```
(*****************************************************************************************************************************)
(* ExtractionMain:                                                                                                           *)
(*                                                                                                                           *)
(*    This module is the "top level" for extraction                                                                          *)
(*                                                                                                                           *)
(*****************************************************************************************************************************)

Require Import Coq.Strings.Ascii.
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.

Require Import Preamble.
Require Import General.

Require Import NaturalDeduction.

Require Import HaskKinds.
Require Import HaskLiteralsAndTyCons.
Require Import HaskCoreVars.
Require Import HaskCoreTypes.
Require Import HaskCore.
Require Import HaskWeakVars.
Require Import HaskWeakTypes.
Require Import HaskWeak.
Require Import HaskStrongTypes.
Require Import HaskStrong.
Require Import HaskProof.
Require Import HaskCoreToWeak.
Require Import HaskWeakToStrong.
Require Import HaskStrongToProof.
Require Import HaskProofToLatex.
Require Import HaskStrongToWeak.
Require Import HaskWeakToCore.
Require Import HaskProofToStrong.

(*Require Import HaskProofFlattener.*)
(*Require Import HaskProofStratified.*)

Open Scope string_scope.
Extraction Language Haskell.
```

```
(*Extract Inductive vec    => "([])" [ "([])" "(:)" ].*)
(*Extract Inductive Tree   => "([])" [ "([])" "(:)" ].*)
(*Extract Inlined Constant map => "Prelude.map".*)

(* I try to reuse Haskell types mostly to get the "deriving Show" aspect *)
Extract Inductive option => "Prelude.Maybe" [ "Prelude.Just" "Prelude.Nothing" ].
Extract Inductive list   => "([])" [ "([])" "(:)" ].
Extract Inductive string => "Prelude.String" [ "[]" "(:)" ].
Extract Inductive prod   => "(,)" [ "(,)" ].
Extract Inductive sum    => "Prelude.Either" [ "Prelude.Left" "Prelude.Right" ].
Extract Inductive sumbool => "Prelude.Bool" [ "Prelude.True" "Prelude.False" ].
Extract Inductive bool    => "Prelude.Bool" [ "Prelude.True" "Prelude.False" ].
Extract Inductive unit    => "()" [ "()" ].
Extract Inlined Constant string_dec => "(==)".
Extract Inlined Constant ascii_dec => "(==)".

Extract Inductive ascii => "Char" [ "you_forgot_to_patch_coq" ] "you_forgot_to_patch_coq".
Extract Constant zero   => "'\000'".
Extract Constant one    => "'\001'".
Extract Constant shift  => "shiftAscii".

Unset Extraction Optimize.
Unset Extraction AutoInline.

Variable Name : Type.  Extract Inlined Constant Name => "Name.Name".
Variable mkSystemName : Unique -> string -> nat -> Name.
  Extract Inlined Constant mkSystemName =>
    "(\u s d -> Name.mkSystemName u (OccName.mkOccName (OccName.varNameDepth (nat2int d)) s))".
Variable mkTyVar : Name -> Kind -> CoreVar.
  Extract Inlined Constant mkTyVar => "(\n k -> Var.mkTyVar n (kindToCoreKind k))".
Variable mkCoVar : Name -> CoreType -> CoreType -> CoreVar.
  Extract Inlined Constant mkCoVar => "(\n t1 t2 -> Var.mkCoVar n (Coercion.mkCoKind t1 t2))".
Variable mkExVar : Name -> CoreType -> CoreVar.
  Extract Inlined Constant mkExVar => "Id.mkLocalId".

Section core2proof.
  Context (ce:@CoreExpr CoreVar).

  Definition Γ : TypeEnv := nil.

  Definition Δ : CoercionEnv Γ := nil.
```

```
  Definition φ : TyVarResolver Γ :=
    fun cv => Error ("unbound tyvar: " +++ toString (cv:CoreVar)).
    (*fun tv => error ("tried to get the representative of an unbound tyvar:" +++ (getCoreVarOccString tv)).*)

  Definition ψ : CoVarResolver Γ Δ :=
    fun cv => Error ("tried to get the representative of an unbound covar!" (*+++ (getTypeVarOccString cv)*)).

  (* We need to be able to resolve unbound exprvars, but we can be sure their types will have no
   * free tyvars in them *)
  Definition ξ (cv:CoreVar) : LeveledHaskType Γ ★:=
    match coreVarToWeakVar cv with
      | WExprVar wev => match weakTypeToTypeOfKind φ wev ★with
                          | Error s => Prelude_error ("Error converting weakType of top-level variable "+++
                                                      toString cv+++": " +++ s)
                          | OK    t => t @@ nil
                        end
      | WTypeVar _   => Prelude_error "top-level xi got a type variable"
      | WCoerVar _   => Prelude_error "top-level xi got a coercion variable"
    end.


  Definition header : string :=
    "\documentclass{article}"+++eol+++
    "\usepackage{amsmath}"+++eol+++
    "\usepackage{amssymb}"+++eol+++
    "\usepackage{proof}"+++eol+++
(*    "\usepackage{mathpartir}   % http://cristal.inria.fr/~remy/latex/"+++eol+++*)
    "\usepackage{trfrac}       % http://www.utdallas.edu/~hamlen/trfrac.sty"+++eol+++
    "\def\code#1#2{\Box_{#1} #2}"+++eol+++
    "\usepackage[paperwidth=\maxdimen,paperheight=\maxdimen]{geometry}"+++eol+++
    "\usepackage[tightpage,active]{preview}"+++eol+++
    "\begin{document}"+++eol+++
    "\setlength\PreviewBorder{5pt}"+++eol.

  Definition footer : string :=
    eol+++"\end{document}"+++
    eol.

  (* core-to-string (-dcoqpass) *)
  Definition coreToStringExpr' (ce:@CoreExpr CoreVar) : ???string :=
```

```
  addErrorMessage ("input CoreSyn: " +++ toString ce)
  (addErrorMessage ("input CoreType: " +++ toString (coreTypeOfCoreExpr ce)) (
    coreExprToWeakExpr ce >>= fun we =>
      addErrorMessage ("WeakExpr: " +++ toString we)
        ((addErrorMessage ("CoreType of WeakExpr: " +++ toString (coreTypeOfCoreExpr (weakExprToCoreExpr we)))
          ((weakTypeOfWeakExpr we) >>= fun t =>
            (addErrorMessage ("WeakType: " +++ toString t)
              ((weakTypeToTypeOfKind φ t ★) >>= fun τ =>
                addErrorMessage ("HaskType: " +++ toString τ)
                ((weakExprToStrongExpr Γ Δ φ ψ ξ (fun _ => false) τ nil we) >>= fun e =>
                  OK (eol+++eol+++eol+++
                      "\begin{preview}"+++eol+++
                      "$\displaystyle "+++
                      toString (nd_ml_toLatexMath (@expr2proof _ _ _ _ _ _ e))+++
                      " $"+++eol+++
                      "\end{preview}"+++eol+++eol+++eol)
              )))))).

Definition coreToStringExpr (ce:@CoreExpr CoreVar) : string :=
  match coreToStringExpr' ce with
    | OK x => x
    | Error s => Prelude_error s
  end.

Definition coreToStringBind (binds:@CoreBind CoreVar) : string :=
  match binds with
    | CoreNonRec _ e => coreToStringExpr e
    | CoreRec    lbe => fold_left (fun x y => x+++eol+++eol+++y) (map (fun x => coreToStringExpr (snd x)) lbe) ""
  end.

Definition coqPassCoreToString (lbinds:list (@CoreBind CoreVar)) : string :=
  header +++
  (fold_left (fun x y => x+++eol+++eol+++y) (map coreToStringBind lbinds) "")
  +++ footer.


(* core-to-core (-fcoqpass) *)
Section CoreToCore.

  Definition mkWeakTypeVar (u:Unique)(k:Kind)                 : WeakTypeVar :=
    weakTypeVar (mkTyVar (mkSystemName u "tv" 0) k) k.
```

```
  Definition mkWeakCoerVar (u:Unique)(k:Kind)(t1 t2:WeakType) : WeakCoerVar :=
    weakCoerVar (mkCoVar (mkSystemName u "cv" 0) (weakTypeToCoreType t1) (weakTypeToCoreType t2)) k t1 t2.
  Definition mkWeakExprVar (u:Unique)(t:WeakType)             : WeakExprVar :=
    weakExprVar (mkExVar (mkSystemName u "ev" 0) (weakTypeToCoreType t)) t.

  Context (hetmet_brak  : WeakExprVar).
  Context (hetmet_esc   : WeakExprVar).
  Context (uniqueSupply : UniqSupply).

  Definition useUniqueSupply {T}(ut:UniqM T) : ???T :=
    match ut with
      uniqM f =>
       f uniqueSupply >>= fun x => OK (snd x)
    end.

  Definition larger : forall ln:list nat, { n:nat & forall n', In n' ln -> gt n n' }.
    intros.
    induction ln.
    exists 0.
    intros.
    inversion H.
    destruct IHln as [n pf].
    exists (plus (S n) a).
    intros.
    destruct H.
    omega.
    fold (@In _ n' ln) in H.
    set (pf n' H) as q.
    omega.
    Defined.

Definition FreshNat : @FreshMonad nat.
  refine {| FMT       := fun T => nat -> prod nat T
          ; FMT_fresh := _
         |}.
  Focus 2.
  intros.
  refine ((S H),_).
  set (larger tl) as q.
  destruct q as [n' pf].
  exists n'.
```

```
intros.
set (pf _ H0) as qq.
omega.

refine {| returnM := fun a (v:a) => _ |}.
  intro n. exact (n,v).
  intros.
  set (X H) as q.
  destruct q as [n' v].
  set (X0 v n') as q'.
  exact q'.
  Defined.

Definition unFresh {T} : @FreshM _ FreshNat T -> T.
  intros.
  destruct X.
  exact 0.
  apply t.
  Defined.

Definition coreToCoreExpr' (ce:@CoreExpr CoreVar) : ???(@CoreExpr CoreVar) :=
  addErrorMessage ("input CoreSyn: " +++ toString ce)
  (addErrorMessage ("input CoreType: " +++ toString (coreTypeOfCoreExpr ce)) (
    coreExprToWeakExpr ce >>= fun we =>
      addErrorMessage ("WeakExpr: " +++ toString we)
        ((addErrorMessage ("CoreType of WeakExpr: " +++ toString (coreTypeOfCoreExpr (weakExprToCoreExpr we)))
          ((weakTypeOfWeakExpr we) >>= fun t =>
            (addErrorMessage ("WeakType: " +++ toString t)
              ((weakTypeToTypeOfKind φ t ★) >>= fun τ =>

                ((weakExprToStrongExpr Γ Δ φ ψ ξ (fun _ => true) τ nil we) >>= fun e =>

                  (addErrorMessage ("HaskStrong...")
                    (let haskProof := @expr2proof _ _ _ _ _ _ e
                     in (* insert HaskProof-to-HaskProof manipulations here *)
                     OK ((@proof2expr nat _ FreshNat _ _ _ _ (fun _ => Prelude_error "unbound unique") _ haskProof) 0)
                     >>= fun e' =>
                       (snd e') >>= fun e'' =>
                     strongExprToWeakExpr hetmet_brak hetmet_esc mkWeakTypeVar mkWeakCoerVar mkWeakExprVar uniqueSupply
                       (projT2 e'') INil
                     >>= fun q =>
```

```
                        OK (weakExprToCoreExpr q)
                  ))))))))))).

    Definition coreToCoreExpr (ce:@CoreExpr CoreVar) : (@CoreExpr CoreVar) :=
      match coreToCoreExpr' ce with
        | OK x    => x
        | Error s => Prelude_error s
      end.

    Definition coreToCoreBind (binds:@CoreBind CoreVar) : @CoreBind CoreVar :=
      match binds with
        | CoreNonRec v e => CoreNonRec v (coreToCoreExpr e)
        | CoreRec    lbe => CoreRec (map (fun ve => match ve with (v,e) => (v,coreToCoreExpr e) end) lbe)
      end.

    Definition coqPassCoreToCore' (lbinds:list (@CoreBind CoreVar)) : list (@CoreBind CoreVar) :=
      map coreToCoreBind lbinds.

  End CoreToCore.

  Definition coqPassCoreToCore
    (hetmet_brak  : CoreVar)
    (hetmet_esc   : CoreVar)
    (uniqueSupply : UniqSupply)
    (lbinds:list (@CoreBind CoreVar)) : list (@CoreBind CoreVar) :=
    match coreVarToWeakVar hetmet_brak with
      | WExprVar hetmet_brak' => match coreVarToWeakVar hetmet_esc with
                                   | WExprVar hetmet_esc' => coqPassCoreToCore' hetmet_brak' hetmet_esc' uniqueSupply lbinds
                                   | _ => Prelude_error "IMPOSSIBLE"
                                 end
      | _ => Prelude_error "IMPOSSIBLE"
    end.

End core2proof.
```