```
Module ModularArithmetic.
  Definition F (m : positive) := { z : Z | z = z mod (Z.pos m) }.
  Program Definition of_Z  m (a:Z) : F m := a mod (Z.pos m).
  Definition to_Z {m} (a:F m) : Z := proj1_sig a.

  Context {m : positive}.
  Definition zero : F m := of_Z m 0.
  Definition one : F m := of_Z m 1.
  Definition add (a b:F m) : F m := of_Z m (to_Z a + to_Z b).
  Definition mul (a b:F m) : F m := of_Z m (to_Z a * to_Z b).
  Definition opp (a  :F m) : F m := of_Z m (0 - to_Z a).
  Definition sub (a b:F m) : F m := add a (opp b).
  Definition inv_with_spec : { inv : F m → F m
                             | inv zero = zero ∧ ( prime (Z.pos m) →
                                    ∀ a, a ≠ zero → mul (inv a) a = one )
                             } := ModularArithmeticPre.inv_impl.
  Definition inv : F m → F m := Eval hnf in proj1_sig inv_with_spec.
  Definition div (a b:F m) : F m := mul a (inv b).
End ModularArithmetic.

Module EdwardsCurve. (* <https://eprint.iacr.org/2008/013.pdf> *)
  Context `{field:@Algebra.Hierarchy.field F Feq Fzero Fone Fopp Fadd Fsub Fmul Finv Fdiv}
              {char_ge_3 : @Ring.char_ge F Feq Fzero Fone Fopp Fadd Fsub Fmul 3}
        {a : F} {nonzero_a : a ≠ 0} {square_a : ∃ sqrt_a, sqrt_a^2 = a}
        {d : F} {nonsquare_d : ∀ x, x^2 ≠ d} {Feq_dec:Decidable.DecidableRel Feq}.

  Definition point := { xy | let '(x, y) := xy in a*x^2 + y^2 = 1 + d*x^2*y^2 }.
  Definition coordinates (P:point) : (F*F) := let (xy, xy_onCurve_proof) := P in xy.
  Program Definition zero : point := (0, 1).
  Program Definition add (P1 P2:point) : point :=
    match coordinates P1, coordinates P2 return (F*F) with
      (x1, y1), (x2, y2) ⇒
      (((x1*y2  +  y1*x2)/(1 + d*x1*x2*y1*y2)) , ((y1*y2 - a*x1*x2)/(1 - d*x1*x2*y1*y2)))
    end.
  Fixpoint mul (n:ℕ) (P : point) : point :=
    match n with O ⇒ zero | S n' ⇒ add P (mul n' P) end.
End EdwardsCurve.

Module EdDSA. (* <https://eprint.iacr.org/2015/677.pdf> *)
  Class EdDSAParametersOK
        {E Eeq Eadd Ezero Eopp} {EscalarMult} {c:ℕ} {B:E} {l : positive} {n b:ℕ}
        {H : ∀ {t}, word t → word (b + b)} {Eenc:E→word b} {Senc:F l→word b}
  :={ EdDSA_group :> @Algebra.Hierarchy.group E Eeq Eadd Ezero Eopp;
      EdDSA_scalarmult:@Algebra.ScalarMult.is_scalarmult E Eeq Eadd Ezero EscalarMult;
      EdDSA_c_valid : c = 2 ∨ c = 3; EdDSA_n_ge_c : n ≥ c; EdDSA_n_le_b : n ≤ b;
      EdDSA_B_not_identity : not (Eeq B Ezero); EdDSA_l_prime : Znumtheory.prime l;
      EdDSA_l_odd : Z.lt 2 l; EdDSA_l_order_B :> Eeq (EscalarMult (Z.to_nat l) B) Ezero }.
  Context `{prm:EdDSAParametersOK}. Notation signature := (word (b + b)).
  Notation secretkey := (word b).  Notation publickey := (word b).
  Infix "^" := Nat.pow. Local Infix "mod" := Nat.modulo. Infix "++" := Word.combine.
  Infix "+" := Eadd. Local Infix "*" := EscalarMult.
  Coercion wordToNat : word ↣ ℕ. Coercion Z.to_nat : Z ↣ ℕ.

  Program Definition curveKey (sk:secretkey) : ℕ :=
    let x := wfirstn n (H sk) in (* hash the key, use first "half" for secret scalar *)
    let x := x - (x mod (2^c)) in (* it is implicitly 0 mod (2^c) *)
              PeanoNat.Nat.setbit x n. (* and the high bit is always set *)
  Definition public (sk:secretkey) : publickey := Eenc (curveKey sk*B).
  Definition prngKey (sk:secretkey) : word b := Word.split2 b b (H sk).
  Program Definition sign (A_:publickey) sk {n} (M : word n) :=
```

```
    let r : ℕ := H (prngKey sk ++ M) in (* secret nonce *)
    let R : E := r * B in (* commitment to nonce *)
    let s : ℕ := curveKey sk in (* secret scalar *)
    let S : F l := F.nat_mod l (r + H (Eenc R ++ A_ ++ M) * s) in
        Eenc R ++ Senc S.
  Definition valid {n} (message : word n) pubkey signature :=
    ∃ A S R, Eenc A = pubkey ∧ Eenc R ++ Senc S = signature ∧
                  Eeq (F.to_nat S * B) (R + (H (Eenc R ++ Eenc A ++ message) mod l) * A).
End EdDSA.

Section Ed25519.
  Definition q : positive := 2^255 - 19.
  Definition a : F q := F.opp 1. Definition d:F q:=F.opp(F.of_Z _ 121665)/(F.of_Z _ 121666).
  Definition E :=E.point(F:=F q)(Feq:=eq)(Fone:=F.one)(Fadd:=F.add)(Fmul:=F.mul)(a:=a)(d:=d).
  Definition b : ℕ := 256. Definition n : ℕ := b - 2. Definition c : ℕ := 3.
  Definition l : positive := 2^252 + 27742317777372353535851937790883648493.
  Program Definition B : E :=
    (F.of_Z q 15112221349535400772501151409588531511454012693041857206046113283949847762202,
     F.of_Z q 4 / F.of_Z q 5)%F.
  Definition Fencode {len} {m} : F m → word len :=
    λ x : F m ⇒ (NToWord _ (Z.to_N (F.to_Z x))).
  Definition signum (x : F q) := Z.testbit (F.to_Z x) 0.
  Definition Eenc : E → word b := λ P ⇒
    let '(x,y) := E.coordinates P in Word.combine (Fencode (len:=b-1) y) (bit (signum x)).
  Definition Senc : Fl → word b := Fencode (len:=b).
End Ed25519.

Module WeierstrassCurves. (* <https://hyperelliptic.org/EFD/g1p/auto-shortw.html> *)
  Context `{field:@Algebra.Hierarchy.field F Feq Fzero Fone Fopp Fadd Fsub Fmul Finv Fdiv}
          {Feq_dec:Decidable.DecidableRel Feq} {a b:F}
          {char_ge_3:@Ring.char_ge F Feq Fzero Fone Fopp Fadd Fsub Fmul 3}.
  Notation "'∞'" := unit : type_scope. Notation "'∞'" := (inr tt) : core_scope.
  Notation "( x , y )" := (inl (pair x y)). Open Scope core_scope.

  Definition point := { P | match P with (x, y) ⇒ y^2 = x^3 + a*x + b | ∞ ⇒ ⊤ end }.
  Definition coordinates (P:point) : (F*F + ∞) := proj1_sig P.
  Program Definition zero : point := ∞.
  Program Definition add (P1 P2:point) : point :=
    match coordinates P1, coordinates P2 return F*F + ∞ with
    | (x1, y1), (x2, y2) ⇒
      if x1 =? x2
      then
        if y2 =? -y1
        then ∞
        else let k := (3*x1^2+a)/(2*y1) in
             let x3 := k^2-x1-x1 in
             let y3 := k*(x1-x3)-y1 in
             (x3, y3)
      else let k := (y2-y1)/(x2-x1) in
           let x3 := k^2-x1-x2 in
           let y3 := k*(x1-x3)-y1 in
           (x3, y3)
    | ∞, ∞ ⇒ ∞
    | ∞, _ ⇒ coordinates P2
    | _, ∞ ⇒ coordinates P1
    end.
  Fixpoint mul (n:ℕ) (P : point) : point :=
    match n with O ⇒ zero | S n' ⇒ add P (mul n' P) end.
End WeierstrassCurves.
```