Third Lesson

Programming with Coq

Typed language

Typed language

Defining new objects

$$o : T$$

Typed language

Defining new objects

$$o : T$$

Checking

$$\texttt{Check } o.$$

Typed language

Defining new objects

$$o : T$$

Checking

$$\texttt{Check } o.$$

$$o : T$$

Functional language

Functional language

Defining functions

$$\texttt{fun}\ x\ :\ T \Rightarrow x\ :\ T \rightarrow T$$

Functional language

Defining functions

$$\texttt{fun}\ x\ :\ T\ \Rightarrow\ x\ :\ T \rightarrow T$$

Checking

$$\texttt{Check fun}\ x\ :\ T\ \Rightarrow\ x.$$

Functional language

Defining functions

$$\texttt{fun}\ x\ :\ T\ \Rightarrow\ x\ :\ T \rightarrow T$$

Checking

$$\texttt{Check fun}\ x\ :\ T\ \Rightarrow\ x.$$

$$\texttt{fun}\ x\ :\ T\ \Rightarrow\ x\ :\ T \rightarrow T$$

Defining new objects

Defining new objects

`Definition` $name$: $type$:= $body$.

Defining new objects

`Definition` *name* : *type* := *body*.

`Check` *name*.

Defining new objects

`Definition` *name* : *type* := *body*.

`Check` *name*.

name : *type*

Defining new objects

`Definition` *name* : *type* := *body*.

`Check` *name*.

name : *type*

`Print` *name*.

Defining new objects

`Definition` *name* : *type* := *body*.

`Check` *name*.

name : *type*

`Print` *name*.

name = *body*

Defining objects

Defining objects

```
Definition five : nat := 5.
```

Defining objects

```
Definition five : nat := 5.

Check five.
```

Defining objects

```
Definition five : nat := 5.

Check five.
  five :  nat
```

Defining objects

```
Definition five : nat := 5.

Check five.

  five :  nat

Print five.
```

Defining objects

```
Definition five : nat := 5.

Check five.

  five : nat

Print five.

  five = 5
```

Defining functions

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.

Check double.
```

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

```
Check double.
```

```
  double : nat → nat
```

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

```
Check double.
```

```
  double : nat → nat
```

```
Print double.
```

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

```
Check double.
```

```
  double : nat → nat
```

```
Print double.
```

```
  double = fun x : nat ⇒ x + x
```

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

Defining functions

$\texttt{Definition double : nat} \longrightarrow \texttt{nat} :=$
$\quad \texttt{fun}\ x\ \texttt{: nat} \Rightarrow x + x.$

Using arguments

$\texttt{Definition double}\ (x : \texttt{nat})\ \texttt{: nat} \longrightarrow \texttt{nat} :=$
$\quad x + x.$

Defining functions

Definition double : nat $\longrightarrow$ nat :=
  fun x : nat $\Rightarrow$ x + x.

Using arguments

Definition double (x : nat) : nat $\longrightarrow$ nat :=
  x + x.

Omitting types

Definition double x := x + x.

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

Using arguments

```
Definition double (x : nat) : nat → nat :=
    x + x.
```

Omitting types

```
Definition double x :=  x + x.
```

Defining functions

Definition double : nat $\longrightarrow$ nat :=
    fun x : nat $\Rightarrow$ x + x.

Using arguments

Definition double (x : nat) : nat $\longrightarrow$ nat :=
    x + x.

Omitting types

Definition double x := x + x.

Defining functions

```
Definition double : nat → nat :=
    fun x : nat ⇒ x + x.
```

Using arguments

```
Definition double (x : nat) : nat → nat :=
    x + x.
```

Omitting types

```
Definition double x := x + x.
```

Proving with definitions

Proving with definitions

Definitions are transparent

Proving with definitions

Definitions are transparent

```
Check erefl.
  erefl : forall (A : Type) (x : A), x = x
```

Proving with definitions

Definitions are transparent

```
Check erefl.
  erefl : forall (A : Type) (x : A), x = x
```

```
Lemma triv10 : double five = 5 + 5.
```

Proving with definitions

Definitions are transparent

```
Check erefl.
  erefl : forall (A : Type) (x : A), x = x

Lemma triv10 : double five = 5 + 5.
Proof. by apply: erefl. Qed.
```

Proving with definitions

Controlling manually:

Proving with definitions

Controlling manually:

Unfolding

`rewrite` $/name$

Proving with definitions

Controlling manually:

Unfolding

`rewrite  /`$name$

Folding

`rewrite -/`$name$

Proving with definitions

```
double five = 5 + 5.
```

Proving with definitions

```
double five = 5 + 5.
```

```
rewrite /double.
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
```

Proving with definitions

```
double five = 5 + 5.
```

```
rewrite /double.
```

```
five + five = 5 + 5.
```

```
rewrite /five.
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
rewrite /five.
        5 + 5 = 5 + 5.
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
rewrite /five.
        5 + 5 = 5 + 5.
rewrite -/five.
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
rewrite /five.
        5 + 5 = 5 + 5.
rewrite -/five.
        five + five = five + five.
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
rewrite /five.
        5 + 5 = 5 + 5.
rewrite -/five.
        five + five = five + five.
rewrite -/(double _).
```

Proving with definitions

```
        double five = 5 + 5.
rewrite /double.
        five + five = 5 + 5.
rewrite /five.
        5 + 5 = 5 + 5.
rewrite -/five.
        five + five = five + five.
rewrite -/(double _).
        double five = double five.
```

Defining data-structures

Defining data-structures

$$
\begin{array}{l}
\texttt{Inductive}\ name\ \texttt{:=} \\
\quad \texttt{C}_1\ \texttt{of}\ T'_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T'_{n_1} \\
\mid\ \texttt{C}_2\ \texttt{of}\ T''_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T''_{n_2} \\
\qquad\quad \ldots \\
\mid\ \texttt{C}_k\ \texttt{of}\ T'^{(k)}_1\ \texttt{\&}\ \ldots\texttt{\&}\ T'^{(k)}_{n_k}\ .
\end{array}
$$

Defining data-structures

```
Inductive name :=
  C1 of T'_1 & ... & T'_n1
| C2 of T''_1 & ... & T''_n2
    ...
| Ck of T'^(k)_1 & ... & T'^(k)_nk.

Check (C1 o1 ... o_n1).
```

$$\texttt{Inductive}\ name\ \texttt{:=}$$
$$\texttt{C}_1\ \texttt{of}\ T'_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T'_{n_1}$$
$$|\ \texttt{C}_2\ \texttt{of}\ T''_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T''_{n_2}$$
$$\ldots$$
$$|\ \texttt{C}_k\ \texttt{of}\ T'^{(k)}_1\ \texttt{\&}\ \ldots\texttt{\&}\ T'^{(k)}_{n_k}.$$

$$\texttt{Check (C}_1\ o_1\ \ldots\ o_{n_1}\texttt{).}$$

Defining data-structures

$$\begin{array}{l}
\texttt{Inductive}\ name\ \texttt{:=} \\
\quad \texttt{C}_1\ \texttt{of}\ T'_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T'_{n_1} \\
\mid\ \texttt{C}_2\ \texttt{of}\ T''_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T''_{n_2} \\
\qquad \ldots \\
\mid\ \texttt{C}_k\ \texttt{of}\ T'^{(k)}_1\ \texttt{\&}\ \ldots\texttt{\&}\ T'^{(k)}_{n_k}\ .
\end{array}$$

$$\texttt{Check (C}_1\ o_1\ \ldots\ o_{n_1}\texttt{).}$$

$$\texttt{(C}_1\ o_1\ \ldots\ o_{n_1}\texttt{)}\ :\ name$$

Defining data-structures

(ssrbool.v)

Boolean

Defining data-structures

(ssrbool.v)

Boolean

```
Inductive bool := true | false.
```

Defining data-structures

(ssrbool.v)

Boolean

```
Inductive bool :=  true | false.

Check true.
  true : bool
```

Defining data-structures

(ssrbool.v)

Boolean

```
Inductive bool := true | false.

Check true.
  true : bool

Check false.
  false : bool
```

Defining data-structures

Conditional Expression

Defining data-structures

Conditional Expression

`if` *test* `then` *then-part* `else` *else-part*

Defining data-structures

Boolean connectors

Defining data-structures

Boolean connectors

```
Definition andb x y :=  if x then y else false.
Notation "x && y" :=  (andb x y).
```

Defining data-structures

Boolean connectors

Definition andb x y := if x then y else false.
Notation "x && y" := (andb x y).

Definition orb x y := if x then true else y.
Notation "x $\|$ y" := (orb x y).

Defining data-structures

Boolean connectors

Definition andb x y := if x then y else false.
Notation "x && y" := (andb x y).

Definition orb x y := if x then true else y.
Notation "x $\|$ y" := (orb x y).

Definition negb x := if x then false else true.
Notation "$\neg\neg$ x" := (negb x).

Proving with booleans

Proving with booleans

Conditional simplications are transparent

Proving with booleans

Conditional simplications are transparent

```
Lemma trivIfTF :
 if true then 1 else 2 = if false then 3 else 1.
```

Proving with booleans

Conditional simplications are transparent

```
Lemma trivIfTF :
 if true then 1 else 2 = if false then 3 else 1.

Proof. by apply: erefl. Qed.
```

Proving with booleans

Controlling simplications manually

Proving with booleans

Controlling simplications manually

Proving with booleans

`forall` x`, true && `x` = `x` && true`

Proving with booleans

`forall` x`, true && ` x ` = ` x ` && true`

`move` $\Rightarrow$ x`.`

Proving with booleans

$$\texttt{forall } x\texttt{, true \&\& } x \texttt{ = } x \texttt{ \&\& true}$$

move $\Rightarrow$ x.

$$\texttt{true \&\& } x \texttt{ = } x \texttt{ \&\& true}$$

Proving with booleans

forall x, true && x = x && true

move $\Rightarrow$ x.

true && x = x && true

rewrite /=.

Proving with booleans

forall x, true && x = x && true

move ⇒ x.

true && x = x && true

rewrite /=.

x = x && true

Proving with booleans

Case analysis

Proving with booleans

Case analysis

Proving with booleans

x `=` x `&& true`

Proving with booleans

x `= ` x `&& true`

`case:` x`.`

Proving with booleans

x `= ` x `&& true`

`case:` x`.`

```
(1/2)      true =  true && true

(2/2)     false = false && true
```

Proving with booleans

```
Lemma andbC : forall x y, x && y = y && x.
Proof. by move => x y; case: x; case: y. Qed.
```

Proving with booleans

Lemma andbC : forall x y, x && y = y && x.

Proof. by move ⇒ x y; case: x; case: y. Qed.

Lemma negb_or : forall x y, $\neg\neg$(x || y) = $\neg\neg$ x && $\neg\neg$ y.

Proof. by move ⇒ x y; case: x; case: y. Qed.

Summary

Summary

`Definition` *name* : *type* := *body*.

`if` *test* `then` *then-part* `else` *else-part*.

`Inductive` *name* := $\texttt{C}_1$ `of` . . . | . . . | $\texttt{C}_k$ `of` . . . .

Summary

Definition *name* : *type* := *body*.

if *test* then *then-part* else *else-part*.

Inductive *name* := C_1 of ... | ... | C_k of ... .

rewrite /*name* -/*name* /=.

case: *term*.

Defining sequences

(seq.v)

Defining sequences

(seq.v)

```
[:: 2; 3; 5; 7]
```

Defining sequences

(seq.v)

```
[:: 2; 3; 5; 7]

[:: false; true]
```

Defining sequences

(seq.v)

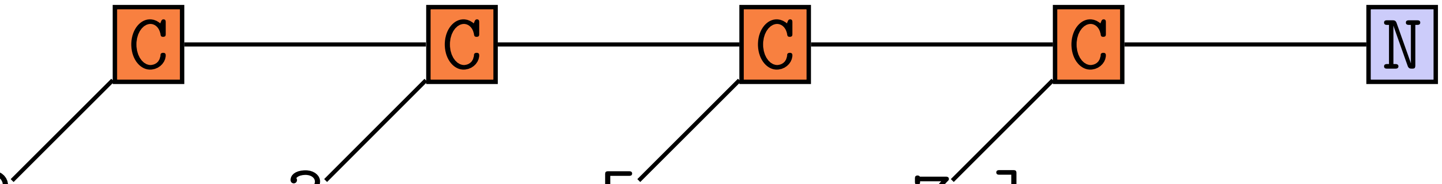
```
[:: 2; 3; 5; 7]:   seq nat

[:: false; true]:   seq bool
```

Defining sequences

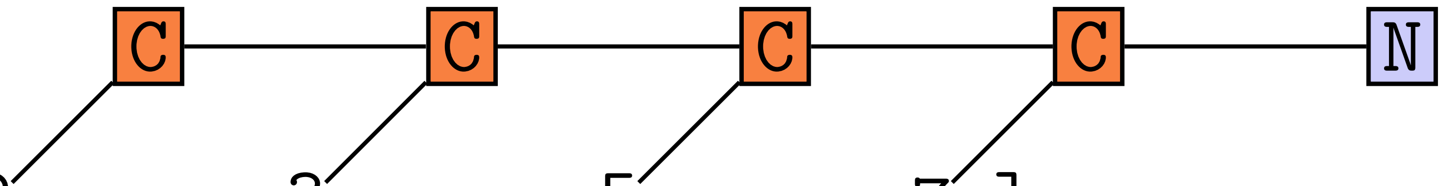
```
[:: 2 ;      3 ;      5 ;      7 ]
```

Defining sequences




```
[:: 2 ; 3 ; 5 ; 7 ]
```

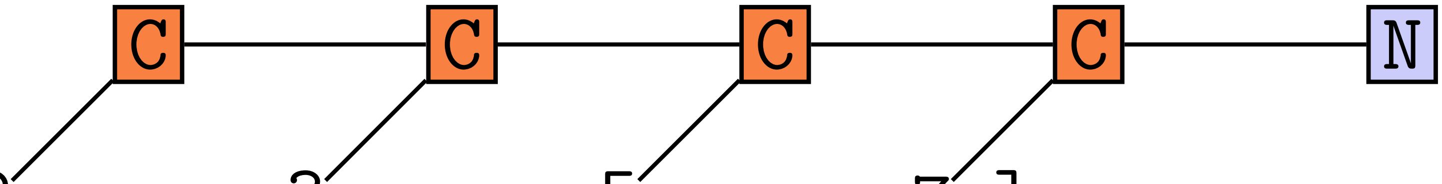
Defining sequences



`[:: 2 ; 3 ; 5 ; 7 ]`

Inductive seq (T: Type) := Nil | Cons of T & seq T.

Defining sequences



`[:: 2 ; 3 ; 5 ; 7 ]`

Inductive seq (T: Type) := Nil | Cons of T & seq T.

Notation "[::]" := Nil.

Notation "h :: t" := (Cons h t).

Notation "[h_1 ; h_2; ... ; h_n]" :=

(h_1 :: h_2 :: ... :: h_n :: [::]).

Pattern Conditional

Pattern Conditional

`if` *term* `is` *pattern* `then` *then-part* `else` *else-part*

Functions on sequences

Getting the tail of a sequence

Functions on sequences

Getting the tail of a sequence

Definition tl (T : Type) (l : seq T) :=
  if l is h :: t then t else [::].

Proving with sequences

Proving with sequences

Conditional simplications are transparent

Proving with sequences

Conditional simplications are transparent

```
Lemma trivtl :
    tl [:: 2; 3; 5; 7] = [:: 3; 5; 7].
```

Proving with sequences

Conditional simplications are transparent

```
Lemma trivtl :
    tl [:: 2; 3; 5; 7] = [:: 3; 5; 7].
Proof. by apply: erefl. Qed.
```

Functions on sequences

Swapping the first two elements a sequence

Functions on sequences

Swapping the first two elements a sequence

Definition swap (T : Type) (l : seq T) :=
  if l is h_1 :: h_2 :: t then h_2 :: h_1 :: t
  else l.

Proving with sequences

`forall` T `(`l`: seq` T`), swap (swap` l`) =` l

Proving with sequences

forall T (l: seq T), swap (swap l) = l

move $\Rightarrow T\ l$.

Proving with sequences

```
forall T (l: seq T), swap (swap l) = l
```

move ⇒ T l.

```
swap (swap l) = l
```

Proving with sequences

`forall` T `(`l`: seq` T`), swap (swap` l`) =` l

`move` $\Rightarrow$ T l.

`swap (swap` l`) =` l

`case:` l.

Proving with sequences

forall T (l: seq T), swap (swap l) = l

move $\Rightarrow$ T l.

swap (swap l) = l

case: l.

(1/2) swap (swap [::]) = [::]

(2/2) forall h t, swap (swap h :: t) = h :: t

Proving with sequences

```
forall T (l: seq T), swap (swap l) = l
```

move ⇒ T l.

swap (swap l) = l

case: l.

(1/2) swap (swap [::]) = [::]

by apply: erefl

(2/2) forall h t, swap (swap h :: t) = h :: t

Proving with sequences

`forall` h t, `swap (swap` h `::` t`) =` h `::` t

Proving with sequences

```
forall h t, swap (swap h :: t) = h :: t
```

```
move => h1 t.
```

Proving with sequences

`forall` h t, `swap (swap` h `::` t`) =` h `::` t

`move` $\Rightarrow$ h_1 t.

`swap (swap` h_1 `::` t`) =` h_1 `::` t

Proving with sequences

$\texttt{forall}\ h\ t\texttt{,}\ \texttt{swap}\ \texttt{(swap}\ h\ \texttt{::}\ t\texttt{)}\ \texttt{=}\ h\ \texttt{::}\ t$

$\texttt{move}\ \Rightarrow\ h_1\ t\texttt{.}$

$\texttt{swap}\ \texttt{(swap}\ h_1\ \texttt{::}\ t\texttt{)}\ \texttt{=}\ h_1\ \texttt{::}\ t$

$\texttt{case:}\ \ t\texttt{.}$

Proving with sequences

forall h t, swap (swap h :: t) = h :: t

move $\Rightarrow$ h_1 t.

swap (swap h_1 :: t) = h_1 :: t

case: t.

(1/2) swap (swap [:: h_1]) = [:: h_1]

(2/2) forall h t, swap (swap h_1 :: h :: t) = h_1 :: h :: t

Proving with sequences

`forall` h t, `swap (swap` h `::` t`)` `=` h `::` t

`move` $\Rightarrow$ h_1 t.

`swap (swap` h_1 `::` t`)` `=` h_1 `::` t

`case:` t.

(1/2) `swap (swap [::` h_1`])` `= [::` h_1`]`

`by apply: erefl.`

(2/2) `forall` h t, `swap (swap` h_1 `::` h `::` t`)` `=` h_1 `::` h `::` t

`by move` $\Rightarrow$ h_2 t; `apply: erefl.`

Defining recursive functions

Defining recursive functions

`Fixpoint` *name* : *type* := *body*.

Functions on sequences

Appending two sequences

Functions on sequences

Appending two sequences

Fixpoint app $(T$: Type) $(l_1\ l_2$: seq $T)$:=
  if l_1 is $h :: t$ then $h ::$ (app $t\ l_2$) else l_2.

Functions on sequences

Appending two sequences

```
Fixpoint app (T : Type) (l1 l2 : seq T) :=
  if l1 is h :: t then h :: (app t l2) else l2.
```

```
Notation "l1 ++ l2" := (app l1 l2).
```

Proving with functions

Proving with functions

`elim:` *term*.

Proving with sequences

forall T (l: seq T), l ++ [::] = l

Proving with sequences

```
forall T (l: seq T), l ++ [::]  = l
```

```
move => T l.
```

Proving with sequences

`forall` T `(`l`: seq` T`),` l `++ [::] =` l

`move` $\Rightarrow$ T l`.`

l `++ [::] =` l

Proving with sequences

```
forall T (l: seq T), l ++ [::]  = l
```

```
move => T l.
```

```
l ++ [::]  = l
```

```
elim:  l.
```

Proving with sequences

$$\texttt{forall}\ T\ (l\texttt{: seq}\ T)\texttt{,}\ l\ \texttt{++ [::] =}\ l$$

$\texttt{move}\Rightarrow T\ l\texttt{.}$

$$l\ \texttt{++ [::] =}\ l$$

$\texttt{elim:}\ \ l\texttt{.}$

(1/2) [::] ++ [::] = [::]

(2/2) $\texttt{forall}\ h\ t\texttt{,}$

$$t\ \texttt{++ [::] =}\ t \rightarrow (h\ \texttt{::}\ t)\ \texttt{++ [::] =}\ h\ \texttt{::}\ t$$

Proving with sequences

```
forall T (l: seq T), l ++ [::] = l
```

```
move => T l.
```

```
l ++ [::] = l
```

```
elim: l.
```

```
(1/2)   [::] ++ [::] = [::]
```

```
by apply: erefl
```

```
(2/2)   forall h t,
    t ++ [::] = t -> (h :: t) ++ [::] = h :: t
```

Proving with sequences

$$\texttt{forall}\ h\ t,$$
$$t\ \texttt{++}\ \texttt{[::]}\ \texttt{=}\ t \longrightarrow (h\ \texttt{::}\ t)\ \texttt{++}\ \texttt{[::]}\ \texttt{=}\ h\ \texttt{::}\ t$$

Proving with sequences

```
forall h t,
t ++ [::] = t -> (h :: t) ++ [::] = h :: t
```

`move` $\Rightarrow h\ t\ IH.$

Proving with sequences

$$\texttt{forall } h\ t,$$
$$t \texttt{ ++ [::] = } t \longrightarrow (h \texttt{ :: } t) \texttt{ ++ [::] = } h \texttt{ :: } t$$

`move` $\Rightarrow h\ t\ IH.$

$$(h \texttt{ :: } t) \texttt{ ++ [::] = } h \texttt{ :: } t$$

Proving with sequences

$$\texttt{forall}\ h\ t,$$
$$t \texttt{ ++ [::] = } t \rightarrow (h \texttt{ :: } t) \texttt{ ++ [::] = } h \texttt{ :: } t$$

move $\Rightarrow h\ t\ IH.$

$$(h \texttt{ :: } t) \texttt{ ++ [::] = } h \texttt{ :: } t$$

```
rewrite /=.
```

Proving with sequences

$\texttt{forall}\ h\ t,$

t `++ [::]` = $t \longrightarrow (h$ `::` $t)$ `++ [::]` = h `::` t

`move` $\Rightarrow h\ t\ IH.$

$(h$ `::` $t)$ `++ [::]` = h `::` t

`rewrite /=.`

h `::` $(t$ `++ [::])` = h `::` t

Proving with sequences

```
    forall h t,
t ++ [::] = t -> (h :: t) ++ [::] = h :: t
```

```
move => h t IH.
```

```
        (h :: t) ++ [::] = h :: t
```

```
rewrite /=.
```

```
    h :: (t ++ [::]) = h :: t
```

```
by rewrite IH.
```

Defining natural numbers

(ssrnat.v)

Defining natural numbers

`(ssrnat.v)`

Defining natural numbers

(ssrnat.v)

```
Inductive nat := O | S of nat.
```

Defining natural numbers

(ssrnat.v)

```
Inductive nat := O | S of nat.

Notation "n .+1" := (S n).
```

Functions on natural numbers

Functions on natural numbers

Definition predn n := if n is m.+1 then m else n.

Functions on natural numbers

```
Definition predn n :=  if n is m.+1 then m else n.
Notation "n .-1" :=  (predn n).
```

Functions on natural numbers

```
Definition predn n := if n is m.+1 then m else n.
Notation "n .-1" := (predn n).
```

```
Fixpoint addn m n :=
    if m is m'.+1 then (addn m' n).+1 else n.
```

Functions on natural numbers

```
Definition predn n := if n is m.+1 then m else n.
Notation "n .-1" := (predn n).

Fixpoint addn m n :=
    if m is m'.+1 then (addn m' n).+1 else n.
Notation "m + n" := (addn m n).
```

Functions on natural numbers

```
Definition predn n := if n is m.+1 then m else n.
Notation "n .-1" := (predn n).

Fixpoint addn m n :=
    if m is m'.+1 then (addn m' n).+1 else n.
Notation "m + n" := (addn m n).

Fixpoint muln m n :=
    if m is m'.+1 then n + muln m' n else m.
```

Functions on natural numbers

```
Definition predn n := if n is m.+1 then m else n.
Notation "n .-1" := (predn n).

Fixpoint addn m n :=
    if m is m'.+1 then (addn m' n).+1 else n.
Notation "m + n" := (addn m n).

Fixpoint muln m n :=
    if m is m'.+1 then n + muln m' n else m.
Notation "m * n" := (muln m n).
```

Functions on natural numbers

```
Definition predn n := if n is m.+1 then m else n.
Notation "n .-1" := (predn n).

Fixpoint addn m n :=
    if m is m'.+1 then (addn m' n).+1 else n.
Notation "m + n" := (addn m n).

Fixpoint muln m n :=
    if m is m'.+1 then n + muln m' n else m.
Notation "m * n" := (muln m n).

Fixpoint size (T : Type) (l : seq T) :=
    if l is h :: t then (size t).+1 else 0.
```

Proving with natural numbers

forall n, n + 0 = n

Proving with natural numbers

`forall` n, n `+` 0 `=` n

`move` $\Rightarrow$ n.

Proving with natural numbers

forall n, n + 0 = n

move ⇒ n.

n + 0 = n

Proving with natural numbers

```
forall n, n + 0 = n
```

```
move => n.
```

```
n + 0 = n
```

```
elim: n.
```

Proving with natural numbers

`forall` n, n `+ 0 =` n

`move` $\Rightarrow$ n.

n `+ 0 =` n

`elim:` n.

`(1/2)` `0 + 0 = 0`

`(2/2)` `forall` n,

n `+ 0 =` n $\rightarrow$ n`.+1 + 0 =` n`.+1`

Proving with natural numbers

forall n, n + 0 = n

move ⇒ n.

n + 0 = n

elim: n.

(1/2) 0 + 0 = 0

by apply: erefl

(2/2) forall n,

n + 0 = n → n.+1 + 0 = n.+1

Summary

Summary

`Inductive` *name* `:=` $\texttt{C}_1$ `of` ... | ... | $\texttt{C}_k$ `of` ...

`Definition` *name* : *type* `:=` *body*.

`if` *test* `then` *then-part* `else` *else-part*.

`if` *term* `is` *pattern* `then` *then-part* `else` *else-part*.

Summary

`Inductive` *name* `:=` $\mathtt{C}_1$ `of` ... | ... | $\mathtt{C}_k$ `of` ...

`Definition` *name* `:` *type* `:=` *body*.

`if` *test* `then` *then-part* `else` *else-part*.

`if` *term* `is` *pattern* `then` *then-part* `else` *else-part*.

`rewrite /`*name* `-/`*name* `/=`.

`case:` *term* `elim:` *term*.