```
(*****************************************************************************************************************************)
(* General: general data structures                                                                                          *)
(*****************************************************************************************************************************)

Require Import Coq.Unicode.Utf8.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Setoids.Setoid.
Require Import Coq.Strings.String.
Require Setoid.
Require Import Coq.Lists.List.
Require Import Preamble.
Generalizable All Variables.
Require Import Omega.

Definition EqDecider T := forall (n1 n2:T),  sumbool (n1=n2) (not (n1=n2)).
Class EqDecidable (T:Type) :=
{ eqd_type           := T
; eqd_dec            :  forall v1 v2:T, sumbool (v1=v2) (not (v1=v2))
}.
Coercion eqd_type : EqDecidable >-> Sortclass.


Class ToString (T:Type) := { toString : T -> string }.
Instance StringToString : ToString string := { toString := fun x => x }.

Class Concatenable {T:Type} :=
 { concatenate : T -> T -> T }.
 Implicit Arguments Concatenable [ ].
Open Scope string_scope.
Open Scope type_scope.
Close Scope list_scope.
Notation "a +++ b" := (@concatenate _ _ a b) (at level 99).
Instance ConcatenableString : Concatenable string := { concatenate := append }.


(*******************************************************************************)
(* Trees                                                                       *)

Inductive Tree (a:Type) : Type :=
  | T_Leaf   : a -> Tree a
```

```
  | T_Branch : Tree a -> Tree a -> Tree a.
Implicit Arguments T_Leaf   [ a ].
Implicit Arguments T_Branch [ a ].

Notation "a ,, b"   := (@T_Branch _ a b)                        : tree_scope.

(* tree-of-option-of-X comes up a lot, so we give it special notations *)
Notation "[ q ]"    := (@T_Leaf (option _) (Some q))            : tree_scope.
Notation "[ ]"      := (@T_Leaf (option _) None)                : tree_scope.
Notation "[]"       := (@T_Leaf (option _) None)                : tree_scope.

Fixpoint InT {A} (a:A) (t:Tree ??A) {struct t} : Prop :=
  match t with
    | T_Leaf None     => False
    | T_Leaf (Some x) => x = a
    | T_Branch b1 b2  => InT a b1 \/ InT a b2
  end.

Open Scope tree_scope.

Fixpoint mapTree {a b:Type}(f:a->b)(t:@Tree a) : @Tree b :=
  match t with
    | T_Leaf x     => T_Leaf (f x)
    | T_Branch l r => T_Branch (mapTree f l) (mapTree f r)
  end.
Fixpoint mapOptionTree {a b:Type}(f:a->b)(t:@Tree ??a) : @Tree ??b :=
  match t with
    | T_Leaf None     => T_Leaf None
    | T_Leaf (Some x) => T_Leaf (Some (f x))
    | T_Branch l r    => T_Branch (mapOptionTree f l) (mapOptionTree f r)
  end.
Fixpoint mapTreeAndFlatten {a b:Type}(f:a->@Tree b)(t:@Tree a) : @Tree b :=
  match t with
    | T_Leaf x        => f x
    | T_Branch l r    => T_Branch (mapTreeAndFlatten f l) (mapTreeAndFlatten f r)
  end.
Fixpoint mapOptionTreeAndFlatten {a b:Type}(f:a->@Tree ??b)(t:@Tree ??a) : @Tree ??b :=
  match t with
    | T_Leaf None     => []
    | T_Leaf (Some x) => f x
    | T_Branch l r    => T_Branch (mapOptionTreeAndFlatten f l) (mapOptionTreeAndFlatten f r)
```

```
  end.
Fixpoint treeReduce {T:Type}{R:Type}(mapLeaf:T->R)(mergeBranches:R->R->R) (t:Tree T) :=
  match t with
  | T_Leaf x => mapLeaf x
  | T_Branch y z => mergeBranches (treeReduce mapLeaf mergeBranches y) (treeReduce mapLeaf mergeBranches z)
  end.
Definition treeDecomposition {D T:Type} (mapLeaf:T->D) (mergeBranches:D->D->D) :=
  forall d:D, { tt:Tree T & d = treeReduce mapLeaf mergeBranches tt }.

Lemma tree_dec_eq :
   forall {Q}(t1 t2:Tree ??Q),
     (forall q1 q2:Q, sumbool (q1=q2) (not (q1=q2))) ->
     sumbool (t1=t2) (not (t1=t2)).
  intro Q.
  intro t1.
  induction t1; intros.

  destruct a; destruct t2.
  destruct o.
  set (X q q0) as X'.
  inversion X'; subst.
  left; auto.
  right; unfold not; intro; apply H. inversion H0; subst. auto.
  right. unfold not; intro; inversion H.
  right. unfold not; intro; inversion H.
  destruct o.
  right. unfold not; intro; inversion H.
  left; auto.
  right. unfold not; intro; inversion H.

  destruct t2.
  right. unfold not; intro; inversion H.
  set (IHt1_1 t2_1 X) as X1.
  set (IHt1_2 t2_2 X) as X2.
  destruct X1; destruct X2; subst.
  left; auto.

  right.
  unfold not; intro H.
  apply n.
  inversion H; auto.
```

```
  right.
  unfold not; intro H.
  apply n.
  inversion H; auto.

  right.
  unfold not; intro H.
  apply n.
  inversion H; auto.
  Defined.

Lemma mapOptionTree_compose : forall A B C (f:A->B)(g:B->C)(l:Tree ??A),
  (mapOptionTree (g ○ f) l) = (mapOptionTree g (mapOptionTree f l)).
  induction l.
    destruct a.
    reflexivity.
    reflexivity.
    simpl.
    rewrite IHl1.
    rewrite IHl2.
    reflexivity.
    Qed.

Lemma mapOptionTree_extensional {A}{B}(f g:A->B) : (forall a, f a = g a) -> (forall t, mapOptionTree f t = mapOptionTree g t).
  intros.
  induction t.
  destruct a; auto.
  simpl; rewrite H; auto.
  simpl; rewrite IHt1; rewrite IHt2; auto.
  Qed.

Fixpoint treeToString {T}{TT:ToString T}(t:Tree ??T) : string :=
match t with
  | T_Leaf None => "[]"
  | T_Leaf (Some s) => "["+++toString s+++"]"
  | T_Branch b1 b2 => treeToString b1 +++ ",," +++ treeToString b2
end.
Instance TreeToString {T}{TT:ToString T} : ToString (Tree ??T) := { toString := treeToString }.

(*******************************************************************************)
```

```
(* Lists                                                                       *)

Notation "a :: b"         := (cons a b)                              : list_scope.
Open Scope list_scope.
Fixpoint leaves {a:Type}(t:Tree (option a)) : list a :=
  match t with
    | (T_Leaf l)     => match l with
                          | None   => nil
                          | Some x => x::nil
                        end
    | (T_Branch l r) => app (leaves l) (leaves r)
  end.
(* weak inverse of "leaves" *)
Fixpoint unleaves {A:Type}(l:list A) : Tree (option A) :=
  match l with
    | nil    => []
    | (a::b) => [a],,(unleaves b)
  end.

(* a map over a list and the conjunction of the results *)
Fixpoint mapProp {A:Type} (f:A->Prop) (l:list A) : Prop :=
  match l with
    | nil => True
    | (a::al) => f a /\ mapProp f al
  end.

Lemma map_id : forall A (l:list A), (map (fun x:A => x) l) = l.
  induction l.
  auto.
  simpl.
  rewrite IHl.
  auto.
  Defined.
Lemma map_app : forall `(f:A->B) l l', map f (app l l') = app (map f l) (map f l').
  intros.
  induction l; auto.
  simpl.
  rewrite IHl.
  auto.
  Defined.
Lemma map_compose : forall A B C (f:A->B)(g:B->C)(l:list A),
```

```
  (map (g ○ f) l) = (map g (map f l)).
  intros.
  induction l.
  simpl; auto.
  simpl.
  rewrite IHl.
  auto.
  Defined.
Lemma list_cannot_be_longer_than_itself : forall `(a:A)(b:list A), b = (a::b) -> False.
  intros.
  induction b.
  inversion H.
  inversion H. apply IHb in H2.
  auto.
  Defined.
Lemma list_cannot_be_longer_than_itself' : forall A (b:list A) (a c:A), b = (a::c::b) -> False.
  induction b.
  intros; inversion H.
  intros.
  inversion H.
  apply IHb in H2.
  auto.
  Defined.

Lemma mapOptionTree_on_nil : forall `(f:A->B) h, [] = mapOptionTree f h -> h=[].
  intros.
  destruct h.
  destruct o. inversion H.
  auto.
  inversion H.
  Defined.

Lemma mapOptionTree_comp a b c (f:a->b) (g:b->c) q : (mapOptionTree g (mapOptionTree f q)) = mapOptionTree (g ○ f) q.
  induction q.
    destruct a0; simpl.
    reflexivity.
    reflexivity.
    simpl.
    rewrite IHq1.
    rewrite IHq2.
    reflexivity.
```

```
  Qed.

Lemma leaves_unleaves {T}(t:list T) : leaves (unleaves t) = t.
  induction t; auto.
  simpl.
  rewrite IHt; auto.
  Qed.

Lemma mapleaves' {T:Type}(t:list T){Q}{f:T->Q} : unleaves (map f t) = mapOptionTree f (unleaves t).
  induction t; simpl in *; auto.
  rewrite IHt; auto.
  Qed.

(* handy facts: map preserves the length of a list *)
Lemma map_on_nil : forall A B (s1:list A) (f:A->B), nil = map f s1 -> s1=nil.
  intros.
  induction s1.
  auto.
  assert False.
  simpl in H.
  inversion H.
  inversion H0.
  Defined.
Lemma map_on_singleton : forall A B (f:A->B) x (s1:list A), (cons x nil) = map f s1 -> { y : A & s1=(cons y nil) & (f y)=x }.
  induction s1.
  intros.
  simpl in H; assert False. inversion H. inversion H0.
  clear IHs1.
  intros.
  exists a.
  simpl in H.
  assert (s1=nil).
    inversion H. apply map_on_nil in H2. auto.
  subst.
  auto.
  assert (s1=nil).
    inversion H. apply map_on_nil in H2. auto.
  subst.
  simpl in H.
  inversion H. auto.
  Defined.
```

```
Lemma map_on_doubleton : forall A B (f:A->B) x y (s1:list A), ((x::y::nil) = map f s1) ->
  { z : A*A & s1=((fst z)::(snd z)::nil) & (f (fst z))=x /\ (f (snd z))=y }.
  intros.
  destruct s1.
  inversion H.
  destruct s1.
  inversion H.
  destruct s1.
  inversion H.
  exists (a,a0); auto.
  simpl in H.
  inversion H.
  Defined.


Lemma mapTree_compose : forall A B C (f:A->B)(g:B->C)(l:Tree A),
  (mapTree (g ○ f) l) = (mapTree g (mapTree f l)).
  induction l.
    reflexivity.
    simpl.
    rewrite IHl1.
    rewrite IHl2.
    reflexivity.
    Defined.

Lemma lmap_compose : forall A B C (f:A->B)(g:B->C)(l:list A),
  (map (g ○ f) l) = (map g (map f l)).
  intros.
  induction l.
  simpl; auto.
  simpl.
  rewrite IHl.
  auto.
  Defined.

(* sends a::b::c::nil to [[[[],,c],,b],,a] *)
Fixpoint unleaves' {A:Type}(l:list A) : Tree (option A) :=
  match l with
    | nil    => []
    | (a::b) => (unleaves' b),,[a]
```

```
  end.

(* sends a::b::c::nil to [[[[],,c],,b],,a] *)
Fixpoint unleaves'' {A:Type}(l:list ??A) : Tree ??A :=
  match l with
    | nil    => []
    | (a::b) => (unleaves'' b),,(T_Leaf a)
  end.

Lemma mapleaves {T:Type}(t:Tree ??T){Q}{f:T->Q} : leaves (mapOptionTree f t) = map f (leaves t).
  induction t.
  destruct a; auto.
  simpl.
  rewrite IHt1.
  rewrite IHt2.
  rewrite map_app.
  auto.
  Qed.

Fixpoint filter {T:Type}(l:list ??T) : list T :=
  match l with
    | nil => nil
    | (None::b) => filter b
    | ((Some x)::b) => x::(filter b)
  end.

Inductive distinct {T:Type} : list T -> Prop :=
| distinct_nil  : distinct nil
| distinct_cons : forall a ax, (@In _ a ax -> False) -> distinct ax -> distinct (a::ax).

Inductive distinctT {T:Type} : Tree ??T -> Prop :=
| distinctT_nil  : distinctT []
| distinctT_leaf : forall t, distinctT [t]
| distinctT_cons : forall t1 t2, (forall v, InT v t1 -> InT v t2 -> False) -> distinctT (t1,,t2).

Lemma in_decidable {VV:Type}{eqdVV:EqDecidable VV} : forall (v:VV)(lv:list VV), sumbool (In v lv) (not (In v lv)).
  intros.
  induction lv.
  right.
  unfold not.
  intros.
```

```
  inversion H.
  destruct IHlv.
  left.
  simpl.
  auto.
  set (eqd_dec v a) as dec.
  destruct dec.
  subst.
  left; simpl; auto.
  right.
  unfold not; intros.
  destruct H.
  subst.
  apply n0; auto.
  apply n.
  apply H.
  Defined.

Lemma distinct_decidable {VV:Type}{eqdVV:EqDecidable VV} : forall (lv:list VV), sumbool (distinct lv) (not (distinct lv)).
  intros.
  induction lv.
  left; apply distinct_nil.
  destruct IHlv.
  set (in_decidable a lv) as dec.
  destruct dec.
  right; unfold not; intros.
  inversion H.
  subst.
  apply H2; auto.
  left.
  apply distinct_cons; auto.
  right.
  unfold not; intros.
  inversion H.
  subst.
  apply n; auto.
  Defined.

Lemma map_preserves_length {A}{B}(f:A->B)(l:list A) : (length l) = (length (map f l)).
  induction l; auto.
  simpl.
```

```
  omega.
  Qed.

(* decidable quality on a list of elements which have decidable equality *)
Definition list_eq_dec : forall {T:Type}(l1 l2:list T)(dec:forall t1 t2:T, sumbool (eq t1 t2) (not (eq t1 t2))),
  sumbool (eq l1 l2) (not (eq l1 l2)).
  intro T.
  intro l1.
  induction l1; intros.
    destruct l2.
    left; reflexivity.
    right; intro H; inversion H.
  destruct l2 as [| b l2].
    right; intro H; inversion H.
  set (IHl1 l2 dec) as eqx.
    destruct eqx.
    subst.
    set (dec a b) as eqy.
    destruct eqy.
      subst.
      left; reflexivity.
      right. intro. inversion H. subst. apply n. auto.
    right.
      intro.
      inversion H.
      apply n.
      auto.
      Defined.

Instance EqDecidableList {T:Type}(eqd:EqDecidable T) : EqDecidable (list T).
  apply Build_EqDecidable.
  intros.
  apply list_eq_dec.
  apply eqd_dec.
  Defined.

(******************************************************************************)
(* Length-Indexed Lists                                                       *)

Inductive vec (A:Type) : nat -> Type :=
| vec_nil  : vec A 0
```

```
| vec_cons : forall n, A -> vec A n -> vec A (S n).

Fixpoint vec2list {n:nat}{t:Type}(v:vec t n) : list t :=
  match v with
    | vec_nil => nil
    | vec_cons n a va => a::(vec2list va)
  end.

Require Import Omega.
Definition vec_get : forall {T:Type}{l:nat}(v:vec T l)(n:nat)(pf:lt n l), T.
  intro T.
  intro len.
  intro v.
  induction v; intros.
    assert False.
    inversion pf.
    inversion H.
  rename n into len.
    destruct n0 as [|n].
    exact a.
    apply (IHv n).
    omega.
    Defined.

Definition vec_zip {n:nat}{A B:Type}(va:vec A n)(vb:vec B n) : vec (A*B) n.
  induction n.
  apply vec_nil.
  inversion va; subst.
  inversion vb; subst.
  apply vec_cons; auto.
  Defined.

Definition vec_map {n:nat}{A B:Type}(f:A->B)(v:vec A n) : vec B n.
  induction n.
  apply vec_nil.
  inversion v; subst.
  apply vec_cons; auto.
  Defined.

Fixpoint vec_In {A:Type} {n:nat} (a:A) (l:vec A n) : Prop :=
  match l with
```

```
    | vec_nil         => False
    | vec_cons _ n m  => (n = a) \/ vec_In a m
  end.
Implicit Arguments vec_nil  [ A   ].
Implicit Arguments vec_cons [ A n ].

Definition append_vec {n:nat}{T:Type}(v:vec T n)(t:T) : vec T (S n).
  induction n.
  apply (vec_cons t vec_nil).
  apply vec_cons; auto.
  Defined.

Definition list2vec {T:Type}(l:list T) : vec T (length l).
  induction l.
  apply vec_nil.
  apply vec_cons; auto.
  Defined.

Definition vec_head {n:nat}{T}(v:vec T (S n)) : T.
  inversion v;  auto.
  Defined.
Definition vec_tail {n:nat}{T}(v:vec T (S n)) : vec T n.
  inversion v;  auto.
  Defined.

Lemma vec_chop {T}{l1 l2:list T}{Q}(v:vec Q (length (app l1 l2))) : vec Q (length l1).
  induction l1.
  apply vec_nil.
  apply vec_cons.
  simpl in *.
  inversion v; subst; auto.
  apply IHl1.
  inversion v; subst; auto.
  Defined.

Lemma vec_chop' {T}{l1 l2:list T}{Q}(v:vec Q (length (app l1 l2))) : vec Q (length l2).
  induction l1.
  apply v.
  simpl in *.
  apply IHl1; clear IHl1.
  inversion v; subst; auto.
```

```
  Defined.

Lemma vec2list_len {T}{n}(v:vec T n) : length (vec2list v) = n.
  induction v; auto.
  simpl.
  omega.
  Qed.

Lemma vec2list_map_list2vec {A B}{n}(f:A->B)(v:vec A n) : map f (vec2list v) = vec2list (vec_map f v).
  induction v; auto.
  simpl. rewrite IHv.
  reflexivity.
  Qed.

Lemma vec2list_list2vec {A}(v:list A) : vec2list (list2vec v) = v.
  induction v; auto.
  set (vec2list (list2vec (a :: v))) as q.
  rewrite <- IHv.
  unfold q.
  clear q.
  simpl.
  reflexivity.
  Qed.

Notation "a ::: b" := (@vec_cons _ _ a b) (at level 20).



(*******************************************************************************)
(* Shaped Trees                                                                *)

(* a ShapedTree is a tree indexed by the shape (but not the leaf values) of another tree; isomorphic to (ITree (fun _ => Q)) *)
Inductive ShapedTree {T:Type}(Q:Type) : Tree ??T -> Type :=
| st_nil    : @ShapedTree T Q []
| st_leaf   : forall {t}, Q -> @ShapedTree T Q [t]
| st_branch : forall {t1}{t2}, @ShapedTree T Q t1 -> @ShapedTree T Q t2 -> @ShapedTree T Q (t1,,t2).

Fixpoint unshape {T:Type}{Q:Type}{idx:Tree ??T}(st:@ShapedTree T Q idx) : Tree ??Q :=
match st with
| st_nil => []
| st_leaf _ q => [q]
```

```
| st_branch _ _ b1 b2 => (unshape b1),,(unshape b2)
end.

Definition mapShapedTree {T}{idx:Tree ??T}{V}{Q}(f:V->Q)(st:ShapedTree V idx) : ShapedTree Q idx.
  induction st.
  apply st_nil.
  apply st_leaf. apply f. apply q.
  apply st_branch; auto.
  Defined.

Definition zip_shapedTrees {T:Type}{Q1 Q2:Type}{idx:Tree ??T}
   (st1:ShapedTree Q1 idx)(st2:ShapedTree Q2 idx) : ShapedTree (Q1*Q2) idx.
  induction idx.
    destruct a.
    apply st_leaf; auto.
    inversion st1.
    inversion st2.
    auto.
    apply st_nil.
    apply st_branch; auto.
    inversion st1; subst; apply IHidx1; auto.
    inversion st2; subst; auto.
    inversion st2; subst; apply IHidx2; auto.
    inversion st1; subst; auto.
  Defined.

Definition build_shapedTree {T:Type}(idx:Tree ??T){Q:Type}(f:T->Q) : ShapedTree Q idx.
  induction idx.
  destruct a.
  apply st_leaf; auto.
  apply st_nil.
  apply st_branch; auto.
  Defined.

Lemma unshape_map : forall {Q}{b}(f:Q->b){T}{idx:Tree ??T}(t:ShapedTree Q idx),
   mapOptionTree f (unshape t) = unshape (mapShapedTree f t).
  intros.
  induction t; auto.
  simpl.
  rewrite IHt1.
  rewrite IHt2.
```

```
  reflexivity.
  Qed.



(*******************************************************************************)
(* Type-Indexed Lists                                                          *)

(* an indexed list *)
Inductive IList (I:Type)(F:I->Type) : list I -> Type :=
| INil      :                                      IList I F nil
| ICons     :   forall i is, F i -> IList I F is -> IList I F (i::is).
Implicit Arguments INil [ I F ].
Implicit Arguments ICons [ I F ].

Notation "a :::: b" := (@ICons _ _ _ _ a b) (at level 20).

Definition ilist_head {T}{F}{x}{y} : IList T F (x::y) -> F x.
  intro il.
  inversion il.
  subst.
  apply X.
  Defined.

Definition ilist_tail {T}{F}{x}{y} : IList T F (x::y) -> IList T F y.
  intro il.
  inversion il.
  subst.
  apply X0.
  Defined.

Definition ilmap {I}{F}{G}{il:list I}(f:forall i:I, F i -> G i) : IList I F il -> IList I G il.
  induction il; intros; auto.
  apply INil.
  inversion X; subst.
  apply ICons; auto.
  Defined.

Lemma ilist_chop {T}{F}{l1 l2:list T}(v:IList T F (app l1 l2)) : IList T F l1.
  induction l1; auto.
```

```
  apply INil.
  apply ICons.
  inversion v; auto.
  apply IHl1.
  inversion v; auto.
  Defined.

Lemma ilist_chop' {T}{F}{l1 l2:list T}(v:IList T F (app l1 l2)) : IList T F l2.
  induction l1; auto.
  apply IHl1.
  inversion v; subst; auto.
  Defined.

Fixpoint ilist_to_list {T}{Z}{l:list T}(il:IList T (fun _ => Z) l) : list Z :=
  match il with
  | INil   => nil
  | a::::b => a::(ilist_to_list b)
  end.

(* a tree indexed by a (Tree (option X)) *)
Inductive ITree (I:Type)(F:I->Type) : Tree ??I -> Type :=
| INone      :                               ITree I F []
| ILeaf      :  forall       i:     I, F i -> ITree I F [i]
| IBranch    :  forall it1 it2:Tree ??I, ITree I F it1 -> ITree I F it2 -> ITree I F (it1,,it2).
Implicit Arguments INil    [ I F ].
Implicit Arguments ILeaf   [ I F ].
Implicit Arguments IBranch [ I F ].

Definition itmap {I}{F}{G}{il:Tree ??I}(f:forall i:I, F i -> G i) : ITree I F il -> ITree I G il.
  induction il; intros; auto.
  destruct a.
  apply ILeaf.
  apply f.
  inversion X; auto.
  apply INone.
  apply IBranch; inversion X; auto.
  Defined.

Fixpoint itree_to_tree {T}{Z}{l:Tree ??T}(il:ITree T (fun _ => Z) l) : Tree ??Z :=
  match il with
  | INone     => []
```

```
  | ILeaf _ a => [a]
  | IBranch _ _ b1 b2 => (itree_to_tree b1),,(itree_to_tree b2)
  end.


(*****************************************************************************)
(* Extensional equality on functions                                         *)

Definition extensionality := fun (t1 t2:Type) => (fun (f:t1->t2) g => forall x:t1, (f x)=(g x)).
Hint Transparent extensionality.
Instance extensionality_Equivalence : forall t1 t2, Equivalence (extensionality t1 t2).
  intros; apply Build_Equivalence;
    intros; compute; intros; auto.
    rewrite H; rewrite H0; auto.
  Defined.
  Add Parametric Morphism (A B C:Type) : (fun f g => g ○ f)
  with signature (extensionality A B ==> extensionality B C ==> extensionality A C) as parametric_morphism_extensionality.
  unfold extensionality; intros; rewrite (H x1); rewrite (H0 (y x1)); auto.
  Defined.
Lemma extensionality_composes : forall t1 t2 t3 (f f':t1->t2) (g g':t2->t3),
  (extensionality _ _ f f') ->
  (extensionality _ _ g g') ->
  (extensionality _ _ (g ○ f) (g' ○ f')).
  intros.
  unfold extensionality.
  unfold extensionality in H.
  unfold extensionality in H0.
  intros.
  rewrite H.
  rewrite H0.
  auto.
  Qed.
```

```
Definition map2 {A}{B}(f:A->B)(t:A*A) : (B*B) := ((f (fst t)), (f (snd t))).


(* string stuff *)
Variable eol : string.
Extract Constant eol  => "'\n':[]".

Class Monad {T:Type->Type} :=
{ returnM : forall {a},     a -> T a
; bindM   : forall {a}{b},  T a -> (a -> T b) -> T b
}.
Implicit Arguments Monad [ ].
Notation "a >>>= b" := (@bindM _ _ _ _ a b) (at level 50, left associativity).

(* the Error monad *)
Inductive OrError (T:Type) :=
| Error : forall error_message:string, OrError T
| OK    : T      -> OrError T.
Notation "??? T" := (OrError T) (at level 10).
Implicit Arguments Error [T].
Implicit Arguments OK [T].

Definition orErrorBind {T:Type} (oe:OrError T) {Q:Type} (f:T -> OrError Q) :=
  match oe with
    | Error s => Error s
    | OK    t => f t
  end.
Notation "a >>= b" := (@orErrorBind _ a _ b) (at level 20).

Definition orErrorBindWithMessage {T:Type} (oe:OrError T) {Q:Type} (f:T -> OrError Q) err_msg :=
  match oe with
    | Error s => Error (err_msg +++ eol +++ "  " +++ s)
    | OK    t => f t
  end.

Notation "a >>=[ S ] b" := (@orErrorBindWithMessage _ a _ b S) (at level 20).

Definition addErrorMessage s {T} (x:OrError T) :=
  x >>=[ s ] (fun y => OK y).
```

```
Inductive Indexed {T:Type}(f:T -> Type) : ???T -> Type :=
| Indexed_Error : forall error_message:string, Indexed f (Error error_message)
| Indexed_OK    : forall t, f t -> Indexed f (OK t)
.


Require Import Coq.Arith.EqNat.
Instance EqDecidableNat : EqDecidable nat.
  apply Build_EqDecidable.
  intros.
  apply eq_nat_dec.
  Defined.

(* for a type with decidable equality, we can maintain lists of distinct elements *)
Section DistinctList.
  Context `{V:EqDecidable}.

  Fixpoint addToDistinctList (cv:V)(cvl:list V) :=
  match cvl with
  | nil      => cv::nil
  | cv'::cvl' => if eqd_dec cv cv' then cvl' else cv'::(addToDistinctList cv cvl')
  end.

  Fixpoint removeFromDistinctList (cv:V)(cvl:list V) :=
  match cvl with
  | nil => nil
  | cv'::cvl' => if eqd_dec cv cv' then removeFromDistinctList cv cvl' else cv'::(removeFromDistinctList cv cvl')
  end.

  Fixpoint removeFromDistinctList' (cvrem:list V)(cvl:list V) :=
  match cvrem with
  | nil        => cvl
  | rem::cvrem' => removeFromDistinctList rem (removeFromDistinctList' cvrem' cvl)
  end.

  Fixpoint mergeDistinctLists (cvl1:list V)(cvl2:list V) :=
  match cvl1 with
  | nil      => cvl2
  | cv'::cvl' => mergeDistinctLists cvl' (addToDistinctList cv' cvl2)
  end.
End DistinctList.
```

```
Lemma list2vecOrFail {T}(l:list T)(n:nat)(error_message:nat->nat->string) : ???(vec T n).
  set (list2vec l) as v.
  destruct (eqd_dec (length l) n).
    rewrite e in v; apply OK; apply v.
    apply (Error (error_message (length l) n)).
    Defined.

(* Uniques *)
Variable UniqSupply      : Type.                   Extract Inlined Constant UniqSupply     => "UniqSupply.UniqSupply".
Variable Unique          : Type.                   Extract Inlined Constant Unique         => "Unique.Unique".
Variable uniqFromSupply  : UniqSupply -> Unique.   Extract Inlined Constant uniqFromSupply => "UniqSupply.uniqFromSupply".
Variable splitUniqSupply : UniqSupply -> UniqSupply * UniqSupply.
    Extract Inlined Constant splitUniqSupply => "UniqSupply.splitUniqSupply".
Variable unique_eq       : forall u1 u2:Unique, sumbool (u1=u2) (u1≠u2).
    Extract Inlined Constant unique_eq => "(==)".
Variable unique_toString : Unique -> string.
    Extract Inlined Constant unique_toString => "show".
Instance EqDecidableUnique : EqDecidable Unique :=
  { eqd_dec := unique_eq }.
Instance EqDecidableToString : ToString Unique :=
  { toString := unique_toString }.

Inductive UniqM {T:Type} : Type :=
 | uniqM    : (UniqSupply -> ???(UniqSupply * T)) -> UniqM.
 Implicit Arguments UniqM [ ].

Instance UniqMonad : Monad UniqM :=
{ returnM := fun T (x:T) => uniqM (fun u => OK (u,x))
; bindM   := fun a b (x:UniqM a) (f:a->UniqM b) =>
  uniqM (fun u =>
    match x with
      | uniqM fa =>
        match fa u with
          | Error s   => Error s
          | OK (u',va) => match f va with
                            | uniqM fb => fb u'
                          end
        end
    end)
}.
```

```
Definition getU : UniqM Unique :=
  uniqM (fun us => let (us1,us2) := splitUniqSupply us in OK (us1,(uniqFromSupply us2))).

Notation "'bind' x = e ; f" := (@bindM _ _ _ _ e (fun x => f)) (x ident, at level 60, right associativity).
Notation "'return' x" := (returnM x) (at level 100).
Notation "'failM'  x" := (uniqM (fun _ => Error x)) (at level 100).
```

```
Record FreshMonad {T:Type} :=
{ FMT       :  Type -> Type
; FMT_Monad :> Monad FMT
; FMT_fresh :  forall tl:list T, FMT { t:T & @In _ t tl -> False }
}.
Implicit Arguments FreshMonad [ ].
Coercion FMT       : FreshMonad >-> Funclass.
```

```
Variable Prelude_error : forall {A}, string -> A.   Extract Inlined Constant Prelude_error => "Prelude.error".
```

```
Ltac eqd_dec_refl X :=
  destruct (eqd_dec X X) as [eqd_dec1 | eqd_dec2];
    [ clear eqd_dec1 | set (eqd_dec2 (refl_equal _)) as eqd_dec2'; inversion eqd_dec2' ].

Lemma unleaves_injective : forall T (t1 t2:list T), unleaves t1 = unleaves t2 -> t1 = t2.
  intros T.
  induction t1; intros.
  destruct t2.
  auto.
  inversion H.
  destruct t2.
  inversion H.
  simpl in H.
```

```
  inversion H.
  set (IHt1 _ H2) as q.
  rewrite q.
  reflexivity.
  Qed.

(* adapted from Adam Chlipala's posting to the coq-club list (thanks!) *)
Definition openVec A n (v: vec A (S n)) : exists a, exists v0, v = a:::v0 :=
  match v in vec _ N return match N return vec A N -> Prop with
                              | O => fun _ => True
                              | S n => fun v => exists a, exists v0, v = a:::v0
                            end v with
    | vec_nil => I
    | a:::v0 => ex_intro _ a (ex_intro _ v0 (refl_equal _))
  end.

Definition nilVec A (v: vec A O) : v = vec_nil :=
  match v in vec _ N return match N return vec A N -> Prop with
                              | O   => fun v => v = vec_nil
                              | S n => fun v => True
                            end v with
    | vec_nil => refl_equal _
    | a:::v0  => I
  end.

Lemma fst_zip : forall T Q n (v1:vec T n)(v2:vec Q n), vec_map (@fst _ _) (vec_zip v1 v2) = v1.
  intros.
  induction n.
  set (nilVec _ v1) as v1'.
  set (nilVec _ v2) as v2'.
  subst.
  simpl.
  reflexivity.
  set (openVec _ _ v1) as v1'.
  set (openVec _ _ v2) as v2'.
  destruct v1'.
  destruct v2'.
  destruct H.
  destruct H0.
  subst.
  simpl.
```

```
  rewrite IHn.
  reflexivity.
  Qed.

Lemma snd_zip : forall T Q n (v1:vec T n)(v2:vec Q n), vec_map (@snd _ _) (vec_zip v1 v2) = v2.
  intros.
  induction n.
  set (nilVec _ v1) as v1'.
  set (nilVec _ v2) as v2'.
  subst.
  simpl.
  reflexivity.
  set (openVec _ _ v1) as v1'.
  set (openVec _ _ v2) as v2'.
  destruct v1'.
  destruct v2'.
  destruct H.
  destruct H0.
  subst.
  simpl.
  rewrite IHn.
  reflexivity.
  Qed.

Fixpoint mapM {M}{mon:Monad M}{T}(ml:list (M T)) : M (list T) :=
  match ml with
    | nil  => return nil
    | a::b => bind a' = a ; bind b' = mapM b ; return a'::b'
  end.

Fixpoint list_to_ilist {T}{F}(f:forall t:T, F t) (l:list T) : IList T F l :=
  match l as L return IList T F L with
  | nil  => INil
  | a::b => ICons a b (f a) (list_to_ilist f b)
  end.

Fixpoint treeM {T}{M}{MT:Monad M}(t:Tree ??(M T)) : M (Tree ??T) :=
  match t with
    | T_Leaf None     => return []
    | T_Leaf (Some x) => bind x' = x ; return [x']
    | T_Branch b1 b2  => bind b1' = treeM b1 ; bind b2' = treeM b2 ; return (b1',,b2')
```

```
  end.


(* escapifies any characters which might cause trouble for LaTeX *)
Variable sanitizeForLatex    : string      -> string.
   Extract Inlined Constant sanitizeForLatex      => "sanitizeForLatex".
Inductive Latex     : Type := rawLatex     : string -> Latex.
Inductive LatexMath : Type := rawLatexMath : string -> LatexMath.

Class ToLatex (T:Type) := { toLatex : T -> Latex }.
Instance ConcatenableLatex : Concatenable Latex :=
  { concatenate := fun l1 l2 => match l1 with rawLatex l1' => match l2 with rawLatex l2' => rawLatex (l1'+++l2') end end }.
Instance LatexToString : ToString Latex := { toString := fun x => match x with rawLatex s => s end }.

Class ToLatexMath (T:Type) := { toLatexMath : T -> LatexMath }.
Instance ConcatenableLatexMath : Concatenable LatexMath :=
  { concatenate := fun l1 l2 =>
    match l1 with rawLatexMath l1' =>
      match l2 with rawLatexMath l2' => rawLatexMath (l1'+++l2')
      end end }.
Instance LatexMathToString : ToString LatexMath := { toString := fun x => match x with rawLatexMath s => s end }.

Instance ToLatexLatexMath : ToLatex LatexMath := { toLatex     := fun l => rawLatex     ("$"+++toString l+++"$") }.
Instance ToLatexMathLatex : ToLatexMath Latex := { toLatexMath := fun l => rawLatexMath ("\text{"+++toString l+++"}") }.

Instance StringToLatex     : ToLatex string := { toLatex := fun x => rawLatex (sanitizeForLatex x) }.
Instance StringToLatexMath : ToLatexMath  string := { toLatexMath := fun x => toLatexMath (toLatex x) }.
Instance LatexToLatex          : ToLatex     Latex      := { toLatex := fun x => x }.
Instance LatexMathToLatexMath  : ToLatexMath LatexMath  := { toLatexMath := fun x => x }.

Fixpoint treeToLatexMath {V}{ToLatexV:ToLatexMath V}(t:Tree ??V) : LatexMath :=
  match t with
    | T_Leaf None     => rawLatexMath "\langle\rangle"
    | T_Leaf (Some x) => (rawLatexMath "\langle")+++toLatexMath x+++(rawLatexMath "\rangle")
    | T_Branch b1 b2  => (rawLatexMath "\langle")+++treeToLatexMath b1+++(rawLatexMath " , ")
      +++treeToLatexMath b2+++(rawLatexMath "\rangle")
  end.
```