

Inductive types and identity types

Michael Shulman

February 7, 2012

1 / 49

Type constructors

For every type constructor, we have rules for:

- ① Constructing types
- ② Constructing terms in those types (introduction)
- ③ Using terms in those types (elimination)
- ④ Eliminating introduced terms (computation)

3 / 49

Negative types

The only negative type we will use is **dependent product**.

- For $A: \text{Type}$ and $B: A \rightarrow \text{Type}$, we have $\prod_{x: A} B(x): \text{Type}$.
- An element of $\prod_{x: A} B(x)$ is a *dependently typed function*, sending each $x: A$ to an element $f(x): B(x)$.
- Coq syntax: `forall (x:A), B x`

When $B(x)$ is independent of x , we have the **function type**

$$(A \rightarrow B) := \prod_{x: A} B$$

4 / 49

Positive types

Positive types are characterized by their introduction rules.

$$\frac{a: A}{\text{inl}(a): A + B} \quad \frac{b: B}{\text{inr}(b): A + B}$$

$$\frac{a: A \quad b: B}{(a, b): A \times B}$$

$$\frac{}{\text{tt}: \text{unit}}$$

The elimination and computation rules can then be deduced.

5 / 49

Non-recursive inductive types

All positive types in Coq are **inductive types**.

```
Inductive W : Type :=
| constr1 : A1 -> A2 -> ... -> Am -> W
| constr2 : B1 -> B2 -> ... -> Bn -> W
| :
| constrk : Z1 -> Z2 -> ... -> Zp -> W.
```

This command causes Coq to:

- 1 create a type `W`
- 2 create functions `constr1` through `constrk` with the specified types
- 3 allow an appropriate form of `match` syntax, and
- 4 implement appropriate computation rules.

6/49

Examples

```
Inductive AplusB : Type :=
| inlAB : A -> AplusB
| inrAB : B -> AplusB.
```

```
Inductive AtimesB : Type :=
| pairAB : A -> B -> AtimesB.
```

```
Inductive unit : Type :=
| tt : unit
```

```
Inductive Empty_set : Type :=
.
```

7/49

Parameters

With **parameters** we can define many related types at once.

```
Inductive sum (A B : Type) : Type :=  
| inl : A -> sum A B  
| inr : B -> sum A B.
```

```
Inductive prod (A B : Type) : Type :=  
| pair : A -> B -> prod A B.
```

Implicit arguments and **notations** make these nicer to use.

8 / 49

Dependent sums

In the presence of dependent types, the constructors can be dependently typed functions.

```
Inductive sigT (A : Type) (P : A -> Type)  
: Type :=  
| existT : forall (a : A), P a -> sigT A P.
```

The type of existT is

$$\prod_{a: A} (P(a) \rightarrow \sum_{x: A} P(x))$$

This is a function of two variables whose output is the type being defined ($\sum_{x: A} P(x)$), but the type of the second input depends on the value of the first.

10 / 49

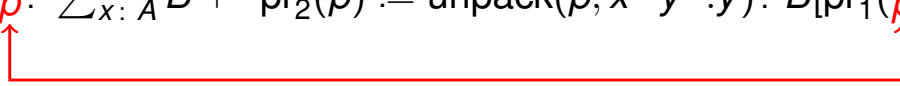
Strong eliminators

The elimination rule for an inductive type W is

$$\frac{\begin{array}{c} \Gamma, p: W \vdash C: \text{Type} \quad \Gamma \vdash p: W \\ \Gamma, (\text{inputs of } \text{constr}_1) \vdash c_1: C[\text{constr}_1(\dots)/p] \\ \vdots \\ \Gamma, (\text{inputs of } \text{constr}_k) \vdash c_k: C[\text{constr}_k(\dots)/p] \end{array}}{\Gamma \vdash \text{match}(p, \dots): C}$$

Note: In general, we must allow the **output type** C to depend on the **value** $p: W$.

Example

$$\begin{array}{l} p: \sum_{x:A} B \vdash \text{pr}_1(p) := \text{unpack}(p, x^A y^B.x): A \\ p: \sum_{x:A} B \vdash \text{pr}_2(p) := \text{unpack}(p, x^A y^B.y): B[\text{pr}_1(p)/x] \end{array}$$


11 / 49

The natural numbers

The natural numbers are generated by 0 and successor s . That is, \mathbb{N} is defined by the ways to **construct** a natural number. Thus it is a **positive type**.

```
Inductive nat : Type :=
| zero : nat
| succ : nat -> nat.
```

A new feature: the **input** of the constructor succ involves something of the type \mathbb{N} being defined!

We intend, of course, that all elements of \mathbb{N} are generated by *successively* applying constructors.

$$0, s(0), s(s(0)), s(s(s(0))), \dots$$

13 / 49

The natural numbers

$$\frac{\Gamma, n: \mathbb{N} \vdash C: \text{Type} \quad \Gamma \vdash n: \mathbb{N} \quad \Gamma \vdash c_0: C[0/n] \quad \Gamma, x: \mathbb{N} \vdash c_s: C[s(x)/n]}{\Gamma \vdash \text{rec}(n, c_0, x^{\mathbb{N}} r^C . c_s): C}$$

But this is not much good; we need to **recurse**.

$$\frac{\Gamma, n: \mathbb{N} \vdash C: \text{Type} \quad \Gamma \vdash n: \mathbb{N} \quad \Gamma \vdash c_0: C[0/n] \quad \Gamma, x: \mathbb{N}, r: C[x/n] \vdash c_s: C[s(x)/n]}{\Gamma \vdash \text{rec}(n, c_0, x^{\mathbb{N}} r^C . c_s): C}$$

The variable r represents the result of the recursive call at x , to be used the computation c_s of the value at $s(x)$.

$$\begin{aligned} \text{rec}(0, c_0, x^{\mathbb{N}} r^C . c_s) &\rightarrow_{\beta} c_0 \\ \text{rec}(s(n), c_0, x^{\mathbb{N}} r^C . c_s) &\rightarrow_{\beta} c_s[n/x, \text{rec}(n, c_0, x^{\mathbb{N}} r^C . c_s)/r] \end{aligned}$$

14 / 49

Addition

We define addition by recursion on the first input.

$$\begin{aligned} 0 + m &:= m \\ s(n) + m &:= s(n + m) \end{aligned}$$

In terms of the rec eliminator, this is

$$n: \mathbb{N}, m: \mathbb{N} \vdash \text{plus}(n, m) := \text{rec}(n, m, x^{\mathbb{N}} r^{\mathbb{N}} . s(r))$$

- When $n = 0$, the result is m .
- When n is a successor $s(x)$, the result is $s(r)$.
(As before, r is the result of the recursive call at x .)

15 / 49

Computing an addition

$$\begin{aligned}ss0 + sss0 &\rightarrow_{\beta} s(s0 + sss0) \\ &\rightarrow_{\beta} s(s(0 + sss0)) \\ &\rightarrow_{\beta} s(s(sss0)) = sssss0.\end{aligned}$$

plus(ss0, sss0)

$$\begin{aligned}&:= \text{rec}(ss0, sss0, x^{\mathbb{N}} r^{\mathcal{C}}.s(r)) \\ &\rightarrow_{\beta} \left(s(r) \right) \left[s0/x, \text{rec}(s0, sss0, x^{\mathbb{N}} r^{\mathcal{C}}.s(r))/r \right] \\ &= s\left(\text{rec}(s0, sss0, x^{\mathbb{N}} r^{\mathcal{C}}.s(r)) \right) \\ &\rightarrow_{\beta} s\left(\left(s(r) \right) \left[0/x, \text{rec}(0, sss0, x^{\mathbb{N}} r^{\mathcal{C}}.s(r))/r \right] \right) \\ &= s\left(s\left(\text{rec}(0, sss0, x^{\mathbb{N}} r^{\mathcal{C}}.s(r)) \right) \right) \\ &\rightarrow_{\beta} s(s(sss0)) = sssss0\end{aligned}$$

16/49

Fixpoints

The “Fixpoint” command in Coq allows traditional-style programming with recursive functions.

```
Fixpoint fac (n : nat) : nat :=
  match n with
  | 0 => 1
  | S n' => (S n') * fac n'
  end.
```

But Coq checks that our functions could be written with “rec” and therefore always terminate. This is necessary for logic to be consistent!

```
Fixpoint oops : Empty_set :=
  oops.
```

18/49

The “limits” of Coq

With recursion over \mathbb{N} in Coq, we can program:

- 1 Simple primitive recursive functions (+, ·, exp, ...).
- 2 Higher-order primitive recursive functions
(**Exercise***: Define the Ackermann function.)
- 3 Any algorithm that we can prove to terminate, e.g. by well-founded induction on some measure.

With a coinductive nontermination monad, we can program:

- 4 All general recursive functions
(But we can only compute them some specified amount.)

With classical axioms (PEM, AC) we can program:

- 5 All mathematical (total) functions
(But they don't compute—they may not be computable!)

NB: These naturals are unary, hence very inefficient. But we can also define binary ones.

19/49

Other recursive inductive types

```
Inductive list (A : Type) : Type :=  
| nil : list A  
| cons : A -> list A -> list A.
```

Contains nil, cons(a,nil), cons(a,cons(b,nil)), ...

```
Inductive btree (A : Type) : Type :=  
| leaf : A -> btree A  
| branch : btree A -> btree A -> btree A.
```

20/49

Programming with inductive datatypes

```
Fixpoint length {A : Type} (l : list A) : nat :=
  match l with
  | nil          => 0
  | cons _ l'   => S (length l')
  end.
```

$$\begin{aligned} \text{length}(\text{cons}(a, \text{cons}(b, \text{nil}))) &\rightarrow_{\beta} \text{s}(\text{length}(\text{cons}(b, \text{nil}))) \\ &\rightarrow_{\beta} \text{s}(\text{s}(\text{length}(\text{nil}))) \\ &\rightarrow_{\beta} \text{s}(\text{s}(0)) \end{aligned}$$

21 / 49

Proof by induction

Recall that **propositions** are just **types** in some sort “Prop”.

$$\frac{\Gamma, n: \mathbb{N} \vdash P: \text{Prop} \quad \Gamma \vdash n: \mathbb{N} \quad \Gamma \vdash c_0: P[0/n] \quad \Gamma, x: \mathbb{N}, r: P[x/n] \vdash c_s: P[s(x)/n]}{\Gamma \vdash \text{rec}(n, c_0, x^{\mathbb{N}} r^{\mathbb{C}}.c_s): P}$$

This is just classical **proof by induction**.

| | | |
|-------------|-----------------------|--------------|
| types | \longleftrightarrow | propositions |
| programming | \longleftrightarrow | proving |
| recursion | \longleftrightarrow | induction |

22 / 49

Example

Theorem

Every natural number is either zero or the successor of some other natural number.

Proof.

Let $P(n) := (n = 0) + \sum_{m: \mathbb{N}} (n = sm)$.

$$\frac{\begin{array}{c} \vdash n: \mathbb{N} \\ \vdash \text{inl}(\text{refl}_0): P(0) \quad x: \mathbb{N}, r: P(x) \vdash \text{inr}(x, \text{refl}_{sx}): P(sx) \end{array}}{\vdash P(n)}$$

□

23 / 49

Inductive proofs

Proof by induction is not something special about the natural numbers. It applies to any inductively defined type, including even non-recursive ones.

25 / 49

Induction on lists

$$\begin{aligned}\text{nil} \# l &:= l \\ \text{cons}(a, l_1) \# l_2 &:= \text{cons}(a, l_1 \# l_2)\end{aligned}$$

Theorem

$$\text{length}(l_1 \# l_2) = \text{length}(l_1) + \text{length}(l_2)$$

Proof.

By induction on l_1 .

- 1 When l_1 is nil, we have

$$\begin{aligned}\text{length}(\text{nil} \# l_2) &= \text{length}(l_2) \\ &= 0 + \text{length}(l_2) \\ &= \text{length}(\text{nil}) + \text{length}(l_2)\end{aligned}$$

26 / 49

Induction on lists

$$\begin{aligned}\text{nil} \# l &:= l \\ \text{cons}(a, l_1) \# l_2 &:= \text{cons}(a, l_1 \# l_2)\end{aligned}$$

Theorem

$$\text{length}(l_1 \# l_2) = \text{length}(l_1) + \text{length}(l_2)$$

Proof.

By induction on l_1 .

- 2 When l_1 is $\text{cons}(a, l'_1)$, we have

$$\begin{aligned}\text{length}(\text{cons}(a, l'_1) \# l_2) &= \text{length}(\text{cons}(a, l'_1 \# l_2)) \\ &= s(\text{length}(l'_1 \# l_2)) \\ &= s(\text{length}(l'_1) + \text{length}(l_2)) \\ &= s(\text{length}(l'_1)) + \text{length}(l_2) \\ &= \text{length}(\text{cons}(a, l'_1)) + \text{length}(l_2) \quad \square\end{aligned}$$

26 / 49

Parameters versus indices

An inductive definition with **parameters**, like

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

actually defines a dependent type

$$\text{list} : \text{Type} \rightarrow \text{Type}$$

But each type $\text{list}(A)$ is separately inductively defined; the constructors don't "hop around" between different A s.

Indices remove this restriction.

28 / 49

Vectors with indices

A **vector** is a list whose length is specified in its type.

```
Inductive vec (A : Type) : nat -> Type :=
| vnil : vec A 0
| vcons : forall (n : nat),
  A -> vec A n -> vec A (S n).
```

- For **each** type A , we inductively define the **family** of types $\text{vec } A \ n$, as n ranges over natural numbers.
- The value of n used in the constructors can vary both **between** constructors and **within** the inputs and outputs of a single constructor.

Thus A is a **parameter**, n is an **index**.

29 / 49

Programming with indices

For any A , we can define a dependently typed function

$$\text{concat}: \prod_{n: \mathbb{N}} \left(\text{vec}(A, n) \rightarrow \prod_{m: \mathbb{N}} \left(\text{vec}(A, m) \rightarrow \text{vec}(A, n+m) \right) \right)$$

as follows:

$$\text{concat}(0, \text{vnil}, m, v) := v$$

$$\text{concat}(s(n), \text{vcons}(a, v_1), m, v_2) := \text{vcons}(a, \text{concat}(n, v_1, m, v_2))$$

- 1 The first clause is well-typed because $0 + m \leftrightarrow_{\beta} m$.
- 2 The second is well-typed because $s(n + m) \leftrightarrow_{\beta} sn + m$.

NB: In each “case”, the indices automatically get specialized to the appropriate values.

The definition **and behavior** of “length” are built into the **type**.

30 / 49

Induction with indices

Theorem

For $v_i: \text{vec}(A, n_i)$, $i = 1, 2, 3$, we have

$$v_1 \text{ ++ } (v_2 \text{ ++ } v_3) = (v_1 \text{ ++ } v_2) \text{ ++ } v_3$$

Proof.

By induction on v_1 .

- 1 If v_1 is vnil , then both sides are $v_2 \text{ ++ } v_3$.
- 2 If v_1 is $\text{vcons}(a, v'_1)$, the LHS is $\text{vcons}(a, v'_1 \text{ ++ } (v_2 \text{ ++ } v_3))$, and the RHS is $\text{vcons}(a, (v'_1 \text{ ++ } v_2) \text{ ++ } v_3)$, which are equal by the inductive hypothesis.

□

31 / 49

Lists with indices

Any inductive definition with parameters:

```
Inductive listP (A : Type) : Type :=  
| nilP : listP A  
| consP : A -> listP A -> listP A.
```

can be rephrased using indices:

```
Inductive listI : Type -> Type :=  
| nilI : forall A, listI A  
| consI : forall A, A -> listI A -> listI A.
```

But the inductive principle we obtain is subtly different.

32/49

Parameters versus indices

With parameters

The type `listP(A)` is separately inductively defined for every `A`. Thus we can use induction to prove something about `listP(A)` for some **particular** `A`.

With indices

The family of types `listI(A)` is jointly inductively defined for all `A`. Thus we can only use induction to prove something about `listI(A)` for **all** `A` at once.

33/49

Parameters versus indices

Define $\text{sum} : \text{listP}(\mathbb{N}) \rightarrow \mathbb{N}$ by

$$\begin{aligned}\text{sum}(\text{nilP}) &:= 0 \\ \text{sum}(\text{consP}(a, \ell)) &:= a + \text{sum}(\ell)\end{aligned}$$

Theorem

$$\text{sum}(\ell_1 \# \ell_2) = \text{sum}(\ell_1) + \text{sum}(\ell_2)$$

Proof.

By induction...

□

With `listI` this is a non-starter.

Proving something about `listI(N)` by induction is like proving “3 is prime” by induction on 3.

35 / 49

What indices can do

Indices give a weaker induction principle because in general, we can't separate the values at different inputs.

In theory, we could have:

```
Inductive listI' : Type -> Type :=
| nilI : forall A, listI' A
| consI : forall A, A -> listI' A -> listI' A
| huh : listI' (ℝ × ℤ) -> listI' ℕ
```

Just like `vec`, we couldn't define this type with parameters.

36 / 49

Parameters versus indices

“If an index could be a parameter, it should be.”

but actually...

If an index could be a parameter, it **might as well** be.

Theorem

We can prove the induction principle of listP from the induction principle of listI .

Proof.

The induction principle of listP says “for any A , any property of elements of $\text{listP}(A)$ can be proven by induction.” But **this** statement is general over all A , hence follows from the induction principle of listI . □

37/49

Trickier induction with indices

Theorem

For any $v: \text{vec}(A, 0)$ we have $v = \text{vnil}$.

Proof.

By induction??

Again, this is like proving “3 is prime” by induction on 3.

38/49

Trickier induction with indices

Theorem

For any $v : \text{vec}(A, 0)$ we have $v = \text{vnil}$.

Proof.

Define $P : \prod_{n: \mathbb{N}} (\text{vec}(A, n) \rightarrow \text{Prop})$ by induction on n :

$$P(0, v) := (v = \text{vnil})$$

$$P(sn, v) := \top$$

Now prove by induction on $v : \text{vec}(A, n)$ that $P(n, v)$ holds.

- ① If v is vnil , then $P(0, v)$ is $(\text{vnil} = \text{vnil})$, which is true.
- ② If v is $\text{vcons}(a, v')$, then $P(0, v)$ is \top , which is true.

Finally, let $n = 0$. □

38 / 49

Non-uniform parameters

As usual, this is an oversimplification. Coq also allows “non-uniform parameters”, which are basically indices that are written like parameters, but treated slightly differently internally. Not really important for us.

39 / 49

Equality types

Definition

The **equality type** (or **identity type** or **path type**) of any type A is the following inductive family:

```
Inductive eq {A : Type} : A -> A -> Type :=
| refl : forall (a:A), eq a a.
```

Notations: $\text{eq}_A(a, b)$ $(a = b)$ $\text{Id}_A(a, b)$ $\text{Paths}_A(a, b)$

- There is only one way to prove that two things are equal; namely, everything is equal to itself.
- A is a parameter; a and b are indices.
- We can make a into a parameter (Paulin-Möhrring equality), but not also b .

41 / 49

Induction on equality

The eliminator for equality is:

$$\frac{\begin{array}{l} \Gamma, x: A, y: A, p: (x = y) \vdash C: \text{Type} \\ \Gamma \vdash a: A \quad \Gamma \vdash b: A \quad \Gamma \vdash p: (a = b) \\ \Gamma, x: A \vdash c: C[y/x, \text{refl}_x/p] \end{array}}{\Gamma \vdash J(x^A.c; p): C}$$

In words:

If $C(x, y, p)$ is a property of pairs of equal elements of A , and $C(x, x, \text{refl}_p)$ holds, then $C(a, b, p)$ holds whenever $p: (a = b)$.

In particular, if C depends only on y , then we have the principle of **substitution of equals for equals**:

If $a = b$ and $C(a)$ holds, then so does $C(b)$.

42 / 49

Properties of equality

Theorem

Equality is transitive.

Proof.

Suppose $p: (a = b)$ and $q: (b = c)$. Then using q , we can substitute c for b in $p: (a = b)$ to obtain $J(b.p, q): (a = c)$. \square

Theorem

Equality is symmetric.

Proof.

Suppose $p: (a = b)$. Then using p , we substitute b for the first copy of a in $\text{refl}_a: (a = a)$ to obtain $J(a.\text{refl}_a, p): (b = a)$. \square

43 / 49

A trickier application

Theorem

$0 \neq 1$.

Proof.

Suppose $p: (0 = 1)$. Define $C: \mathbb{N} \rightarrow \text{Type}$ by “recursion”:

$$C(0) := \text{unit}$$

$$C(sn) := \emptyset$$

Now we have $\text{tt}: C(0)$. Using p , we can substitute 1 for 0 in this to obtain a term in $C(1) = \emptyset$. \square

NB: This proof is **not** by “induction on p ”. We cannot do induction on p , since its type is not fully general. Instead we **apply** to p the already proved theorem of substitution.

45 / 49

A non-application

Theorem

For any p : $(a = a)$ we have $p = \text{refl}_a$.

Proof.

By induction, it suffices to assume that p is refl_a . But then we have $\text{refl}_{\text{refl}_a} : (p = \text{refl}_a)$. □

This is **not valid**.

The type of p is not fully general.

We are trying to prove “3 is prime” by induction on 3.

46 / 49

Intensional equality types

There are ways to formulate the rules of inductive type families so that $p = \text{refl}_a$ becomes provable. One such way is implemented (by default) in the proof assistant Agda.

Or, we could just add it as an axiom.

But I find it much more natural just to take seriously the rule we teach our incoming freshmen: *when you prove something by induction, the statement must be fully general*.

Of course, I’m biased, because this is what makes the homotopy interpretation possible. We’ll see that for most types arising in real-world programming, the rule $p = \text{refl}_a$ does hold automatically, so this merely expands the scope of the theory.

48 / 49

Resources

If you're serious about following along in Coq, then at this point I recommend starting to read some standard tutorials. Unfortunately (for a mathematician), these are all written by people working in verified computer programming.

- Benjamin Pierce et. al., *Software foundations* (<http://www.cis.upenn.edu/~bcpierce/sf/>)
- Adam Chlipala, *Certified programming with dependent types* (<http://adam.chlipala.net/cpdt/>)
- Yves Bertot and Pierre Castéran, *The Coq'Art*
- The Coq web site: <http://coq.inria.fr/>