Chapter 1

Library AugmentedSimplexCategory

```
Require Import Category.Lib.
Require Import Theory.Category.

Require Import Category.Instance.Simplex.NaturalNumberArithmetic.
Require Import Category.Instance.Simplex.Stdfinset.
Require Import Category.Instance.Simplex.FinType.

Require Import ssreflect.
Require Import ssrfun.
Require Import ssrbool.

Require Import mathcomp.ssreflect.seq.
Require Import mathcomp.ssreflect.ssrnat.
Require Import mathcomp.ssreflect.eqtype.
Require Import mathcomp.ssreflect.fintype.
Require Import mathcomp.ssreflect.finfun.
Require Import mathcomp.ssreflect.tuple.

Set Primitive Projections.
Set Universe Polymorphism.
```

This file defines the coface and codegeneracy maps of the simplex category Δ, (including the proofs that they are monotonic) and proves that the simplicial identities hold. It builds on the contents of stdfinset.v to incorporate monotonicity conditions.

Letting $[n]$ denote the n-th finite ordinal {0,... n-1}, and letting $i \in [n{+}1]$, the i-th coface map $\delta\_i : [n] \to [n{+}1]$ is the unique monotonic injection whose image does not contain i; that is, $\delta\_i(x) = x$ if $x < i$, else $\delta\_i(x) = x{+}1$ `if` $x \geq i$.

We define δ_i in terms of the *lift* and *bump* functions from the ssreflect fintype library.

Again, letting $i \in [n{+}1]$, the i-th codegeneracy map $\sigma\_i : [n{+}2] \to [n{+}1]$, (denoted σ_i), is the unique monotonic surjection such that the preimage of i contains two elements; that is, $\sigma\_i(x) = x$ if $x \leq i$; else, $\sigma\_i(x) = x$-1 if $x > i$. These functions satisfy the following equations, called the simplicial identities.

$\delta_j \circ \delta_i = \delta_i \circ \delta_{j-1};\ i < j$

$\sigma_j \circ \sigma_i = \sigma_i \circ \sigma_{j+1};\ i \leq j$
$\sigma_j \circ \delta_i = \delta_i \circ \sigma_(j-1); i < j$
$\sigma_j \circ \delta_j = id = \sigma_j \circ \delta_{j+1}$
$\sigma_j \circ \delta_i = \delta_{i-1};\ i > j+1$

which we prove in this file. References for this material include "Simplicial Objects in Algebraic Topology" by Peter May, or "Simplicial Homotopy Theory" by Goerss and Jardine. The above five equations are taken from page 1 of May's book, except that in his book they occur dualized, i.e., they are meant to be interpreted in the opposite category to our simplex category.

```
Definition monotonic {n m : nat} (f : 'I_m^n) : bool :=
  pairwise (fun i j : 'I_m => leq i j) (tuple_of_finfun f).

Definition monotonicP {n m : nat} (f : 'I_m^n)
  : reflect (∀ i j : 'I_n, i ≤ j → f i ≤ f j) (monotonic f).
Proof.
  rewrite /monotonic.
  apply/(iffP (tuple_pairwiseP _ _ _ _)); intro H.
  { intros i j ineq.
    assert (k := (H i j (mem_ord_enum i) (mem_ord_enum j))).
    rewrite leq_eqVlt in ineq ; move/orP : ineq; intro ineq;
      destruct ineq as [eq | lt].
    {

      move/eqP: eq; intro eq; by rewrite (val_inj eq). }
    assert (z := k lt).
    rewrite 2! tnth_tuple_of_finfun in z.
    exact: z.
  }
  intros i j _ _ ineq.
  rewrite 2! tnth_tuple_of_finfun. apply: H; exact: ltnW.
Qed.

Proposition monotonic_fold_equiv (n m : nat) (f : 'I_m^n) :
  monotonic f =
    let fg := tuple.tval (tuple_of_finfun f) in
    if fg is x :: xs then
      foldr andb true (pairmap (fun i j : 'I_m => leq i j) x xs)
    else true.
Proof.
  apply: pairmap_trans_pairwise; rewrite /nat_of_ord; by apply: leq_trans.
Qed.

Proposition idmap_monotonic (n : nat) : @monotonic n _ (finfun id).
Proof.
```

```
  apply/monotonicP ⇒ i j.
  by rewrite 2! ffunE.
Qed.

Record monotonic_fn_sig (n m : nat) :=
  { fun_of_monotonic_fn :> 'I_m^n ;
    _ : monotonic fun_of_monotonic_fn }.
Arguments fun_of_monotonic_fn {n} {m} _.
```

The following definition records monotonic_fn with the hint database for subtypes, which means that we can apply lemmas about general subtypes to monotonic functions. For example, we can "apply val_inj" to conclude that two monotonic functions are equal if the underlying (finite) functions are equal after the monotonicity property is forgotten. Monotonic functions are also equipped with a Boolean comparison function "==" automatically which is inherited from the underlying comparison function for finfuns.

```
Canonical Structure monotonic_fn (n m : nat) :=
  [subType for (@fun_of_monotonic_fn n m) ].
Definition monotonic_fn_eqMixin (n m : nat) :=
  [eqMixin of (monotonic_fn n m) by <:].
Canonical Structure monotonic_fn_eqType (n m : nat) :=
  EqType (monotonic_fn n m) (monotonic_fn_eqMixin n m).

Definition comp {aT : finType} {rT sT : Type} (f : {ffun aT → rT})
  (g : rT → sT)
  : {ffun aT → sT}
  := [ffun x ⇒ g (f x)].

Proposition comp_assoc {aT : finType} {rT sT mT: Type}
  (f : {ffun aT → rT}) (g : rT → sT) (h : sT → mT)
  : comp f (g \; h) = comp (comp f g) h.
Proof.
  apply: eq_ffun; simpl; intro; apply: f_equal; by rewrite ffunE.
Qed.

Definition comp_mon (n m k : nat) (g : @monotonic_fn m k)
  (f : @monotonic_fn n m)
  : @monotonic_fn n k.
Proof.
  ∃ (comp (fun_of_monotonic_fn f) (fun_of_monotonic_fn g)).
  apply/monotonicP.
  intros i j ineq; simpl.
  rewrite 2! ffunE.
  move/monotonicP : (valP g). intro H; apply: H.
  by move/monotonicP : (valP f); intro H; apply H.
Defined.

Definition id_mon (n : nat) : @monotonic_fn n n :=
```

```
  Sub (finfun (@id 'I_n)) (idmap_monotonic n).

Program Definition finord : Category :=
  {|
    obj := nat;
    hom := fun n m ⇒ @monotonic_fn n m;
    homset := fun _ _ ⇒ {| equiv := eq |};
    Category.id := id_mon;
    compose := comp_mon;
    |}.
Next Obligation.
  apply: val_inj; simpl; apply ffunP; intro i; rewrite 2! ffunE; done. Qed.
Next Obligation.
  apply: val_inj; simpl; apply ffunP; intro i; rewrite 2! ffunE; done. Qed.
Next Obligation.
  apply: val_inj; simpl; apply ffunP; intro i; rewrite 4! ffunE; done. Qed.
Next Obligation.
  apply: val_inj; simpl; apply ffunP; intro i; rewrite 4! ffunE; done. Qed.

Notation Δ := finord.
```