

Definition *firstf* (*D*:Type) : *D* → *DS D -m>DS D*:=
fun (*d*:*D*) ⇒ *fmon_cte* (*DS D*) (*O2*:=*DS D*) (*cons d* (0:*DS D*)).

Lemma *firstf_simpl* : ∀ *D* (*a*:*D*) (*s*:*DS D*), *firstf a s* = *cons a* (0:*DS D*).

Lemma *firstf_cont* : ∀ *D* (*a*:*D*) *c*, *firstf a* (*lub c*) ≤ *lub* (*firstf a* @ *c*).
Hint *Resolve firstf_cont*.

Definition *First* (*D*:Type) : *DS D -m> DS D* := *DSCase* (*firstf* (*D*:=*D*)).

Definition *first D* (*s*:*DS D*) := *First D s*.

Lemma *first_simpl* : ∀ *D* (*s*:*DS D*), *first s* = *DScase* (*firstf* (*D*:=*D*)) *s*.

Lemma *first_le_compat* : ∀ *D* (*s t*:*DS D*), *s* ≤ *t* → *first s* ≤ *first t*.
Hint *Resolve first_le_compat*.

Lemma *first_eq_compat* : ∀ *D* (*s t*:*DS D*), *s* == *t* → *first s* == *first t*.
Hint *Resolve first_eq_compat*.

Lemma *first_cons* : ∀ *D a* (*s*:*DS D*), *first* (*cons a s*) = *cons a* (0:*DS D*).

Lemma *first_bot* : ∀ *D*, *first* (*D*:=*D*) 0 ≤ 0.

Lemma *first_cons_elim* : ∀ *D a* (*s t*:*DS D*),
first t == *cons a s* → ∃ *u*, *t* == *cons a u* ∧ *s*==(0:*DS D*).

Add Morphism first with *signature Oeq* ==> *Oeq* as *first_eq_compat_morph*.

Add Morphism first with *signature Ole* ++> *Ole* as *first_le_compat_morph*.

Lemma *is_cons_first* : ∀ *D* (*s*:*DS D*), *is_cons s* → *is_cons* (*first s*).
Hint *Resolve is_cons_first*.

Lemma *first_is_cons* : ∀ *D* (*s*:*DS D*), *is_cons* (*first s*) → *is_cons s*.
Hint Immediate *first_is_cons*.

Lemma *first_cont* : ∀ *D*, *continuous* (*First D*).
Hint *Resolve first_cont*.

Definition *FIRST* (*D*:Type) : *DS D -c> DS D*.

Lemma *FIRST_simpl* : ∀ *D s*, *FIRST D s* = *first s*.

- rem returns the stream without the first element

Definition *remf D* (*d*: *D*) : *DS D -m> DS D* := *fmon_id* (*DS D*).

Lemma *remf_simpl* : ∀ *D* (*a*:*D*) *s*, *remf a s* = *s*.

Lemma *remf_cont* : ∀ *D* (*a*:*D*) *s*, *remf a* (*lub s*) ≤ *lub* (*remf a* @ *s*).
Hint *Resolve remf_cont*.

Definition *Rem D* : *DS D -m> DS D* := *DSCase* (*remf* (*D*:=*D*)).
Definition *rem D* (*s*:*DS D*) := *Rem D s*.

Lemma *rem_simpl* : ∀ *D* (*s*:*DS D*), *rem s* = *DScase* (*remf* (*D*:=*D*)) *s*.

Lemma *rem_cons* : ∀ *D* (*a*:*D*) *s*, *rem* (*cons a s*) = *s*.

Lemma *rem_bot* : ∀ *D*, *rem* (*D*:=*D*) 0 ≤ 0.

Lemma *rem_le_compat* : ∀ *D* (*s t*:*DS D*), *s*≤*t* → *rem s* ≤ *rem t*.
Hint *Resolve rem_le_compat*.

Lemma *rem_eq_compat* : ∀ *D* (*s t*:*DS D*), *s*==*t* → *rem s* == *rem t*.
Hint *Resolve rem_eq_compat*.

Add Morphism rem with *signature Oeq* ==> *Oeq* as *rem_eq_compat_morph*.

Add Morphism rem with *signature Ole* ++> *Ole* as *rem_le_compat_morph*.

Lemma *rem_is_cons* : ∀ *D* (*s*:*DS D*), *is_cons* (*rem s*) → *is_cons s*.
Hint Immediate *rem_is_cons*.

Lemma *rem_cont* : ∀ *D*, *continuous* (*Rem D*).
Hint *Resolve rem_cont*.

Definition *REM* (*D*:Type) : *DS D -c> DS D*.

Lemma *REM_simpl* : ∀ *D* (*s*:*DS D*), *REM D s* = *rem s*.

- app s t concatenates the first element of s to t

Definition appf D (t:DS D) (d: D) : DS D -m>DS D := fmon_cte (DS D) (Cons d t).

Lemma appf_simpl D (t:DS D) : ∀ a s, appf t a s = cons a t.

Definition Appf : ∀ D, DS D -m> D -o> (DS D -m> DS D).

Lemma Appf_simpl : ∀ D t, Appf D t = appf t.

Lemma appf_cont D (t:DS D) : ∀ a c, appf t a (lub c) ≤ lub (appf t a @ c).
Hint Resolve appf_cont.

Lemma appf_cont_par : ∀ D, continuous (D2:=D -O→ (DS D -M→DS D)) (Appf D).
Hint Resolve appf_cont_par.

Definition AppI : ∀ D, DS D -m> DS D -m> DS D.

Lemma AppI_simpl : ∀ D s t, AppI D t s = DScase (appf t) s.

Definition App (D:Type) := fmon_shift (AppI D).

Lemma App_simpl : ∀ D s t, App D s t = DScase (appf t) s.

Definition app D s t := App D s t.

Lemma app_simpl : ∀ D (s t:DS D), app s t = DScase (appf t) s.

Lemma app_cons : ∀ D a (s t:DS D), app (cons a s) t = cons a t.

Lemma app_bot : ∀ D (s:DS D), app 0 s ≤ 0.

Lemma app_mon_left : ∀ D (s t u : DS D), s ≤ t → app s u ≤ app t u.

Lemma app_cons_elim : ∀ D a (s t u:DS D), app t u == cons a s →
∃ t', t == cons a t' ∧ s == u.

Lemma app_mon_right : ∀ D (s t u : DS D), t ≤ u → app s t ≤ app s u.

Hint Resolve first_cons first_bot app_cons app_bot
app_mon_left app_mon_right rem_cons rem_bot.

Lemma app_le_compat : ∀ D (s t u v:DS D), s ≤ t → u ≤ v → app s u ≤ app t v.
Hint Immediate app_le_compat.

Lemma app_eq_compat : ∀ D (s t u v:DS D), s == t → u == v → app s u == app t v.
Hint Immediate app_eq_compat.

Add Morphism app with signature Oeq ==> Oeq ==> Oeq as app_eq_compat_morph.

Add Morphism app with signature Ole ++> Ole ++> Ole as app_le_compat_morph.

Lemma is_cons_app : ∀ D (x y : DS D), is_cons x → is_cons (app x y).
Hint Resolve is_cons_app.

Lemma app_is_cons : ∀ D (x y : DS D), is_cons (app x y) → is_cons x.

Lemma app_cont : ∀ D, continuous2 (App D).
Hint Resolve app_cont.

Definition APP (D:Type) : DS D -c> DS D -C→ DS D := continuous2_cont (app_cont (D:=D)).

Lemma APP_simpl : ∀ D (s t : DS D), APP D s t = app s t.

4.9.1 Basic equalities

Lemma first_eq_bot : ∀ D, first (D:=D) 0 == 0.

Lemma rem_eq_bot : ∀ D, rem (D:=D) 0 == 0.

Lemma app_eq_bot : ∀ D (s:DS D), app 0 s == 0.
Hint Resolve first_eq_bot rem_eq_bot app_eq_bot.

Lemma DSle_app_bot_right_first : ∀ D (s:DS D), app s 0 ≤ first s.

Lemma *DSle_first_app_bot_right* : ∀ *D* (*s*:*DS D*), *first s* ≤ *app s* 0.

Lemma *app_bot_right_first* : ∀ *D* (*s*:*DS D*), *app s* 0 == *first s*.

Lemma *DSle_first_app_first* : ∀ *D* (*x y*:*DS D*), *first* (*app x y*) ≤ *first x*.

Lemma *DSle_first_first_app* : ∀ *D* (*x y*:*DS D*), *first x* ≤ *first* (*app x y*).

Lemma *first_app_first* : ∀ *D* (*x y*:*DS D*), *first* (*app x y*)==*first x*.

Hint *Resolve app_bot_right_first first_app_first*.

Lemma *DSle_app_first_rem* : ∀ *D* (*x*:*DS D*), *app* (*first x*) (*rem x*) ≤ *x*.

Lemma *DSle_app_first_rem_sym* : ∀ *D* (*x*:*DS D*), *x* ≤ *app* (*first x*) (*rem x*).

Lemma *app_first_rem* : ∀ *D* (*x*:*DS D*), *app* (*first x*) (*rem x*) == *x*.
Hint *Resolve app_first_rem*.

Lemma *rem_app* : ∀ *D* (*x y*:*DS D*), *is_cons x* → *rem* (*app x y*) == *y*.
Hint *Resolve rem_app*.

Lemma *rem_app_le* : ∀ *D* (*x y*:*DS D*), *rem* (*app x y*) ≤ *y*.
Hint *Resolve rem_app_le*.

Lemma *is_cons_rem_app* : ∀ *D* (*x y* : *DS D*), *is_cons x* → *is_cons y* → *is_cons* (*rem* (*app x y*)).
Hint *Resolve is_cons_rem_app*.

Lemma *rem_app_is_cons* : ∀ *D* (*x y* : *DS D*), *is_cons* (*rem* (*app x y*)) → *is_cons y*.

Lemma *first_first_eq* : ∀ *D* (*s*:*DS D*), *first* (*first s*) == *first s*.
Hint *Resolve first_first_eq*.

Lemma *app_app_first* : ∀ *D* (*s t* : *DS D*), *app* (*first s*) *t* == *app s t*.

4.10 Proof by co-recursion

Lemma *DS_bisimulation* : ∀ *D* (*R*: *DS D* → *DS D* → Prop),
(∀ *x1 x2 y1 y2*, *R x1 y1* → *x1*==*x2* → *y1*==*y2* → *R x2 y2*)
→ (∀ (*x y*:*DS D*), (*is_cons x* ∨ *is_cons y*) → *R x y* → *first x* == *first y*)
→ (∀ (*x y*:*DS D*), (*is_cons x* ∨ *is_cons y*) → *R x y* → *R* (*rem x*) (*rem y*))
→ ∀ *x y*, *R x y* → *x* == *y*.

Lemma *DS_bisimulation2* : ∀ *D* (*R*: *DS D* → *DS D* → Prop),
(∀ *x1 x2 y1 y2*, *R x1 y1* → *x1*==*x2* → *y1*==*y2* → *R x2 y2*)
→ (∀ (*x y*:*DS D*), (*is_cons x* ∨ *is_cons y*) → *R x y* → *first x* == *first y*)
→ (∀ (*x y*:*DS D*), (*is_cons* (*rem x*) ∨ *is_cons* (*rem y*)) → *R x y* → *first* (*rem x*) == *first* (*rem y*))
→ (∀ (*x y*:*DS D*), (*is_cons* (*rem x*) ∨ *is_cons* (*rem y*)) → *R x y* → *R* (*rem* (*rem x*)) (*rem* (*rem y*)))
→ ∀ *x y*, *R x y* → *x* == *y*.

4.11 Finiteness of streams

CoInductive *infinite* (*D*:Type) (*s*: *DS D*) : Prop :=
inf_intro : *is_cons s* → *infinite* (*rem s*) → *infinite s*.

Lemma *infinite_le_compat* : ∀ *D* (*s t*:*DS D*), *s* ≤ *t* → *infinite s* → *infinite t*.

Add Morphism infinite with *signature Oeq* ==> *iff* as *infinite_morph*.

Lemma *not_infiniteBot* : ∀ *D*, ¬ *infinite* (0:*DS D*).
Hint *Resolve not_infiniteBot*.

Inductive *finite* (*D*:Type) (*s*:*DS D*) : Prop :=
fin_bot : *s* ≤ 0 → *finite s* | *fin_cons* : *finite* (*rem s*) → *finite s*.

Lemma *finite_mon* : ∀ *D* (*s t*:*DS D*), *s* ≤ *t* → *finite t* → *finite s*.

Add Morphism finite with *signature Oeq* ==> *iff* as *finite_morph*.

Lemma *not_finite_infinite* : ∀ *D* (*s*:*DS D*), *finite s* → ¬ *infinite s*.

4.12 Mapping a function on a stream

Section *MapStream*.
Variable *D D'*: Type.
Variable *F* : *D* → *D'*.

Definition *mapf* : (*DS D* -*C*→ *DS D'*) -*m*> *D*-*O*→ *DS D* -*C*→*DS D'*.

Lemma *mapf_simpl* : ∀ *f*, *mapf f* = fun *a* ⇒ *CONS* (*F a*) @_ *f*.

Definition *Mapf* : (*DS D* -*C*→ *DS D'*) -*c*> *D*-*O*→ *DS D* -*C*→*DS D'*.

Lemma *Mapf_simpl* : ∀ *f*, *Mapf f* = fun *a* ⇒ *CONS* (*F a*) @_ *f*.

Definition *MAP* : *DS D* -*C*→ *DS D'* := *FIXP* (*DS D*-*C*→*DS D'*) (*DSCASE D D'* @_ *Mapf*).

Lemma *MAP_eq* : *MAP* == *DSCASE D D'* (*Mapf MAP*).

Definition *map* (*s*: *DS D*) := *MAP s*.

Lemma *map_eq* : ∀ *s*:*DS D*, *map s* == *DScase* (fun *a* ⇒ *Cons* (*F a*) @ (*fcontit MAP*)) *s*.

Lemma *map_bot* : *map* 0 == 0.

Lemma *map_eq_cons* : ∀ *a s*,
map (*cons a s*) == *cons* (*F a*) (*map s*).

Lemma *map_le_compat* : ∀ *s t*, *s* ≤ *t* → *map s* ≤ *map t*.

Lemma *map_eq_compat* : ∀ *s t*, *s* == *t* → *map s* == *map t*.

Add Morphism map with *signature Oeq* ==> *Oeq* as *map_eq_compat_morph_local*.

Lemma *is_cons_map* : ∀ (*s*:*DS D*), *is_cons s* → *is_cons* (*map s*).
Hint *Resolve is_cons_map*.

Lemma *map_is_cons* : ∀ *s*, *is_cons* (*map s*) → *is_cons s*.
Hint Immediate *map_is_cons*.

End *MapStream*.
Hint *Resolve map_bot map_eq_cons map_le_compat map_eq_compat is_cons_map*.

Add Morphism map with *signature eq* ==> *Oeq* ==> *Oeq* as *map_eq_compat_morph*.

4.13 Filtering a stream

Section *FilterStream*.
Variable *D* : Type.
Variable *P* : *D* → Prop.
Variable *Pdec* : ∀ *x*, {*P x*}+{˜ *P x*}.

Definition *filterf* : (*DS D* -*C*→ *DS D*) -*m*> *D*-*O*→ *DS D* -*C*→*DS D*.

Lemma *filterf_simpl* : ∀ *f*, *filterf f* = fun *a* ⇒ if *Pdec a* then *CONS a* @_ *f* else *f*.

Definition *Filterf* : (*DS D* -*C*→ *DS D*) -*c*> *D*-*O*→ *DS D* -*C*→*DS D*.

Lemma *Filterf_simpl* : ∀ *f*, *Filterf f* = fun *a* ⇒ if *Pdec a* then *CONS a* @_ *f* else *f*.

Definition *FILTER* : *DS D* -*C*→ *DS D* := *FIXP* (*DS D*-*C*→*DS D*) (*DSCASE D D* @_ *Filterf*).

Lemma *FILTER_eq* : *FILTER* == *DSCASE D D* (*Filterf FILTER*).

Definition *filter* (*s*: *DS D*) := *FILTER s*.

Lemma *filter_bot* : *filter* 0 == 0.

Lemma *filter_eq_cons* : ∀ *a s*,
filter (*cons a s*) == if *Pdec a* then *cons a* (*filter s*) else *filter s*.

Lemma *filter_le_compat* : ∀ *s t*, *s* ≤ *t* → *filter s* ≤ *filter t*.

Lemma *filter_eq_compat* : ∀ *s t*, *s* == *t* → *filter s* == *filter t*.

End *FilterStream*.
Hint *Resolve filter_bot filter_eq_cons filter_le_compat filter_eq_compat*.

Require Export *Cpo_streams_type*.

5 Cpo_nat.v: Domains of natural numbers

5.1 Definition

- Natural numbers are a particular case of streams over the trivial type unit

Lemma $unit\_eq : \forall\ x : unit,\ x = tt.$
Hint $Resolve\ unit\_eq.$

Definition $DN : cpo := DS\ unit.$

Definition $DNS\ (n : DN) : DN := cons\ tt\ n.$

5.2 Embedding of usual natural numbers

Fixpoint $nat2DN\ (n{:}nat) : DN :=$ match n with $0 \Rightarrow 0 \mid S\ p \Rightarrow DNS\ (nat2DN\ p)$ end.

5.3 Infinite element

CoFixpoint $DNinf : DN := DNS\ DNinf.$

Lemma $DNinf\_inv : DNinf = DNS\ DNinf.$

Lemma $DNleinf : \forall\ n{:}DN,\ n \leq DNinf.$

Hint $Resolve\ DNleinf.$

5.4 Properties of basic operators

Lemma $DNEps\_left : \forall\ x\ y{:}DN,\ x == y \rightarrow (Eps\ x : DN) == y.$

Lemma $DNEps\_right : \forall\ x\ y{:}DN,\ x == y \rightarrow x == Eps\ y.$

Hint Immediate $DNEps\_left\ DNEps\_right.$

Lemma $DNS\_le\_compat : \forall\ (x\ y{:}DN)\ ,\ x \leq y \rightarrow DNS\ x \leq DNS\ y.$
Hint $Resolve\ DNS\_le\_compat.$

Lemma $DNS\_eq\_compat : \forall\ (x\ y{:}DN)\ ,\ x{==}y \rightarrow DNS\ x == DNS\ y.$
Hint $Resolve\ DNS\_eq\_compat.$

Add Morphism DNS with *signature Oeq ==> Oeq* as *DNS_eq_compat_morph.*

Lemma $DNS\_le\_simpl : \forall\ x\ y : DN,\ DNS\ x \leq DNS\ y \rightarrow x \leq y.$
Hint Immediate $DNS\_le\_simpl.$

Lemma $DNS\_eq\_simpl : \forall\ x\ y : DN,\ DNS\ x == DNS\ y \rightarrow x == y.$
Hint Immediate $DNS\_eq\_simpl.$

5.5 Simulation principles

Lemma $DNle\_rec : \forall\ R : DN \rightarrow DN \rightarrow$ Prop,
$(\forall\ x1\ x2\ y1\ y2{:}DN,\ R\ x1\ y1 \rightarrow x1{==}x2 \rightarrow y1{==}y2 \rightarrow R\ x2\ y2) \rightarrow$
$(\forall\ s\ (y{:}DN),\ R\ (DNS\ s)\ y \rightarrow \exists\ t,\ y{==}\ DNS\ t \wedge R\ s\ t)$
$\rightarrow \forall\ x\ y : DN\ ,\ R\ x\ y \rightarrow x \leq y.$

Lemma $DNeq\_rec : \forall\ R : DN \rightarrow DN \rightarrow$ Prop,
$(\forall\ x1\ x2\ y1\ y2{:}DN,\ R\ x1\ y1 \rightarrow x1{==}x2 \rightarrow y1{==}y2 \rightarrow R\ x2\ y2) \rightarrow$
$(\forall\ s\ (y{:}DN),\ R\ (DNS\ s)\ y \rightarrow \exists\ t,\ y{==}DNS\ t \wedge R\ s\ t)$->
$(\forall\ s\ (x{:}DN),\ R\ x\ (DNS\ s) \rightarrow \exists\ t,\ x{==}DNS\ t \wedge R\ t\ s)$
$\rightarrow \forall\ x\ y : DN\ ,\ R\ x\ y \rightarrow x == y.$

5.6 More properties on basic functions

Lemma *DNle_n_Sn* : ∀ *x*, *x* ≤ *DNS x*.
Hint *Resolve DNle_n_Sn*.

Lemma *DNinf_le_intro* : ∀ *x*, *DNS x* ≤ *x* → *DNinf* ≤ *x*.
Hint Immediate *DNinf_le_intro*.

Lemma *is_cons_S* : ∀ *n*, *is_cons n* → *n* == *DNS* (*rem n*).

Lemma *infinite_S* : ∀ *n* : *DN*, *DNS n* ≤ *n* → *infinite n*.

5.7 Addition

CoFixpoint *add* (*n m* : *DN*) : *DN* :=
    match *n* with (*Eps n'*) ⇒ *Eps* (*add m n'*)
        | (*Con _ n'*) ⇒ *DNS* (*add m n'*)
    end.

Lemma *add_inv* : ∀ (*n m* : *DN*),
  add n m = match *n* with (*Eps n'*) ⇒ *Eps* (*add m n'*)
        | (*Con _ n'*) ⇒ *DNS* (*add m n'*)
  end.

Lemma *add_decomp_elim* : ∀ *a* (*s x y*:*DN*), *decomp a s* (*add x y*) →
    ∃ *t*, ∃ *u*, *s* = *add t u* ∧
      (*x*==*DNS u* ∧ *y*==*t* ∨ *x*==*t* ∧ *y* == *DNS u*).

Lemma *add_eqCons* : ∀ (*s x y*:*DN*), *add x y* == *DNS s* →
    ∃ *t*, ∃ *u*, *s* == *add t u* ∧
      (*x*==*DNS u* ∧ *y*==*t* ∨ *x* == *t* ∧ *y* == *DNS u*).

Lemma *addS* :
    ∀ *x' x*, *DNS x'* ≤ *x*
  → ∀ *y*, (∃ *s*, *add x y* == *DNS s*) ∧ (∃ *s*, *add y x* == *DNS s*).

Lemma *addS_sym* :
  ∀ *x y v*:*DN*, *add x y* == *DNS v* → ∃ *t'*, *add y x* == *DNS t'*.

DNSn x n = S^n x

Fixpoint *DNSn* (*x*:*DN*) (*n*:*nat*) {*struct n*} :*DN* :=
  match *n* with 0 ⇒ *x* | *S p* ⇒ *DNSn* (*DNS x*) *p* end.

Lemma *DNSnS* : ∀ *n x*, *DNSn* (*DNS x*) *n* = *DNS* (*DNSn x n*).
Hint *Resolve DNSnS*.

Lemma *DNSn_mon* : ∀ (*n*:*nat*) (*x y*:*DN*), *x*≤*y* → *DNSn x n* ≤ *DNSn y n*.
Hint *Resolve DNSn_mon*.

Lemma *DNSn_eq_compat* : ∀ (*n*:*nat*) (*x y*:*DN*), *x*==*y* → *DNSn x n* == *DNSn y n*.
Hint *Resolve DNSn_eq_compat*.

Condition S^l x <= z & y <= S^l t ensuring x+y <= z+t
Definition *compat* (*x y z t*:*DN*) := ∃ *l*:*nat*, (*DNSn x l* ≤ *z* ∧ *y* ≤ *DNSn t l*).

Lemma *compatS* :
  ∀ *x y z t x' y' z' t'*,
  compat x y z t →
    (*x*==*DNS y'* ∧ *y*==*x'* ∨ *x*==*x'* ∧ *y*==*DNS y'*)
  → (*z*==*DNS t'* ∧ *t*==*z'* ∨ *z*==*z'* ∧ *t*==*DNS t'*)
  → (*compat x' y' z' t'* ∨ *compat x' y' t' z'* ∨ *compat y' x' z' t'* ∨ *compat y' x' t' z'*).

Lemma *compat_addS* : ∀ *n m p q v*:*DN*,
   (*compat n m p q*) → *add n m* == *DNS v* → ∃ *t*, *add p q* == *DNS t*.

Lemma *add_compat* :
  ∀ *n m p q*,

$(compat\ n\ m\ p\ q \lor compat\ n\ m\ q\ p \lor compat\ m\ n\ p\ q \lor compat\ m\ n\ q\ p)$
$\rightarrow add\ n\ m \leq add\ p\ q.$

Lemma *add_mon* : ∀ *n p m q*, *n*≤*p* → *m* ≤ *q* → *add n m* ≤ *add p q*.

Hint *Resolve add_mon*.

Definition *Add* : *DN -m> DN -m> DN* := *le_compat2_mon add_mon*.

Lemma *Add_simpl* : ∀ *x y*, *Add x y* = *add x y*.

Add Morphism add with *signature Oeq ==> Oeq ==> Oeq* as *add_eq_compat*.

Lemma *add_le_sym* : ∀ *n m*, *add n m* ≤ *add m n*.
Hint *Resolve add_le_sym*.

Lemma *add_sym* : ∀ *n m*, *add n m* == *add m n*.

Lemma *addS_shift* : ∀ *n m*, *add* (*DNS n*) *m* == *add n* (*DNS m*).
Hint *Resolve addS_shift*.

Lemma *addS_simpl* : ∀ *n m*, *add* (*DNS n*) *m* == *DNS* (*add n m*).
Hint *Resolve addS_simpl*.

Lemma *not_le_S_0* : ∀ *n*, ¬ (*DNS n* ≤ 0).
Hint *Resolve not_le_S_0*.

Lemma *add_n_0* : ∀ *n*, *add* 0 *n* == *n*.

Lemma *add_continuous_right* : ∀ *b*, *continuous* (*Add b*).

Lemma *add_continuous2* : *continuous2 Add*.

Lemma *add_continuous* : *continuous* (*D2*:=*DN-M*→*DN*) *Add*.

5.8 Length of a stream

Definition *LENGTH* (*D*:Type) : *DS D -c> DN* := *MAP* (fun *x*:*D*⇒*tt*).

Definition *length* (*D*:Type) (*s*:*DS D*) : *DN* := *LENGTH D s*.

Lemma *length_simpl* : ∀ (*D*:Type) (*s*:*DS D*), *length s* = *map* (fun *x*:*D*⇒*tt*) *s*.

Lemma *length_eq_cons* : ∀ *D a* (*s*:*DS D*), *length* (*cons a s*) == *DNS* (*length s*).

Lemma *length_nil* : ∀ *D*, *length* (0:*DS D*) == 0.

Hint *Resolve length_eq_cons length_nil*.

Lemma *length_le_compat* : ∀ *D* (*s t* : *DS D*), *s* ≤ *t* → *length s* ≤ *length t*.
Hint *Resolve length_le_compat*.

Lemma *length_eq_compat* : ∀ *D* (*s t* : *DS D*), *s* == *t* → *length s* == *length t*.
Hint *Resolve length_eq_compat*.

Add Morphism length with *signature Oeq ==> Oeq* as *length_morph*.

Lemma *is_cons_length* : ∀ *D* (*s*:*DS D*), *is_cons s* → *is_cons* (*length s*).
Hint *Resolve is_cons_length*.

Lemma *length_is_cons* : ∀ *D* (*s*:*DS D*), *is_cons* (*length s*) → *is_cons s*.
Hint Immediate *length_is_cons*.

Lemma *length_rem* : ∀ *D* (*s*:*DS D*), *length* (*rem s*) == *rem* (*length s*).
Hint *Resolve length_rem*.

Lemma *infinite_length* : ∀ *D* (*s*:*DS D*), *infinite s* → *infinite* (*length s*).
Hint *Resolve infinite_length*.

Lemma *length_infinite* : ∀ *D* (*s*:*DS D*), *infinite* (*length s*) → *infinite s*.
Hint Immediate *length_infinite*.

6 System.v: Formalisation of Kahn networks

Require Export *Cpo_streams_type*.

6.1 Definition of nodes

Definition of a multiple node :

- index for inputs with associated types
- index for outputs with associated types
- continuous function on corresponding streams

Definition *DS_fam* (*I*:Type)(*SI*:*I* → Type) (*i*:*I*) := *DS* (*SI* *i*).

Definition *DS_prod* (*I*:Type)(*SI*:*I* → Type) := *Dprodi* (*DS_fam* *SI*).

- A node is a continuous function from inpits to outputs

Definition *node_fun* (*I* *O* : Type) (*SI* : *I* → Type) (*SO* : *O* → Type) :*cpo*
  := *DS_prod SI -C→ DS_prod SO*.

- node with a single output

Definition *snode_fun* (*I* : Type) (*SI* : *I* → Type) (*SO* : Type) : *cpo* := *DS_prod SI -C→ DS SO*.

6.2 Definition of a system

- Each link is either an input link or is associated to the output of a simple node, each input of that node is associated to a link with the apropriate type

Definition *inlSL* (*LI LO*:Type) (*SL*:(*LI*+*LO*)->Type) (*i*:*LI*) := *SL* (*inl LO i*).
Definition *inrSL* (*LI LO*:Type) (*SL*:(*LI*+*LO*)->Type) (*o*:*LO*) := *SL* (*inr LI o*).

A system associates a continuous functions to a set of typed output links

Definition *system* (*LI LO*:Type) (*SL*:*LI*+*LO*→Type)
  := *Dprodi* (fun (*o*:*LO*) ⇒ *DS_prod SL -C→ DS* (*inrSL SL o*)).

6.3 Semantics of a system

Each system defines a new node with inputs for the inputs of the system

6.3.1 Definition of the equations

Equations are a continuous functional on links

Definition *eqn_of_system* : ∀ (*LI LO*:Type) (*SL*:*LI*+*LO*→Type),
  system SL → *DS_prod* (*inlSL SL*) → *DS_prod SL -m> DS_prod SL*.

Lemma *eqn_of_system_simpl* : ∀ (*LI LO*:Type) (*SL*:*LI*+*LO*→Type)(*s*:*system SL*)
  (*init* : *DS_prod* (*inlSL SL*)) (*X*:*DS_prod SL*),
  eqn_of_system s init X =
    fun *l* : *LI*+*LO* ⇒
    match *l* return (*DS* (*SL l*)) with
      inl i ⇒ *init i*
    | *inr o* ⇒ *s o X*

end.

Lemma *eqn_of_system_cont* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)(*s*:*system SL*)
(*init* : *DS_prod* (*inlSL SL*)), *continuous* (*eqn_of_system s init*).
Hint *Resolve eqn_of_system_cont*.

Definition *EQN_of_system* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type),
system SL $\rightarrow$ *DS_prod* (*inlSL SL*) $\rightarrow$ *DS_prod SL* -*c*> *DS_prod SL*.

Lemma *EQN_of_system_simpl* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)(*s*:*system SL*)
(*init* : *DS_prod* (*inlSL SL*)) (*X*:*DS_prod SL*),
EQN_of_system s init X = *eqn_of_system s init X*.

6.3.2 Properties of the equations

The equations are monotonic with respect to the inputs and the system

Lemma *EQN_of_system_mon* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)
(*s1 s2* : *system SL*) (*init1 init2* : *DS_prod* (*inlSL SL*)),
s1 $\le$ *s2* $\rightarrow$ *init1* $\le$ *init2* $\rightarrow$ *EQN_of_system s1 init1* $\le$ *EQN_of_system s2 init2*.

Definition *Eqn_of_system* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type),
(*system SL*) -*m*> *DS_prod* (*inlSL SL*) -*M*$\rightarrow$ *DS_prod SL* -*C*$\rightarrow$ *DS_prod SL*.

Lemma *Eqn_of_system_simpl* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)(*s*:*system SL*)
(*init*:*DS_prod* (*inlSL SL*)), *Eqn_of_system SL s init* = *EQN_of_system s init*.

The equations are continuous with respect to the inputs

Lemma *Eqn_of_system_cont* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type),
continuous2 (*Eqn_of_system SL*).
Hint *Resolve Eqn_of_system_cont*.

Lemma *Eqn_of_system_cont2* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)(*s*:*system SL*),
continuous (*Eqn_of_system SL s*).

Lemma *Eqn_of_system_cont1* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type),
continuous (*Eqn_of_system SL*).

Definition *EQN_of_SYSTEM* (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)
: *system SL* -*c*> *DS_prod* (*inlSL SL*) -*C*$\rightarrow$ *DS_prod SL* -*C*$\rightarrow$ *DS_prod SL*
:= *continuous2_cont* (*Eqn_of_system_cont* (*SL*:=*SL*)).

6.3.3 Solution of the equations

The solution is defined as the smallest fixpoint of the equations it is a monotonic function of the inputs

Definition *sol_of_system* (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)
: *system SL* -*c*> *DS_prod* (*inlSL SL*) -*C*$\rightarrow$ *DS_prod SL* := *FIXP* (*DS_prod SL*) @@_ *EQN_of_SYSTEM SL*.

Lemma *sol_of_system_simpl* :
$\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type) (*s*:*system SL*) (*init*:*DS_prod* (*inlSL SL*)),
sol_of_system SL s init = *FIXP* (*DS_prod SL*) (*EQN_of_system s init*).

Lemma *sol_of_system_eq* : $\forall$ (*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type) (*s*:*system SL*) (*init*:*DS_prod* (*inlSL SL*)),
sol_of_system SL s init == *eqn_of_system s init* (*sol_of_system SL s init*).

6.3.4 New node from the system

We can choose an arbitrary set of outputs
Definition *node_of_system* (*O*:Type)(*LI LO*:Type) (*SL*:*LI*+*LO*$\rightarrow$Type)(*indO* : *O* $\rightarrow$ *LO*) :
system SL -*C*$\rightarrow$ *node_fun* (fun *i* : *LI* $\Rightarrow$ *SL* (*inl LO i*)) (fun *o* : *O* $\Rightarrow$ *SL* (*inr LI* (*indO o*)))
:= *DLIFTi* (*DS_fam SL*) (fun *o* $\Rightarrow$ *inr LI* (*indO o*)) @@_ (*sol_of_system SL*).

Require Export *Systems*.
Require Export *Cpo_nat*.

7 Example.v: example from Kahn's IFIT 74 paper

7.1 Definitions of nodes

Definition *D* := *nat*.

- f U V = app U (app V (f (rem U) (rem V)))

Definition *Funf* : (*DS D* -*C*→ *DS D* -*C*→ *DS D*) -*c*> (*DS D*) -*C*→ (*DS D*) -*C*→ *DS D*.

Lemma *Funf_eq* : ∀ (*f* : *DS D* -*C*→ *DS D* -*C*→ *DS D*) (*U V* : *DS D*),
Funf f U V == *app U* (*app V* (*f* (*rem U*) (*rem V*))).

Definition *f* : *DS D* -*C*→ *DS D* -*C*→ *DS D* := *FIXP* _ *Funf*.

Lemma *f_eq* : ∀ (*p1 p2*:*DS D*),
f p1 p2 == *app p1* (*app p2* (*f* (*rem p1*) (*rem p2*))).
Hint *Resolve f_eq*.

Lemma *first_f_eq* : ∀ (*p1 p2*:*DS D*), *first* (*f p1 p2*) == *first p1*.
Hint *Resolve first_f_eq*.

Lemma *rem_f_eq* : ∀ (*p1 p2*:*DS D*),
is_cons p1 → *rem* (*f p1 p2*) == *app p2* (*f* (*rem p1*) (*rem p2*)).
Hint *Resolve rem_f_eq*.

Lemma *is_cons_f*: ∀ (*p1 p2*:*DS D*), *is_cons p1* → *is_cons* (*f p1 p2*).

Lemma *is_cons_rem_f*: ∀ (*p1 p2*:*DS D*), *is_cons p1* → *is_cons p2* → *is_cons* (*rem* (*f p1 p2*)).

Lemma *f_is_cons* : ∀ (*p1 p2*:*DS D*), *is_cons* (*f p1 p2*) → *is_cons p1*.

Lemma *rem_f_is_cons* : ∀ (*p1 p2*:*DS D*), *is_cons* (*rem* (*f p1 p2*)) → *is_cons p2*.

- g1 U = app U g1 (rem (rem U)))

Definition *Fung1* : (*DS D* -*C*→ *DS D*) -*c*> *DS D* -*C*→ *DS D*.

Lemma *Fung1_eq* : ∀ (*g1* : *DS D* -*c*> *DS D*) (*p*:*DS D*),
Fung1 g1 p == *app p* (*g1* (*rem* (*rem p*))).

Lemma *Fung1_pair_eq* : ∀ (*g1* : *DS D* -*C*→ *DS D*) (*p*:*DS D*),
Fung1 g1 p == *app p* (*g1* (*rem* (*rem p*))).

Definition *g1* : *DS D* -*c*> *DS D* := *FIXP* (*DS D* -*C*→ *DS D*) *Fung1*.

Lemma *g1_eq* : ∀ (*p*:*DS D*),
g1 p == *app p* (*g1* (*rem* (*rem p*))).
Hint *Resolve g1_eq*.

Lemma *first_g1_eq* : ∀ (*p*:*DS D*), *first* (*g1 p*) == *first p*.

Lemma *rem_g1_eq* : ∀ (*p*:*DS D*), *is_cons p* → *rem* (*g1 p*) == *g1* (*rem* (*rem p*)).

Lemma *is_cons_g1* : ∀ (*p*:*DS D*), *is_cons p* → *is_cons* (*g1 p*).
Hint *Resolve is_cons_g1*.

Lemma *g1_is_cons* : ∀ (*p*:*DS D*), *is_cons* (*g1 p*) → *is_cons p*.

- g2 U = app (rem U) (g2 (rem (rem U)))

Definition *Fung2* : (*DS D* -*C*→ *DS D*) -*c*> *DS D* -*C*→ *DS D*.

Lemma *Fung2_eq* : ∀ (*g2* : *DS D* -*c*> *DS D*) (*p*:*DS D*),
Fung2 g2 p == *app* (*rem p*) (*g2* (*rem* (*rem p*))).

Definition *g2* : *DS D -c> DS D* := *FIXP* (*DS D -C→ DS D*) *Fung2*.

Lemma *g2_eq* : ∀ (*p*:*DS D*), *g2 p* == *app* (*rem p*) (*g2* (*rem* (*rem p*))).
Hint *Resolve g2_eq*.

Lemma *first_g2_eq* : ∀ (*p*:*DS D*), *first* (*g2 p*) == *first* (*rem p*).

Lemma *rem_g2_eq* : ∀ (*p*:*DS D*), *is_cons* (*rem p*) → *rem* (*g2 p*) == *g2* (*rem* (*rem p*)).

Lemma *is_cons_g2* : ∀ (*p*:*DS D*), *is_cons* (*rem p*) → *is_cons* (*g2 p*).
Hint *Resolve is_cons_g2*.

Lemma *g2_is_cons* : ∀ (*p*:*DS D*), *is_cons* (*g2 p*) → *is_cons* (*rem p*).

- h n s = cons n s

Definition *h* (*n*:*nat*) : *DS D -c> DS D* := *CONS n*.

7.2 Definition of the system

Inductive *LI* : Type := .
Inductive *LO* : Type := *X* | *Y* | *Z* | *T1* | *T2*.

Definition *SL* : *LI*+*LO* → Type := fun *x* ⇒ *D*.

- X = f Y Z; Y = h 0 T1; Z = h 1 T2; T1 = g1 X; T2 = g2 X

```
Definition sys : system SL :=
    fun x : LO ⇒ match x with
                X ⇒ (f @2_ (PROJ (DS_fam SL) (inr LI Y))) (PROJ (DS_fam SL) (inr LI Z))
              | Y ⇒ h O @_ PROJ (DS_fam SL) (inr LI T1)
              | Z ⇒ h 1 @_ PROJ (DS_fam SL) (inr LI T2)
              | T1 ⇒ g1 @_ PROJ (DS_fam SL) (inr LI X)
              | T2 ⇒ g2 @_ PROJ (DS_fam SL) (inr LI X)
                         end.
```

- The system has no inputs

Definition *init* : *Dprodi* (*DS_fam* (*inlSL SL*)) := fun *i* : *LI* ⇒ match *i* with end.

- Equation associated to the system

Definition *EQN_sys* : *Dprodi* (*DS_fam SL*) -*c*> *Dprodi* (*DS_fam SL*)
:= *EQN_of_system sys init*.

- The result is given on the X node

Definition *result* : *DS D* := *sol_of_system SL sys init* (*inr LI X*).

7.3 Properties

- X == f (h O (g1 X)) (h 1 (g2 X))

Definition *FunX* : *DS D -c> DS D* := (*f* @2_ (*h O* @_ *g1*)) (*h* 1 @_ *g2*).

Lemma *FunX_simpl* : ∀ (*s*:*DS D*), *FunX s* = *f* (*h O* (*g1 s*)) (*h* 1 (*g2 s*)).

Lemma *eqn_sys_FunX* :
∀ *p* : *Dprodi* (*DS_fam SL*),
fcont_compn EQN_sys 2 *p* (*inr LI X*) == *FunX* (*p* (*inr LI X*)).

Lemma *result_eq* : *result* == *FIXP* (*DS D*) *FunX*.

Lemma *lem1* : ∀ *s*:*DS D*, *FunX s* == *cons O* (*cons* 1 (*f* (*g1 s*) (*g2 s*))).

Lemma *R_is_cons* : ∀ *s t*, *t* == *f* (*g1 s*) (*g2 s*) → *is_cons s* → *is_cons t*.

Lemma *R_is_cons_rem* : ∀ *s t*, *t* == *f* (*g1 s*) (*g2 s*)
→ *is_cons* (*rem s*) → *is_cons* (*rem t*).

Lemma *R_is_cons_inv* : ∀ *s t*, *t* == *f* (*g1 s*) (*g2 s*) → *is_cons t* → *is_cons s*.

Lemma *R_is_cons_rem_inv* : ∀ *s t*, *t* == *f* (*g1 s*) (*g2 s*)
→ *is_cons* (*rem t*) → *is_cons* (*rem s*).

Lemma *lem2* : ∀ *s*:*DS D*, *s* == *f* (*g1 s*) (*g2 s*).

- result is an infinite stream alternating 0 and 1

Lemma *result_alt* : *result* == *cons O* (*cons* 1 *result*).

Lemma *result_inf* : *infinite result*.

Require Export *Systems*.
Require Export *Cpo_nat*.
Require Export *Arith*.
Require Export *Euclid*.

8 Sieve.v: Example of Sieve of Eratosthenes

- sift (cons a s) == cons a (sift (filter (div a) s))

8.1 Preliminaries on divisibility

Definition *div* (*n m*:*nat*) := ∃ *q*, *m* = *q* × *n*.

Lemma *div0* : ∀ *q n r*, *r*=*q* × *n* → *r* < *n* → *r* = *O*.

Lemma *div0bis* : ∀ *q q' n r*, *q* × *n* = *q'* × *n* + *r* → *r* < *n* → *r* = *O*.

Definition *div_dec* : ∀ *n m*, {*div n m*}+ {~ *div n m*}.

8.2 Definition of the system

- o = sift i is recursively defined by: o = app i Y ; Y = sift X; X = fdiv i

Definition *LI* : Type := *unit*.
Definition *i* := *tt*.

Inductive *LO* : Type := *X* | *Y* | *o*.

Definition *D* := *nat*.

Definition *SL* : *LI*+*LO* → Type := fun *i* ⇒ *D*.

8.2.1 Node corresponding to the division

Definition *fdiv* : *DS D* -c> *DS D* := *DSCASE D D* (fun *a* ⇒ *FILTER* (*div a*) (*div_dec a*)).

Lemma *fdiv_cons* : ∀ *a s*, *fdiv* (*cons a s*) = *filter* (*div a*) (*div_dec a*) *s*.

8.2.2 Definition of the system parameterized by sift

Definition *Funsift* : (*DS D -C→ DS D*) *-m> system SL*.

Lemma *Funsift_simpl* : ∀ *fs x*, *Funsift fs x* = match *x* with *X* ⇒ *fdiv* @_ *PROJ* (*DS_fam SL*) (*inl LO i*)
| *Y* ⇒ *fs* @_ *PROJ* (*DS_fam SL*) (*inr LI X*)
| *o* ⇒ (*APP D* @2_ *PROJ* (*DS_fam SL*) (*inl LO i*))
(*PROJ* (*DS_fam SL*) (*inr LI Y*))
end.

Definition *FunSift* : (*DS D -C→ DS D*) *-c> system SL*.

Lemma *FunSift_simpl* : ∀ *fs x*, *FunSift fs x* = match *x* with *X* ⇒ *fdiv* @_ *PROJ* (*DS_fam SL*) (*inl LO i*)
| *Y* ⇒ *fs* @_ *PROJ* (*DS_fam SL*) (*inr LI X*)
| *o* ⇒ (*APP D* @2_ *PROJ* (*DS_fam SL*) (*inl LO i*))
(*PROJ* (*DS_fam SL*) (*inr LI Y*))
end.

- *focus* restrict to the input and output observed

Definition *focus* : (*DS_prod* (*inlSL SL*) *-C→ DS_prod SL*) *-c> DS D -C→ DS D* :=
PROJ (*DS_fam SL*) (*inr LI o*) @@_ *fcont_SEQ* (*DS D*) (*DS_prod* (*inlSL SL*)) (*DS_prod SL*) (*PAIR1* (*DS D*)).

Lemma *focus_simpl* : ∀ (*f* : *DS_prod* (*inlSL SL*) *-C→ DS_prod SL*) (*s*:*DS D*),
focus f s = *f* (*pair1 s*) (*inr LI o*).

8.2.3 Definition and properties of *sift*

Definition *sift* : *DS D -C→ DS D* :=
FIXP (*DS D -C→DS D*) (*focus* @_ (*sol_of_system SL* @_ *FunSift*)).

Lemma *sift_eq* : *sift* == *focus* (*sol_of_system SL* (*FunSift sift*)).
Hint *Resolve sift_eq*.

Lemma *sift_le_compat* : ∀ *x y*, $x \leq y \rightarrow$ *sift x* ≤ *sift y*.
Hint *Resolve sift_le_compat*.

Lemma *sift_eq_compat* : ∀ *x y*, *x* == *y* → *sift x* == *sift y*.
Hint *Resolve sift_eq_compat*.

Lemma *sol_of_system_i* : ∀ *s* : *DS D*, *sol_of_system SL* (*FunSift sift*) (*pair1 s*) (*inl LO i*) == *s*.

Lemma *sol_of_system_X* : ∀ *s* : *DS D*,
sol_of_system SL (*FunSift sift*) (*pair1 s*) (*inr LI X*) == *fdiv s*.

Lemma *sol_of_system_Y* : ∀ *s* : *DS D*,
sol_of_system SL (*FunSift sift*) (*pair1 s*) (*inr LI Y*) == *sift* (*fdiv s*).

Lemma *sol_of_system_o* : ∀ *s* : *DS D*,
sol_of_system SL (*FunSift sift*) (*pair1 s*) (*inr LI o*) == *app s* (*sift* (*fdiv s*)).

Lemma *sift_prop* : ∀ *a s*, *sift* (*cons a s*) == *cons a* (*sift* (*filter* (*div a*) (*div_dec a*) *s*)).

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

[2] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*. North-Holland, 1974.

[3] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*. North-Holland, 1977.

[4] Christine Paulin-Mohring. A constructive denotational semantics for kahn networks in Coq. Submitted to the volume in memory of Gilles Kahn, 2007.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. .