Types in mathematical proofs

Georges Gonthier

CENTRE DE RECHERCHE COMMUN
INRIA
MICROSOFT RESEARCH

Legacy

- Interactive theorem proving
- Parametric polymorphism
- Type inference

Some Context

I'm interested in

- interactive proofs
- of mathematical theorems

Finite Group Theory

ISSN 0002-9920

Notices

of the American Mathematical Society

December 2008

Volume 55, Number 11

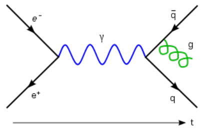
Formal Proof
page 1370

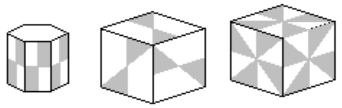
Formal Proof—The Four-Color Theorem
page 1382

Formal Proof—Theory and Practice
page 1395

Formal Proof—Getting Started
page 1408

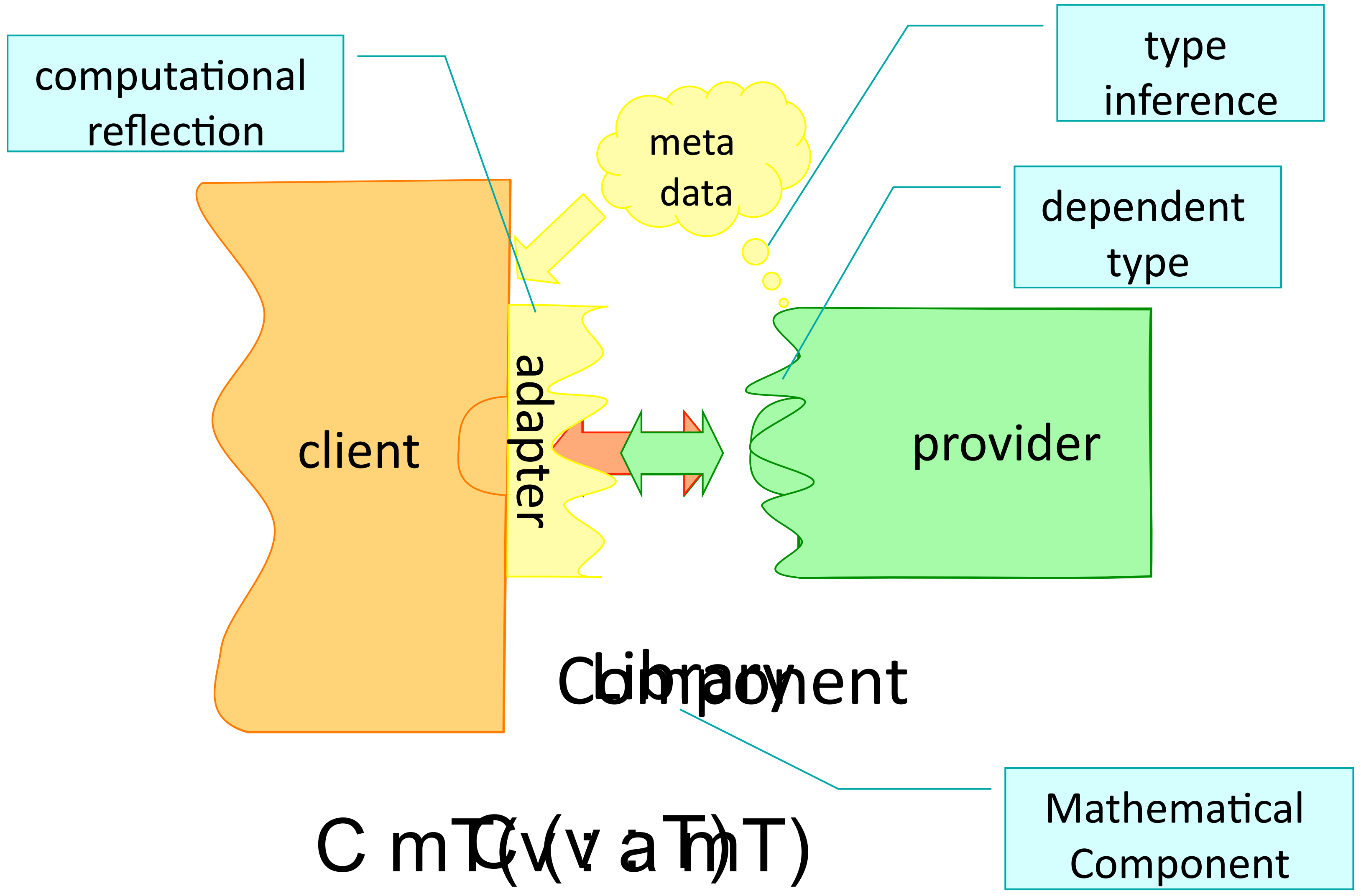
Four-color hypermaps (see page 1427)





CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

The “library” problem



Methodology

- Formalize in the Logic rather than in the Proof Assistant
- Use both type and form to represent intent
- Formalize dynamics (simplification and proof rules) as well as the statics of a theory.

Who does what

- The logic/type theory: CiC
  - Propositions as types: programs as proof
  - Reflection: programs in proofs
- The system: Coq/ssreflect
  - Type reconstruction, term reconstruction
  - Notation / elision
  - Proof scripting : ssreflect
- The library : Ssreflect
  - Components

Tool review

- Data (inductive) types / propositions
- Computational reflection
  - compute values, types, and propositions
- Dependent types
  - first-class Structures
- Type / value inference
  - controlled by Coercion / Canonical / Structure
- User notations

Math vs. Computer Math

$$|AB| = \sum_{\sigma\in S_n} (-1)^{\sigma} \prod_i \Big(\sum_j A_{i,j} B_{j,i\sigma}\Big)$$

$$= \sum_{\rho} \prod_i A_{i,i\rho} \sum_{\sigma\in S_n} (-1)^{\sigma} \prod_i B_{i\rho,i\sigma}$$

$$= \sum_{\rho\in S_n} \prod_i A_{i,i\rho} \sum_{\sigma\in S_n} (-1)^{\sigma} \prod_j B_{j,j\rho^{-1}\sigma} \qquad i = j\rho^{-1}$$

$$+ \sum_{\rho\notin S_n} \prod_i A_{i,i\rho} \sum_{\sigma\in S_n} (-1)^{\sigma} \prod_i B_{i\rho,i\sigma}$$

$$=\Big(\sum_{\rho\in S_n} (-1)^{\rho} \prod_i A_{i,i\rho}\Big) \Big(\sum_{\tau\in S_n} (-1)^{\tau} \prod_j B_{j,j\tau}\Big) \qquad \sigma = \rho\tau$$

$$+ \sum_{\rho\notin S_n} \prod_i A_{i,i\rho} \,|(B_{i\rho,j})|$$

$$= |A|\,|B|$$

Big Operators

$$\sum_{i<n} a_i X^i$$

$$\sum_{d \mid n} \Phi(n/d)\, m^d$$

$$\bigwedge_{i=1}^{n}{}_{GCD}\; Q_i(X)$$

$$\sum_{\sigma \in S_n} (-1)^\sigma \prod_i A_{i,i\sigma}$$

$$\bigcap_{\substack{H<G \\ H \text{ maximal}}} H$$

$$\bigoplus_{V_i \approx W} V_i$$

```
\bigcap_{H < G \atop H {\rm\ maximal}} H
```

```
Definition determinant n (A : 'M_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
```

Notation

```
Definition bigop R I op idx r P (F : I -> R) : R :=
  foldr (fun i x => if P i then op (F i) x else x) idx r.
```

- Present the options

```
Notation "\big[ op / idx ]_ ( i <- r | P ) F" :=
  (bigop op idx r (fun i => P) (fun i => F)).
```

- Hide or fill the options

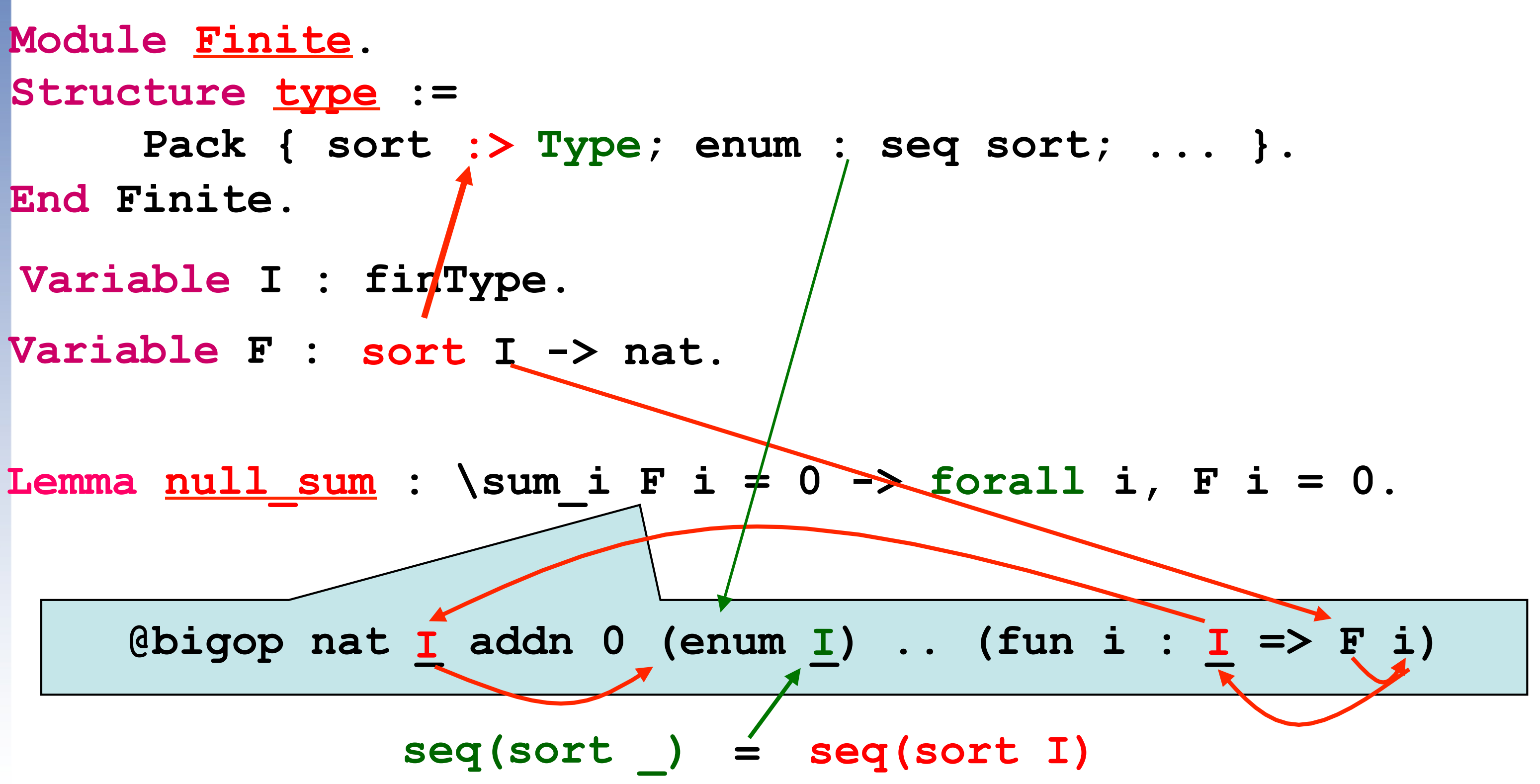
```
Notation "\big[ op / idx ]_ ( i <- r ) F" :=
  (\big[op/idx]_(i <- r | true) F).

Notation "\sum_ ( i <- r ) F" :=
              (\big[addn/0%nat]_(i <- r) F) : nat_scope.
```

- Generic filling

```
Notation "\big[ op / idx ]_ i F" :=
  (\big[op/idx]_(i <- Finite.enum _) F).

Notation "\sum_ i F" :=
  (\big[GRing.add _/GRing.zero _]_i F) : ring_scope.
```

Inferred Notation

- Polymorphism with dependent records

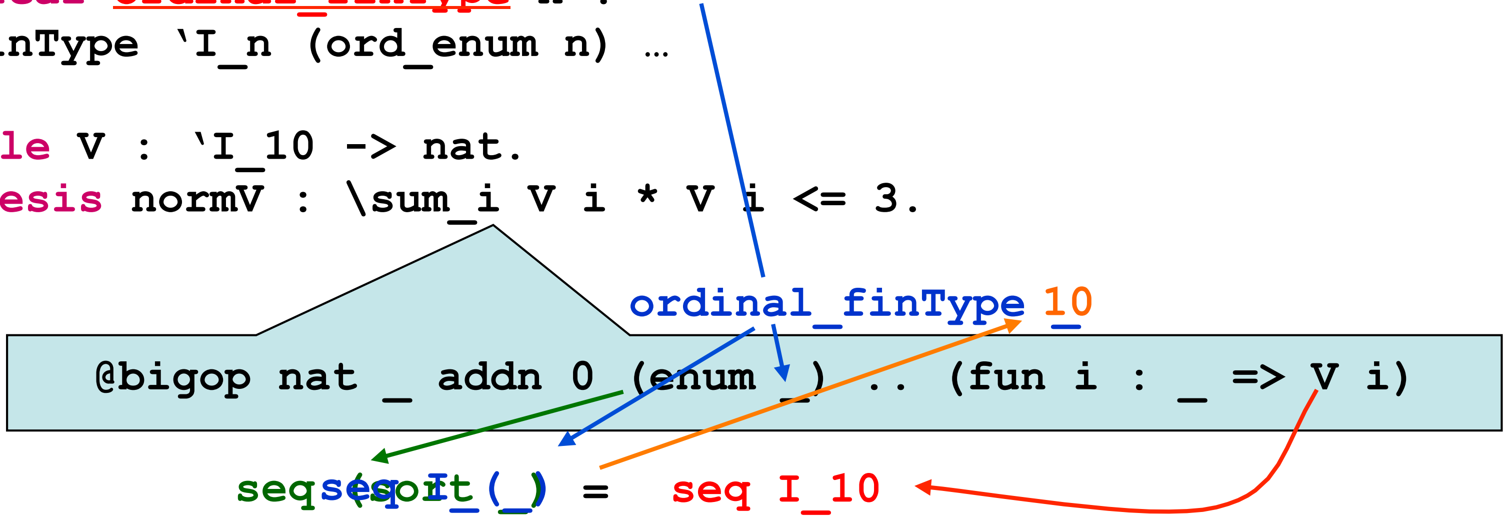


```
Module Finite.
Structure type :=
      Pack { sort :> Type; enum : seq sort; ... }.
End Finite.

 Variable I : finType.

Variable F :  sort I -> nat.


Lemma null_sum : \sum_i F i = 0 -> forall i, F i = 0.


    @bigop nat I addn 0 (enum I) .. (fun i : I => F i)

              seq(sort _)  =  seq(sort I)
```

Canonical Notation

- Use ad hoc interpretation

```
Inductive ordinal n := Ordinal i of i < n.
Notation "`I_ n" := (ordinal n).

Definition ord_enum n : seq `I_n := …
Canonical ordinal_finType n :=
     FinType `I_n (ord_enum n) …

Variable V : `I_10 -> nat.
Hypothesis normV : \sum_i V i * V i <= 3.
```



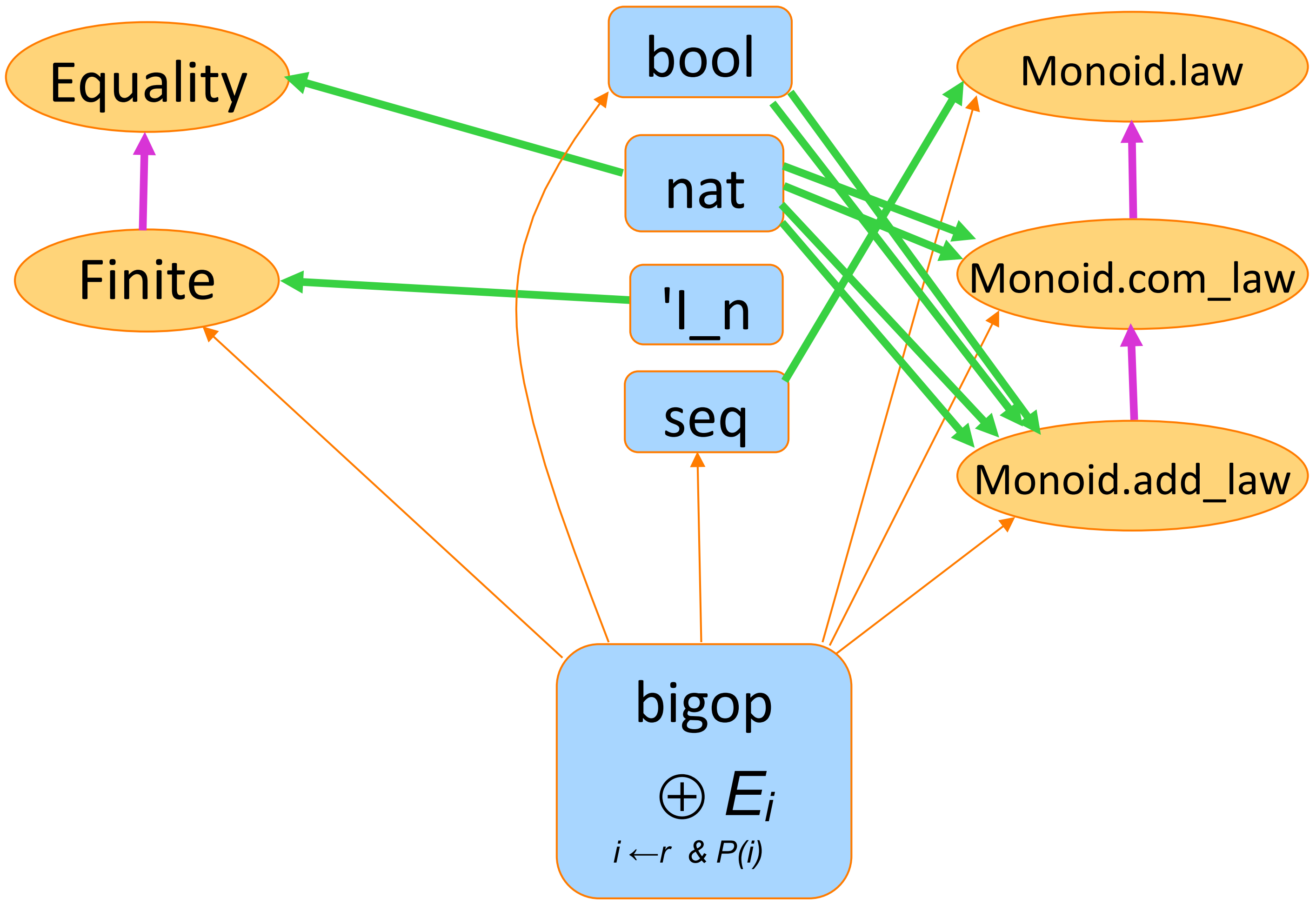
```
@bigop nat _ addn 0 (enum _) .. (fun i : _ => V i)
```

Generic Lemmas

- Pull, split, reindex, exchange …

```
Lemma bigD1 : forall (I : finType) (j : I) P F,
 P j -> \big[*%M/1]_(i | P i) F i
  = F j * \big[*%M/1]_(i | P i && (i != j)) F i.

Lemma big_split : forall I (r : list I) P F1 F2,
 \big[*%M/1]_(i <- r | P i) (F1 i * F2 i) =
  \big[*%M/1]_(i <- r | P i) F1 i * \big[*%M/1]_(i <- r | P i) F2 i.

Lemma reindex : forall (I J : finType) (h : J -> I) P F,
 {on P, bijective h} ->
 \big[*%M/1]_(i | P i) F i = \big[*%M/1]_(j | P (h j)) F (h j).

Lemma bigA_distr_bigA : forall (I J : finType) F,
 \big[*%M/1]_(i : I) \big[+%M/0]_(j : J) F i j
  = \big[+%M/0]_(f : {ffun I -> J}) \big[*%M/1]_(i) F i (f i).
```

Interfacing big ops



Operator structures

- Polymorphism for values!

```
Structure law : Type :=
 Law {
  operator :> T -> T -> T;
  _ : associative operator;
  _ : left_id idx operator;
  _ : right_id idx operator
 }.
```

```
Structure com_law : Type :=
 AbelianLaw {
   com_operator :> law;
   _ : commutative com_operator
 }.
```

```
Canonical addn_monoid := Monoid.Law addnA add0n addn0.
Canonical addn_abeloid := Monoid.ComLaw addnC.
Canonical muln_monoid := Monoid.Law mulnA mul1n muln1.
…

Canonical ring_add_monoid := Monoid.Law addrA add0r addr0.
Canonical ring_add_abeloid := Monoid.ComLaw addrC.
…
```

The Equality interface

```
Module Equality.
Definition axiom T op := forall x y : T, reflect (x = y) (op
   x y).
Record mixin_of T :=
   Mixin {op : rel T; _ : axiom T op}.
Structure type :=
   Pack {sort :> Type; class : mixin_of sort}.
End Equality.
Definition eq_op T := Equality.op (Equality.class T).
Notation eqType := Equality.type.
Notation "x == y" := (eq_op x y).
```

Building up (telescopes)

- Finite (enumerable) types:

```
Structure finType := FinType {
    finCarrier : eqType;
    enum : seq finCarrier; _ : ...}
```

#|T|, #|A|, A \subset B, ...

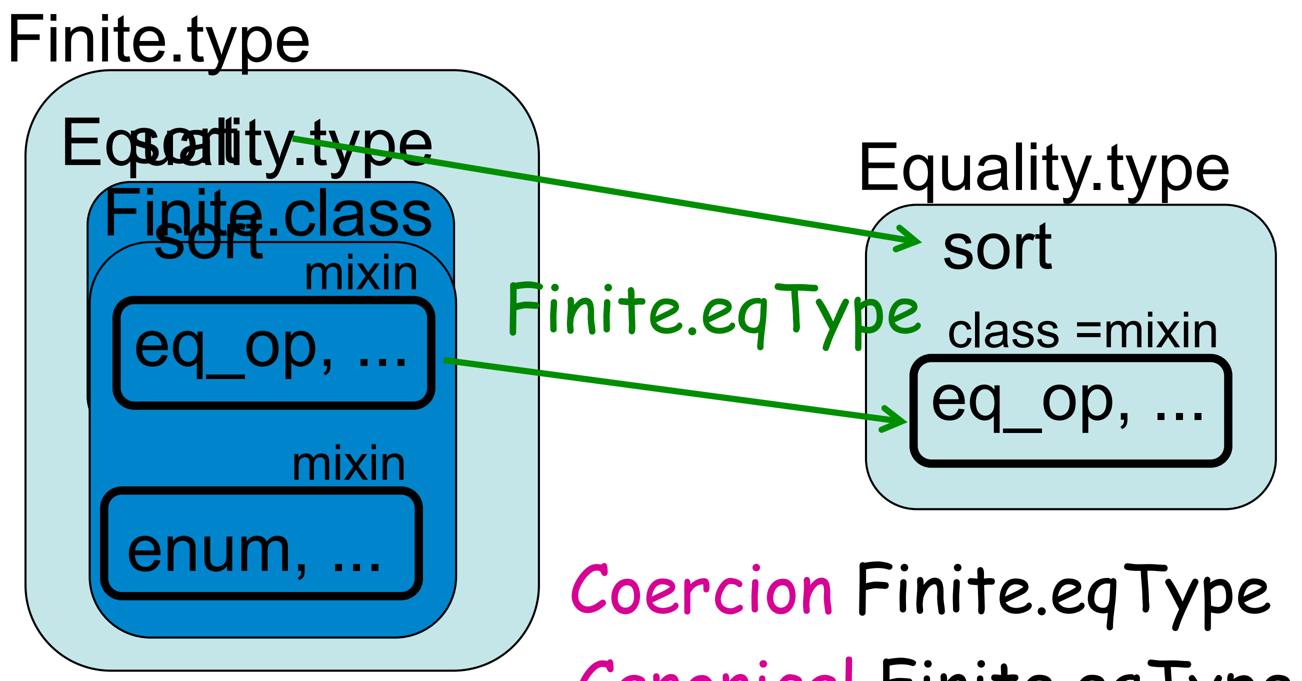
- Finite functions

```
Inductive finfun (aT : finType) rT :=
   Finfun of #|aT|.-tuple rT.
```

Packed classes

- Telescopes are easy to code, but...
  - Single inheritance only
  - Coercion chains: aT ≡ eqSort (finCarrier aT)
  - Wrong head constructor eqSort (finCarrier aT)
- Solution: use classes (dictionaries) and mixins.

Class structures

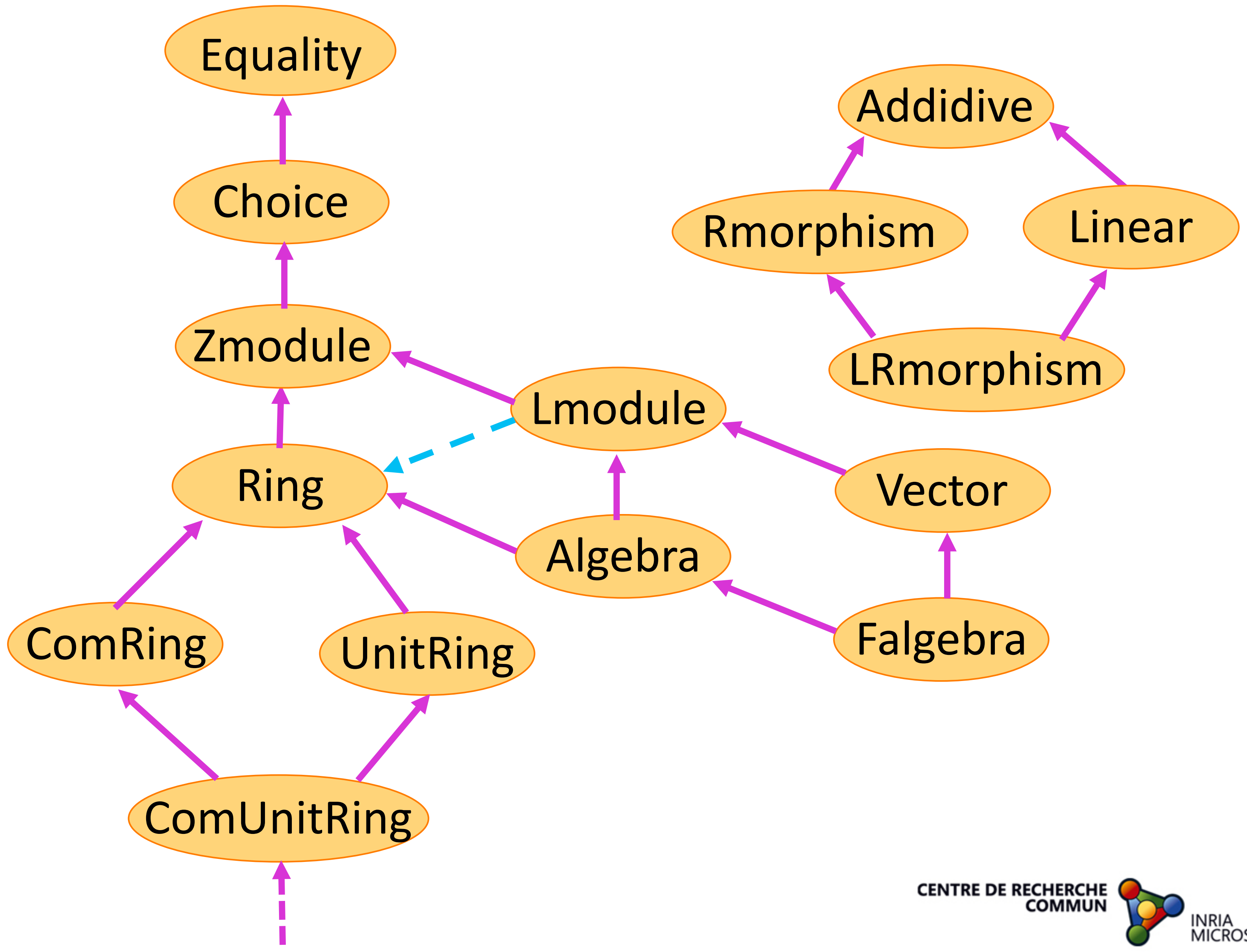


Coercion Finite.eqType

Canonical Finite.eqType

aT ≡ Finite.sort aT≡ Equality.sort (Finite.eqType aT)

Inheritance graph



Linear operator interface

- Encapsulate f (λv) = λ(f v)

```
Module Linear.
Section ClassDef.
Variables (R : ringType) (U V : lmodType R).
Definition mixin_of (f : U -> V)  :=
   forall a, {morph f : u / a *: u}.
Record class_of f : Prop :=
   Class {base : additive f; mixin : mixin_of f}.
Structure map :=
  Pack {apply :> U -> V; class : class_of apply}.
Structure additive cT := Additive (base (class cT)).
End Linear.
```

General linear operators

- Encapsulate f (λv) = λ^σ(f v)

```
Module Linear….
Variables (R : ringType) (U : lmodType R) (V : zmodType).
Variable (s : R -> V -> V).
Definition mixin_of (f : U -> V)  :=
  forall a, {morph f : u / a *: u >-> s a u}.
Record class_of f : Prop :=
  Class {base : additive f; mixin : mixin_of f}.
Structure map := Pack {apply :> Type; class : class_of apply}.
…

 (* horner_morph mulCx_nu P := (map nu P).[x] *)
Fact …:  scalable_for (nu \; *%R) (horner_morph mulCx_nu).
```

General linearity

- Rewrite f (λv) = λ^σ(f v) in *both* directions

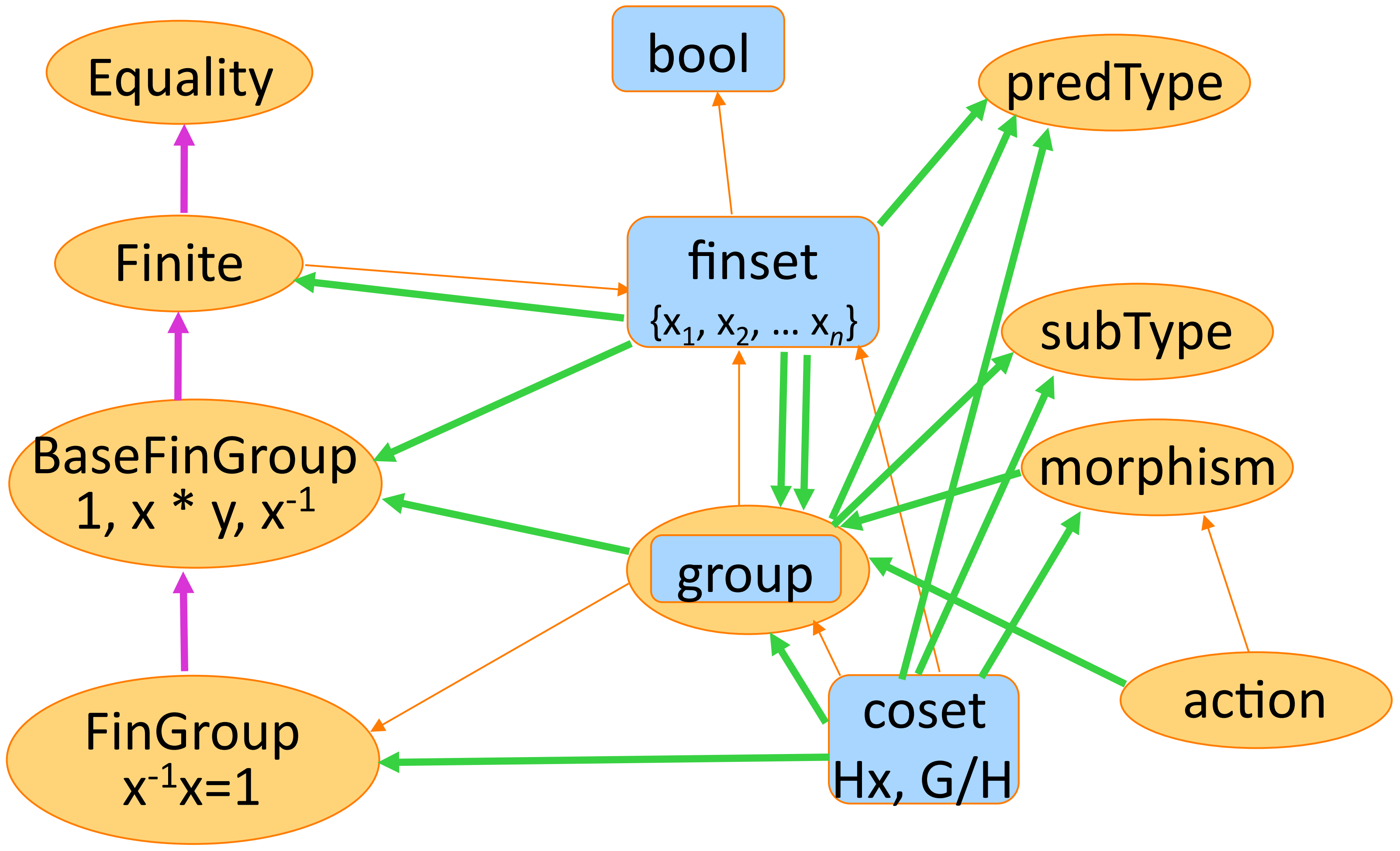
```
Variables (R : ringType) (U : lmodType R) (V : zmodType).
Variables (s : R -> V -> V) (S : ringType) (h : S -> V -> V).
Variable h_law : Scale.law h.

Lemma linearZ c a (h_c := Scale.op h_law c)
                          (f : Linear.map_for U s a h_c) u :
  f (a *: u) = h_c (Linear.wrap f u).
```

Deep matching

- Adjoin a unification constraint to Linear

```
Module Linear. …
Definition map_class := map.
Definition map_at (a : R) := map.
Structure map_for a s_a :=
   MapFor {map_for_map : map; _ : s a = s_a}.
Canonical unify_map_at a (f : map_at a) :=
   MapFor f (erefl (s a)).
Structure wrapped := Wrap {unwrap : map}.
Coercion wrap (f : map_class) := Wrap f.
Canonical wrap.
```

Interfacing groups



A web of basic notions

group H	$\{1\} \cup H^2 = H$
normaliser N (H)	$\{x \quad \mid Hx = xH \text{ (or } H^x = H)\}$
normal subgroup $H \trianglelefteq G$	$H \leq G \leq N \; (H)$
factor group G / H	$\{Hx \mid x \in N_G(H)\}$
morphism $\varphi : G \to H$	$\varphi(xy) = (\varphi x)(\varphi y)$ if $x, y \in G$
action $\alpha : S \to G \to S$	$a(xy)_\alpha = ax_\alpha y_\alpha$ if $x, y \in G$

+ group set A $\quad AB, 1, A^{-1}$ pointwise

+ group type $\quad xy, 1, x^{-1}$

Groups are sets

- Need x∈G & x∈H → groups are not types
- Group theory is really subgroup theory.
- In Coq :

```
Variable gT : finGroupType.
Definition group_set (G : {set gT}) :=
   1∪ G*G ⊆ G.
Structure group :=
   Group { gval :> {set gT}; _ : group_set gval }.
```

Phantom Types

- Matrices from

```
Inductive ordinal n := Ordinal i & i < n.
```

- ?? `matrix T m n := finfun`
  `(pair_finType (ordinal_finType m) (ordinal_finType n))`
  `T`
- Use `Inductive phantom T (x : T) := Phantom.`

```
finfun_of aT rT of phantom (aT -> rT) := finfun aT rT
Notation "{ 'ffun' fT }" := (finfun_of (Phantom fT)).
```

- Now `matrix T m n := {ffun 'I_m * 'I_n -> T}`

Property inference

- The statements only contain sets.

  Theorem third_iso : isog ((G / K) / (H / K)) (G / H).

- The group properties are inferred.

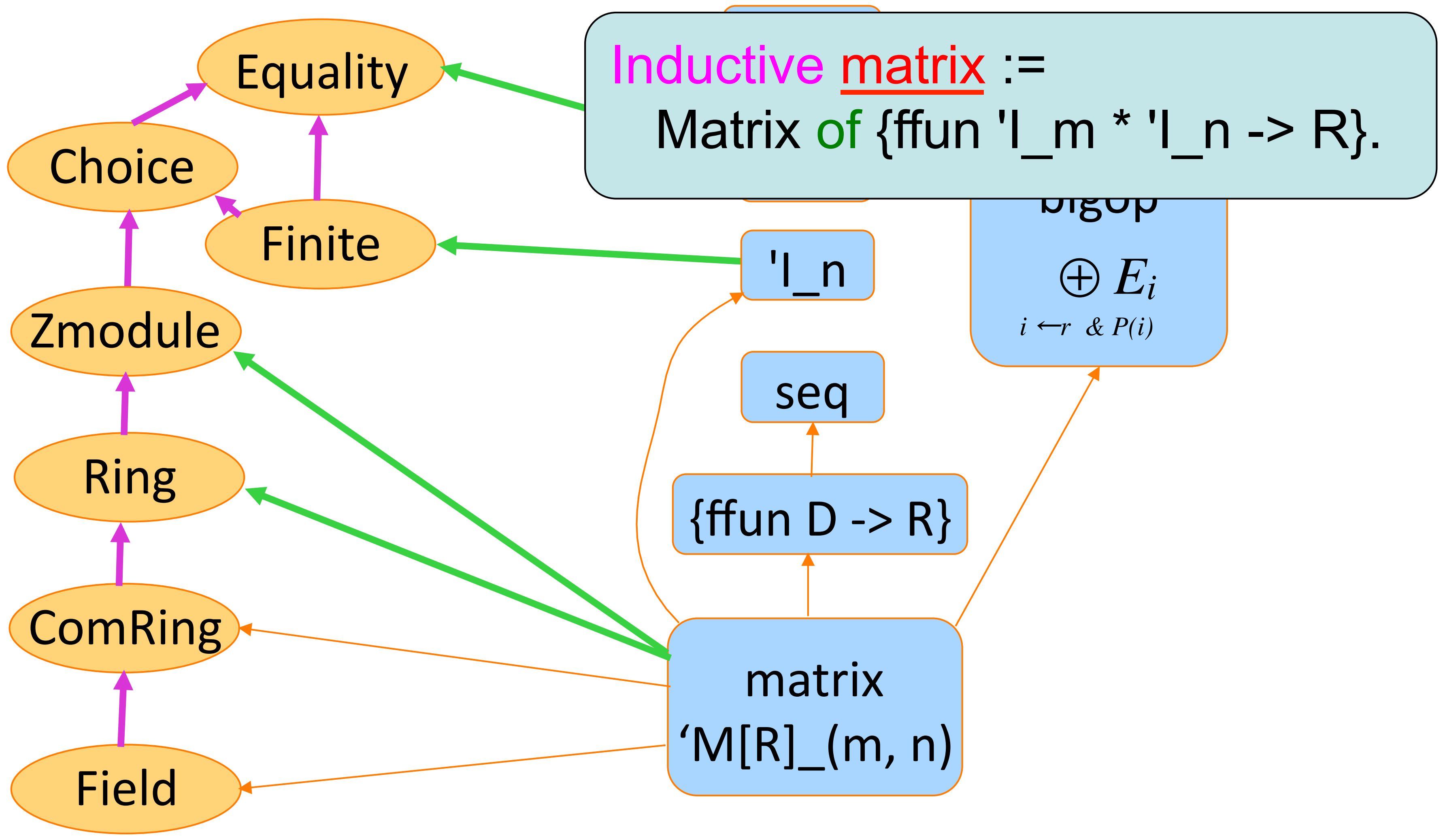
  Canonical Structure setI_group G H :=

  Group (setI_closed G H : group_set (G∩H)).

$$\frac{\text{G, H inst group}}{\text{G}\cap\text{H inst group}} \qquad \frac{\text{H inst group}}{\text{./H inst morphism}}$$

$$\frac{\text{G inst group} \quad \varphi \text{ inst morphism}}{\varphi\text{(G) inst group}}$$

Interfacing matrices



```
Inductive matrix :=
  Matrix of {ffun 'I_m * 'I_n -> R}.
```

Direct sums

- In math:

  $S = A + \sum_i B_i$ is direct

  iff $\text{rank } S = \text{rank } A + \sum_i \text{rank } B_i$

- In Coq:

```
Lemma mxdirectP :
    forall n (S : 'M_n) (E : mxsum_expr S S),
  reflect (\rank E = mxsum_rank E) (mxdirect E).
```

- This is generic in the *shape* of S

Quotation by type inference

```
Structure mxsum := Mxsu
  mxsum_val : 'M_n;
  mxsum_rank : nat;
  _ : mxsum_spec mxsum_
}.

Fact binary_mxsum_proo
   mxsum_spec (mxsum_v
Canonical binary_mx

Canonical trivial_m

Definition mxdirect_def
  \rank (mxsum_val S) ==
Notation mxdirect S := (mxdirect_def (Phantom 'M_n S)).
```

```
Let D := (A + B)%MS.
mxdirect D
     → @mxdirect_def ?S (Phantom 'M_n D)
  unwrap (mxsum_val ?S) ≟ D
             mxsum_val ?S ≟ wrap D
    proper_mxsum_val ?P ≟ D
      unwrap (mxsum_val ?S₁) ≟ A
                 mxsum_val ?S₁ ≟ wrap A
                 mxsum_val ?S₁ ≟ Wrap A
   S₁ ← trivial_mxsum A      S₂ ← trivial_mxsum B
S ← sum_mxsum (binary_mxsum
                       (trivial_mxsum A)
(trivial_mxsum A))
```

```
mx
S
```

```
Let D := (A + B)%MS.
mxdirect D
     → @mxdirect_def ?S (Phantom 'M_n D)
  mxsum_val ?S ≟ D
S ← trivial_mxsum D
```

S).

Circular inequalities

```
rankEP : \rank 1%:M = (\sum_(ZxH \in clPqH) #|ZxH|)%N
 cl1 : 1%g \in clPqH
 dxB : mxdirect (<<B (b 1%g)>> + \sum_(i \in clPqH^#) <<B (b i)>>)
 defB1 : (<<B (b 1%g)>> :=: mxvec 1%:M)%MS
 Bfree : forall x : {set coset_of Z}, x \in clPqH^# -> row_free (B (b x))
 ===============================

...
...
...
```

```
have Bfree_if: forall ZxH, ZxH \in clPqH^# ->
                    \rank <<B (b ZxH)>> <= #|ZxH| ?= iff row_free (B (b ZxH)) by...
have B1_if: \rank <<B (b 1%g)>> <= 1 ?= iff (<<B (b 1%g)>> == mxvec 1%:M)%MS by ...
have rankEP: \rank (1%:M : 'A[F]_q) = (\sum_(ZxH \in clPqH) #|ZxH|)%N by ...
have cl1: 1%g \in clPqH by ...
have{B1_if Bfree_if}:= leqif_add B1_if (leqif_sum Bfree_if).
case/(leqif_trans (mxrank_sum_leqif _)) => _ /=.
rewrite -{1}(big_setD1 _ cl1) sumB {}rankEP (big_setD1 1%g) // cards1 eqxx.
move/esym; case/and3P=> dxB; move/eqmxP=> defB1; move/forall_inP=> /= Bfree.
```

Conclusions

- Advanced mathematics is also a Software Engineering challenge.
- Higher-order type theory provides a rich language for organising formalisations.
- Dependent type reconstruction is user-programmable

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH