Lecture 5: Inductive types

Łukasz Czajka

Primitive recursive functions

We consider n-argument functions $f : \mathbb{N}^n \to \mathbb{N}$.

Primitive recursive functions

We consider n-argument functions $f : \mathbb{N}^n \to \mathbb{N}$.

The basic primitive recursive functions consist of:

- constant functions $c^n_m(x_1, \ldots, x_n) = m$ for a fixed $m \in \mathbb{N}$,
- successor function $S(x) = x + 1$,
- identity functions $\mathtt{id}^n_k(x_1, \ldots, x_n) = x_k$.

Primitive recursive functions

We consider n-argument functions $f : \mathbb{N}^n \to \mathbb{N}$.

The basic primitive recursive functions consist of:

- constant functions $c^n_m(x_1, \ldots, x_n) = m$ for a fixed $m \in \mathbb{N}$,
- successor function $S(x) = x + 1$,
- identity functions $\mathtt{id}^n_k(x_1, \ldots, x_n) = x_k$.

The class of primitive recursive functions is the smallest class of functions containing the basic primitive recursive functions and closed under the operations of composition and primitive recursion.

Primitive recursive functions

We consider n-argument functions $f : \mathbb{N}^n \to \mathbb{N}$.

The basic primitive recursive functions consist of:

- constant functions $c^n_m(x_1, \ldots, x_n) = m$ for a fixed $m \in \mathbb{N}$,
- successor function $S(x) = x + 1$,
- identity functions $\mathtt{id}^n_k(x_1, \ldots, x_n) = x_k$.

The class of primitive recursive functions is the smallest class of functions containing the basic primitive recursive functions and closed under the operations of composition and primitive recursion.

- **Composition**. If $f : \mathbb{N}^n \to \mathbb{N}$ and $g_i : \mathbb{N}^m \to \mathbb{N}$ for $i = 1, \ldots, n$, then the function $h : \mathbb{N}^m \to \mathbb{N}$ satisfying

  $$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

  is the composition of f with $g_1, \ldots, g_n$.

Primitive recursive functions

We consider n-argument functions $f : \mathbb{N}^n \to \mathbb{N}$.

The basic primitive recursive functions consist of:

- constant functions $c^n_m(x_1, \ldots, x_n) = m$ for a fixed $m \in \mathbb{N}$,
- successor function $S(x) = x + 1$,
- identity functions $\mathtt{id}^n_k(x_1, \ldots, x_n) = x_k$.

The class of primitive recursive functions is the smallest class of functions containing the basic primitive recursive functions and closed under the operations of composition and primitive recursion.

- **Composition**. If $f : \mathbb{N}^n \to \mathbb{N}$ and $g_i : \mathbb{N}^m \to \mathbb{N}$ for $i = 1, \ldots, n$, then the function $h : \mathbb{N}^m \to \mathbb{N}$ satisfying

$$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

  is the composition of f with $g_1, \ldots, g_n$.
- **Primitive recursion.** If $h : \mathbb{N}^{n+1} \to \mathbb{N}$ and $g : \mathbb{N}^{n-1} \to \mathbb{N}$ then the unique function $f : \mathbb{N}^n \to \mathbb{N}$ satisfying

$$\begin{aligned} f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(S(x), x_2, \ldots, x_n) &= h(x, f(x, x_2, \ldots, x_n), x_2, \ldots, x_n) \end{aligned}$$

  is defined by primitive recursion from g and h.

Digression: partial recursive functions

The class of partial recursive functions is the smallest class of partial functions containing the basic primitive recursive functions and closed under composition, primitive recursion and minimisation.

Digression: partial recursive functions

The class of partial recursive functions is the smallest class of partial functions containing the basic primitive recursive functions and closed under composition, primitive recursion and minimisation.

Minimisation. If $h : \mathbb{N}^{n+1} \rightarrowtail \mathbb{N}$ then the minimisation f of h is defined as follows:

- if there exists $m \in \mathbb{N}$ such that $h(x_1, \ldots, x_n, m) = 0$ and for all $i < m$ the value $h(x_1, \ldots, x_n, i)$ is defined and nonzero, then $f(x_1, \ldots, x_n) = m$;
- if such an m does not exist, then $f(x_1, \ldots, x_n)$ is undefined.

Digression: partial recursive functions

The class of partial recursive functions is the smallest class of partial functions containing the basic primitive recursive functions and closed under composition, primitive recursion and minimisation.

Minimisation. If $h : \mathbb{N}^{n+1} \rightarrowtail \mathbb{N}$ then the minimisation f of h is defined as follows:

- if there exists $m \in \mathbb{N}$ such that $h(x_1, \dots, x_n, m) = 0$ and for all $i < m$ the value $h(x_1, \dots, x_n, i)$ is defined and nonzero, then $f(x_1, \dots, x_n) = m$;
- if such an m does not exist, then $f(x_1, \dots, x_n)$ is undefined.

The total recursive functions are those partial recursive functions which happen to be total. These are exactly the computable functions on natural numbers.

Primitive recursion

If $h : \mathbb{N}^{n+1} \to \mathbb{N}$ and $g : \mathbb{N}^{n-1} \to \mathbb{N}$ then the unique function $f : \mathbb{N}^n \to \mathbb{N}$ satisfying

$$
\begin{aligned}
f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\
f(S(x), x_2, \ldots, x_n) &= h(x, f(x, x_2, \ldots, x_n), x_2, \ldots, x_n)
\end{aligned}
$$

is defined by primitive recursion from g and h.

Primitive recursion

If $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $g \in \mathbb{N}$ then the unique function $f : \mathbb{N} \to \mathbb{N}$ satisfying

$$\begin{aligned} f(0) &= g \\ f(S(x)) &= h(x, f(x)) \end{aligned}$$

is defined by primitive recursion from g and h.

Higher-type primitive recursion

If α : `Type` and $h : \mathbb{N} \to \alpha \to \alpha$ and $a : \alpha$ then the unique function $f : \mathbb{N} \to \alpha$ satisfying

$$\begin{aligned} f0 &= a \\ f(Sn) &= hn(fn) \end{aligned}$$

is defined from a and h by $\underline{\text{primitive recursion into type } \alpha}$.

Higher-type primitive recursion

If $\alpha : \texttt{Type}$ and $h : \mathbb{N} \to \alpha \to \alpha$ and $a : \alpha$ then the unique function $f : \mathbb{N} \to \alpha$ satisfying

$$\begin{aligned} f0 &= a \\ f(Sn) &= hn(fn) \end{aligned}$$

is defined from a and h by <u>primitive recursion into type α</u>.

$\texttt{nat\_srec}_\alpha$ `: ` α `-> (nat ->` α `->` α`) -> nat ->` α
$\texttt{nat\_srec}_\alpha$ `a h 0` $\to_\iota$ `a`
$\texttt{nat\_srec}_\alpha$ `a h (S n)` $\to_\iota$ `h n (`$\texttt{nat\_srec}_\alpha$ `a h n)`

Higher-type primitive recursion

If α : Type and $h : \mathbb{N} \to \alpha \to \alpha$ and $a : \alpha$ then the unique function $f : \mathbb{N} \to \alpha$ satisfying

$$\begin{aligned} f0 &= a \\ f(Sn) &= hn(fn) \end{aligned}$$

is defined from a and h by primitive recursion into type α.

$\texttt{nat\_srec}_\alpha$ `:` α `->` `(nat ->` α `->` α`) -> nat ->` α
$\texttt{nat\_srec}_\alpha$ `a h 0` $\to_\iota$ `a`
$\texttt{nat\_srec}_\alpha$ `a h (S n)` $\to_\iota$ `h n (`$\texttt{nat\_srec}_\alpha$ `a h n)`

Using this generalised `nat_srec` and higher-order functions it is possible to define the (non-primitive-recursive) Ackermann function.

Higher-type primitive recursion

Let $g : \mathbb{N}^{n-1} \to \mathbb{N}$ and $h : \mathbb{N}^{n+1} \to \mathbb{N}$. By currying/uncurrying we identify $\mathbb{N}^k \to \mathbb{N}$ with $\mathbb{N} \to \mathbb{N} \to \ldots \to \mathbb{N}$ where $\mathbb{N}$ occurs $k+1$ times.

Higher-type primitive recursion

Let $g : \mathbb{N}^{n-1} \to \mathbb{N}$ and $h : \mathbb{N}^{n+1} \to \mathbb{N}$. By currying/uncurrying we identify $\mathbb{N}^k \to \mathbb{N}$ with $\mathbb{N} \to \mathbb{N} \to \ldots \to \mathbb{N}$ where $\mathbb{N}$ occurs $k+1$ times.

The function $f : \mathbb{N}^n \to \mathbb{N}$ defined from g and

$$h'(x, y) = \lambda x_2 \ldots x_n . h(x, y(x_2, \ldots, x_n), x_2, \ldots, x_n)$$

by primitive recursion into type $\mathbb{N}^{n-1} \to \mathbb{N}$, satisfies:

$$\begin{aligned} &f 0 x_2 \ldots x_n = g(x_2, \ldots, x_n) \\ &f(Sn) x_2 \ldots x_n = h(x, f n x_2 \ldots x_n, x_2, \ldots, x_n) \end{aligned}$$

Non-dependent recursors

```
Inductive nat := 0 : nat | S : nat -> nat.
```

$\texttt{nat\_srec}_\alpha$: α -> (nat -> α -> α) -> nat -> α
$\texttt{nat\_srec}_\alpha$ a f 0 $\to_\iota$ a
$\texttt{nat\_srec}_\alpha$ a f (S n) $\to_\iota$ f n ($\texttt{nat\_srec}_\alpha$ a f n)

Non-dependent recursors

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

$\texttt{list\_srec}_{A,\alpha}$: α -> (A -> list A -> α -> α) -> list A -> α

$\texttt{list\_srec}_{A,\alpha}$ a f nil $\rightarrow_\iota$ a

$\texttt{list\_srec}_{A,\alpha}$ a f (cons x l) $\rightarrow_\iota$ f x l ($\texttt{list\_srec}_{A,\alpha}$ a f l)

Non-dependent recursors

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

list_src$_{A,\alpha}$: α -> (A -> list A -> α -> α) -> list A -> α
list_srec$_{A,\alpha}$ a f nil $\rightarrow_\iota$ a
list_srec$_{A,\alpha}$ a f (cons x l) $\rightarrow_\iota$ f x l (list_srec$_{A,\alpha}$ a f l)

List.fold_right : (A -> α -> α) -> α -> list A -> α
List.fold_right f a = list_srec a (fun x _ acc => f x acc)

Non-dependent recursors

```
Inductive tree (A : Set) :=
| leaf : A -> tree A
| node1 : A -> tree A -> tree A
| node2 : A -> tree A -> tree A -> tree A
| nodeN : (nat -> tree A) -> tree A.
```

$$
\begin{array}{l}
\mathtt{tree\_srec}_{A,\alpha} : \mathtt{(A \to}\ \alpha\mathtt{)} \to \mathtt{(A \to tree\ A \to}\ \alpha \to \alpha\mathtt{)} \to \\
\quad \mathtt{(A \to tree\ A \to}\ \alpha \to \mathtt{tree\ A} \to \alpha \to \alpha\mathtt{)} \to \\
\quad \mathtt{((nat \to tree\ A) \to (nat \to}\ \alpha\mathtt{)} \to \alpha\mathtt{)} \to \mathtt{tree\ A} \to \alpha \\
\mathtt{tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{(leaf\ x)} \to_\iota f_1\ \mathtt{x} \\
\mathtt{tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{(node1\ x\ t)} \to_\iota \\
\quad f_2\ \mathtt{x\ t\ (tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{t)} \\
\mathtt{tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{(node2\ x\ l\ r)} \to_\iota \\
\quad f_3\ \mathtt{x\ l\ (tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{l)} \\
\qquad\quad \mathtt{r\ (tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{r)} \\
\mathtt{tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{(nodeN\ h)} \to_\iota \\
\quad f_4\ \mathtt{h\ (fun\ n \Rightarrow (tree\_srec}_{A,\alpha}\ f_1\ f_2\ f_3\ f_4\ \mathtt{(h\ n)))}
\end{array}
$$

Non-dependent recursors

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

$\texttt{or\_srec}_{A,B,\alpha}$ `: (A ->` α`) -> (B ->` α`) -> A \/ B ->` α

$\texttt{or\_srec}_{A,B,\alpha}\ f_1\ f_2\ (\texttt{or\_introl x}) \to_\iota f_1\ \texttt{x}$

$\texttt{or\_srec}_{A,B,\alpha}\ f_1\ f_2\ (\texttt{or\_intror x}) \to_\iota f_2\ \texttt{x}$

Non-dependent recursors

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

$\texttt{or\_srec}_{A,B,\alpha}$: `(A ->` α`) -> (B ->` α`) -> A \/ B ->` α

$\texttt{or\_srec}_{A,B,\alpha}\ f_1\ f_2\ \texttt{(or\_introl x)} \to_\iota f_1\ \texttt{x}$

$\texttt{or\_srec}_{A,B,\alpha}\ f_1\ f_2\ \texttt{(or\_intror x)} \to_\iota f_2\ \texttt{x}$

Recall the higher-order encoding of disjunction:

$$\varphi \vee \psi := \forall P.(\varphi \to P) \to (\psi \to P) \to P$$

Non-dependent recursors

```
Inductive vector (A : Set) : nat -> Set :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).
```

$\texttt{vector\_srec}_{A,\alpha}$ `:` α `-> (A -> forall n, vector A n ->` α `->` α`) ->`
    `forall n, vector A n ->` α

$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2\ \texttt{0 nil} \rightarrow_\iota f_1$

$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2\ \texttt{(S n) (cons x n v)} \rightarrow_\iota$
    $f_2\ \texttt{x n v (vector\_srec}_{A,\alpha}\ f_1\ f_2\ \texttt{n v)}$

Non-dependent recursors

```
Inductive vector (A : Set) : nat -> Set :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).
```

$\texttt{vector\_srec}_{A,\alpha}$ `:` α `-> (A -> forall n, vector A n ->` α `->` α`) ->`
    `forall n, vector A n ->` α
$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `0 nil` $\to_\iota\ f_1$
$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `(S n) (cons x n v)` $\to_\iota$
    f_2 `x n v (`$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `n v)`

`vector_rec : forall (A : Set) (P : nat -> Set),`
    `P 0 -> (A -> forall n, vector A n -> P n -> P (S n)) ->`
    `forall n, vector A n -> P n`
`vector_rec A P` $f_1\ f_2$ `0 nil` $\to_\iota\ f_1$
`vector_rec A P` $f_1\ f_2$ `(S n) (cons x n v)` $\to_\iota$
    f_2 `x n v (vector_rec A P` $f_1\ f_2$ `n v)`

Recursors

```
Inductive Even : nat -> Prop :=
| Even_0 : Even 0
| Even_1 : forall n, Even n -> Even (S (S n)).

Even_rec : forall P : nat -> Prop,
    P 0 -> (forall n, Even n -> P n -> P (S (S n))) ->
    forall n, Even n -> P n
```

Even_rec P f_1 f_2 0 Even_0 $\to_\iota$ f_1

Even_rec P f_1 f_2 (S (S n)) (Even_1 n e) $\to_\iota$
    f_2 n e (Even_rec P f_1 f_2 n e)

Dependent elimination

```
Inductive Even : nat -> Prop :=
| Even_0 : Even 0
| Even_1 : forall n, Even n -> Even (S (S n)).
```

Even_elim : forall P : forall n, Even n -> Prop,
    P 0 Even_0 ->
    (forall n (e : Even n),
        P n e -> P (S (S n)) (Even_1 n e)) ->
    forall n (e : Even n), P n e

Even_elim P f_1 f_2 0 Even_0 $\to_\iota$ f_1

Even_elim P f_1 f_2 (S (S n)) (Even_1 n e) $\to_\iota$
    f_2 n e (Even_elim P f_1 f_2 n e)

Dependent elimination

```
Inductive nat : Set := 0 : nat | S : nat -> nat.
```

$$
\begin{array}{l}
\texttt{nat\_srec}_\alpha \texttt{ : } \alpha \texttt{ -> (nat -> } \alpha \texttt{ -> } \alpha \texttt{) -> nat -> } \alpha \\
\texttt{nat\_srec}_\alpha \texttt{ a f 0 } \rightarrow_\iota \texttt{ a} \\
\texttt{nat\_srec}_\alpha \texttt{ a f (S n) } \rightarrow_\iota \texttt{ f n (nat\_srec}_\alpha \texttt{ a f n)}
\end{array}
$$

$$
\begin{array}{l}
\texttt{nat\_elim : forall P : nat -> Set,} \\
\qquad \texttt{P 0 -> (forall n, P n -> P (S n)) -> forall n, P n} \\
\texttt{nat\_elim P a f 0 } \rightarrow_\iota \texttt{ a} \\
\texttt{nat\_elim P a f (S n) } \rightarrow_\iota \texttt{ f n (nat\_elim P a f n)}
\end{array}
$$

Dependent elimination

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

$$\texttt{list\_srec}_{A,\alpha}\ \texttt{:}\ \alpha\ \texttt{->}\ \texttt{(A -> list A ->}\ \alpha\ \texttt{->}\ \alpha\texttt{) -> list A ->}\ \alpha$$

$$\texttt{list\_srec}_{A,\alpha}\ \texttt{a f nil} \to_\iota \texttt{a}$$

$$\texttt{list\_srec}_{A,\alpha}\ \texttt{a f (cons x l)} \to_\iota \texttt{f x l (list\_srec}_{A,\alpha}\ \texttt{a f l)}$$

```
list_elim : forall A (P : list A -> Set),
    P nil -> (forall x l, P l -> P (cons x l)) ->
    forall l, P l
```

$$\texttt{list\_elim A P a f nil} \to_\iota \texttt{a}$$

$$\texttt{list\_elim A P a f (cons x l)} \to_\iota \texttt{f x l (list\_elim A P a f l)}$$

Dependent elimination

```
Inductive tree (A : Set) : Set :=
| leaf : A -> tree A
| node1 : A -> tree A -> tree A
| node2 : A -> tree A -> tree A -> tree A
| nodeN : (nat -> tree A) -> tree A.
```

```
tree_srec_{A,α} : (A -> α) -> (A -> tree A -> α -> α) ->
    (A -> tree A -> α -> tree A -> α -> α) ->
    ((nat -> tree A) -> (nat -> α) -> α) -> tree A -> α
```

```
tree_elim : forall (A : Set) (P : tree A -> Set),
    (forall x, P (leaf x)) ->
    (forall x t, P t -> P (node1 x t)) ->
    (forall x l, P l -> forall r, P r -> P (node2 x l r)) ->
    (forall (f : nat -> tree A),
        (forall n, P (f n)) -> P (nodeN f)) ->
    forall t : tree A, P t
```

Dependent elimination

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

$\texttt{or\_srec}_{A,B,\alpha}$: (A -> α) -> (B -> α) -> A \/ B -> α
$\texttt{or\_srec}_{A,B,\alpha}$ f_1 f_2 (or_introl x) $\rightarrow_\iota$ f_1 x
$\texttt{or\_srec}_{A,B,\alpha}$ f_1 f_2 (or_intror x) $\rightarrow_\iota$ f_2 x

```
or_elim : forall (A B : Prop) (P : A \/ B -> Prop),
    (forall x : A, P (or_introl x)) ->
    (forall x : B, P (or_intror x)) ->
    forall p : A \/ B, P p
```

or_elim A B P f_1 f_2 (or_introl x) $\rightarrow_\iota$ f_1 x
or_elim A B P f_1 f_2 (or_intror x) $\rightarrow_\iota$ f_2 x

Dependent elimination

```
Inductive vector (A : Set) : nat -> Set :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).
```

$\texttt{vector\_srec}_{A,\alpha}$ `:` α `-> (A -> forall n, vector A n ->` α `->` α`) ->`
    `forall n, vector A n ->` α

$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `0 nil` $\to_\iota f_1$

$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `(S n) (cons x n v)` $\to_\iota$
    f_2 `x n e (`$\texttt{vector\_srec}_{A,\alpha}\ f_1\ f_2$ `n v)`

```
vector_elim : forall (A : Set),
    forall (P : forall n : nat, vector A n -> Set),
    P 0 nil ->
    (forall x n (v : vector A n),
        P n v -> P (S n) (cons x n v)) ->
    forall (n : nat) (v : vector A n), P n v
```

`vector_elim A P` $f_1\ f_2$ `0 nil` $\to_\iota f_1$

`vector_elim A P` $f_1\ f_2$ `(S n) (cons x n v)` $\to_\iota$
    f_2 `x n v (vector_elim A P` $f_1\ f_2$ `n v)`

Dependent elimination = `fix` + `match`

In contrast to e.g. Martin-Löf type theory, in Coq the eliminators are not primitive constants, but can be implemented using `fix` and `match`.

Dependent elimination = `fix` + `match`

In contrast to e.g. Martin-Löf type theory, in Coq the eliminators are not primitive constants, but can be implemented using `fix` and `match`.

- `fix` implements the "structural recursion" aspect of an eliminator.

Dependent elimination = `fix` + `match`

In contrast to e.g. Martin-Löf type theory, in Coq the eliminators are not primitive constants, but can be implemented using `fix` and `match`.

- `fix` implements the "structural recursion" aspect of an eliminator.
- `match` implements the "reasoning by cases" aspect of an eliminator.

The `fix` expressions

`fix F :` τ `:= b`

where $\tau = \forall(x_1 : \sigma_1) \ldots (x_k : \sigma_k).\rho$ and σ_k is an inductive type.

The `fix` expressions

`fix F : ` τ ` := b`

where $\tau = \forall(x_1 : \sigma_1) \ldots (x_k : \sigma_k).\rho$ and σ_k is an inductive type.

- The k-th argument is the <u>principal argument</u>.

The `fix` expressions

```
fix F : τ := b
```

where $\tau = \forall(x_1 : \sigma_1)\ldots(x_k : \sigma_k).\rho$ and σ_k is an inductive type.

- The k-th argument is the principal argument.
- The definition needs to be guarded, meaning that in each recursive call the principal argument needs to structurally decrease.

The `fix` expressions

`fix F : τ := b`

where $\tau = \forall (x_1 : \sigma_1) \ldots (x_k : \sigma_k).\rho$ and σ_k is an inductive type.

- The k-th argument is the principal argument.
- The definition needs to be guarded, meaning that in each recursive call the principal argument needs to structurally decrease.
- For precise definitions see: Giménez, "Codifying guarded definitions with recursive schemes", TYPES 1994.

Reduction of fixpoints

```
(fix F : τ := b) a_1 ... a_k →ι
    b[(fix F : τ := b)/F] a_1 ... a_k
```

where a_k is the principal argument and it begins with a constructor: $a_k = ct_1 \ldots t_n$.

Reduction of fixpoints

```
(fix F : τ := b) a_1 ... a_k →ι
    b[(fix F : τ := b)/F] a_1 ... a_k
```

where a_k is the principal argument and it begins with a constructor: $a_k = ct_1 \ldots t_n$.

NOTE:

`(fix F (x : A) : τ := b) ≡ (fix F : A -> τ := fun x => b)`

Reduction of fixpoints

```
(fix F : τ := b) a_1 ... a_k →ι
    b[(fix F : τ := b)/F] a_1 ... a_k
```

where a_k is the principal argument and it begins with a constructor: $a_k = ct_1 \ldots t_n$.

NOTE:

`(fix F (x : A) : τ := b)` ≡ `(fix F : A -> τ := fun x => b)`

BTW, this "argument shifting" also works for `Definition`, `Lemma`, etc.:

```
Definition F (x : A) : τ := b.
≡
Definition F : A -> τ := fun x => b.
```

The `match` expressions

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

nat_elim : forall P : nat -> Set,
    P O -> (forall n, P n -> P (S n)) -> forall n, P n
nat_elim P a f O →ι a
nat_elim P a f (S n) →ι f n (nat_elim P a f n)

match n as x in nat return P x with
| O => a
| S n => b
end
```

The `match` expressions

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

nat_elim : forall P : nat -> Set,
    P O -> (forall n, P n -> P (S n)) -> forall n, P n
nat_elim P a f O →ι a
nat_elim P a f (S n) →ι f n (nat_elim P a f n)

match n as x in nat return P x with
| O => a
| S n => b
end
```

```
P : nat -> Set
a : P O
(fun n => b) : forall n, P (S n)
```

The match expressions

```
Inductive nat : Set := 0 : nat | S : nat -> nat.
```

```
nat_elim : forall P : nat -> Set,
    P 0 -> (forall n, P n -> P (S n)) -> forall n, P n
nat_elim P a f 0 →ι a
nat_elim P a f (S n) →ι f n (nat_elim P a f n)
```

```
match n as x in nat return P x with
| 0 => a
| S n => b
end
```

```
P : nat -> Set
a : P 0
(fun n => b) : forall n, P (S n)
```

```
nat_elim P a f :=
   fix F (n : nat) : P n := match n as x return P x with
                            | 0 => a
                            | S n => f n (F n)
                            end
```

The `match` expressions

```
match n as x in nat return P x with
| 0 => a
| S y => b
end
```

```
P : nat -> Set
a : P 0
(fun y => b) : forall y, P (S y)
```

The type of the entire match expression is `P n`.

The `match` expressions

```
match n as x in nat return P x with
| 0 => a
| S y => b
end
```

```
P : nat -> Set
a : P 0
(fun y => b) : forall y, P (S y)
```

The type of the entire match expression is `P n`.

```
match 0 as x in nat return P x with
| 0 => a
| S y => b
end
```
$\rightarrow_\iota$ `a`

```
match S n as x in nat return P x with
| 0 => a
| S y => b
end
```
$\rightarrow_\iota$ `b[n/y]`

The `match` expressions

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

list_elim : forall A (P : list A -> Set),
    P nil -> (forall x l, P l -> P (cons x l)) ->
    forall l, P l
list_elim A P a f nil →ι a
list_elim A P a f (cons x l) →ι f x l (list_elim A P a f l)

match l as x in list _ return P x with
| nil => a
| cons x l' => b
end

P : list A -> Set
a : P nil
(fun x l' => b) : forall x l', P (cons x l')
```

The type of the entire match expression is `P l`.

The `match` expressions

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

list_elim : forall A (P : list A -> Set),
    P nil -> (forall x l, P l -> P (cons x l)) ->
    forall l, P l

list_elim A P a f :=
   fix F (l : list A) : P l :=
     match l as x return P x with
     | nil => a
     | cons x l' => f x l' (F l')
     end
```

The `match` expressions

```
Inductive vector (A : Set) : nat -> Set :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).

vector_elim : forall (A : Set),
    forall (P : forall n : nat, vector A n -> Set),
    P 0 nil ->
    (forall x n (v : vector A n),
        P n v -> P (S n) (cons x n v)) ->
    forall (n : nat) (v : vector A n), P n v

match v as x in vector _ m return P m x with
| nil => a
| cons x y u => b
end

P : forall n, vector A n -> Set
a : P 0 nil
(fun x y u => b) : forall x y (u : vector A y), P (S y) (cons x y u)
```

If the actual type of `v` is `vector A n`, then the type of the entire match expression is `P n v`.

The `match` expressions

```
Inductive vector (A : Set) : nat -> Set :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).

vector_elim : forall (A : Set),
    forall (P : forall n : nat, vector A n -> Set),
    P 0 nil ->
    (forall x n (v : vector A n),
        P n v -> P (S n) (cons x n v)) ->
    forall (n : nat) (v : vector A n), P n v
vector_elim A P f1 f2 0 nil →ι f1
vector_elim A P f1 f2 (S n) (cons x n v) →ι
    f2 x n v (vector_elim A P f1 f2 n v)

vector_elim A P f1 f2 :=
   fix F (n : nat) (v : vector A n) {struct v} : P n v :=
     match v as x in vector _ m return P m x with
     | nil => f1
     | cons x y u => f2 x y u (F y u)
     end
```

General form of an inductive definition

```
Inductive I (p₁ : ρ₁) ... (p_k : ρ_k) :
   forall (i₁ : A₁) ... (i_m : A_m), 𝒰 :=
| c₁ : forall (x_{1,1} : τ_{1,1}) ... (x_{1,l₁} : τ_{1,l₁}), I p₁...p_k u_{1,1}...u_{1,m}
...
| c_n : forall (x_{n,1} : τ_{n,1}) ... (x_{n,l_n} : τ_{n,l_n}), I p₁...p_k u_{n,1}...u_{n,m}.
```

$$
\begin{array}{l}
\texttt{Inductive I } (p_1 : \rho_1) \ldots (p_k : \rho_k) : \\
\quad \texttt{forall } (i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U} := \\
\mid c_1 : \texttt{forall } (x_{1,1} : \tau_{1,1}) \ldots (x_{1,l_1} : \tau_{1,l_1}),\ \texttt{I}\ p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m} \\
\ldots \\
\mid c_n : \texttt{forall } (x_{n,1} : \tau_{n,1}) \ldots (x_{n,l_n} : \tau_{n,l_n}),\ \texttt{I}\ p_1 \ldots p_k\ u_{n,1} \ldots u_{n,m}.
\end{array}
$$

General form of an inductive definition

$$
\begin{array}{l}
\texttt{Inductive I } (p_1 : \rho_1) \ldots (p_k : \rho_k) : \\
\quad \texttt{forall } (i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U} := \\
\mid c_1 : \texttt{forall } (x_{1,1} : \tau_{1,1}) \ldots (x_{1,l_1} : \tau_{1,l_1}),\ \texttt{I}\ p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m} \\
\ldots \\
\mid c_n : \texttt{forall } (x_{n,1} : \tau_{n,1}) \ldots (x_{n,l_n} : \tau_{n,l_n}),\ \texttt{I}\ p_1 \ldots p_k\ u_{n,1} \ldots u_{n,m}.
\end{array}
$$

- Parameters: $p_1, \ldots, p_k$.

General form of an inductive definition

```
Inductive I (p₁ : ρ₁) ... (pₖ : ρₖ) :
   forall (i₁ : A₁) ... (iₘ : Aₘ), 𝒰 :=
| c₁ : forall (x₁,₁ : τ₁,₁) ... (x₁,ₗ₁ : τ₁,ₗ₁), I p₁ ... pₖ u₁,₁ ... u₁,ₘ
...
| cₙ : forall (xₙ,₁ : τₙ,₁) ... (xₙ,ₗₙ : τₙ,ₗₙ), I p₁ ... pₖ uₙ,₁ ... uₙ,ₘ.
```

- Parameters: $p_1, \ldots, p_k$.
- Indices: $i_1, \ldots, i_m$.

General form of an inductive definition

$$
\begin{array}{l}
\texttt{Inductive I } (p_1 : \rho_1) \ldots (p_k : \rho_k) : \\
\quad \texttt{forall } (i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U} \texttt{ :=} \\
\texttt{| } c_1 : \texttt{forall } (x_{1,1} : \tau_{1,1}) \ldots (x_{1,l_1} : \tau_{1,l_1}),\ \texttt{I } p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m} \\
\ldots \\
\texttt{| } c_n : \texttt{forall } (x_{n,1} : \tau_{n,1}) \ldots (x_{n,l_n} : \tau_{n,l_n}),\ \texttt{I } p_1 \ldots p_k\ u_{n,1} \ldots u_{n,m}.
\end{array}
$$

- Parameters: $p_1, \ldots, p_k$.
- Indices: $i_1, \ldots, i_m$.
- Arity: `forall` $(i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U}$.

General form of an inductive definition

$$
\begin{array}{l}
\texttt{Inductive I } (p_1 : \rho_1) \ldots (p_k : \rho_k) : \\
\quad \texttt{forall } (i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U} := \\
\mid c_1 : \texttt{forall } (x_{1,1} : \tau_{1,1}) \ldots (x_{1,l_1} : \tau_{1,l_1}),\ \texttt{I } p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m} \\
\ldots \\
\mid c_n : \texttt{forall } (x_{n,1} : \tau_{n,1}) \ldots (x_{n,l_n} : \tau_{n,l_n}),\ \texttt{I } p_1 \ldots p_k\ u_{n,1} \ldots u_{n,m}.
\end{array}
$$

- Parameters: $p_1, \ldots, p_k$.
- Indices: $i_1, \ldots, i_m$.
- Arity: `forall` $(i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U}$.
- Constructors: $c_1, \ldots, c_n$.

General form of an inductive definition

```
Inductive I (p₁ : ρ₁) ... (p_k : ρ_k) :
   forall (i₁ : A₁) ... (i_m : A_m), 𝒰 :=
| c₁ : forall (x₁,₁ : τ₁,₁) ... (x₁,l₁ : τ₁,l₁), I p₁...p_k u₁,₁...u₁,m
...
| c_n : forall (x_n,₁ : τ_n,₁) ... (x_n,l_n : τ_n,l_n), I p₁...p_k u_n,₁...u_n,m.
```

- Parameters: $p_1, \ldots, p_k$.
- Indices: $i_1, \ldots, i_m$.
- Arity: `forall` $(i_1 : A_1) \ldots (i_m : A_m), \mathcal{U}$.
- Constructors: $c_1, \ldots, c_n$.
- The declared type of the constructor c_j is
  `forall` $(x_{1,1} : \tau_{1,1}) \ldots (x_{1,1} : \tau_{1,l_1})$,
  $\mathtt{I}\ p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m}$

General form of an inductive definition

```
Inductive I (p₁ : ρ₁) ... (pₖ : ρₖ) :
   forall (i₁ : A₁) ... (iₘ : Aₘ), 𝒰 :=
| c₁ : forall (x₁,₁ : τ₁,₁) ... (x₁,ₗ₁ : τ₁,ₗ₁), I p₁...pₖ u₁,₁...u₁,ₘ
...
| cₙ : forall (xₙ,₁ : τₙ,₁) ... (xₙ,ₗₙ : τₙ,ₗₙ), I p₁...pₖ uₙ,₁...uₙ,ₘ.
```

- Parameters: $p_1, \ldots, p_k$.
- Indices: $i_1, \ldots, i_m$.
- Arity: `forall` $(i_1 : A_1) \ldots (i_m : A_m)$, $\mathcal{U}$.
- Constructors: $c_1, \ldots, c_n$.
- The declared type of the constructor c_j is
  `forall` $(x_{1,1} : \tau_{1,1}) \ldots (x_{1,1} : \tau_{1,l_1})$,
  I $p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m}$
- The type of constructor c_j is actually
  `forall` $(p_1 : \rho_1) \ldots (p_k : \rho_k)$,
  `forall` $(x_{1,1} : \tau_{1,1}) \ldots (x_{1,1} : \tau_{1,l_1})$,
  I $p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m}$
  i.e., the declared type of c_j with quantification over the parameters prepended.

The strict positivity restriction

```
Inductive I (p₁ : ρ₁) ... (p_k : ρ_k) :
   forall (i₁ : A₁) ... (i_m : A_m), 𝒰 :=
| c₁ : forall (x_{1,1} : τ_{1,1}) ... (x_{1,l₁} : τ_{1,l₁}), I p₁...p_k u_{1,1}...u_{1,m}
...
| c_n : forall (x_{n,1} : τ_{n,1}) ... (x_{n,l_n} : τ_{n,l_n}), I p₁...p_k u_{n,1}...u_{n,m}.
```

The strict positivity restriction

$\texttt{Inductive I } (p_1 \texttt{ : } \rho_1) \texttt{ ... } (p_k \texttt{ : } \rho_k) \texttt{ :}$
$\quad \texttt{forall } (i_1 \texttt{ : } A_1) \texttt{ ... } (i_m \texttt{ : } A_m) \texttt{, } \mathcal{U} \texttt{ :=}$
$\texttt{| } c_1 \texttt{ : forall } (x_{1,1} \texttt{ : } \tau_{1,1}) \texttt{ ... } (x_{1,l_1} \texttt{ : } \tau_{1,l_1}) \texttt{, I } p_1 \ldots p_k \; u_{1,1} \ldots u_{1,m}$
$\ldots$
$\texttt{| } c_n \texttt{ : forall } (x_{n,1} \texttt{ : } \tau_{n,1}) \texttt{ ... } (x_{n,l_n} \texttt{ : } \tau_{n,l_n}) \texttt{, I } p_1 \ldots p_k \; u_{n,1} \ldots u_{n,m} \texttt{.}$

The declared type of each constructor c_j
$\texttt{forall } (x_{j,1} \texttt{ : } \tau_{j,1}) \texttt{ ... } (x_{j,l_j} \texttt{ : } \tau_{j,l_j}) \texttt{, I } p_1 \ldots p_k \; u_{j,1} \ldots u_{j,m} \texttt{.}$
is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \ldots, p_k, u_{j,1}, \ldots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \ldots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:
  $\texttt{forall } (y_1 \texttt{ : } \sigma_1) \texttt{ ... } (y_r \texttt{ : } \sigma_r) \texttt{, I } p_1 \ldots p_k \; w_1 \ldots w_m \texttt{.}$
  where I does not occur in $\sigma_1, \ldots, \sigma_r$, $p_1, \ldots, p_k$, $w_1, \ldots, w_m$.

The strict positivity restriction

```
Inductive I (p₁ : ρ₁) ... (p_k : ρ_k) :
   forall (i₁ : A₁) ... (i_m : A_m), 𝒰 :=
| c₁ : forall (x_{1,1} : τ_{1,1}) ... (x_{1,l₁} : τ_{1,l₁}), I p₁...p_k u_{1,1}...u_{1,m}
...
| c_n : forall (x_{n,1} : τ_{n,1}) ... (x_{n,l_n} : τ_{n,l_n}), I p₁...p_k u_{n,1}...u_{n,m}.
```

The declared type of each constructor c_j

`forall` $(x_{j,1} : \tau_{j,1}) \dots (x_{j,l_j} : \tau_{j,l_j})$, `I` $p_1 \dots p_k\ u_{j,1} \dots u_{j,m}$.

is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \dots, p_k, u_{j,1}, \dots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \dots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:

  `forall` $(y_1 : \sigma_1) \dots (y_r : \sigma_r)$, `I` $p_1 \dots p_k\ w_1 \dots w_m$.

  where I does not occur in $\sigma_1, \dots, \sigma_r$, $p_1, \dots, p_k$, $w_1, \dots, w_m$.

Actually, in Coq the strict positivity condition is slightly more liberal than described above, with I allowed to occur, under certain restrictions, in the parameters of another inductive type in the target of a constructor type. See the Coq reference manual for details: `https://coq.inria.fr/distrib/current/refman/language/cic.html#inductive-definitions`.

The strict positivity restriction

The declared type of each constructor c_j

`forall` $(x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j})$, `I` $p_1 \ldots p_k\ u_{j,1} \ldots u_{j,m}$.

is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \ldots, p_k, u_{j,1}, \ldots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \ldots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:

  `forall` $(y_1 : \sigma_1) \ldots (y_r : \sigma_r)$, `I` $p_1 \ldots p_k\ w_1 \ldots w_m$.

  where I does not occur in $\sigma_1, \ldots, \sigma_r$, $p_1, \ldots, p_k$, $w_1, \ldots, w_m$.

This is valid:

```
Inductive I : Set := c : (nat -> I) -> I.
```

The strict positivity restriction

The declared type of each constructor c_j

`forall` $(x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j})$, `I` $p_1 \ldots p_k \; u_{j,1} \ldots u_{j,m}$.

is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \ldots, p_k, u_{j,1}, \ldots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \ldots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:

  `forall` $(y_1 : \sigma_1) \ldots (y_r : \sigma_r)$, `I` $p_1 \ldots p_k \; w_1 \ldots w_m$.

  where I does not occur in $\sigma_1, \ldots, \sigma_r$, $p_1, \ldots, p_k$, $w_1, \ldots, w_m$.

This is valid:

```
Inductive I : Set := c : (nat -> I) -> I.
```

This is not valid:

```
Inductive I : Set := c : (I -> I) -> I.
```

The strict positivity restriction

The declared type of each constructor c_j

`forall` $(x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j})$, `I` $p_1 \ldots p_k \; u_{j,1} \ldots u_{j,m}$.

is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \ldots, p_k, u_{j,1}, \ldots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \ldots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:

  `forall` $(y_1 : \sigma_1) \ldots (y_r : \sigma_r)$, `I` $p_1 \ldots p_k \; w_1 \ldots w_m$.

  where I does not occur in $\sigma_1, \ldots, \sigma_r$, $p_1, \ldots, p_k$, $w_1, \ldots, w_m$.

This is valid:

```
Inductive I : Set := c : (nat -> I) -> I.
```

This is not valid:

```
Inductive I : Set := c : (I -> I) -> I.
```

This is not valid either:

```
Inductive I : Set := c : ((I -> nat) -> I) -> I.
```

The strict positivity restriction

```
Inductive I (p₁ : ρ₁) ... (pₖ : ρₖ) :
   forall (i₁ : A₁) ... (iₘ : Aₘ), 𝒰 :=
| c₁ : forall (x₁,₁ : τ₁,₁) ... (x₁,ₗ₁ : τ₁,ₗ₁), I p₁...pₖ u₁,₁...u₁,ₘ
...
| cₙ : forall (xₙ,₁ : τₙ,₁) ... (xₙ,ₗₙ : τₙ,ₗₙ), I p₁...pₖ uₙ,₁...uₙ,ₘ.
```

The declared type of each constructor c_j

$\texttt{forall}\ (x_{j,1}\ :\ \tau_{j,1})\ \ldots\ (x_{j,l_j}\ :\ \tau_{j,l_j}),\ \texttt{I}\ p_1 \ldots p_k\ u_{j,1} \ldots u_{j,m}.$

is required to satisfy the positivity condition, meaning that:

- I does not occur in any of $p_1, \ldots, p_k, u_{j,1}, \ldots, u_{j,m}$; and
- I may occur only strictly positively in each $\tau_{j,i}$ for $i = 1, \ldots, l_j$, i.e., either I does not occur in $\tau_{j,i}$ at all or $\tau_{j,i}$ has the form:

  $\texttt{forall}\ (y_1\ :\ \sigma_1)\ \ldots\ (y_r\ :\ \sigma_r),\ \texttt{I}\ p_1 \ldots p_k\ w_1 \ldots w_m.$

  where I does not occur in $\sigma_1, \ldots, \sigma_r$, $p_1, \ldots, p_k$, $w_1, \ldots, w_m$.

Relaxing the strict positivity restriction may result in undecidability of type checking or even inconsistency!

Universe constraints

```
Inductive I (p₁ : ρ₁) ... (p_k : ρ_k) :
   forall (i₁ : A₁) ... (i_m : A_m), 𝒰 :=
| c₁ : forall (x_{1,1} : τ_{1,1}) ... (x_{1,l₁} : τ_{1,l₁}), I p₁ ... p_k u_{1,1} ... u_{1,m}
...
| c_n : forall (x_{n,1} : τ_{n,1}) ... (x_{n,l_n} : τ_{n,l_n}), I p₁ ... p_k u_{n,1} ... u_{n,m}.
```

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall (i_1 : A_1) \ldots (i_m : A_m).\mathcal{U} \vdash \\ (\forall (x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j}).I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}$$

Universe constraints

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\begin{aligned}&\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U} \vdash \\ &\qquad (\forall (x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j}).I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}\end{aligned}$$

Universe constraints

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall(p_1 : \rho_1)\ldots(p_k : \rho_k)(i_1 : A_1)\ldots(i_m : A_m).\mathcal{U} \vdash \\ (\forall(x_{j,1} : \tau_{j,1})\ldots(x_{j,l_j} : \tau_{j,l_j}).I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}$$

This is valid:

```
Inductive I (A : Set) : Set := cI : I A.
```

because $A : \texttt{Set}, I : \texttt{Set} \to \texttt{Set} \vdash I\,A : \texttt{Set}$.

Universe constraints

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U} \vdash \\ (\forall (x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j}).I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}$$

This is valid:

```
Inductive I (A : Set) : Set := cI : I A.
```

because $A : \texttt{Set}, I : \texttt{Set} \to \texttt{Set} \vdash I\,A : \texttt{Set}$.

This is not valid:

```
Inductive I : Set -> Set := cI : forall A : Set, I A.
```

because `forall A : Set, I A` is in $\texttt{Type}_1$ but not in `Set`.

Universe constraints

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U} \vdash \\ (\forall (x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j}).I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}$$

This is valid:

```
Inductive I : Prop := cI : forall A : Prop, A -> I.
```

because `Prop` is impredicative.

Universe constraints

The declared type of each constructor needs to be in $\mathcal{U}$:

$$\Gamma, p_1 : \rho_1, \ldots, p_k : \rho_k, I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U} \vdash \\ (\forall (x_{j,1} : \tau_{j,1}) \ldots (x_{j,l_j} : \tau_{j,l_j}). I p_1 \ldots p_k u_{j,1} \ldots u_{j,m}) : \mathcal{U}$$

This is valid:

```
Inductive I : Prop := cI : forall A : Prop, A -> I.
```

because `Prop` is impredicative.

This is valid:

```
Inductive I : Type := cI : forall A : Type, A -> I.
```

because the inferred implicit indices for `Type` are:

$\texttt{Inductive I : Type}_{i+1}\texttt{ := cI : forall A : Type}_{i}\texttt{, A -> I.}$

Large inductive types

A large inductive type is an inductive type such that the declared type of one of its constructors quantifies over a universe.

Large inductive types

A large inductive type is an inductive type such that the declared type of one of its constructors quantifies over a universe.

Examples:

```
Inductive I (A : Set) : Prop := cI : forall B : Set, B -> I A.

Inductive I (A : Set) : Type := cI : forall B : Set, B -> I A.

Inductive I (A : Set) : Prop := cI : forall B : Type, B -> I A.
```

Large inductive types

A large inductive type is an inductive type such that the declared type of one of its constructors quantifies over a universe.

Examples:

```
Inductive I (A : Set) : Prop := cI : forall B : Set, B -> I A.

Inductive I (A : Set) : Type := cI : forall B : Set, B -> I A.

Inductive I (A : Set) : Prop := cI : forall B : Type, B -> I A.
```

Large inductive types either are in `Prop` or they are in $\texttt{Type}_i$ with the universes quantified over in declared constructor types all in $\texttt{Type}_i$.

General form of `match` expressions

```
match t as x in I _ ... _ i_1 ... i_m return P i_1 ... i_m x with
| c_1 _ ... _ x_{1,1} ... x_{1,l_1} => b_1
...
| c_n _ ... _ x_{n,1} ... x_{n,l_n} => b_n
end
```

General form of match expressions

match t as x in I _ … _ i_1 … i_m return P i_1 … i_m x with
| c_1 _ … _ $x_{1,1}$ … x_{1,l_1} => b_1
…
| c_n _ … _ $x_{n,1}$ … x_{n,l_n} => b_n
end

where

Inductive I (p_1 : ρ_1) … (p_k : ρ_k) :
  forall (i_1 : A_1) … (i_m : A_m), $\mathcal{U}$:=
| c_1 : forall ($x_{1,1}$: $\tau_{1,1}$) … (x_{1,l_1} : τ_{1,l_1}), I $p_1 \dots p_k$ $u_{1,1} \dots u_{1,m}$
…
| c_n : forall ($x_{n,1}$: $\tau_{n,1}$) … (x_{n,l_n} : τ_{n,l_n}), I $p_1 \dots p_k$ $u_{n,1} \dots u_{n,m}$.

General form of `match` expressions

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,l₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,lₙ => bₙ
end
```

where

```
Inductive I (p₁ : ρ₁) ... (pₖ : ρₖ) :
   forall (i₁ : A₁) ... (iₘ : Aₘ), 𝒰 :=
| c₁ : forall (x₁,₁ : τ₁,₁) ... (x₁,l₁ : τ₁,l₁), I p₁...pₖ u₁,₁...u₁,ₘ
...
| cₙ : forall (xₙ,₁ : τₙ,₁) ... (xₙ,lₙ : τₙ,lₙ), I p₁...pₖ uₙ,₁...uₙ,ₘ.
```

- $t : I q_1 \dots q_k a_1 \dots a_m$.

General form of `match` expressions

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,l₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,lₙ => bₙ
end
```

$$\texttt{match t as x in I \_ ... \_ } i_1 \ldots i_m \texttt{ return P } i_1 \ldots i_m \texttt{ x with}$$
$$\texttt{| } c_1 \texttt{ \_ ... \_ } x_{1,1} \ldots x_{1,l_1} \texttt{ => } b_1$$
$$\ldots$$
$$\texttt{| } c_n \texttt{ \_ ... \_ } x_{n,1} \ldots x_{n,l_n} \texttt{ => } b_n$$
$$\texttt{end}$$

where

$$\texttt{Inductive I } (p_1 : \rho_1) \ldots (p_k : \rho_k) :$$
$$\quad \texttt{forall } (i_1 : A_1) \ldots (i_m : A_m),\ \mathcal{U} :=$$
$$\texttt{| } c_1 : \texttt{forall } (x_{1,1} : \tau_{1,1}) \ldots (x_{1,l_1} : \tau_{1,l_1}),\ \texttt{I}\ p_1 \ldots p_k\ u_{1,1} \ldots u_{1,m}$$
$$\ldots$$
$$\texttt{| } c_n : \texttt{forall } (x_{n,1} : \tau_{n,1}) \ldots (x_{n,l_n} : \tau_{n,l_n}),\ \texttt{I}\ p_1 \ldots p_k\ u_{n,1} \ldots u_{n,m}\,.$$

- $t : I q_1 \ldots q_k a_1 \ldots a_m$.
- $x, i_1, \ldots, i_m$ are fresh variables.

General form of `match` expressions

$\texttt{match t as x in I \_ ... \_}\ i_1\ \texttt{...}\ i_m\ \texttt{return P}\ i_1\ \texttt{...}\ i_m\ \texttt{x with}$
$\texttt{|}\ c_1\ \texttt{\_ ... \_}\ x_{1,1}\ \texttt{...}\ x_{1,l_1}\ \texttt{=>}\ b_1$
...
$\texttt{|}\ c_n\ \texttt{\_ ... \_}\ x_{n,1}\ \texttt{...}\ x_{n,l_n}\ \texttt{=>}\ b_n$
`end`

where

$\texttt{Inductive I}\ (p_1\ \texttt{:}\ \rho_1)\ \texttt{...}\ (p_k\ \texttt{:}\ \rho_k)\ \texttt{:}$
$\quad \texttt{forall}\ (i_1\ \texttt{:}\ A_1)\ \texttt{...}\ (i_m\ \texttt{:}\ A_m)\texttt{,}\ \mathcal{U}\ \texttt{:=}$
$\texttt{|}\ c_1\ \texttt{: forall}\ (x_{1,1}\ \texttt{:}\ \tau_{1,1})\ \texttt{...}\ (x_{1,l_1}\ \texttt{:}\ \tau_{1,l_1})\texttt{, I}\ p_1 \dots p_k\ u_{1,1} \dots u_{1,m}$
...
$\texttt{|}\ c_n\ \texttt{: forall}\ (x_{n,1}\ \texttt{:}\ \tau_{n,1})\ \texttt{...}\ (x_{n,l_n}\ \texttt{:}\ \tau_{n,l_n})\texttt{, I}\ p_1 \dots p_k\ u_{n,1} \dots u_{n,m}\texttt{.}$

- $t : I q_1 \dots q_k a_1 \dots a_m$.
- $x, i_1, \dots, i_m$ are fresh variables.
- The motive P has type $\forall (i_1 : A'_1) \dots (i_m : A'_m). I q_1 \dots q_k i_1 \dots i_m \to \mathcal{U}'$ where $A'_j = A_j[q_1/p_1] \dots [q_k/p_k]$.

General form of `match` expressions

```
match t as x in I _ ... _ i1 ... im return P i1 ... im x with
| c1 _ ... _ x1,1 ... x1,l1 => b1
...
| cn _ ... _ xn,1 ... xn,ln => bn
end
```

where

```
Inductive I (p1 : ρ1) ... (pk : ρk) :
   forall (i1 : A1) ... (im : Am), 𝒰 :=
| c1 : forall (x1,1 : τ1,1) ... (x1,l1 : τ1,l1), I p1...pk u1,1...u1,m
...
| cn : forall (xn,1 : τn,1) ... (xn,ln : τn,ln), I p1...pk un,1...un,m.
```

- $t : I q_1 \dots q_k a_1 \dots a_m$.
- $x, i_1, \dots, i_m$ are fresh variables.
- The motive P has type $\forall (i_1 : A'_1) \dots (i_m : A'_m). I q_1 \dots q_k i_1 \dots i_m \to \mathcal{U}'$ where $A'_j = A_j[q_1/p_1] \dots [q_k/p_k]$.
- Branch b_j is typed as

$$x_{j,1} : \tau'_{j,1}, \dots, x_{j,l_j} : \tau'_{j,l_j} \vdash b_j : P u'_{j,1} \dots u'_{j,m} (c_j q_1 \dots q_k x_{j,1} \dots x_{j,l_j})$$

where $\tau'_{j,r} = \tau_{j,r}[q_1/p_1] \dots [q_k/p_k]$ and $u'_{j,r} = u_{j,r}[q_1/p_1] \dots [q_k/p_k]$.

General form of `match` expressions

`match t as x in I _ ... _` i_1 `...` i_m `return P` i_1 `...` i_m `x with`
`| ` c_1 `_ ... _` $x_{1,1}$ `...` x_{1,l_1} `=>` b_1
`...`
`| ` c_n `_ ... _` $x_{n,1}$ `...` x_{n,l_n} `=>` b_n
`end`

where

`Inductive I (`p_1 `:` ρ_1`) ... (`p_k `:` ρ_k`) :`
`  forall (`i_1 `:` A_1`) ... (`i_m `:` A_m`),` $\mathcal{U}$ `:=`
`| ` c_1 `: forall (`$x_{1,1}$ `:` $\tau_{1,1}$`) ... (`x_{1,l_1} `:` τ_{1,l_1}`), I` $p_1 \ldots p_k \; u_{1,1} \ldots u_{1,m}$
`...`
`| ` c_n `: forall (`$x_{n,1}$ `:` $\tau_{n,1}$`) ... (`x_{n,l_n} `:` τ_{n,l_n}`), I` $p_1 \ldots p_k \; u_{n,1} \ldots u_{n,m}$`.`

- $t : I q_1 \ldots q_k a_1 \ldots a_m$.
- $x, i_1, \ldots, i_m$ are fresh variables.
- The <u>motive</u> P has type $\forall (i_1 : A'_1) \ldots (i_m : A'_m). I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$ where $A'_j = A_j[q_1/p_1] \ldots [q_k/p_k]$.
- Branch b_j is typed as

$$x_{j,1} : \tau'_{j,1}, \ldots, x_{j,l_j} : \tau'_{j,l_j} \vdash b_j : P u'_{j,1} \ldots u'_{j,m} (c_j q_1 \ldots q_k x_{j,1} \ldots x_{j,l_j})$$

  where $\tau'_{j,r} = \tau_{j,r}[q_1/p_1] \ldots [q_k/p_k]$ and $u'_{j,r} = u_{j,r}[q_1/p_1] \ldots [q_k/p_k]$.
- The type of the entire `match` expression is $P a_1 \ldots a_m t$.

Valid elimination universes

```
match t as x in I _ … _ i₁ … iₘ return P i₁ … iₘ x with
| c₁ _ … _ x₁,₁ … x₁,l₁ => b₁
…
| cₙ _ … _ xₙ,₁ … xₙ,lₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.

Valid elimination universes

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,ₗ₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,ₗₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m). I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.

Valid elimination universes

```
match t as x in I _ … _ i₁ … iₘ return P i₁ … iₘ x with
| c₁ _ … _ x₁,₁ … x₁,ₗ₁ => b₁
…
| cₙ _ … _ xₙ,₁ … xₙ,ₗₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}.$
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'.$
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.
- If $\mathcal{U} = \mathtt{Type}_j$ for some $j \geq 0$ then $\mathcal{U}'$ can be arbitrary.

Valid elimination universes

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,ₗ₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,ₗₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m). I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.
- If $\mathcal{U} = \texttt{Type}_j$ for some $j \geq 0$ then $\mathcal{U}'$ can be arbitrary.
- If $\mathcal{U} = \texttt{Prop}$ then $\mathcal{U}' = \texttt{Prop}$.

Valid elimination universes

```
match t as x in I _ ... _ i1 ... im return P i1 ... im x with
| c1 _ ... _ x1,1 ... x1,l1 => b1
...
| cn _ ... _ xn,1 ... xn,ln => bn
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.
- If $\mathcal{U} = \texttt{Type}_j$ for some $j \geq 0$ then $\mathcal{U}'$ can be arbitrary.
- If $\mathcal{U} = \texttt{Prop}$ then $\mathcal{U}' = \texttt{Prop}$.
  - Elimination from `Prop` to $\texttt{Type}_j$ for $j \geq 0$ is not allowed.

Valid elimination universes

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,ₗ₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,ₗₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.
- If $\mathcal{U} = \texttt{Type}_j$ for some $j \geq 0$ then $\mathcal{U}'$ can be arbitrary.
- If $\mathcal{U} = \texttt{Prop}$ then $\mathcal{U}' = \texttt{Prop}$.
  - Elimination from `Prop` to $\texttt{Type}_j$ for $j \geq 0$ is not allowed.
  - In other words, matching on proofs to produce programs/types (elements of a type in any of the "computational" universes) is not allowed.

Valid elimination universes

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c₁ _ ... _ x₁,₁ ... x₁,ₗ₁ => b₁
...
| cₙ _ ... _ xₙ,₁ ... xₙ,ₗₙ => bₙ
end
```

- $I : \forall (p_1 : \rho_1) \ldots (p_k : \rho_k)(i_1 : A_1) \ldots (i_m : A_m).\mathcal{U}$.
- $P : \forall (i_1 : A'_1) \ldots (i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- Elimination from $\mathcal{U}$ into $\mathcal{U}'$.
- If $\mathcal{U} = \texttt{Type}_j$ for some $j \geq 0$ then $\mathcal{U}'$ can be arbitrary.
- If $\mathcal{U} = \texttt{Prop}$ then $\mathcal{U}' = \texttt{Prop}$.
  - Elimination from `Prop` to $\texttt{Type}_j$ for $j \geq 0$ is not allowed.
  - In other words, matching on proofs to produce programs/types (elements of a type in any of the "computational" universes) is not allowed.
  - Except in one situation…

Small propositional inductive types

A small propositional inductive type is an inductive type I in `Prop` with at most one constructor with declared type

$$\forall(x_1 : \tau_1)\dots(x_l : \tau_l).I p_1 \dots p_k u_1 \dots u_m$$

such that all τ_j are in `Prop`.

Small propositional inductive types

A small propositional inductive type is an inductive type I in `Prop` with at most one constructor with declared type

$$\forall (x_1 : \tau_1) \ldots (x_l : \tau_l) . I p_1 \ldots p_k u_1 \ldots u_m$$

such that all τ_j are in `Prop`.

Examples:

```
Inductive False :=

Inductive and (A B : Prop) : Prop := conj : A -> B -> A /\ B.

Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : x = x.
```

Small propositional inductive types

A small propositional inductive type is an inductive type I in `Prop` with at most one constructor with declared type

$$\forall (x_1 : \tau_1) \dots (x_l : \tau_l). I p_1 \dots p_k u_1 \dots u_m$$

such that all τ_j are in `Prop`.

Small propositional inductive types

A small propositional inductive type is an inductive type I in `Prop` with at most one constructor with declared type

$$\forall(x_1 : \tau_1) \dots (x_l : \tau_l). I p_1 \dots p_k u_1 \dots u_m$$

such that all τ_j are in `Prop`.

```
match t as x in I _ ... _ i_1 ... i_m return P i_1 ... i_m x with
| c _ ... _ x_1 ... x_l => b
end
```

- $I : \forall(p_1 : \rho_1) \dots (p_k : \rho_k)(i_1 : A_1) \dots (i_m : A_m).\mathcal{U}.$
- $P : \forall(i_1 : A'_1) \dots (i_m : A'_m). I q_1 \dots q_k i_1 \dots i_m \to \mathcal{U}'.$

Small propositional inductive types

A small propositional inductive type is an inductive type I in `Prop` with at most one constructor with declared type

$$\forall(x_1 : \tau_1)\ldots(x_l : \tau_l).I p_1 \ldots p_k u_1 \ldots u_m$$

such that all τ_j are in `Prop`.

```
match t as x in I _ ... _ i₁ ... iₘ return P i₁ ... iₘ x with
| c _ ... _ x₁ ... xₗ => b
end
```

- $I : \forall(p_1 : \rho_1)\ldots(p_k : \rho_k)(i_1 : A_1)\ldots(i_m : A_m).\mathcal{U}$.
- $P : \forall(i_1 : A'_1)\ldots(i_m : A'_m).I q_1 \ldots q_k i_1 \ldots i_m \to \mathcal{U}'$.
- If $\mathcal{U} =$ `Prop` and I is a small propositional inductive type then $\mathcal{U}'$ can be arbitrary.

Large elimination

Large elimination (or strong elimination) is elimination from $\mathcal{U}$ to $\mathcal{U}'$ when $\mathcal{U} : \mathcal{U}'$.

Large elimination

Large elimination (or strong elimination) is elimination from $\mathcal{U}$ to $\mathcal{U}'$ when $\mathcal{U} : \mathcal{U}'$.

Examples:

```
match t in bool return Set with
| true => nat
| false => bool
end

match t in bool return Prop with
| true => True
| false => False
end
```

Note: `bool` : `Set`, `Set` : $\texttt{Type}_1$ and `Prop` : $\texttt{Type}_1$, so in both examples $\mathcal{U} =$ `Set` and $\mathcal{U}' = \texttt{Type}_1$.

Large elimination

Large elimination (or strong elimination) is elimination from $\mathcal{U}$ to $\mathcal{U}'$ when $\mathcal{U} : \mathcal{U}'$.

- Unrestricted large elimination of large inductive types from an impredicative universe leads to inconsistency.

  Jacobs, "The inconsistency of higher-order extensions of Martin-Löf's type theory", Journal of Philosophical Logic, 1989

Large elimination

Large elimination (or strong elimination) is elimination from $\mathcal{U}$ to $\mathcal{U}'$ when $\mathcal{U} : \mathcal{U}'$.

- Unrestricted large elimination of large inductive types from an impredicative universe leads to inconsistency.

  Jacobs, "The inconsistency of higher-order extensions of Martin-Löf's type theory", Journal of Philosophical Logic, 1989
- In Coq, large elimination from `Prop` is allowed only for small propositional inductive types.

Large elimination

Large elimination (or strong elimination) is elimination from $\mathcal{U}$ to $\mathcal{U}'$ when $\mathcal{U} : \mathcal{U}'$.

- Unrestricted large elimination of large inductive types from an impredicative universe leads to inconsistency.

  Jacobs, "The inconsistency of higher-order extensions of Martin-Löf's type theory", Journal of Philosophical Logic, 1989
- In Coq, large elimination from `Prop` is allowed only for small propositional inductive types.
- In Coq, without large elimination from `Set` it is not possible to prove the distinctness of constructors, e.g., that $0 \neq 1$.