Librairy overview - a first guided tour

Laurence Rideau
14 March
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:

\[
\text{nat} := 0 \mid S \text{ of } \text{nat}
\]
ssrnat: Type and Arithmetic Operators

Inductive type:

- \( \text{nat} := 0 \mid S \text{ of nat} \)  
  standard Coq

- Notations: "0" and "\( n + 1 \)" (generalized till \( .+4 \)).
ssrnat: Type and Arithmetic Operators

Inductive type:

- \texttt{nat} := 0 \mid S \ of \ \texttt{nat}  

Notations: "0" and "n.+1" (generalized till .+4).

Arithmetic operators:

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

Laurence Rideau
Librairy overview - a first guided tour
**ssrnat: Type and Arithmetic Operators**

Inductive type:
- \( \text{nat} := 0 \mid S \text{ of nat} \) 

- 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:

- \( \text{nat} := 0 \mid S \text{ of nat} \)  
- 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."
 Explicit rewriting rules for simplification:

- \textit{addn0}: \( n + 0 = n \)
- \textit{add0n}: \( 0 + n = n \)
Explicit rewriting rules for simplification:

- **addn0**: \( n + 0 = n \)
- **add0n**: \( 0 + n = n \)
- **addn1**: \( n + 1 = n + 1 \)
- **add1n**: \( 1 + n = n + 1 \)
ssrnat: Rewriting Rules

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

- **Other rewriting rules**:
  - addnC: \( m + n = n + m \) \hspace{1cm} \text{commutativity}
  - addnA: \( m + (n + p) = (m + n) + p \) \hspace{1cm} \text{associativity}
ssrnat: Rewriting Rules

Explicit rewriting rules for simplification:

- **addn0**: $n + 0 = n$
- **add0n**: $0 + n = n$
- **addn1**: $n + 1 = n + 1$
- **add1n**: $1 + n = n + 1$
- **addnS**: $m + n + 1 = (m + n) + 1$
- **addSn**: $m + 1 + n = (m + n) + 1$

Other rewriting rules:

- **addnC**: $m + n = n + m$  \[\text{commutativity}\]
- **addnA**: $m + (n + p) = (m + n) + p$  \[\text{associativity}\]
- **addnK**: $(n + p) - p = n$  \[\text{cancellation}\]
- **addKn**: $(p + n) - p = n$
Explicit rewriting rules for simplification:

- **addn0**: $n + 0 = n$
- **add0n**: $0 + n = n$
- **addn1**: $n + 1 = n + 1$
- **add1n**: $1 + n = n + 1$
- **addnS**: $m + n + 1 = (m + n) + 1$
- **addSn**: $m + 1 + n = (m + n) + 1$

Other rewriting rules:

- **addnC**: $m + n = n + m$ \hspace{1cm} \text{commutativity}$
- **addnA**: $m + (n + p) = (m + n) + p$ \hspace{1cm} \text{associativity}$
- **addnK**: $(n + p) - p = n$ \hspace{1cm} \text{cancellation}$
- **addKn**: $(p + n) - p = n$
- **addnI**: $p + m = p + n \Rightarrow m = n$ \hspace{1cm} \text{injectivity}$
- **addIn**: $m + p = n + p \Rightarrow m = n$
ssrnat: Comparison Operators

- "\(\leq\)"
- "\(<\)"
- "\(\geq\)"  

**Notation** "\(m < n\)" := (\(m + 1 \leq n\) )
ssrnat: Comparison Operators

- "<="
- "<"
- ">="  ">

Notation "m < n" := (m.+1 <= n)

For propositions, implicitly: (m <= n) = true.
ssrnat: Comparison Operators

- "\less_equal"
- "\less"
- "\greater_equal"

\textbf{Notation "m < n"} := (m.+1 \leq n)

\textbf{Boolean functions: leq: nat -> nat -> bool.}
For propositions, implicitly: \((m \leq n) = \text{true}\).

\textbf{Reflection with standard Coq:}
- leP: \(\text{le } m n \Leftrightarrow \text{leq } m n = \text{true}\)
- ltP: \(\text{lt } m n \Leftrightarrow \text{leq } m.+1 n = \text{true}\)
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:** $0 \leq n$
- **lt0Sn:** $0 < n + 1$
- **leqnn:** $n \leq n$
- **ltSn:** $n < n + 1$
- **leqnSn:** $n \leq n + 1$
- **leq_trans:** $m \leq p \Rightarrow p \leq n \Rightarrow m \leq n$ \hspace{1cm} transitivity
- **lt_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:
  - \( \text{leq0n}: \ 0 \leq n \)
  - \( \text{lt0Sn}: \ 0 < n + 1 \)
  - \( \text{leqnn}: \ n \leq n \)
  - \( \text{ltSn}: \ n < n + 1 \)
  - \( \text{leqSn}: \ n \leq n + 1 \)
  - \( \text{leqtrans}: \ m \leq p \Rightarrow p \leq n \Rightarrow m \leq n \) (transitivity)
  - \( \text{lttrans}: \ m < p \Rightarrow p < n \Rightarrow m < n \)
  - \( \text{leqlttrans}: \ m \leq p \Rightarrow p < n \Rightarrow m < n \)

But actually rewriting rules!
div

Divisibility for natural numbers

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

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

- Some properties:
  - divn_eq: $m = (m \%/ d) \ast d + (m \%% d)$
  - dvdn_eq: $(d \%| m) = ((m \%/ d) \ast d == m)$
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:
  - \texttt{gcdn} \quad \text{A function computing the gcd of 2 numbers}
  - \texttt{coprime} \quad \text{Definition} \ coprime \ m \ n := \ gcdn \ m \ n == 1.

- The chinese remainder theorem

  \textbf{Lemma} \ chinese: \ \texttt{forall} \ x \ y, \\
  \quad (x == y \mod m1 * m2) = \\
  \quad (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 \( m \) \( m > 1 \), the sorted list of prime divisors of \( m > 1 \), else [::].
- prime_decomp \( m \) \( m > 1 \), the list of prime factors of \( m > 1 \), sorted by primes.
- divisors \( m \) \( m > 0 \), the sorted list of divisors of \( m > 0 \), else [::].

**Lemma dvdn_divisors** :
forall \( d m \), 0 < \( m \) -> (d \( \% \mid m \)) = (d \in \text{divisors} \( m \)).

- \( \Phi n \) \( n \) the Euler totient : \#\{i < n \mid i \text{ and } n \text{ are coprime}\}.

Laurence Rideau
Library overview - a first guided tour
prime

- prime p
  - p is a prime.
- primes m
  - the sorted list of prime divisors of \( m > 1 \), else \( \vvdots \).
- 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 \( \vvdots \).

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

- \( \Phi n \)
  - the Euler totient : \#\{i < n | i and n are coprime\}.

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

- \( \binom{n}{m} \)  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 :

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

Laurence Rideau Library overview - a first guided tour
binomial

- \( \binom{n}{m} \) the binomial coefficient choose \( m \) among \( n \) a Fixpoint definition, using the Pascal’s triangle property.

- **Lemma** bin_factd :
  
  \[
  \forall n \ m, \ 0 < n \rightarrow \binom{n}{m} = n!' \div (m!' \times (n - m)')!.
  \]

- **Lemma** prime_dvd_bin : \( \forall k \ p, \) prime \( p \rightarrow 0 < k < p \rightarrow p \mid \binom{p}{k}. \)
Outline

1. Walk around in natural numbers area

2. About finite objects
   - seq
   - fintype
   - tuple
   - finfun
   - finset
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".
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 \texttt{lt\_i\_n} the element of 'I\_n with (nat) value i
- \texttt{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 \( \text{lt}_i_n \) is the element of ’I_n with (nat) value i
- \( \text{ord_enum} \ n \) enumeration is 0, ..., n.-1
- ’I_n coerces to nat integer arithmetic available
- \( \text{ord0} \)
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.
Fintype of natural numbers: "'I_n" is \( \{ k \mid k < n \} \).

- Ordinal \( \text{lt}_i_n \) the element of 'I_n with (nat) value i
- \( \text{ord_enum} \ n \) enumeration is 0, ..., n.-1
- 'I_n coerces to nat integer arithmetic available
- \( \text{ord0} \)
- \( \text{lshift}: 'I_n \rightarrow 'I_(m + n) \) same value
- \( \text{rshift}: 'I_n \rightarrow 'I_(m + n) \) value i + m.
- \( \text{split}: 'I_(m + n) \rightarrow 'I_m + 'I_n \)
- \( \text{unsplit}: 'I_m + 'I_n \rightarrow 'I_(m + n) \)
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).
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)
tuples

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>.  

Laurence Rideau Library overview - a first guided tour
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.
tuples

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 \( (\text{size } s = n) \).
- Coerces to \( (\text{seq } T) \),
  (i.e. all operations for seq (size, nth, ...) are available)
- \([\text{tuple } x_1; \ldots; x_n]\) the explicit \( n \)-tuple \( \langle x_1; \ldots; x_n \rangle \).
- \([\text{tuple of } s]\) where \( s \) has a known size.

A **central** element for the definition of finite functions!
A finfun is given by its graph: 

\[
\begin{array}{c}
aT \\
\end{array} 
\rightarrow 
\begin{array}{c}
tupleT \\
\end{array} 
\]

The expression 

\[
[f \Rightarrow \text{expr}] 
\]

to build the finfun associated to 

\( f : aT \rightarrow rT \).

Lemma fgraph_map : \( f : aT \rightarrow rT \), 

\( \text{fgraph} f = \text{tuple of map} f (\text{enum} aT) \).

Lemma ffunE : \( f : aT \rightarrow rT \), 

\( \text{finfun} f = f \).
A finfun is given by his graph: \( \{ ffun aT \to rT \} \)
{ffun aT -> rT} : Type for functions (aT -> rT) where aT is a finType structure.

A finfun is given by his graph: #|aT|. \(\sim\) tupleT.

- [ffun x => expr]
  to build the finfun associated to (fun x => expr)
{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)].
\{\textit{ffun} \ aT \rightarrow \ rT\} : \text{Type for functions (aT \rightarrow \ rT)}
where aT is a \text{finType} structure.

A finfun is given by his graph: \#|aT|. – \textit{tupleT}.

- \[\text{ffun} \ x \Rightarrow \text{expr}\]
  to build the finfun associated to (fun x => expr)

- \textbf{Lemma} \ fgraph\_map : \textit{forall} \ f : \textit{fT},
  fgraph \ f = [\text{tuple of map} \ f \ (\text{enum aT})].

- \textbf{Lemma} \ ffun\_E : \textit{forall} \ f : \textit{aT} \rightarrow \textit{rT}, \textit{finfun} \ f =1 \ f.
Sets over a **finite Type**

- The finsets are finite functions with boolean values.

\[
F : \text{finType} \\
E : \text{finset } F
\]
finset

- Type of finsets: \( \{ \text{set } T \} \) where \( T \) has a \texttt{finType} structure.
finset

- Type of finsets: \( \{ \text{set } T \} \) where \( T \) has a finType structure.

- the type \( \{ \text{set } T \} \) itself is equipped with a finType structure
  \( \Rightarrow \) we get equality, and we can form \( \{ \text{set } \{ \text{set } T \} \} \)
finset

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

- the type \{set T\} itself is equipped with a finType structure
  \[\Rightarrow\] we get equality, and we can form \{set \{set T\}\}

- \(x \in A, \text{mem } A x\) belonging predicate
finset

- Type of finsets: \{\text{set } T\} where $T$ has a \texttt{finType structure}.
- the type \{\text{set } T\} itself is equipped with a finType structure
  \Rightarrow we get equality, and we can form \{\text{set } \{\text{set } T\}\}
- $x \ \text{\texttt{\textbackslash in}} \ A$, \text{mem } A \ x \ \text{belonging predicate}
- \texttt{[set x \mid C]} \ \text{the set of } x \ \text{such that } C \ \text{holds}
- \texttt{[set x1; ..; xn]} \ \text{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
- \[set x \mid C\] the set of x such that C holds
- \[set x1; ..; xn\] the explicit set \(<x1; ..; xn>\).
- set0 the empty set
- x :|: A , A \: x add, remove an element
Type of finsets: \{\text{set } T\} where \(T\) has a \text{finType} structure.

the type \{\text{set } T\} itself is equipped with a \text{finType} structure

\Rightarrow \text{we get equality, and we can form } \{\text{set } \{\text{set } T\}\}\}

\(x \ \text{\textbackslash in } A\), \text{mem } A \ x \ \text{belonging predicate}

\{\text{set } x \ | \ C\} \ \text{the set of } x \ \text{such that } C\ \text{holds}

\{\text{set } x1; \ldots; xn\} \ \text{the explicit set } <x1; \ldots; xn>.

\text{set0} \ \text{the empty set}

\(A :|: B\), \(A :&: B\), \(A :\setminus: B\), \(\sim : A\)

\text{Union, Intersection, Difference and Complement}

\(x \ |: A\), \(A \ \setminus : x\) \ \text{add, remove an element}

and a lot of \text{lemmas}

(!! \text{naming conventions at the end of the file header} !!)