Inductive types and identity types

Michael Shulman

February 7, 2012

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)

Negative types

The only negative type we will use is dependent product.

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

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

$$(A \to B) \coloneqq \prod_{x\colon A} B$$

Positive types

Positive types are characterized by their introduction rules.

$$\frac{a\colon A}{\mathsf{inl}(a)\colon A + B} \qquad \frac{b\colon B}{\mathsf{inr}(b)\colon A + B}$$

$$\frac{a\colon A \qquad b\colon B}{(a, b)\colon A \times B}$$

$$\frac{}{\mathsf{tt}\colon \mathsf{unit}}$$

The elimination and computation rules can then be deduced.

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.

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 :=
.
```


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.

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

$$\prod_{a:\,A} \left(P(a) \to \sum_{x:\,A} P(x) \right)$$

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.

Strong eliminators

The elimination rule for an inductive type W is

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

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

Example

$$p\colon \textstyle\sum_{x\colon A} B \;\vdash\; \mathrm{pr}_1(p) := \mathrm{unpack}(p, x^A y^B.x)\colon A$$

$$p\colon \textstyle\sum_{x\colon A} B \;\vdash\; \mathrm{pr}_2(p) := \mathrm{unpack}(p, x^A y^B.y)\colon B[\mathrm{pr}_1(p)/x]$$

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.

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

The natural numbers

$$\frac{\begin{array}{c}\Gamma, n\colon \mathbb{N} \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash n\colon \mathbb{N} \\ \Gamma \vdash c_0\colon C[0/n] \qquad \Gamma, x\colon \mathbb{N} \vdash c_{\mathrm{s}}\colon C[\mathrm{s}(x)/n]\end{array}}{\Gamma \vdash \mathrm{rec}(n, c_0, x^{\mathbb{N}}\, r^C.c_{\mathrm{s}})\colon C}$$

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

$$\frac{\begin{array}{c}\Gamma, n\colon \mathbb{N} \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash n\colon \mathbb{N} \\ \Gamma \vdash c_0\colon C[0/n] \qquad \Gamma, x\colon \mathbb{N}, r\colon C[x/n] \vdash c_{\mathrm{s}}\colon C[\mathrm{s}(x)/n]\end{array}}{\Gamma \vdash \mathrm{rec}(n, c_0, x^{\mathbb{N}}\, r^C.c_{\mathrm{s}})\colon C}$$

The variable r represents the result of the recursive call at x, to be used the computation c_{s} of the value at $\mathrm{s}(x)$.

$$\mathrm{rec}(0, c_0, x^{\mathbb{N}}\, r^C.c_{\mathrm{s}}) \to_\beta c_0$$

$$\mathrm{rec}(\mathrm{s}(n), c_0, x^{\mathbb{N}}\, r^C.c_{\mathrm{s}}) \to_\beta c_{\mathrm{s}}[n/x, \mathrm{rec}(n, c_0, x^{\mathbb{N}}\, r^C.c_{\mathrm{s}})/r]$$

Addition

We define addition by recursion on the first input.

$$0 + m := m$$
$$\mathrm{s}(n) + m := \mathrm{s}(n + m)$$

In terms of the rec eliminator, this is

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

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

Computing an addition

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

$$\begin{aligned}&\mathsf{plus}(\mathsf{ss0}, \mathsf{sss0})\\ &\quad := \mathsf{rec}(\mathsf{ss0}, \mathsf{sss0}, x^{\mathbb{N}}\, r^{C}.\mathsf{s}(r))\\ &\quad \to_\beta \Big(\mathsf{s}(r)\Big)\Big[\mathsf{s0}/x, \mathsf{rec}(\mathsf{s0}, \mathsf{sss0}, x^{\mathbb{N}}\, r^{C}.\mathsf{s}(r))/r\Big]\\ &\quad = \mathsf{s}\Big(\mathsf{rec}(\mathsf{s0}, \mathsf{sss0}, x^{\mathbb{N}}\, r^{C}.\mathsf{s}(r))\Big)\\ &\quad \to_\beta \mathsf{s}\bigg(\Big(\mathsf{s}(r)\Big)\Big[0/x, \mathsf{rec}(0, \mathsf{sss0}, x^{\mathbb{N}}\, r^{C}.\mathsf{s}(r))/r\Big]\bigg)\\ &\quad = \mathsf{s}\Big(\mathsf{s}\Big(\mathsf{rec}(0, \mathsf{sss0}, x^{\mathbb{N}}\, r^{C}.\mathsf{s}(r))\Big)\Big)\\ &\quad \to_\beta \mathsf{s}(\mathsf{s}(\mathsf{sss0})) = \mathsf{sssss0}\end{aligned}$$

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.
```


The “limits” of Coq

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

1. Simple primitive recursive functions ($+$, $\cdot$, 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.

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.
```

Programming with inductive datatypes

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

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

Proof by induction

Recall that propositions are just types in some sort "Prop".

$$\frac{\begin{array}{c}\Gamma, n\colon \mathbb{N} \vdash P\colon \mathsf{Prop} \qquad \Gamma \vdash n\colon \mathbb{N}\\ \Gamma \vdash c_0\colon P[0/n] \qquad \Gamma, x\colon \mathbb{N}, r\colon P[x/n] \vdash c_\mathsf{s}\colon P[\mathsf{s}(x)/n]\end{array}}{\Gamma \vdash \mathsf{rec}(n, c_0, x^{\mathbb{N}}\, r^{C}.c_\mathsf{s})\colon P}$$

This is just classical proof by induction.

types	$\longleftrightarrow$	propositions
programming	$\longleftrightarrow$	proving
recursion	$\longleftrightarrow$	induction

Example

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

Proof.
Let $P(n) :\equiv (n = 0) + \sum_{m\colon \mathbb{N}}(n = \mathsf{s}m)$.

$$\frac{\begin{array}{c}\vdash n\colon \mathbb{N}\\ \vdash \mathsf{inl}(\mathsf{refl}_0)\colon P(0) \qquad x\colon \mathbb{N}, r\colon P(x) \vdash \mathsf{inr}(x, \mathsf{refl}_{\mathsf{s}x})\colon P(\mathsf{s}x)\end{array}}{\vdash P(n)}$$

□

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.

Induction on lists

$$\mathsf{nil} +\!\!+ \ell := \ell$$
$$\mathsf{cons}(a, \ell_1) +\!\!+ \ell_2 := \mathsf{cons}(a, \ell_1 +\!\!+ \ell_2)$$

Theorem
$\mathsf{length}(\ell_1 +\!\!+ \ell_2) = \mathsf{length}(\ell_1) + \mathsf{length}(\ell_2)$

Proof.
By induction on ℓ_1.

1. When ℓ_1 is nil, we have

$$\begin{aligned}\mathsf{length}(\mathsf{nil} +\!\!+ \ell_2) &= \mathsf{length}(\ell_2)\\ &= 0 + \mathsf{length}(\ell_2)\\ &= \mathsf{length}(\mathsf{nil}) + \mathsf{length}(\ell_2)\end{aligned}$$

Induction on lists

$$\mathsf{nil} +\!\!+ \ell := \ell$$
$$\mathsf{cons}(a, \ell_1) +\!\!+ \ell_2 := \mathsf{cons}(a, \ell_1 +\!\!+ \ell_2)$$

Theorem
$\mathsf{length}(\ell_1 +\!\!+ \ell_2) = \mathsf{length}(\ell_1) + \mathsf{length}(\ell_2)$

Proof.
By induction on ℓ_1.

2. When ℓ_1 is $\mathsf{cons}(a, \ell_1')$, we have

$$\begin{aligned}\mathsf{length}(\mathsf{cons}(a, \ell_1') +\!\!+ \ell_2) &= \mathsf{length}(\mathsf{cons}(a, \ell_1' +\!\!+ \ell_2))\\ &= \mathsf{s}(\mathsf{length}(\ell_1' +\!\!+ \ell_2))\\ &= \mathsf{s}(\mathsf{length}(\ell_1') + \mathsf{length}(\ell_2))\\ &= \mathsf{s}(\mathsf{length}(\ell_1')) + \mathsf{length}(\ell_2)\\ &= \mathsf{length}(\mathsf{cons}(a, \ell_1')) + \mathsf{length}(\ell_2) \quad \square\end{aligned}$$

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

$$\mathsf{list} : \mathsf{Type} \to \mathsf{Type}$$

But each type list(A) is separately inductively defined; the constructors don't "hop around" between different As.

Indices remove this restriction.

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 `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.

Programming with indices

For any A, we can define a dependently typed function

$$\text{concat} : \prod_{n:\mathbb{N}} \Big(\text{vec}(A,n) \to \prod_{m:\mathbb{N}} \Big(\text{vec}(A,m) \to \text{vec}(A,n+m)\Big)\Big)$$

as follows:

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

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

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

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

The definition and behavior of "length" are built into the type.

Induction with indices

Theorem

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

$$v_1 +\!\!+ (v_2 +\!\!+ v_3) = (v_1 +\!\!+ v_2) +\!\!+ v_3$$

Proof.

By induction on v_1.

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

□

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.

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.

Parameters versus indices

Define $\mathsf{sum}\colon \mathsf{listP}(\mathbb{N}) \to \mathbb{N}$ by

$$\mathsf{sum}(\mathsf{nilP}) := 0$$
$$\mathsf{sum}(\mathsf{consP}(a, \ell)) := a + \mathsf{sum}(\ell)$$

Theorem
$\mathsf{sum}(\ell_1 +\!\!+ \ell_2) = \mathsf{sum}(\ell_1) + \mathsf{sum}(\ell_2)$

Proof.
By induction… □

With `listI` this is a non-starter.

Proving something about `listI`$(\mathbb{N})$ by induction is like proving "3 is prime" by induction on 3.

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.

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 `listP`(A) can be proven by induction." But this statement is general over all A, hence follows from the induction principle of `listI`. □

Trickier induction with indices

Theorem
For any $v\colon \mathsf{vec}(A, 0)$ *we have* $v = \mathsf{vnil}$.

Proof.
By induction??

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

Trickier induction with indices

Theorem
For any $v\colon \mathrm{vec}(A,0)$ we have $v = \mathrm{vnil}$.

Proof.
Define $P\colon \prod_{n\colon \mathbb{N}}(\mathrm{vec}(A,n) \to \mathrm{Prop})$ by induction on n:

$$P(0,v) :\equiv (v = \mathrm{vnil})$$
$$P(\mathrm{s}n,v) :\equiv \top$$

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

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

Finally, let $n = 0$. □

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.

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

- 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-Möhring equality), but not also b.

Induction on equality

The eliminator for equality is:

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

In words:

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

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 $C(b)$.

Properties of equality

Theorem
Equality is transitive.

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

Theorem
Equality is symmetric.

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

A trickier application

Theorem
$0 \neq 1$.

Proof.
Suppose $p\colon (0 = 1)$. Define $C\colon \mathbb{N} \to \mathsf{Type}$ by "recursion":

$$C(0) := \mathsf{unit}$$
$$C(sn) := \emptyset$$

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

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.

A non-application

Theorem
For any $p\colon (a = a)$ we have $p = \mathsf{refl}_a$.

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

This is not valid.
The type of p is not fully general.
We are trying to prove "3 is prime" by induction on 3.

Intensional equality types

There are ways to formulate the rules of inductive type families so that $p = \mathsf{refl}_a$ 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 $p = \mathsf{refl}_a$ does hold automatically, so this merely expands the scope of the theory.

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/`