MPRI 2-7-2: Proof Assistants

Bruno Barras

Sep 27, 2019

Update!

Update!

- No lecture on October 11th:
  5th lecture (Matthieu's first) on October 18th

Advertisement! Internships

- Stitching functions (application to HoTT)
  define $f + g : \mathbb{Z} \to X$ by stitching $f : \mathbb{Z}^- \to X$ and $g : \mathbb{Z}^+ \to X$

```
stitch :∀(f:Z->X)(g:∀n:Z, {m:X | n=0->m=f 0}), Z->X
```

- Relating Set and Type Theory (HoTT's univalence helps!)
  Find a restriction of ZF such that:
  Type Theoretic Universes have the same strength as Grothendieck Universes (aka inaccessible cardinals)
- Reduction machines for the λ-calculus
  Strong Call-by-Need

(implementation in the Type Theory of DEDUKTI)

Recap

Last week:

- Universes
  - Stratification of types to avoid paradoxes `Type`$_i$
  - Impredicativity of `Prop`: `(forall P:Prop, P): Prop` !
- Impredicative encodings of datatypes is limited
  - Cannot prove $0 \neq 1$, or the induction scheme
- Dependent types
  - $\Pi x : A.\, B$ and $\Sigma x : A, B$ dependent products and sums
    $f : \Pi x : A.\, B \vdash f\ a : B[a/x]$ and
    $c : \Sigma x : A.\, B \vdash \pi_2(c) : B[\pi_1(c)/x]$
  - $\forall x : A.\, B = \Pi x : A.\, B$ and $\exists x : A.\, B \approx \Sigma x : A.\, B$

Overview

Type of Natural Numbers

Martin-Löf scheme (form/intro/elim/comp):

- 1 formation rule:

$$\frac{}{\vdash \mathbb{N} : \textbf{Type}}$$

- 2 introduction rules:

$$\frac{}{\vdash 0 : \mathbb{N}} \qquad \frac{\vdash n : \mathbb{N}}{\vdash S(n) : \mathbb{N}}$$

- 1 elimination rule (least type closed by 0 and S)

$$\frac{\vdash P : \mathbb{N} \to \textbf{Type} \quad \vdash m : \mathbb{N} \quad \vdash f_0 : P(0) \quad \vdash f_S : \Pi n{:}\mathbb{N}.\, P(n) \to P(S(n))}{\vdash Rec(f_0, f_S, m) : P(m)}$$

- 2 computation rules

$$Rec(f_0, f_S, 0) = f_0 \quad Rec(f_0, f_S, S(n)) = f_S(n, Rec(f_0, f_S, n))$$

Dependent vs non-dependent elimination

The induction scheme:

$$\frac{\vdash P : \mathbb{N} \to \textbf{Type} \quad \vdash n : \mathbb{N} \qquad \vdash f_0 : P(0) \quad \vdash f_S : \Pi n{:}\mathbb{N}.\, P(n) \to P(S(n))}{\vdash Rec(f_0, f_S, n) : P(n)}$$

If we drop the dependent types (P is a constant type):

$$\frac{\vdash P : \textbf{Type} \quad \vdash n : \mathbb{N} \qquad \vdash f_0 : P \quad \vdash f_S : \mathbb{N} \to P \to P}{\vdash Rec(f_0, f_S, n) : P}$$

$\Rightarrow$ This is the recursor of Gödel's T!

Conclusions:

- Induction scheme and recursor is another instance of the Curry-Howard isomorphism
- The recursor of Gödel's T is a non-dependent specialization of the induction scheme

Inductive types in Coq

Coq provides the user with a general mechanism:

- Inductive type specified by the introduction rules (called constructors)
- A dependent induction/recursion scheme is derived systematically (called eliminator)
- Computation rules derived systematically (ι-reduction)

Comparison with Martin-Löf's inductive types:

- Coq checks the definition preserves consistency (but not complete!)
  $\Rightarrow$ Strictly positive inductive definitions
- Coq allows impredicative inductive definitions (defined later...)
- Coq uses style of Pure Type Systems

Natural numbers in Coq

Declaration of the natural numbers:

```
Inductive nat : Type :=
| O : nat | S : nat -> nat.
```

(or `Inductive nat := O | S (n:nat).`)

which defines

- a type $\Gamma \vdash \texttt{nat} : \textbf{Type}$
- a set of introduction rules for this type : constructors

$$\Gamma \vdash O : \texttt{nat} \qquad \frac{\Gamma \vdash n : \texttt{nat}}{\Gamma \vdash S\ n : \texttt{nat}}$$

Natural numbers in Coq: elimination

which defines also

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

$$\frac{\begin{array}{cc}\vdash t : \texttt{nat} & x{:}\texttt{nat} \vdash A(x) : \texttt{Type} \\ \vdash t_1 : A(O) & n{:}\texttt{nat} \vdash t_2 : A(S\ n)\end{array}}{\vdash \texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end} : A(t)}$$

- reduction rules preserve typing (ι-reduction)

$$(\texttt{match}\ O\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end}) \rightarrow_\iota t_1$$
$$(\texttt{match}\ S\ m\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end}) \rightarrow_\iota t_2[m/n]$$

Natural numbers in Coq: case-analysis

- We obtain case analysis and construction by cases :

$$
\begin{array}{l}
\lambda P : \texttt{nat} \to s. \\
\lambda H_O : P(O). \\
\lambda H_S : \forall m : \texttt{nat}. P(S\ m). \\
\lambda n : \texttt{nat}. \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\quad\ |\ O\ \texttt{=>}\ H_O \\
\quad\ |\ S\ m\ \texttt{=>}\ H_S\ m \\
\quad \texttt{end}
\end{array}
$$

- is a proof of

$$\forall P : \texttt{nat} \to s. P(O) \to (\forall m : \texttt{nat}. P(S\ m)) \to \forall n : \texttt{nat}. P(n)$$

How to derive the standard recursion scheme ?

Fixpoint : from case analysis to induction

case-analysis

$\lambda P : \texttt{nat} \to \texttt{Type},$
$\lambda H_O : P(O),$
$\lambda H_S : \forall m : \texttt{nat}, P(S\ m),$
$\lambda n : \texttt{nat},$
$\texttt{match}\ n\ \texttt{return}\ P(n)\ \texttt{with}$
$\quad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m$
$\texttt{end}$

has type

$\forall P : \texttt{nat} \to \texttt{Type},$
$P(O) \to$
$(\forall m : \texttt{nat}, P(S\ m)) \to$
$\forall n : \texttt{nat}, P(n)$

recursor

$\lambda P : \texttt{nat} \to \texttt{Type},$
$\lambda H_O : P(O),$
$\lambda H_S : \forall m : \texttt{nat}, P(m) \to P(S\ m),$
$\texttt{fix}\ f\ (n : \texttt{nat})\ :\ P(n)\ :=$
$\quad \texttt{match}\ n\ \texttt{return}\ P(n)\ \texttt{with}$
$\quad\quad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\ (f\ m)$
$\quad \texttt{end}$

has type

$\forall P : \texttt{nat} \to \texttt{Type},$
$P(O) \to$
$(\forall m : \texttt{nat}, P(m) \to P(S\ m)) \to$
$\forall n : \texttt{nat}, P(n)$

Fixpoint operator : well-foundness

Requirement of the Calculus of Inductive Constructions :

- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are *structurally* smaller

Example of recursor on natural numbers

$$
\begin{array}{l}
\lambda P : \texttt{nat} \to s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \texttt{nat}, P(m) \to P(S\ m), \\
\texttt{fix}\ f\ (n : \texttt{nat})\ :\ P(n)\ := \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\ (f\ m) \\
\quad \texttt{end}
\end{array}
$$

is correct with respect to CCI : recursive call on m which is structurally smaller than n in the inductive `nat`.

Inductive types with parameters

Example of lists

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

which defines

- a family of types $\dfrac{}{\Gamma \vdash \mathit{list} : \mathbf{Type} \to \mathbf{Type}}$
- a set of introduction rules for the types in this family

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathtt{nil}_A : \mathit{list}\ A} \qquad \frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash l : \mathit{list}\ A}{\Gamma \vdash \mathtt{cons}_A\ a\ l : \mathit{list}\ A}$$

Inductive types with parameters

Example of lists : elimination

- An elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

$$\frac{\Gamma \vdash l : \mathit{list}\ A \quad \Gamma, x : \mathit{list}\ A \vdash C(x) : s \qquad \Gamma \vdash t_1 : C(\mathtt{nil}) \quad \Gamma, a : A, l : \mathit{list}\ A \vdash t_2 : C(\mathtt{cons}_A\ a\ l)}{\Gamma \vdash \left(\begin{array}{l}\mathtt{match}\ l\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil} \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right) : C(l)}$$

- reduction rules which preserve typing (ι-reduction)

$$\left(\begin{array}{l}\mathtt{match}\ \mathtt{nil}_A\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil} \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right) \to_\iota t_1$$

$$\left(\begin{array}{l}\mathtt{match}\ \mathtt{cons}_A\ a'\ l'\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil} \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right) \to_\iota t_2[a', l'/a, l]$$

Infinitely branching trees in Coq

Declaration of the infinitely branching trees:

```
Inductive tree (A:Type) : Type :=
| Leaf : tree A
| Node : A -> (nat -> tree A) -> tree A.
tree is defined
tree_rect is defined
tree_ind is defined
tree_rec is defined

tree_rect =
fun (A : Type) (P : tree A->Type) (f : P (Leaf A))
 (f0 : forall (a : A) (t : nat -> tree A),
      (forall n:nat, P (t n)) -> P (Node A a t)) =>
fix F (t : tree A) : P t :=
  match t as t0 return (P t0) with
  | Leaf => f
  | Node y t0 => f0 y t0 (fun n : nat => F (t0 n))
  end
```

Mutual inductive types

Define several types referring one to each other:

```
Inductive tree (A:Type) :=
| node (_:A) (_:forest A)
with forest (A:Type) :=
| nil
| next (_:tree A) (_:forest A).
```

Type of finitely branching trees

Trees and forests: elimination

Elimination: pattern-matching and fixpoint

- Pattern-matching as usual
- Mutual fix:

```
fix size (t:tree A) : nat :=
  match t with node _ f => S(sizef f) end
with sizef (f:forest A) : nat :=
  match f with
  | nil => 0
  | next t f' => size t + sizef f'
  end
```

Beware: by default Coq generates schemes that do not consider cross-recursion! (use command `Scheme`)

Elimination scheme

$$
\begin{array}{l}
\lambda(A : \mathbf{Type})(P : \texttt{tree}\ A \to s)(Q : \texttt{forest}\ A \to s), \\
\lambda H_{node}\ H_{nil}\ H_{next}, \\
\texttt{fix}\ f\ (t : \texttt{tree}\ A)\ :\ P(t)\ := \\
\quad \texttt{match}\ t\ \texttt{return}\ P(t)\ \texttt{with} \\
\qquad \texttt{node}\ x\ f \Rightarrow H_{node}\ x\ f\ (g\ f) \\
\quad \texttt{end} \\
\texttt{with}\ g\ (f : \texttt{forest}\ A)\ :\ Q(f)\ := \\
\quad \texttt{match}\ f\ \texttt{return}\ Q(f)\ \texttt{with} \\
\qquad \texttt{nil} \Rightarrow H_{nil} \mid \texttt{next}\ t\ f' \Rightarrow H_{next}\ t\ (f\ t)\ f'\ (g\ f') \\
\quad \texttt{end} \\
\texttt{in}\ f
\end{array}
$$

$$
\begin{array}{ll}
f : & \forall(A : \mathbf{Type})(P : \texttt{tree}\ A \to s)(Q : \texttt{forest}\ A \to s), \\
& (\forall x\ f, Q(f) \to P(\texttt{node}\ x\ f)) \to \\
& Q(\texttt{nil}) \to \\
& (\forall t, P(t) \to \forall f, Q(f) \to Q(\texttt{next}\ t\ f)) \to \\
& \forall t : \texttt{tree}\ A, P(t)
\end{array}
$$

Nested inductive types

Note: forests are lists of trees!

Reuse list definition:

```
Inductive tree (A:Type) :=
| node (_:A) (_:list (tree A)).
```

(in principle, closer to infinitely branching trees)

Elimination

Nested fix:

$$\lambda(A : \textbf{Type})(P : \texttt{tree}\ A \to s)(Q : \texttt{list}(\texttt{tree}\ A) \to s),$$
$$\lambda H_{node}\ H_{nil}\ H_{cons},$$
$$\texttt{fix}\ f\ (t : \texttt{tree}\ A)\ :\ P(t)\ :=$$
$$\texttt{match}\ t\ \texttt{return}\ P(t)\ \texttt{with}$$
$$\texttt{node}\ x\ f \Rightarrow \texttt{list\_rec}\ (\texttt{tree}\ A)\ Q\ H_{nil}\ (\lambda x\ l\ (H_l : Q(l)), H_{cons}\ x\ (f\ x)\ l\ H_l)$$
$$\texttt{end}$$

$$f : \forall(A : \textbf{Type})(P : \texttt{tree}\ A \to s)(Q : \texttt{list}(\texttt{tree}\ A) \to s),$$
$$(\forall x\ f, Q(f) \to P(\texttt{node}\ x\ f)) \to$$
$$Q(\texttt{nil}) \to$$
$$(\forall t, P(t) \to \forall f, Q(f) \to Q(\texttt{cons}\ t\ f)) \to$$
$$\forall t : \texttt{tree}\ A, P(t)$$

Mutual vs nested inductive types

Mutual and nested inductive types provide similar features:

- Paulin has shown reduction of nested to mutual
- Converse reduction also holds

However, their usage is different (FAQ!):

- Cannot use mutual fix with nested inductive defs
- Cannot use nested fix with mutual inductive defs

Logical connectives

Disjunction example

```
Inductive or (A:Prop) (B:Prop) : Prop :=
| or_introl : A -> or A B
| or_intror : B -> or A B.
```

- General elimination rule

$$\frac{\Gamma \vdash t : or\ A\ B \quad \Gamma, x : or\ A\ B \vdash C(x) : \mathbf{Prop} \qquad \Gamma, p : A \vdash t_1 : C\,(or\_introl\ p) \quad \Gamma, q : B \vdash t_2 : C\,(or\_intror\ q))}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ C(x)\ \texttt{with} \\ \quad or\_introl\ p \Rightarrow t_1 \mid or\_intror\ q \Rightarrow t_2 \\ \texttt{end}\end{array}\right) : C(t)}$$

More logical connectives

The other logical connectives:

```
Inductive and (A:Prop) (B:Prop) : Prop :=
| conj : A -> B -> and A B.
Inductive True : Prop := I.
Inductive False : Prop := .
Inductive ex (A:Type)(P:A->Prop) : Prop :=
| ex_intro : forall (x:A), P x -> ex A P.
```

Exercise: guess the type of the generated eliminator.

Limitations of parameters

Defining a predicate:

```
Inductive even (n:nat) : Prop :=
  even_i (half:nat) (_:half+half=n).
```

Inductive types with parameters are some kind of “template”

```
Inductive listnat :=
  nilnat | consnat (_:nat) (_:listnat).
Inductive listbool :=
  nilbool | consbool (_:bool) (_:listbool).
```

No dependency between both types.

But in the definition of `even:nat->Prop` as an inductive type/set

$$\frac{}{\texttt{E}_0\texttt{:even 0}} \qquad \frac{\texttt{e:even n}}{\texttt{E}_{ss}\texttt{(e):even (S (S n))}}$$

`even (S (S O))` depends on `even O`.

Inductive families

Family = indexed type

`P : nat -> Type` represents the type family $(P(n))_{n \in \mathbb{N}}$

Inductive family:

- Constructors do not inhabit uniformly the members of the family
- Recursive arguments can change the value of the index

Even numbers:

```
Inductive even : nat -> Prop :=
  E0 : even O
| ESS (n:nat) (e:even n) : even (S (S n)).
```

Syntax very close to inference rules!

Elimination scheme

Elimination scheme: minimality of predicate, rule-induction

```
even_ind : forall (P:nat->Prop),
 P O -> (forall n, P n -> P (S (S n))) ->
 forall n, even n -> P n.
```

Seems the analogous of nat's dependent scheme

Elimination scheme

Elimination scheme: minimality of predicate, rule-induction

```
even_ind : forall (P:nat->Prop),
 P O -> (forall n, P n -> P (S (S n))) ->
 forall n, even n -> P n.
```

Seems the analogous of nat's dependent scheme (NOT!)

Even's dependent scheme (refers to constructors `E0` and `ESS`):

```
 forall (P : forall n, even n -> Prop),
 P 0 E0 ->
 (forall n (e:even n), P n e -> P (S (S n)) (ESS n e)) ->
 forall n (e:even n), P n e
```

Definable in Coq, but not automatically generated (why? wait and see...)

Defining the dependent elimination scheme

Even more complex return clause: in

```
Definition even_ind_dep (P:forall n , even n -> Prop)
  (h0:P 0 E0)
  (hSS:forall n e, P n e -> P (S (S n)) (ESS n e))
  : forall n, even n -> P n :=
  fix F n e :=
  match e as e' in even k return P k e' with
  | E0 => h0 :      P 0 E0
  | ESS k e' =>
    hSS k e' (F k e') : P (S (S k)) (ESS k e')
  end
```

Notation `as e' in even k return P k e'` is just a way to write the term `fun k e' => P k e'`.
Becomes natural with time...

Equality: the paradigmatic indexed family

Propositional equality is defined as:

```
Inductive eq (A : Type) (a : A) : A -> Prop :=
  eq_refl : eq A a a.
Notation "x = y" := (eq x y).
```

Its dependent elimination principle is of the form:

$$\frac{\Gamma \vdash e : eq\ A\ t\ u \qquad \Gamma, y : A, e' : eq\ A\ t\ y \vdash C(y, e') : s \qquad \Gamma \vdash t : C(t, \text{eq\_refl}_{A,t})}{\Gamma \vdash \left(\begin{array}{l} \texttt{match}\ e\ \texttt{as}\ e'\ \texttt{in}\ eq\ \_\ y\ \texttt{return}\ C(y, e')\ \texttt{with} \\ \quad \text{eq\_refl} \Rightarrow t \\ \texttt{end} \end{array} \right) : C(u, e)}$$

Tactics related to equality

Tactics:

- `f_equal` (congruence) $\frac{x=y}{f(x)=f(y)}$
- `discriminate` (constructor discrimination) $\frac{C(t_1,\ldots,t_n)=D(u_1,\ldots,u_k)}{A}$
- `injection` (injectivity of constructors) $\frac{C(t_1,..,t_n)=C(u_1,\ldots,u_n)}{t_1=u_1 \quad \ldots \quad t_n=u_n}$
- `inversion` (necessary conditions) $\frac{\text{even } (S(Sn))}{\text{even } n}$
- `rewrite` (substitution) $\frac{x=y \quad P(y)}{P(x)}$
- `symmetry`, `transitivity`

Inductive types with parameters and index

Example of vectors with size

```
Inductive vect (A:Type) : nat -> Type :=
| niln : vect A O
| consn :
    A -> forall n:nat, vect A n -> vect A (S n).
```

which defines

- a family of types-predicates:
  $\Gamma \vdash \mathit{vect} : \mathbf{Type} \to \mathit{nat} \to \mathbf{Type}$
- a set of introduction rules for the types in this family

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathtt{niln}_A : \mathit{vect}\ A\ O}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : \mathit{nat} \quad \Gamma \vdash l : \mathit{vect}\ A\ n}{\Gamma \vdash \mathtt{consn}_A\ a\ n\ l : \mathit{vect}\ A\ (S\ n)}$$

Inductive types with parameters and index

vectors : elimination

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathit{vect}\ A\ n \qquad \Gamma, m{:}\mathit{nat}, x{:}\mathit{vect}\ A\ m \vdash C(m,x) : s \\ \Gamma \vdash t_1 : C(O, \mathtt{niln}_A) \\ \Gamma, a{:}A, n{:}\mathit{nat}, l{:}\mathit{vect}\ A\ n \vdash t_2 : C(S\ n, \mathtt{consn}_A\ a\ n\ l)\end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathtt{match}\ v\ \mathtt{as}\ x\ \mathtt{in}\ \mathit{vect}\ \_\ p\ \mathtt{return}\ C(p,x)\ \mathtt{with} \\ \quad \mathtt{niln} \Rightarrow t_1 \mid \mathtt{consn}\ a\ n\ l \Rightarrow t_2 \\ \mathtt{end}\end{array}\right) : C(n,v)}$$

Inductive types with parameters and index

- reduction rules preserve typing (ι-reduction)

$$
\begin{array}{l}
\left(\begin{array}{l}
\texttt{match}\ \texttt{niln}_A\ \texttt{as}\ x\ \texttt{in}\ \mathit{vect}\ \_\ p\ \texttt{return}\ C(x,p)\ \texttt{with} \\
\quad \texttt{niln} \Rightarrow t_1 \mid \texttt{consn}\ a\ n\ l \Rightarrow t_2 \\
\texttt{end}
\end{array}\right) \\
\rightarrow_\iota \quad t_1 \\
\left(\begin{array}{l}
\texttt{match}\ \texttt{consn}_A\ a'\ n'\ l'\ \texttt{as}\ x\ \texttt{in}\ \mathit{vect}\ \_\ p\ \texttt{return}\ C(x,p)\ \texttt{with} \\
\quad \texttt{niln} \Rightarrow t_1 \mid \texttt{consn}\ a\ n\ l \Rightarrow t_2 \\
\texttt{end}
\end{array}\right) \\
\rightarrow_\iota \quad t_2[a', n', l'/a, n, l]
\end{array}
$$

Non-uniform parameters

Non-uniform parameter:

- Instance in constructor return type: like parameters
- Instance in constructor arguments: arbitrary value

```
Inductive mistery (A:Type) :=
| C0 : A -> mistery A
| CS : mistery (A*A) -> mistery A.
```

What's this type?

Non-uniform parameters

Non-uniform parameter:

- Instance in constructor return type: like parameters
- Instance in constructor arguments: arbitrary value

```
Inductive mistery (A:Type) :=
| C0 : A -> mistery A
| CS : mistery (A*A) -> mistery A.
```

What's this type?

```
Definition t4 : mistery nat :=
  CS nat (CS (nat*nat) (C0 _ ((1,2),(3,4)))).
```

... a 2^n-tuple of A (for some n)

Non-uniform parameters

Non-uniform parameter:

- Instance in constructor return type: like parameters
- Instance in constructor arguments: arbitrary value

```
Inductive mistery (A:Type) :=
| C0 : A -> mistery A
| CS : mistery (A*A) -> mistery A.
```

What's this type?

```
Definition t4 : mistery nat :=
  CS nat (CS (nat*nat) (C0 _ ((1,2),(3,4)))).
```

... a 2^n-tuple of A (for some n) = a complete binary tree

Elimination rules

Pattern-matching:

$$\frac{\Gamma \vdash e : \mathit{tuple}\ A \quad \Gamma, h : \mathit{tuple}\ A \vdash P(h) : s \qquad \Gamma, x : A \vdash t_0 : P(H0\ A\ x) \quad \Gamma, h : \mathit{tuple}(A * A) \vdash t_S : P(HS\ A\ h)}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ e\ \texttt{as}\ h\ \texttt{return}\ P(h)\ \texttt{with} \\ \qquad H0\ x \Rightarrow t_0 \\ \quad \mid \quad HS\ h \Rightarrow t_S \\ \texttt{end}\end{array}\right) : P(e)}$$

Elimination:

```
tuple_rect :
  forall (P:forall A, tuple A -> Type),
  (forall A x, P A (H0 A x)) ->
  (forall A h, P (A*A) h -> P A (HS A h)) ->
  forall A (h:tuple A), P A h.
```

Encoding inductive families

Non-uniform parameters can encode inductive families:

```
Inductive even (n:nat) : Prop :=
  E0' (_:n=0)
| ESS' (k:nat) (e:even k) (_:n=S (S k)).
Definition E0 : even 0 := E0' 0 eq_refl.
Definition ESS n e : even (S (S n)) :=
  ESS' (S (S n)) n e eq_refl.
```

Next week...

- Termination of fixpoints
  Guard condition
- Valid inductive definitions
  (Strict) Positivity