Inductive Families (Part 2)

Niels van Duijl
Erik Oosting

Peter Dybjer, Formal Aspects of Computing 6(4), 1994, Sections 4-6

Contents

- **Section 3 continued**
- **Section 6: Simultaneous Induction**
- **Section 4: Recursive Definitions**
- **Section 5: Lots of examples!**

Elimination rule

- Recursion principle!
- Major premise ***c***
- One minor premise ***e*** for each constructor, corresponding to each induction step

```
forall P : nat -> Set,
P 0 ->
(forall n : nat, P n -> P (S n)) ->
forall n : nat, P n
```

$$
\begin{aligned}
\mathit{elim} :\; & (A :: \sigma) \\
& (C : \; (a :: \alpha[A]) \\
& \qquad\; (c : P_A(a)) \\
& \qquad\; \mathit{set}) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha[A]) \\
& (c : P_A(a)) \\
& C(a, c).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{nrec} :\; & (C : (c : N)\mathit{set}) \\
& (e_1 : C(0)) \\
& (e_2 : \; (u : N) \\
& \qquad\; (v : C(u)) \\
& \qquad\; C(s(u))) \\
& (c : N) \\
& C(c).
\end{aligned}
$$

Section 3 continued

Equality rule

- One equality rule for each constructor
- First parentheses should be read as ∀

$$
\begin{array}{ll}
 & (A, C, e, b, u)elim_{A,C}(e, p[A, b], intro_A(b, u)) \\
= & (A, C, e, b, u)e_j(b, u, v) \\
: & (A :: \sigma) \\
 & (C : \ (a :: \alpha[A]) \\
 & \qquad (c : P_A(a)) \\
 & \qquad set) \\
 & (e :: \epsilon[A]) \\
 & (b :: \beta[A]) \\
 & (u :: \gamma[A, b]) \\
 & C(p[A, b], intro_A(b, u)),
\end{array}
$$

where v_i is

$$(x)elim_{A,C}(e, p_i[A, b, x], u_i(x)).$$

Lists of a certain length. The equality rule for the constructor nil' is

$$
\begin{array}{ll}
 & (A, C, e_1, e_2)listrec'_{A,C}(e_1, e_2, 0, nil'_A) \\
= & (A, C, e_1, e_2)e_1 \\
: & (A : set) \\
 & (C : (a : N)(c : List'_A(a))set) \\
 & (e_1 : C(0, nil'_A)) \\
 & (e_2 : \ (b_1 : N) \\
 & \qquad (b_2 : A) \\
 & \qquad (u : List'_A(b_1)) \\
 & \qquad (v : C(b_1, u)) \\
 & \qquad C(s(b_1), cons'_A(b_1, b_2, u))) \\
 & C(0, nil'_A).
\end{array}
$$

The equality rule for the constructor $cons$ is

$$
\begin{array}{ll}
 & (A, C, e_1, e_2, b_1, b_2, u)listrec'_{A,C}(e_1, e_2, cons'_A(b_1, b_2, u)) \\
= & (A, C, e_1, e_2, b_1, b_2, u)e_2(b_1, b_2, u, listrec'_{A,C}(e_1, e_2, u)) \\
: & (A : set) \\
 & (C : (a : N)(c : List'_A(a))set) \\
 & (e_1 : C(0, nil'_A)) \\
 & (e_2 : \ (b_1 : N) \\
 & \qquad (b_2 : A) \\
 & \qquad (u : List'_A(b_1)) \\
 & \qquad (v : C(b_1, u)) \\
 & \qquad C(s(b_1), cons'_A(b_1, b_2, u))) \\
 & (b_1 : N) \\
 & (b_2 : A) \\
 & (u : List'_A(b_1)) \\
 & C(s(b_1), cons'_A(b_1, b_2, u))
\end{array}
$$

Example of a function: forgetlength

$$\begin{aligned} & \mathit{forgetlength} \\ = \quad & (A)\mathit{listrec}'_{A,(a,c)\mathit{List}_A}(\mathit{nil}_A, (b_1, b_2, u, v)\mathit{cons}_A(b_2, v)) \\ : \quad & (A : \mathit{set})(a : N)(c : \mathit{List}'_A(n))\mathit{List}_A. \end{aligned}$$

$$\begin{aligned} \mathit{forgetlength}_A(0, \mathit{nil}'_A) &= \mathit{nil}_A, \\ \mathit{forgetlength}_A(s(b_1), \mathit{cons}'_A(b_1, b_2, u)) &= \mathit{cons}_A(b_2, \mathit{forgetlength}_A(b_1, u)). \end{aligned}$$

Section 3 continued

Overview of rules

- **A** :: **σ** means
  $A_1,\dots,A_n$: $\sigma_1,\dots,\sigma_n$

$$
\begin{aligned}
P :\quad & (A :: \sigma)\\
& (a :: \alpha[A])\\
& set,
\end{aligned}
$$

$$
\begin{aligned}
intro :\quad & (A :: \sigma)\\
& (b :: \beta[A])\\
& (u :: \gamma[A, b])\\
& P_A(p[A, b]),
\end{aligned}
$$

$$
\begin{aligned}
elim :\quad & (A :: \sigma)\\
& (C : \quad (a :: \alpha[A])\\
& \qquad\quad (c : P_A(a))\\
& \qquad\quad set)\\
& (e :: \epsilon[A])\\
& (a :: \alpha[A])\\
& (c : P_A(a))\\
& C(a, c).
\end{aligned}
$$

$$
\begin{aligned}
& (A, C, e, b, u)elim_{A,C}(e, p[A, b], intro_A(b, u))\\
= \ & (A, C, e, b, u)e_j(b, u, v)\\
: \ & (A :: \sigma)\\
& (C : \quad (a :: \alpha[A])\\
& \qquad\quad (c : P_A(a))\\
& \qquad\quad set)\\
& (e :: \epsilon[A])\\
& (b :: \beta[A])\\
& (u :: \gamma[A, b])\\
& C(p[A, b], intro_A(b, u)),
\end{aligned}
$$

Section 6: Simultaneous Induction

- **Very similar to the rules from Section 3**
    - But now they can depend on each other

Rules (in comparison to normal induction)

- Rules are indexed by a variable ***K***
- For elimination, we need to consider *all* formation rules that we defined our types with in ***C***.

$$\phi_k[A] \text{ is } \; (a :: \alpha_k[A]) \; (P_{k\,A}(a)) \; set.$$

$$
\begin{aligned}
P :\ & (A :: \sigma) \\
& (a :: \alpha[A]) \\
& set, \\
intro :\ & (A :: \sigma) \\
& (b :: \beta[A]) \\
& (u :: \gamma[A, b]) \\
& P_A(p[A, b]), \\
elim :\ & (A :: \sigma) \\
& (C :\ (a :: \alpha[A]) \\
& \qquad (c : P_A(a)) \\
& \qquad set) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha[A]) \\
& (c : P_A(a)) \\
& C(a, c).
\end{aligned}
$$

$$
\begin{aligned}
P_k :\ & (A :: \sigma) \\
& (a :: \alpha_k[A]) \\
& set, \\
intro :\ & (A :: \sigma) \\
& (b :: \beta[A]) \\
& (u :: \gamma[A, b]) \\
& P_{k\,A}(p[A, b]), \\
elim_l :\ & (A :: \sigma) \\
& (C :: \phi[A]) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha_l[A]) \\
& (c : P_{l\,A}(a)) \\
& C(a, c).
\end{aligned}
$$

Example: Even and odd numbers

$$Even : (a : N)set,$$
$$Odd : (a : N)set.$$

$$intro_1 : Even(0),$$

$$intro_2 : (b : N)(u : Even(b))Odd(s(b)),$$

$$intro_3 : (b : N)(u : Odd(b))Even(s(b)).$$

$$P_k : (A :: \sigma)(a :: \alpha_k[A])set,$$

$$intro : (A :: \sigma)(b :: \beta[A])(u :: \gamma[A, b])P_{kA}(p[A, b]),$$

Elimination for even and odd numbers

$$
\begin{aligned}
\mathit{evenelim} :\ & (C_1 : (a : N)(\mathit{Even}(a))\mathit{set}) \\
& (C_2 : (a : N)(\mathit{Odd}(a))\mathit{set}) \\
& (e_1 : \quad C_1(0, \mathit{intro}_1)) \\
& (e_2 : \quad (b : N) \\
& \qquad\quad (u : \mathit{Even}(b)) \\
& \qquad\quad (v : C_1(b, u)) \\
& \qquad\quad C_2(s(b), \mathit{intro}_2(b, u))) \\
& (e_3 : \quad (b : N) \\
& \qquad\quad (u : \mathit{Odd}(b)) \\
& \qquad\quad (v : C_2(b, u)) \\
& \qquad\quad C_1(s(b), \mathit{intro}_3(b, u))) \\
& (a : N) \\
& (c : \mathit{Even}(a)) \\
& C_1(a, c),
\end{aligned}
$$

$$
\begin{aligned}
\mathit{oddelim} :\ & (C_1 : (a : N)(\mathit{Even}(a))\mathit{set}) \\
& (C_2 : (a : N)(\mathit{Odd}(a))\mathit{set}) \\
& (e_1 : \quad C_1(0, \mathit{intro}_1)) \\
& (e_2 : \quad (b : N) \\
& \qquad\quad (u : \mathit{Even}(b)) \\
& \qquad\quad (v : C_1(b, u)) \\
& \qquad\quad C_2(s(b), \mathit{intro}_2(b, u))) \\
& (e_3 : \quad (b : N) \\
& \qquad\quad (u : \mathit{Odd}(b)) \\
& \qquad\quad (v : C_2(b, u)) \\
& \qquad\quad C_1(s(b), \mathit{intro}_3(b, u))) \\
& (a : N) \\
& (c : \mathit{Odd}(a)) \\
& C_2(a, c).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{elim}_l :\ & (A :: \sigma) \\
& (C :: \phi[A]) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha_l[A]) \\
& (c : P_{l\,A}(a)) \\
& C(a, c).
\end{aligned}
$$

Section 4: Recursive Definitions

Introduction

- Using elim, we can only eliminate to types in set
- Problem: we don’t have type:type
- Solution: introduce a new scheme
- Induction vs. Recursion

$$
\begin{aligned}
elim :\quad & (A :: \sigma) \\
& (C : \quad (a :: \alpha[A]) \\
& \qquad\qquad (c : P_A(a)) \\
& \qquad\quad type\ ) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha[A]) \\
& (c : P_A(a)) \\
& C(a, c).
\end{aligned}
$$

A new elimination scheme

- ***B*** are the parameters of ***f***
- ***a*** and ***Q[B]*** are used for our set former ***P***
- (***Q[B]*** is a sequence of constants)
- ***c*** are the major premises
- ψ is a type under the previous assumptions

$$
\begin{aligned}
elim :\quad & (A :: \sigma) \\
& (C : \quad (a :: \alpha[A]) \\
& \qquad\quad (c : P_A(a)) \\
& \qquad\quad set) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha[A]) \\
& (c : P_A(a)) \\
& C(a, c).
\end{aligned}
$$

$$
\begin{aligned}
f :\quad & (B :: \tau) \\
& (a :: \alpha[Q[B]]) \\
& (c :: P_{Q[B]}(a)) \\
& \psi[B, a, c],
\end{aligned}
$$

Example: forgetlength

$\tau = set$, $Q[B] = B : \tau$, and $\psi[B, a, c] = List_B$.

$$f : (B :: \tau)\ (a :: \alpha[Q[B]])\ (c :: P_{Q[B]}(a))\ \psi[B, a, c],$$

$$forgetlength : (B : set),\ (a : N),\ (c : List'_B(a)),\ List_B$$

Equality

- **A** becomes **Q[B]**
- **elim$_{A,C}$** becomes **f$_B$**
- **e** and **C** disappeared, as they are not used in **f**

- New eq rule:

$$
\begin{aligned}
& (B, \quad b, u) f_B(p[Q[B], b], intro_{Q[B]}(b, u)) \\
= \; & (B, \quad b, u) e_j(b, u, v) \\
: \; & (B :: \tau) \\
& (b :: \beta[Q[B]]) \\
& (u :: \gamma[Q[B], b]) \\
& \psi[B, p[Q[B], b], intro_{Q[B]}(b, u)],
\end{aligned}
$$

where v_i is

$$(x) f_B(p_i[A, b, x], u_i(x)),$$

- Eq rule from Section 3:

$$
\begin{aligned}
& (A, C, e, b, u) elim_{A,C}(e, p[A, b], intro_A(b, u)) \\
= \; & (A, C, e, b, u) e_j(b, u, v) \\
: \; & (A :: \sigma) \\
& (C : \quad (a :: \alpha[A]) \\
& \qquad\quad (c : P_A(a)) \\
& \qquad\quad set) \\
& (e :: \epsilon[A]) \\
& (b :: \beta[A]) \\
& (u :: \gamma[A, b]) \\
& C(p[A, b], intro_A(b, u)),
\end{aligned}
$$

where v_i is

$$(x) elim_{A,C}(e, p_i[A, b, x], u_i(x)).$$

Section 5

- Predicate Logic
  - Implication
  - Equality
- Generalised Induction
  - Well-Orderings (W-types)
  - Well-Founded part of a Relation
- Finite Sets and *n*-Tuples
- Untyped λ-Calculus

Section 5

- Predicate Logic
  - Implication
  - **Equality**
- Generalised Induction
  - **Well-Orderings (W-types)**
  - **Well-Founded part of a Relation**
- **Finite Sets and *n*-Tuples**
- **Untyped λ-Calculus**

Two types of equality

- Martin-Löf and Paulin
- Difference in parameters and indices
- Martin-Löf:
  - *A* parameter
  - a_1 , a_2 : *A* indices
- Paulin:
  - *A* and a_1 : *A* parameters
  - a_2 : *A* index

- In coq:

```
Inductive eq (A : Set) : A -> A -> Prop :=
  eq_refl : forall a : A, eq A a a.
```

```
Inductive eq (A : Set) (a : A) : A -> Prop :=
  eq_refl : eq A a a.
```

Formation Rule

- Equality à la Martin-Löf

$$I : (A : set)\ (a_1 : A)\ (a_2 : A)\ set.$$

- Equality à la Paulin

$$I' : (A : set)\ (a_1 : A)\ (a_2 : A)\ set.$$

- General Scheme:

$$P : (A :: \sigma)\ (a :: \alpha[A])\ set,$$

- Coq:

```
Inductive eq (A : Set) : A -> A -> Prop :=
Inductive eq (A : Set) (a : A) : A -> Prop :=
```

- The difference is in what *A* and *a* are, but this is invisible!

Equality using the inductive scheme

Introduction rule

- Equality à la Martin-Löf

$$r: \quad (A : set) \quad (b : A) \quad I_A(b, b).$$

- Equality à la Paulin

$$r': \quad (A : set) \quad (a_1 : A) \quad I'_{A,a_1}(a_1).$$

- General Scheme:

$$intro: \quad (A :: \sigma) \quad (b :: \beta[A]) \quad (u :: \gamma[A, b]) \quad P_A(p[A, b]),$$

- Coq:

```
eq_refl : forall a : A, eq A a a.
eq_refl : eq A a a.
```

Equality using the inductive scheme

Elimination rule

- Equality à la Martin-Löf

$$J : (A : set)\ (C : (a_1 : A)(a_2 : A)set)\ (e : (b : A)C(b,b))\ (a_1 : A)\ (a_2 : A)\ (c : I_A(a_1,a_2))\ C(a_1,a_2).$$

- Equality à la Paulin

$$J' : (A : set)\ (a_1 : A)\ (C : (a_2 : A)set)\ (e : C(a_1))\ (a_2 : A)\ (c : I'_{A,a_1}(a_2))\ C(a_2).$$

- General Scheme:

$$elim : (A :: \sigma)\ (C : (a :: \alpha[A])\ (c : P_A(a))\ set)\ (e :: \epsilon[A])\ (a :: \alpha[A])\ (c : P_A(a))\ C(a,c).$$

- Coq:

```
forall (A : Set) (P : A -> A -> Prop),
(forall a : A, P a a) -> forall a y : A, eq A a y -> P a y
```

```
forall (A : Set) (a : A) (P : A -> Prop),
P a -> forall a0 : A, eq A a a0 -> P a0
```

Coq

Formation rule.

$$W : \; (A_1 : set)\; (A_2 : (A_1)set)\; set.$$

Introduction rule.

$$sup : \; (A_1 : set)\; (A_2 : (A_1)set)\; (b : A_1)\; (u : (x : A_2(b))W_{A_1,A_2})\; W_{A_1,A_2}.$$

Elimination rule.

$$T : \; (A_1 : set)\; (A_2 : (A_1)set)\; (C : (W_{A_1,A_2})set)\; (e : \; (b : A_1)\; (u : (x : A_2(b))W_{A_1,A_2})\; (v : (x : A_2(b))C(u(x)))\; C(sup(b,u)))\; (c : W_{A_1,A_2})\; C(c).$$

```
Inductive W (A : Set) (B : A -> Set) : Set :=
    sup : forall x : A, (B x -> W A B) -> W A B.
```

```
forall (A : Set) (B : A -> Set) (P : W A B -> Prop),
(forall (x : A) (w : B x -> W A B),
 (forall b : B x, P (w b)) -> P (sup A B x w)) ->
forall w : W A B, P w
```

Rules

Formation rule.

$$W : (A_1 : set)\ (A_2 : (A_1)set)\ set.$$

Introduction rule.

$$sup : (A_1 : set)\ (A_2 : (A_1)set)\ (b : A_1)\ (u : (x : A_2(b))W_{A_1,A_2})\ W_{A_1,A_2}.$$

Elimination rule.

$$T : (A_1 : set)\ (A_2 : (A_1)set)\ (C : (W_{A_1,A_2})set)\ (e : (b : A_1)\ (u : (x : A_2(b))W_{A_1,A_2})\ (v : (x : A_2(b))C(u(x)))\ C(sup(b,u)))\ (c : W_{A_1,A_2})\ C(c).$$

$$P : (A :: \sigma)\ (a :: \alpha[A])\ set,$$

$$intro : (A :: \sigma)\ (b :: \beta[A])\ (u :: \gamma[A,b])\ P_A(p[A,b]),$$

$$elim : (A :: \sigma)\ (C : (a :: \alpha[A])\ (c : P_A(a))\ set)\ (e :: \epsilon[A])\ (a :: \alpha[A])\ (c : P_A(a))\ C(a,c).$$

Well-Founded part of a Relation

What is it?

- Accessibility
- Given a binary relation **R** over an arbitrary type **A**, the accessible elements of **R** are those **a:A** from which there is no infinite chain $\mathbf{a\ R\ a_1\ R\ a_2\ R\ a_3}$, etc.
- If $\mathbf{A_1}$ is a set and $\mathbf{A_2}$ is a binary relation on that set, then $\mathbf{Acc_{A1,A2}(a)}$ is true iff **a** is in the well-founded part of $\mathbf{A_2}$.
- The entire relation is well-founded if every element of **A** is accessible.

Well-Founded part of a Relation

Coq

Formation rule.

$$
\begin{aligned}
Acc :\ & (A_1 : set) \\
& (A_2 : (A_1)(A_1)set) \\
& (a : A_1) \\
& set.
\end{aligned}
$$

Introduction rule.

$$
\begin{aligned}
acc :\ & (A_1 : set) \\
& (A_2 : (A_1)(A_1)set) \\
& (b : A_1) \\
& (u : (x_1 : A_1)(x_2 : A_2(x_1, b))Acc_{A_1,A_2}(x_1)) \\
& Acc_{A_1,A_2}(b).
\end{aligned}
$$

Elimination rule.

$$
\begin{aligned}
accrec :\ & (A_1 : \quad set) \\
& (A_2 : \quad (A_1)(A_1)set) \\
& (C : \quad (a : A_1) \\
& \qquad (c : Acc_{A_1,A_2}(a)) \\
& \qquad set) \\
& (e : \quad (b : A_1) \\
& \qquad (u : (x_1 : A_1)(x_2 : A_2(x_1, b))Acc_{A_1,A_2}(x_1)) \\
& \qquad (v : (x_1 : A_1)(x_2 : A_2(x_1, b))C(x_1, u(x_1, x_2))) \\
& \qquad C(b, acc_{A_1,A_2}(b, u))) \\
& (a : \quad A_1) \\
& (c : \quad Acc_{A_1,A_2}(a)) \\
& C(a, c) \quad .
\end{aligned}
$$

```
Inductive Acc {A : Type} (R : A -> A -> Prop) (x : A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x.
```

```
forall (A : Type) (R : A -> A -> Prop)
  (P : forall x : A, Acc R x -> Prop),
(forall (x : A)
   (a : forall y : A, R y x -> Acc R y),
 (forall (y : A) (r : R y x), P y (a y r)) ->
 P x (Acc_intro x a)) ->
forall (x : A) (a : Acc R x), P x a
```

Well-Founded part of a Relation

Elimination rule

- For Well-Founded relations, we have:

$$(\forall x \in X)\,[(\forall y \in X)\,[y\,R\,x \implies P(y)] \implies P(x)] \quad \text{implies} \quad (\forall x \in X)\,P(x).$$

- Equivalent to Acc_ind in Coq:

```
forall (A : Type) (R : A -> A -> Prop)
  (P : A -> Prop),
(forall x : A,
 (forall y : A, R y x -> Acc R y) ->
 (forall y : A, R y x -> P y) -> P x) ->
forall x : A, Acc R x -> P x
```

Acc_inv_dep in Coq:

```
forall (A : Type) (R : A -> A -> Prop)
  (P : forall x : A, Acc R x -> Prop),
(forall (x : A)
   (a : forall y : A, R y x -> Acc R y),
 (forall (y : A) (r : R y x), P y (a y r)) ->
 P x (Acc_intro x a)) ->
forall (x : A) (a : Acc R x), P x a
```

Well-Founded part of a Relation

Rules

Formation rule.

$$
\begin{aligned}
Acc : \; & (A_1 : set) \\
& (A_2 : (A_1)(A_1)set) \\
& (a : A_1) \\
& set.
\end{aligned}
$$

Introduction rule.

$$
\begin{aligned}
acc : \; & (A_1 : set) \\
& (A_2 : (A_1)(A_1)set) \\
& (b : A_1) \\
& (u : (x_1 : A_1)(x_2 : A_2(x_1, b))Acc_{A_1,A_2}(x_1)) \\
& Acc_{A_1,A_2}(b).
\end{aligned}
$$

Elimination rule.

$$
\begin{aligned}
accrec : \; & (A_1 : && set) \\
& (A_2 : && (A_1)(A_1)set) \\
& (C : && (a : A_1) \\
& && (c : Acc_{A_1,A_2}(a)) \\
& && set) \\
& (e : && (b : A_1) \\
& && (u : (x_1 : A_1)(x_2 : A_2(x_1, b))Acc_{A_1,A_2}(x_1)) \\
& && (v : (x_1 : A_1)(x_2 : A_2(x_1, b))C(x_1, u(x_1, x_2))) \\
& && C(b, acc_{A_1,A_2}(b, u))) \\
& (a : && A_1) \\
& (c : && Acc_{A_1,A_2}(a)) \\
& C(a, c) && .
\end{aligned}
$$

$$
\begin{aligned}
P : \; & (A :: \sigma) \\
& (a :: \alpha[A]) \\
& set, \\
intro : \; & (A :: \sigma) \\
& (b :: \beta[A]) \\
& (u :: \gamma[A, b]) \\
& P_A(p[A, b]), \\
elim : \; & (A :: \sigma) \\
& (C : \; (a :: \alpha[A]) \\
& \qquad (c : P_A(a)) \\
& \qquad set) \\
& (e :: \epsilon[A]) \\
& (a :: \alpha[A]) \\
& (c : P_A(a)) \\
& C(a, c).
\end{aligned}
$$

A practical example

Finite Sets and ~~SnocVecs~~ *n*-Tuples

- Finite Sets of size N force you to specify a number between 0 and (N - 1)
- We can then define an *n*-Tuple, and use the finite sets to access elements of the tuple

N'-formation:

$$N' : (N)set$$

N'-introduction:

$$0' \; : \; (n : N)N'(s(n)),$$
$$s' \; : \; (n : N)(N'(n))N'(s(n)).$$

$$Tuple : (A : set)(n : N)set,$$

$$A^0 = \top,$$
$$A^{s(n)} = A^n \times A.$$

```
Inductive Fin : nat -> Set :=
| F1 : forall {n: nat}, Fin (S n)
| FS : forall {n: nat}, Fin n -> Fin (S n).

Inductive SnocVec (A: Set) : nat -> Set :=
| SNil : SnocVec A 0
| SCons : forall (n: nat), (SnocVec A n) -> A -> SnocVec A (S n).
```

A practical example

Using Fins and *n*-Tuples

- A mapping function to map over all elements
- A projection function to access elements in an *n*-Tuple

$$map : (A, B : set)(f : (A)B)(n : N)(A^n)B^n$$

$$\begin{aligned} f^0(as) &= \langle\rangle, \\ f^{s(n)}(as) &= \langle f^n(fst(as)), f(snd(as))\rangle. \end{aligned}$$

```
Fixpoint Snoc_map {A B : Set} {n: nat} (f : A -> B)
  (inp: SnocVec A n) {struct inp} : SnocVec B n :=
  match inp with
  | SNil _ => SNil B
  | SCons _ m rest l => SCons B m (Snoc_map f rest) (f l)
  end.
```

$$\pi : (A : set)(n : N)(i : N_n)(A^n)A$$

$$\begin{aligned} \pi^{0_n}_{s(n)}(as) &= snd(as), \\ \pi^{s_n(i)}_{s(n)}(as) &= \pi^{i}_{n}(fst(as)). \end{aligned}$$

```
Fixpoint proj {n : nat} {A : Set} (inp : SnocVec A n) (idx : Fin n) : A.
  induction idx.
  inversion_clear inp.
  exact H0.
  apply IHidx.
  inversion_clear inp.
  exact H.
Defined.
```

A practical example

Some generators for n-Tuples

- *id(n)* gives us all the elements of *Fin n* in an n-tuple
- *up(n)* is similar but for the successors of all elements of *Fin n* (in *Fin (S n)*)

$$id : (n : N)N^n_n$$

$$\begin{aligned} id_0 &= \langle\rangle, \\ id_{s(n)} &= \langle s^n_n(id_n), 0_n\rangle. \end{aligned}$$

```
Fixpoint idctx (n : nat) {struct n} : SnocVec (Fin n) n.
  induction n.
  exact (SNil (Fin 0)).
  apply SCons.
  apply (Snoc_map FS).
  exact IHn.
  exact F1.
Defined.
```

$$\uparrow : (n : N)N^n_{s(n)}$$

$$\begin{aligned} \uparrow_0 &= \langle\rangle, \\ \uparrow_{s(n)} &= \langle s^n_{s(n)}(\uparrow_n), s_{s(n)}(0_n)\rangle. \end{aligned}$$

```
(* basically idctx + 1 *)
Definition up := fun (n : nat) => Snoc_map FS (idctx n).
```

A practical example

DeBruijn-based untyped lambda calculus

- No names, so α-equivalence is trivial
- Has the number of free variables as a parameter
- Λ(0) is a fully bound term

Λ-formation:

$$\Lambda : (N)set.$$

Λ-introduction:

$$\begin{aligned} var &: (n : N)(i : N_n)\Lambda_n, \\ \lambda &: (n : N)(\Lambda_{s(n)})\Lambda_n, \\ ap &: (n : N)(\Lambda_n)(\Lambda_n)\Lambda_n. \end{aligned}$$

```
Inductive Lambda (free: nat) : Set :=
| Var : Fin free -> Lambda free
| Abs : Lambda (S free) -> Lambda free
| App : Lambda free -> Lambda free -> Lambda free.
```

A practical example

Substitution

- Basically just replace variables by their definitions
- When we enter a lambda term, our

$$sub : (n : N)(g : \Lambda_n)(m : N)(fs : \Lambda^n_m)\Lambda_m,$$

$$\begin{aligned}
sub_n(var_n(i), m, fs) &= \pi^i_n(fs), \\
sub_n(\lambda_n(g), m, fs) &= \lambda_m(sub_{s(n)}(g, s(m), \langle lift^n_m(fs), var_{s(m)}(0_m)\rangle)), \\
sub_n(ap_n(h, f), m, fs) &= ap_m(sub_n(h, m, fs), sub_n(f, c, fs)),
\end{aligned}$$

```
Fixpoint sub {n : nat} (g: Lambda n) (m: nat) (fs : SnocVec (Lambda m) n) {struct g}: Lambda m :=
  match g with
  | Var _ i => proj fs i
  | Abs _ body => Abs m (sub body (S m) (SCons (Lambda (S m)) n (Snoc_map lift fs) (Var (S m) F1)))
  | App _ e1 e2 => App m (sub e1 m fs) (sub e1 m fs)
  end.
```

A practical example

Some helper functions

- Lift renames all *Var i* by *Var (i+1)* in a lambda expression for all free variables in that expression
- *rename* is just a helper function that helps lift apply numbers to lambda expressions

$$lift : (n : N)(\Lambda_n)\Lambda_{s(n)}$$

$$lift_n(f) = rename(n, f, \uparrow_n),$$

$$rename : (n : N)(g : \Lambda_n)(m : N)(is : N^n_m)\Lambda_m,$$

$$\begin{aligned} rename_n(var_n(i), m, is) &= var(\pi^i_n(is)), \\ rename_n(\lambda_n(g), m, is) &= \lambda_m(rename_{s(n)}(g, s(m), \langle s^n_m(is), 0_m\rangle)), \\ rename_n(ap_n(h, f), m, fs) &= ap_m(rename_n(h, m, fs), rename_n(f, c, fs)). \end{aligned}$$

```
Fixpoint rename (n: nat) (g: Lambda n) (m : nat) (ids: SnocVec (Fin m) n) {struct g} : Lambda m :=
  match g with
  | Var _ i => Var m (proj ids i)
  | Abs _ b => Abs m (rename (S n) b (S m) (SCons (Fin (S m)) n (Snoc_map FS ids) F1))
  | App _ e1 e2 => App m (rename n e1 m ids) (rename n e2 m ids)
  end.

Definition lift := fun {n : nat} (f : Lambda n) => rename n f (S n) (up n).
```

And finally...

A simple prover!

- A bit primitive
- Proves β-equivalence!

∼-introduction:

$$
\begin{aligned}
varcong &: (n : N)(i : N_n)var_n(i) \sim_n var_n(i),\\
\xi &: (n : N)(g, g' : \mathscr{F}_{s(n)})(g \sim_{s(n)} g')\lambda(g) \sim_n \lambda(g'),\\
apcong &: (n : N)(h, h' : \mathscr{F}_n)(h \sim_n h')(f, f' : \mathscr{F}_n)(f \sim_n f')\\
&\quad ap_n(h, f) \sim_n ap_n(h', f'),\\
\beta &: (n : N)(g : \mathscr{F}_{s(n)})(f : \mathscr{F}_n)\\
&\quad ap(\lambda_n(g), f) \sim_n sub_{s(n)}(g, n, \langle var_n^n(id_n), f\rangle),\\
trans &: (n : N)(f, g, h : \Lambda_n)(f \sim_n g)(g \sim_n h)f \sim_n h,\\
sym &: (n : N)(f, g : \Lambda_n)(f \sim_n g)g \sim_n f.
\end{aligned}
$$

```
Inductive equiv {n: nat} : Lambda n -> Lambda n -> Prop :=
| varcong : forall (i: Fin n), equiv (Var n i) (Var n i)
| abscong : forall (g1 g2 : Lambda (S n)), equiv g1 g2 -> equiv (Abs n g1) (Abs n g2)
| apcong : forall (h1 h2 f1 f2 : Lambda n), equiv h1 h2 -> equiv f1 f2 -> equiv (App n h1 f2) (App n h2 f2)
| betared : forall (g: Lambda (S n)) (f: Lambda n),
    equiv (App n (Abs n g) f) (sub g n (SCons (Lambda n) n (Snoc_map (Var n) (idctx n)) f))
| eqtrans : forall (a b c : Lambda n), equiv a b -> equiv b c -> equiv a c
| eqsym : forall (l1 l2 : Lambda n), equiv l1 l2 -> equiv l2 l1
.
```

The end