Programowanie z typami zależnymi i dowodzenie twierdzeń

Typowanie i typy indukcyjne

17.03.2015

Logika zdaniowa

Implikacja i kwantyfikator ogólny są "wbudowane"

```
Goal (forall x:nat, P x → Q x) →
                          forall x:nat, P x → R x → Q x.
```

```
intro.
apply H; assumption.
```

Logika zdaniowa

Implikacja i kwantyfikator ogólny są "wbudowane"

```
Goal (forall x:nat, P x → Q x) →
                          forall x:nat, P x → R x → Q x.
```

```
intro.
apply H; assumption.
```

Logika zdaniowa

```
Inductive True : Prop := I : True
```

```
Inductive False : Prop :=
```

```
Print not.
```

```
not = fun A : Prop ⇒ A → False
  : Prop → Prop
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  conj : A → B → A ∧ B
```

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

Logika zdaniowa

```
Inductive True : Prop := I : True
```

```
Inductive False : Prop :=
```

```
Print not.
```

```
not = fun A : Prop ⇒ A → False
  : Prop → Prop
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  conj : A → B → A ∧ B
```

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

Logika zdaniowa

```
Inductive True : Prop := I : True
```

```
Inductive False : Prop :=
```

```
Print not.
```

```
not = fun A : Prop ⇒ A → False
  : Prop → Prop
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  conj : A → B → A ∧ B
```

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

Logika zdaniowa

```
Inductive True : Prop := I : True
Inductive False : Prop :=

Print not.
  not = fun A : Prop ⇒ A → False
    : Prop → Prop

Inductive and (A : Prop) (B : Prop) : Prop :=
  conj : A → B → A ∧ B

Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

Logika zdaniowa

```
Inductive True : Prop := I : True
Inductive False : Prop :=

Print not.
  not = fun A : Prop ⇒ A → False
    : Prop → Prop

Inductive and (A : Prop) (B : Prop) : Prop :=
  conj : A → B → A ∧ B

Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

Logika zdaniowa - podstawowe dowody

```
Theorem obvious : True.
  apply I.
Qed.
```

```
Theorem obvious' : True.
  constructor.
Qed.
```

```
Theorem False_imp : False → 2 + 2 = 5.
  destruct 1.
Qed.
```

Logika zdaniowa - podstawowe dowody

```
Theorem obvious : True.
  apply I.
Qed.
```

```
Theorem obvious' : True.
  constructor.
Qed.
```

```
Theorem False_imp : False → 2 + 2 = 5.
  destruct 1.
Qed.
```

Logika zdaniowa - podstawowe dowody

```
Theorem obvious : True.
  apply I.
Qed.
```

```
Theorem obvious' : True.
  constructor.
Qed.
```

```
Theorem False_imp : False → 2 + 2 = 5.
  destruct 1.
Qed.
```

Dowód dla and

```
Section Propositional.
  Variables P Q R : Prop.

  Theorem and_comm : P ∧ Q → Q ∧ P.

    destruct 1.

  H : P
  H0 : Q
  ============================
   Q ∧ P

    split.
```

Dowód dla and

```
Section Propositional.
  Variables P Q R : Prop.

  Theorem and_comm : P ∧ Q → Q ∧ P.

    destruct 1.

  H : P
  H0 : Q
  ============================
   Q ∧ P

    split.
```

```
2 subgoals

  H : P
  H0 : Q
  ============================
   Q

subgoal 2 is

   P


    assumption.
    assumption.
  Qed.
```

Dowód dla or

Theorem or_comm : $P \vee Q \to Q \vee P$.

```
  destruct 1.
```

```
2 subgoals
```

```
  H : P
  ============================
   Q ∨ P
```

```
subgoal 2 is
```

```
 Q ∨ P
```

```
  right; assumption.
```

Dowód dla or

```
Theorem or_comm : P ∨ Q → Q ∨ P.

  destruct 1.

2 subgoals

  H : P
  ============================
   Q ∨ P

subgoal 2 is

 Q ∨ P

  right; assumption.
```

```
1 subgoal
```

$H : Q$

==========

$Q \vee P$

```
  left; assumption.
Qed.
```

```
End Propositional.
```

Logika zdaniowa - taktyki tauto i inituition

`tauto` i `intuition`. Używając systemu sekwentów Dyckhoffa:

- `tauto` udowadnia instancje zdaniowych tautologii intuinicjonistycznych; rozumie spójniki logiczne, nie rozumie lematów
- `intuition` lub `intuition` *tac* działa jak `tauto`, ale jak już nie ma co robić, próbuje *tac* lub `auto with *`

`firstorder` — eksperymentalne rozszerzenie `tauto` na logikę pierwszego rzędu

Logika zdaniowa - taktyki tauto i inituition

`tauto` i `intuition`. Używając systemu sekwentów Dyckhoffa:

- `tauto` udowadnia instancje zdaniowych tautologii intuinicjonistycznych; rozumie spójniki logiczne, nie rozumie lematów
- `intuition` lub `intuition` *tac*
  działa jak `tauto`, ale jak już nie ma co robić, próbuje *tac* lub `auto with *`

`firstorder` — eksperymentalne rozszerzenie `tauto` na logikę pierwszego rzędu

Logika zdaniowa - taktyki tauto i inituition

`tauto` i `intuition`. Używając systemu sekwentów Dyckhoffa:

- `tauto` udowadnia instancje zdaniowych tautologii intuinicjonistycznych; rozumie spójniki logiczne, nie rozumie lematów
- `intuition` lub `intuition` *tac*
  działa jak `tauto`, ale jak już nie ma co robić, próbuje *tac* lub `auto with *`

`firstorder` — eksperymentalne rozszerzenie `tauto` na logikę pierwszego rzędu

Logika zdaniowa - taktyki tauto i inituition

`tauto` i `intuition`. Używając systemu sekwentów Dyckhoffa:

- `tauto` udowadnia instancje zdaniowych tautologii intuinicjonistycznych; rozumie spójniki logiczne, nie rozumie lematów
- `intuition` lub `intuition` *tac*
  działa jak `tauto`, ale jak już nie ma co robić, próbuje *tac* lub `auto with *`

`firstorder` — eksperymentalne rozszerzenie `tauto` na logikę pierwszego rzędu

Logika zdaniowa - taktyki tauto i inituition

`tauto` i `intuition`. Używając systemu sekwentów Dyckhoffa:

- `tauto` udowadnia instancje zdaniowych tautologii intuinicjonistycznych; rozumie spójniki logiczne, nie rozumie lematów
- `intuition` lub `intuition` *tac*
  działa jak `tauto`, ale jak już nie ma co robić, próbuje *tac* lub `auto with *`

`firstorder` — eksperymentalne rozszerzenie `tauto` na logikę pierwszego rzędu

Logika pierwszego rzędu

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

```
Theorem exist1 : ∃ x : nat, x + 1 = 2.

  exists 1.

  ============================
   1 + 1 = 2

  reflexivity.
Qed.
```

Logika pierwszego rzędu

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

```
Theorem exist1 : ∃ x : nat, x + 1 = 2.

  exists 1.

  ============================
   1 + 1 = 2

  reflexivity.
Qed.
```

Theorem exist2 : $\forall\ n\ m$: **nat**, $(\exists\ x$: **nat**, $n + x = m) \rightarrow n \leq m$.

```
  destruct 1.
```

n : **nat**
m : **nat**
x : **nat**
$H : n + x = m$
==========
$n \leq m$

```
  crush.
Qed.
```

Theorem exist2 : $\forall\ n\ m$: **nat**, $(\exists\ x$: **nat**, $n + x = m) \rightarrow n \leq m$.

destruct 1.

n : **nat**
m : **nat**
x : **nat**
$H : n + x = m$
==========
$n \leq m$

crush.
Qed.

Równość definiowalna

$$\texttt{Inductive } \mathbf{eq}\ (A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$$

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                        P x -> forall y : A, x = y -> P y
```

Równość definiowalna

$$\texttt{Inductive } \mathbf{eq}\ (A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$$

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
crush.
Qed.
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.

Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
crush.
Qed.
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
crush.
Qed.
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:
plus O O = O

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
crush.
Qed.
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:
plus O O = O

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
crush.
Qed.
```

Równość definiowalna i konwersja

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.

Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:
plus O O = O

```
  reflexivity.
```

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n

crush.
Qed.
```

Predykaty rekurencyjne

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

Predykaty rekurencyjne

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

Predykaty rekurencyjne

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

Predykaty rekurencyjne

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

Dowód indukcyjny dla Even c.d.

subgoal 2 is:

even $m \to$ **even** (S (S n) + m)

```
intro.
```

m : **nat**
n : **nat**
H : **even** n
$IHeven$: **even** $m \to$ **even** $(n + m)$
$H0$: **even** m
==========

even (S (S n) + m)

```
simpl; constructor.
```

==========

even $(n + m)$

Dowód indukcyjny dla Even c.d.

subgoal 2 is:

even $m \to$ **even** (S (S n) + m)

```
intro.
```

m : **nat**
n : **nat**
H : **even** n
$IHeven$: **even** $m \to$ **even** $(n + m)$
$H0$: **even** m
==========
even (S (S n) + m)

```
simpl; constructor.
```

==========
even $(n + m)$

Dowód indukcyjny dla Even c.d.

`subgoal 2 is:`

even $m \to$ **even** (S (S n) + m)

`intro.`

m : **nat**
n : **nat**
H : **even** n
$IHeven$: **even** $m \to$ **even** $(n + m)$
$H0$: **even** m
==========
even (S (S n) + m)

`simpl; constructor.`

==========
even $(n + m)$

Dowód indukcyjny dla Even c.d.

`subgoal 2 is:`

even $m \to$ **even** (S (S n) + m)

`intro.`

m : **nat**
n : **nat**
H : **even** n
$IHeven$: **even** $m \to$ **even** $(n + m)$
$H0$: **even** m
==========
even (S (S n) + m)

`simpl; constructor.`

==========
even $(n + m)$

Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą "być w" `bool`.

Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą "być w" `bool`.

Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą "być w" `bool`.

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Typ produktowy - przypomnienie

- kumulatywność sortów (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

- reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają ...

  - $s_1 \leq s_2$, or
  - $s_2 = \mathit{Prop}$

- reguła podtypowania

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Składnia formalna

Assuming Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$, the general typing rules are, for $1 \leq j \leq k$ and $1 \leq i \leq n$:

$$\frac{\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \quad \in E}{(I_j : A_j) \in E}$$

$$\frac{\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \quad \in E}{(c_i : C_i) \in E}$$

Parametry i zwykłe argumenty

An inductive definition $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ admits r inductive parameters if each type of constructors $(c : C)$ in Γ_C is such that

$$C \equiv \forall p_1 : P_1, \ldots, \forall p_r : P_r, \forall a_1 : A_1, \ldots \forall a_n : A_n, (I\ p_1\ \ldots p_r\ t_1 \ldots t_q)$$

with I one of the inductive definitions in Γ_I. We say that n is the number of real arguments of the constructor c.

We use the following notation:

$$\mathsf{Ind}(\Gamma)[r](\Gamma_I := \Gamma_C)$$

and

$$\Gamma_P = [p_1 : P_1; \ldots; p_r : P_r]$$

The latter will be called the *context of parameters*.

Parametry i zwykłe argumenty

An inductive definition $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ admits r inductive parameters if each type of constructors $(c : C)$ in Γ_C is such that

$$C \equiv \forall p_1 : P_1, \ldots, \forall p_r : P_r, \forall a_1 : A_1, \ldots \forall a_n : A_n, (I\ p_1\ \ldots p_r\ t_1 \ldots t_q)$$

with I one of the inductive definitions in Γ_I. We say that n is the number of real arguments of the constructor c.

We use the following notation:

$$\mathsf{Ind}(\Gamma)[r](\Gamma_I := \Gamma_C)$$

and

$$\Gamma_P = [p_1 : P_1; \ldots; p_r : P_r]$$

The latter will be called the *context of parameters*.

Przykłady

The List definition has 1 parameter:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A)\ )$$

This is also the case for this more complex definition where there is a recursive argument on a different instance of List:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ (A \to A) \to \mathsf{List}\ A)\ )$$

But the following definition has 0 parameters:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ (A * A))\ )$$

Przykłady

The List definition has 1 parameter:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A)\ )\end{aligned}$$

This is also the case for this more complex definition where there is a recursive argument on a different instance of List:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ (A \to A) \to \mathsf{List}\ A)\ )\end{aligned}$$

But the following definition has 0 parameters:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ (A * A))\ )\end{aligned}$$

Przykłady

The List definition has 1 parameter:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A)\ )\end{aligned}$$

This is also the case for this more complex definition where there is a recursive argument on a different instance of List:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ (A \to A) \to \mathsf{List}\ A)\ )\end{aligned}$$

But the following definition has 0 parameters:

$$\begin{aligned}\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A),\\ \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ (A * A))\ )\end{aligned}$$

Zasady budowy definicji indukcyjnych

A type is an *arity of sort* s if it converts to the sort s or to a product $\forall\ x : T, U$ with U an arity of sort s.

To put it simply: $\forall\ \overrightarrow{x : T}, s$.

For instance $A \to$ Set or $\forall\ A$: Prop, $A \to$ Prop are arities of sort respectively Set and Prop.

A *type of constructor of* I is either a term $(I\ t_1 \ldots\ t_n)$ or $\forall x : T, C$ with C recursively a *type of constructor of* I.

To put it simply: $\forall\ \overrightarrow{x : T}, (I\ t_1 \ldots\ t_n)$

Zasady budowy definicji indukcyjnych

A type is an *arity of sort* s if it converts to the sort s or to a product $\forall\ x : T, U$ with U an arity of sort s.

To put it simply: $\forall\ \overrightarrow{x : T}, s$.

For instance $A \to$ Set or $\forall\ A$: Prop, $A \to$ Prop are arities of sort respectively Set and Prop.

A *type of constructor of* I is either a term $(I\ t_1 \ldots\ t_n)$ or $\forall x : T, C$ with C recursively a *type of constructor of* I.

To put it simply: $\forall\ \overrightarrow{x : T}, (I\ t_1 \ldots\ t_n)$

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor C will be said to *satisfy the positivity condition* for a constant I in the following cases:

- $C = (I\ t_1 \ldots\ t_n)$ and I does not occur free in any t_i
- $C = \forall\ x : T, V$ and I occurs only strictly positively in T and the type V satisfies the positivity condition for I

To put it simply: $C = \forall\ \overrightarrow{x : T},\ (I\ t_1 \ldots\ t_n)$, where I occurs only strictly positively in every T

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor C will be said to *satisfy the positivity condition* for a constant I in the following cases:

- $C = (I\ t_1 \ldots\ t_n)$ and I does not occur free in any t_i
- $C = \forall\ x : T, V$ and I occurs only strictly positively in T and the type V satisfies the positivity condition for I

To put it simply: $C = \forall\ \overrightarrow{x : T},\ (I\ t_1 \ldots\ t_n)$, where I occurs only strictly positively in every T

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor C will be said to *satisfy the positivity condition* for a constant I in the following cases:

- $C = (I\ t_1 \ldots\ t_n)$ and I does not occur free in any t_i
- $C = \forall\ x : T, V$ and I occurs only strictly positively in T and the type V satisfies the positivity condition for I

To put it simply: $C = \forall\ \overrightarrow{x : T},\ (I\ t_1 \ldots\ t_n)$, where I occurs only strictly positively in every T

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor C will be said to *satisfy the positivity condition* for a constant I in the following cases:

- $C = (I\ t_1 \ldots\ t_n)$ and I does not occur free in any t_i
- $C = \forall\ x : T, V$ and I occurs only strictly positively in T and the type V satisfies the positivity condition for I

To put it simply: $C = \forall\ \overrightarrow{x : T},\ (I\ t_1 \ldots\ t_n)$, where I occurs only strictly positively in every T

Zasady budowy definicji indukcyjnych (c.d.)

The constant I *occurs strictly positively* in T in the following cases:

- I does not occur in T
- T converts to $(I\ t_1 \ldots\ t_n)$ and I does not occur in any of t_i
- T converts to $\forall\ x : U, V$ and I does not occur in type U but occurs strictly positively in type V
- T converts to $(I'\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I' is the name of an inductive declaration of the form
  $\mathrm{Ind}(\Gamma)[m](I : A := c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_1; \ldots;$
  $c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_n\ )$
  (in particular, it is not mutually defined and it has m parameters) and I does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I' satisfy the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

The constant I *occurs strictly positively* in T in the following cases:

- I does not occur in T
- T converts to $(I\ t_1 \ldots\ t_n)$ and I does not occur in any of t_i
- T converts to $\forall\ x : U, V$ and I does not occur in type U but occurs strictly positively in type V
- T converts to $(I'\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I' is the name of an inductive declaration of the form
  $\mathrm{Ind}(\Gamma)[m](I : A := c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_1; \ldots;$
  $c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_n\ )$
  (in particular, it is not mutually defined and it has m parameters) and I does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I' satisfy the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

The constant I *occurs strictly positively* in T in the following cases:

- I does not occur in T
- T converts to $(I\ t_1 \ldots\ t_n)$ and I does not occur in any of t_i
- T converts to $\forall\ x : U, V$ and I does not occur in type U but occurs strictly positively in type V
- T converts to $(I'\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I' is the name of an inductive declaration of the form
  $\mathrm{Ind}(\Gamma)[m](I : A := c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_1; \ldots;$
  $c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_n\ )$
  (in particular, it is not mutually defined and it has m parameters) and I does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I' satisfy the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

The constant I *occurs strictly positively* in T in the following cases:

- I does not occur in T
- T converts to $(I\ t_1 \ldots\ t_n)$ and I does not occur in any of t_i
- T converts to $\forall\ x : U, V$ and I does not occur in type U but occurs strictly positively in type V
- T converts to $(I'\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I' is the name of an inductive declaration of the form
  $\mathsf{Ind}(\Gamma)[m](I : A := c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_1; \ldots;$
  $c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_n\ )$
  (in particular, it is not mutually defined and it has m parameters) and I does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I' satisfy the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor T of I' *satisfies the nested positivity condition* for a constant I in the following cases:

- $T = (I'\ b_1 \ldots b_m\ u_1 \ldots\ u_p)$, I' is an inductive definition with m parameters and I does not occur in any u_i
- $T = \forall\ x : U, V$ and I occurs only strictly positively in U and the type V satisfies the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

The type of constructor T of I' *satisfies the nested positivity condition* for a constant I in the following cases:

- $T = (I'\ b_1 \ldots b_m\ u_1 \ldots\ u_p)$, I' is an inductive definition with m parameters and I does not occur in any u_i
- $T = \forall\ x : U, V$ and I occurs only strictly positively in U and the type V satisfies the nested positivity condition for I

Zasady budowy definicji indukcyjnych (c.d.)

Let E be an environment and $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ are contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ and Γ_C is $[c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma; \Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E; \mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C))[\Gamma]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$.

Zasady budowy definicji indukcyjnych (c.d.)

Let E be an environment and $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ are contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ and Γ_C is $[c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma; \Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E; \mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C))[\Gamma]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$.

Zasady budowy definicji indukcyjnych (c.d.)

Let E be an environment and $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ are contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ and Γ_C is $[c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma; \Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E; \mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C))[\Gamma]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$.

Zasady budowy definicji indukcyjnych (c.d.)

Let E be an environment and $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ are contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ and Γ_C is $[c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma; \Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E; \mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C))[\Gamma]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$.

Przykład: "drzewa arności list"

```
Inductive nat_tree : Set :=
| NNode' : nat → list nat_tree → nat_tree.
```

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

Akceptowane przez Coqa, ale nie akceptowalne według definicji z manuala....

Przykład: "drzewa arności list"

```
Inductive nat_tree : Set :=
| NNode' : nat → list nat_tree → nat_tree.
```

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

Akceptowane przez Coqa, ale nie akceptowalne według definicji z manuala....

Przykład: "drzewa arności list"

```
Inductive nat_tree : Set :=
| NNode' : nat → list nat_tree → nat_tree.
```

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

Akceptowane przez Coqa, ale nie akceptowalne według definicji z manuala....

Przykład: "drzewa arności list" c.d.

Warunki z manuala powinny sprowadzać się do sprawdzenia poprawności takich typów:

```
Inductive nat_tree : Set :=
| NNode' : nat → list_nat_tree → nat_tree.

with list_nat_tree : Set :=
| nil : list_nat_tree
| cons : nat_tree → list_nat_tree → list_nat_tree.
```

Przykład: "drzewa arności list" c.d.

Warunki z manuala powinny sprowadzać się do sprawdzenia poprawności takich typów:

```
Inductive nat_tree : Set :=
| NNode' : nat → list_nat_tree → nat_tree.

with list_nat_tree : Set :=
| nil : list_nat_tree
| cons : nat_tree → list_nat_tree → list_nat_tree.
```

Polimorfizm definicji indukcyjnych

Jeśli definicja jest w sorcie *Type* (i ma parametry typu *Type*),

`Definition I (P`$_1$ `:` *Type*`) ... (P`$_n$ `:` *Type*`) : ∀..., Type :=`
`...`

to w typowaniu instancji „opuszcza się" sort tak nisko jak „można".

`(I` t_1 `...` t_k`)` `:` *Set*

Formalnie używana jest dodatkowa reguła typowania **Ind-Family**, która chwilowo zastępuje **Ind-Const** i **App**, dając typ mniejszy w sensie $\leq$

Co to znaczy „można"?

- definicja z nowymi sortami musi być poprawna, czyli
- pusty typ lub mała definicja indukcyjna o jednym konstruktorze są w `Prop`, mała definicja indukcyjna o wielu konstruktorach jest w `Set`, pozostałe w `Type`

Polimorfizm definicji indukcyjnych

Jeśli definicja jest w sorcie *Type* (i ma parametry typu *Type*),

`Definition I (P`$_1$ `:` *Type*`) ... (P`$_n$ `:` *Type*`) : ∀..., Type :=`
`...`

to w typowaniu instancji „opuszcza się" sort tak nisko jak „można".

`(I` t_1 `...` t_k`)` `:` *Set*

Formalnie używana jest dodatkowa reguła typowania **Ind-Family**, która chwilowo zastępuje **Ind-Const** i **App**, dając typ mniejszy w sensie $\leq$

Co to znaczy „można"?

- definicja z nowymi sortami musi być poprawna, czyli
- pusty typ lub mała definicja indukcyjna o jednym konstruktorze są w `Prop`, mała definicja indukcyjna o wielu konstruktorach jest w `Set`, pozostałe w `Type`

Polimorfizm definicji indukcyjnych

Jeśli definicja jest w sorcie *Type* (i ma parametry typu *Type*),

`Definition I (P`$_1$ `:` *Type*`) ... (P`$_n$ `:` *Type*`) : ∀..., Type :=`
`...`

to w typowaniu instancji „opuszcza się" sort tak nisko jak „można".

`(I` t_1 `...` t_k`)` `:` *Set*

Formalnie używana jest dodatkowa reguła typowania **Ind-Family**, która chwilowo zastępuje **Ind-Const** i **App**, dając typ mniejszy w sensie $\leq$

Co to znaczy „można"?

- definicja z nowymi sortami musi być poprawna, czyli
- pusty typ lub mała definicja indukcyjna o jednym konstruktorze są w `Prop`, mała definicja indukcyjna o wielu konstruktorach jest w `Set`, pozostałe w `Type`

Polimorfizm definicji indukcyjnych

Jeśli definicja jest w sorcie *Type* (i ma parametry typu *Type*),

`Definition I (P`$_1$ `:` *Type*`) ... (P`$_n$ `:` *Type*`) : ∀..., Type :=`
`...`

to w typowaniu instancji „opuszcza się" sort tak nisko jak „można".

`(I` t_1 `...` t_k`)` `:` *Set*

Formalnie używana jest dodatkowa reguła typowania **Ind-Family**, która chwilowo zastępuje **Ind-Const** i **App**, dając typ mniejszy w sensie $\leq$

Co to znaczy „można"?

- definicja z nowymi sortami musi być poprawna, czyli
- pusty typ lub mała definicja indukcyjna o jednym konstruktorze są w `Prop`, mała definicja indukcyjna o wielu konstruktorach jest w `Set`, pozostałe w `Type`

Polimorfizm definicji indukcyjnych

Jeśli definicja jest w sorcie *Type* (i ma parametry typu *Type*),

`Definition I (P`$_1$ `:` *Type*`) ... (P`$_n$ `:` *Type*`) :` $\forall$`..., Type :=`
`...`

to w typowaniu instancji „opuszcza się" sort tak nisko jak „można".

`(I` t_1 `...` t_k`)` `:` *Set*

Formalnie używana jest dodatkowa reguła typowania **Ind-Family**, która chwilowo zastępuje **Ind-Const** i **App**, dając typ mniejszy w sensie $\leq$

Co to znaczy „można"?

- definicja z nowymi sortami musi być poprawna, czyli
- pusty typ lub mała definicja indukcyjna o jednym konstruktorze są w `Prop`, mała definicja indukcyjna o wielu konstruktorach jest w `Set`, pozostałe w `Type`

Przykłady polimorfizmu sortów - **prod**

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

```
Check (fun A:Set ⇒ prod A).
fun A : Set ⇒ prod A : Set → Type → Type
```

```
Check (fun A:Prop ⇒ prod A A).
fun A : Prop ⇒ prod A A : Prop → Prop
```

```
Check (fun (A:Prop) (B:Set) ⇒ prod A B).
fun (A : Prop) (B : Set) ⇒ prod A B : Prop → Set → Set
```

```
Check (fun (A:Type) (B:Prop) ⇒ prod A B).
fun (A : Type) (B : Prop) ⇒ prod A B : Type → Prop → Type
```

Przykłady polimorfizmu sortów - **prod**

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

```
Check (fun A:Set ⇒ prod A).
fun A : Set ⇒ prod A : Set → Type → Type
```

```
Check (fun A:Prop ⇒ prod A A).
fun A : Prop ⇒ prod A A : Prop → Prop
```

```
Check (fun (A:Prop) (B:Set) ⇒ prod A B).
fun (A : Prop) (B : Set) ⇒ prod A B : Prop → Set → Set
```

```
Check (fun (A:Type) (B:Prop) ⇒ prod A B).
fun (A : Type) (B : Prop) ⇒ prod A B : Type → Prop → Type
```

Przykłady polimorfizmu sortów - **prod**

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

```
Check (fun A:Set ⇒ prod A).
fun A : Set ⇒ prod A : Set → Type → Type
```

```
Check (fun A:Prop ⇒ prod A A).
fun A : Prop ⇒ prod A A : Prop → Prop
```

```
Check (fun (A:Prop) (B:Set) ⇒ prod A B).
fun (A : Prop) (B : Set) ⇒ prod A B : Prop → Set → Set
```

```
Check (fun (A:Type) (B:Prop) ⇒ prod A B).
fun (A : Type) (B : Prop) ⇒ prod A B : Type → Prop → Type
```

Przykłady polimorfizmu sortów - **prod**

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

```
Check (fun A:Set ⇒ prod A).
fun A : Set ⇒ prod A : Set → Type → Type
```

```
Check (fun A:Prop ⇒ prod A A).
fun A : Prop ⇒ prod A A : Prop → Prop
```

```
Check (fun (A:Prop) (B:Set) ⇒ prod A B).
fun (A : Prop) (B : Set) ⇒ prod A B : Prop → Set → Set
```

```
Check (fun (A:Type) (B:Prop) ⇒ prod A B).
fun (A : Type) (B : Prop) ⇒ prod A B : Type → Prop → Type
```

Przykłady polimorfizmu sortów - **prod**

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

Check (fun A:Set ⇒ prod A).
fun A : Set ⇒ prod A : Set → Type → Type

Check (fun A:Prop ⇒ prod A A).
fun A : Prop ⇒ prod A A : Prop → Prop

Check (fun (A:Prop) (B:Set) ⇒ prod A B).
fun (A : Prop) (B : Set) ⇒ prod A B : Prop → Set → Set

Check (fun (A:Type) (B:Prop) ⇒ prod A B).
fun (A : Type) (B : Prop) ⇒ prod A B : Type → Prop → Type
```