Librairy overview - a first guided tour

Laurence Rideau
14 March

MAP INTERNATIONAL SPRING SCHOOL ON FORMALIZATION OF MATHEMATICS 2012

SOPHIA ANTIPOLIS, FRANCE / 12-16 MARCH

Inria informatics mathematics — ihp Institut Henri Poincaré — MICROSOFT RESEARCH INRIA JOINT CENTRE

Outline

http://coqfinitgroup.gforge.inria.fr/ssreflect-1.3/

1. Walk around in natural numbers area
2. About finite objects

Outline

1. Walk around in natural numbers area
   - ssrnat
   - div
   - prime
   - binomial
2. About finite objects

ssrnat: Type and Arithmetic Operators

Inductive type:

- `nat := O | S of nat` standard Coq

ssrnat: Type and Arithmetic Operators

Inductive type:

- `nat := O | S of nat` standard Coq
- Notations: "`0`" and "*n*`.+1`" (generalized till `.+4`).

ssrnat: Type and Arithmetic Operators

Inductive type:

- `nat := O | S of nat` standard Coq
- Notations: "0" and "n.+1" (generalized till .+4).

Arithmetic operators:

- "+" (addn), "-" (subn), "*" (muln), "^" (expn)

ssrnat: Type and Arithmetic Operators

Inductive type:

- `nat := O | S of nat` standard Coq
- Notations: "0" and "n.+1" (generalized till .+4).

Arithmetic operators:

- "+" (addn), "-" (subn), "*" (muln), "^" (expn)
- Convertible to plus, minus, and mult.

ssrnat: Type and Arithmetic Operators

Inductive type:

- `nat := O | S of nat` standard Coq
- Notations: "0" and "n.+1" (generalized till .+4).

Arithmetic operators:

- "+" (`addn`), "-" (`subn`), "*" (`muln`), "^" (`expn`)
- Convertible to `plus`, `minus`, and `mult`.
- Locked to prevent simplification.
  Tactic: "`unlock addn.`"

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$
  - addn1: $n + 1 = n.{+}1$
    add1n: $1 + n = n.{+}1$ till .+4

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$
  - addn1: $n + 1 = n.+1$
    add1n: $1 + n = n.+1$ till .+4
  - addnS: $m + n.+1 = (m + n).+1$
    addSn: $m.+1 + n = (m + n).+1$

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$
  - addn1: $n + 1 = n$.+1
    add1n: $1 + n = n$.+1 — till .+4
  - addnS: $m + n$.+1 $= (m + n)$.+1
    addSn: m.+1 $+ n = (m + n)$.+1
- Other rewriting rules:
  - addnC: $m + n = n + m$ — commutativity
    addnA: $m + (n + p) = (m + n) + p$ — associativity

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$
  - addn1: $n + 1 = n.+1$
    add1n: $1 + n = n.+1$ till .+4
  - addnS: $m + n.+1 = (m + n).+1$
    addSn: $m.+1 + n = (m + n).+1$
- Other rewriting rules:
  - addnC: $m + n = n + m$ commutativity
    addnA: $m + (n + p) = (m + n) + p$ associativity
  - addnK: $(n + p) - p = n$ cancellation
    addKn: $(p + n) - p = n$

ssrnat: Rewriting Rules

- Explicit rewriting rules for simplification:
  - addn0: $n + 0 = n$
    add0n: $0 + n = n$
  - addn1: $n + 1 = n.+1$
    add1n: $1 + n = n.+1$ till .+4
  - addnS: $m + n.+1 = (m + n).+1$
    addSn: $m.+1 + n = (m + n).+1$
- Other rewriting rules:
  - addnC: $m + n = n + m$ commutativity
    addnA: $m + (n + p) = (m + n) + p$ associativity
  - addnK: $(n + p) - p = n$ cancellation
    addKn: $(p + n) - p = n$
  - addnI: $p + m = p + n \Rightarrow m = n$ injectivity
    addIn: $m + p = n + p \Rightarrow m = n$

ssrnat: Comparison Operators

- "<=" leq
  "<" `Notation "m < n":= (m.+1 <= n)`
  ">=" ">"

ssrnat: Comparison Operators

- "<=" leq
  "<" Notation "m < n":= (m.+1 <= n)
  ">=" ">"
- Boolean functions: `leq: nat -> nat -> bool`.
  For propositions, implicitly: (m <= n) = true.

ssrnat: Comparison Operators

- "<=" `leq`
  "<" `Notation "m < n":= (m.+1 <= n)`
  ">=" ">"
- Boolean functions: `leq: nat -> nat -> bool`.
  For propositions, implicitly: `(`m `<=` n`) = true`.
- Reflection with standard Coq:
  - `leP: le` m $n \Leftrightarrow$ `leq` m n `= true`
  - `ltP: lt` m $n \Leftrightarrow$ `leq` m`.+1` n `= true`

ssrnat: Comparison Operators

- Usual properties:
  - leq0n: $0 \leq n$
  - ltn0Sn: $0 < n$.+1

ssrnat: Comparison Operators

- Usual properties:
  - leq0n: $0 \leq n$
  - ltn0Sn: $0 < n.+1$
  - leqnn: $n \leq n$
  - ltnSn: $n < n.+1$
  - leqnSn: $n \leq n.+1$

ssrnat: Comparison Operators

- Usual properties:
  - leq0n: $0 \leq n$
  - ltn0Sn: $0 < n.{+1}$
  - leqnn: $n \leq n$
  - ltnSn: $n < n.{+1}$
  - leqnSn: $n \leq n.{+1}$
  - leq_trans: $m \leq p \Rightarrow p \leq n \Rightarrow m \leq n$ transitivity
  - ltn_trans: $m < p \Rightarrow p < n \Rightarrow m < n$
  - leq_ltn_trans: $m \leq p \Rightarrow p < n \Rightarrow m < n$

ssrnat: Comparison Operators

- Usual properties:
  - leq0n: $0 \leq n$
  - ltn0Sn: $0 < n.+1$
  - leqnn: $n \leq n$
  - ltnSn: $n < n.+1$
  - leqnSn: $n \leq n.+1$
  - leq_trans: $m \leq p \Rightarrow p \leq n \Rightarrow m \leq n$ transitivity
  - ltn_trans: $m < p \Rightarrow p < n \Rightarrow m < n$
  - leq_ltn_trans: $m \leq p \Rightarrow p < n \Rightarrow m < n$

But actually rewriting rules!

div

Divisibility for natural numbers

- Operators:
  - "%/" (divn) quotient
  - "%%" (modn) remainder
  - "%|" (dvdn) divisor predicate

div

Divisibility for natural numbers

- Operators:
  - "%/" (divn) quotient
  - "%%" (modn) remainder
  - "%|" (dvdn) divisor predicate
- Some properties:
  - divn_eq: $m = (m \mathbin{\%/} d) * d + (m \mathbin{\%\%} d)$
  - dvdn_eq: $(d \mathbin{\%|} m) = ((m \mathbin{\%/} d) * d == m)$

div

More definitions

- Definitions using divisibility:
  - gcdn — A function computing the gcd of 2 numbers
  - coprime — `Definition coprime m n := gcdn m n == 1.`

div
More definitions

- Definitions using divisibility:
  - gcdn — A function computing the gcd of 2 numbers
  - coprime — `Definition coprime m n := gcdn m n == 1.`
- The chinese remainder theorem

```
Lemma chinese: forall x y,
  (x == y %[mod m1 * m2]) =
  (x == y %[mod m1]) && (x == y %[mod m2]).
```

prime

- prime p — p is a prime.
- primes m — the sorted list of prime divisors of $m > 1$, else [::].
- prime_decomp m — the list of prime factors of $m > 1$, sorted by primes.
- divisors m — the sorted list of divisors of $m > 0$, else [::].

prime

- prime p — p is a prime.
- primes m — the sorted list of prime divisors of $m > 1$, else [::].
- prime_decomp m — the list of prime factors of $m > 1$, sorted by primes.
- divisors m — the sorted list of divisors of $m > 0$, else [::].

```
Lemma dvdn_divisors :
forall d m, 0 < m -> (d %| m)= (d \in divisors m).
```

prime

- prime p — p is a prime.
- primes m — the sorted list of prime divisors of $m > 1$, else [::].
- prime_decomp m — the list of prime factors of $m > 1$, sorted by primes.
- divisors m — the sorted list of divisors of $m > 0$, else [::].

```
Lemma dvdn_divisors :
forall d m, 0 < m -> (d %| m)= (d \in divisors m).
```

- Φ n — the Euler totient : $\sharp|\{i < n \mid$ i and n are coprime$\}|$.

prime

- prime p — p is a prime.
- primes m — the sorted list of prime divisors of $m > 1$, else [::].
- prime_decomp m — the list of prime factors of $m > 1$, sorted by primes.
- divisors m — the sorted list of divisors of $m > 0$, else [::].

```
Lemma dvdn_divisors :
forall d m, 0 < m -> (d %| m)= (d \in divisors m).
```

- Φ n — the Euler totient : $\sharp|\{i < n \mid \text{i and n are coprime}\}|$.

```
Lemma phi_coprime : forall m n,
coprime m n -> phi (m * n)= phi m * phi n.
```

binomial

- 'C(n, m) the binomial coefficient choose m among n
  a `Fixpoint` definition, using the Pascal's triangle property.

binomial

- 'C(n, m) the binomial coefficient choose m among n
  a `Fixpoint` definition, using the Pascal's triangle property.
- `Lemma bin_factd :`

```
forall n m, 0 < n ->
    'C(n, m)= n`! %/ (m`! * (n - m)`!).
```

binomial

- 'C(n, m) the binomial coefficient choose m among n
  a Fixpoint definition, using the Pascal's triangle property.
- ```
  Lemma bin_factd :
            forall n m, 0 < n ->
                 'C(n, m)= n`! %/ (m`! * (n - m)`!).
  ```
- ```
  Lemma prime_dvd_bin : forall k p,
    prime p -> 0 < k < p -> p %| 'C(p, k).
  ```

Outline

1. Walk around in natural numbers area
2. About finite objects
   - seq
   - fintype
   - tuple
   - finfun
   - finset

seq

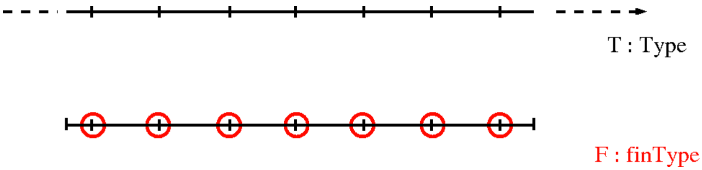
http://coqfinitgroup.gforge.inria.fr/ssreflect-1.3/seq.html

seq

http://coqfinitgroup.gforge.inria.fr/ssreflect-1.3/seq.html

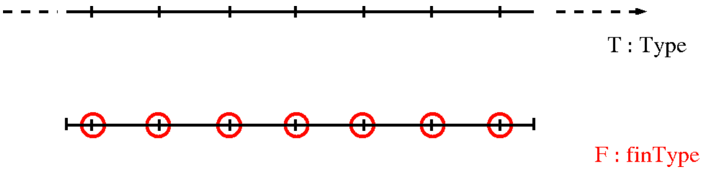
Always read the file header!

fintype



Types with finitely many elements,
supplying a duplicate-free sequence of all the elements.

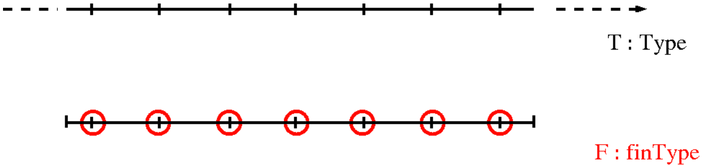
fintype



Types with finitely many elements,
supplying a duplicate-free sequence of all the elements.

- Properties : decidable equality (eqtype), countable, choice.

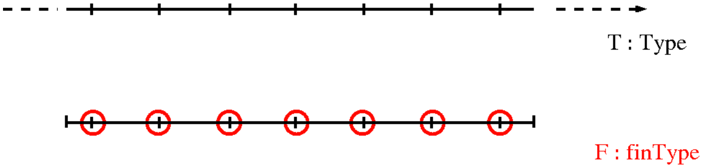
fintype



Types with finitely many elements,
supplying a duplicate-free sequence of all the elements.

- Properties : decidable equality (eqtype), countable, choice.
- Functions: "`card`", "`enum`", "`pick`".

fintype



Types with finitely many elements,
supplying a duplicate-free sequence of all the elements.

- Properties : decidable equality (eqtype), countable, choice.
- Functions: "`card`", "`enum`", "`pick`".
- boolean version of quantifiers: forallb and existb
  with their reflection lemma forallP and existP

Ordinals

Fintype of natural numbers: "'I_n" is $\{k \mid k < n\}$.

- `Ordinal lt_i_n` the element of `'I_n` with (nat) value i
- `ord_enum n` enumeration is 0, ..., n.-1

Ordinals

Fintype of natural numbers: "'I_n" is $\{k \mid k < n\}$.

- Ordinal lt_i_n — the element of 'I_n with (nat) value i
- ord_enum n — enumeration is 0, ..., n.-1
- 'I_n coerces to nat — integer arithmetic available

Ordinals

Fintype of natural numbers: "'I_n" is $\{k \mid k < n\}$.

- Ordinal lt_i_n — the element of 'I_n with (nat) value i
- ord_enum n — enumeration is 0, ..., n.-1
- 'I_n coerces to nat — integer arithmetic available
- ord0

Ordinals

Fintype of natural numbers: "'I_n" is $\{k \mid k < n\}$.

- Ordinal lt_i_n — the element of 'I_n with (nat) value i
- ord_enum n — enumeration is 0, ..., n.-1
- 'I_n coerces to nat — integer arithmetic available
- ord0
- lshift : 'I_n -> 'I_(m + n) — same value
- rshift : 'I_n -> 'I_(m + n) — value i + m.

Ordinals

Fintype of natural numbers: "'I_n" is $\{k \mid k < n\}$.

- `Ordinal lt_i_n` the element of `'I_n` with (nat) value i
- `ord_enum n` enumeration is 0, ..., n.-1
- `'I_n` coerces to nat integer arithmetic available
- `ord0`
- `lshift : 'I_n -> 'I_(m + n)` same value
- `rshift : 'I_n -> 'I_(m + n)` value i + m.
- `split : 'I_(m + n) -> 'I_m + 'I_n`
- `unsplit : 'I_m + 'I_n -> 'I_(m + n)`

tuple

Lists with a fixed (known) length

- `n.-tuple T` the type of n-tuples of elements of type T.
  a sequence s with the proof that (size s = n).

tuple

Lists with a fixed (known) length

- `n.-tuple T` the type of n-tuples of elements of type T.
  a sequence s with the proof that (size s = n).
- Coerces to `(seq T)`,
  (i.e. all operations for seq (size, nth, ...) are available)

tuple

Lists with a fixed (known) length

- `n.-tuple T` the type of n-tuples of elements of type T.
  a sequence s with the proof that (size s = n).
- Coerces to `(seq T)`,
  (i.e. all operations for seq (size, nth, ...) are available)
- `[tuple x1; ..; xn]` the explicit n.-tuple `<x1; ..; xn>`.

tuple

Lists with a fixed (known) length

- `n.-tuple T` the type of n-tuples of elements of type T.
  a sequence s with the proof that (size s = n).
- Coerces to `(seq T)`,
  (i.e. all operations for seq (size, nth, ...) are available)
- `[tuple x1; ..; xn]` the explicit n.-tuple `<x1; ..; xn>`.
- `[tuple of s]` where s has a known size.

tuple

Lists with a fixed (known) length

- `n.-tuple T` the type of n-tuples of elements of type T. a sequence s with the proof that (size s = n).
- Coerces to `(seq T)`, (i.e. all operations for seq (size, nth, ...) are available)
- `[tuple x1; ..; xn]` the explicit n.-tuple `<x1; ..; xn>`.
- `[tuple of s]` where s has a known size.

A central element for the definition of finite functions!

finfun

`{ffun aT -> rT}` : Type for functions `(aT -> rT)` where aT is a finType structure.

finfun

`{ffun aT -> rT}` : Type for functions `(aT -> rT)` where aT is a finType structure.

A finfun is given by his graph: $\#|aT|.-tupleT$.

finfun

`{ffun aT -> rT}` : Type for functions `(aT -> rT)` where aT is a finType structure.

A finfun is given by his graph: $\#|aT|.-tupleT$.

- `[ffun x => expr]` to build the finfun associated to `(fun x => expr)`

finfun

`{ffun aT -> rT}` : Type for functions `(aT -> rT)` where aT is a finType structure.

A finfun is given by his graph: $\#|aT|.-tupleT$.

- `[ffun x => expr]` to build the finfun associated to `(fun x => expr)`
- `Lemma fgraph_map : forall f : fT, fgraph f = [tuple of map f (enum aT)].`

finfun

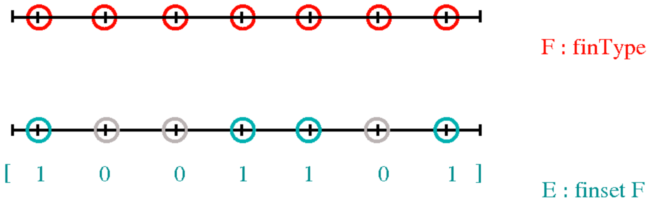
`{ffun aT -> rT}` : Type for functions `(aT -> rT)` where aT is a finType structure.

A finfun is given by his graph: $\#|aT|. - tupleT$.

- `[ffun x => expr]` to build the finfun associated to `(fun x => expr)`
- `Lemma fgraph_map : forall f : fT, fgraph f = [tuple of map f (enum aT)].`
- `Lemma ffunE : forall f : aT -> rT, finfun f =1 f.`

finset

- Sets over a finite Type
- The finsets are finite functions with boolean values.



finset

- Type of finsets: `{set T}` where T has a finType structure.

finset

- Type of finsets: `{set T}` where T has a finType structure.
- the type `{set T}` itself is equipped with a finType structure
  $\Rightarrow$ we get equality, and we can form `{set {set T}}`

finset

- Type of finsets: `{set T}` where T has a finType structure.
- the type `{set T}` itself is equipped with a finType structure
  ⇒ we get equality, and we can form `{set {set T}}`
- x \in A , `mem A x` belonging predicate

finset

- Type of finsets: `{set T}` where T has a finType structure.
- the type `{set T}` itself is equipped with a finType structure
  ⇒ we get equality, and we can form `{set {set T}}`
- x \in A , `mem A x` — belonging predicate
- `[set x | C]` — the set of x such that C holds
- `[set x1; ..; xn]` — the explicit set `<x1; ..; xn>`.

finset

- Type of finsets: `{set T}` where T has a finType structure.
- the type `{set T}` itself is equipped with a finType structure
  ⇒ we get equality, and we can form `{set {set T}}`
- x \in A , `mem A x` — belonging predicate
- `[set x | C]` — the set of x such that C holds
- `[set x1; ..; xn]` — the explicit set <x1; ..; xn>.
- `set0` — the empty set
- `A :|: B , A :&: B , A :\: B, ~: A`
  Union, Intersection, Difference and Complement
- `x |: A , A \: x` — add, remove an element

finset

- Type of finsets: `{set T}` where T has a finType structure.
- the type `{set T}` itself is equipped with a finType structure
  ⇒ we get equality, and we can form `{set {set T}}`
- `x \in A` , `mem A x` — belonging predicate
- `[set x | C]` — the set of x such that C holds
- `[set x1; ..; xn]` — the explicit set `<x1; ..; xn>`.
- `set0` — the empty set
- `A :|: B` , `A :&: B` , `A :\: B`, `~: A`
  Union, Intersection, Difference and Complement
- `x |: A` , `A \: x` — add, remove an element
- and a lot of lemmas
  (!! *naming conventions at the end of the file header* !!)