

# Logika i teoria typów

Wykład 11

11 stycznia 2020

## Ko-rekursja i ko-indukcja

# Strumienie

Jaki sens ma równanie  $\mathbf{Stream} = \mathbf{int} \times \mathbf{Stream}$ ?

Można je potraktować podobnie jak równanie  $\mathbf{int} = \mathbf{1} \oplus \mathbf{int}$ ,  
i rozważać typ rekurencyjny (indukcyjny)

$$\mathbf{Stream} = \mu p. \mathbf{int} \times p.$$

Ale to jest *najmniejsze* rozwiązanie równania: typ pusty.

Najmniejszy punkt stały  $\mu = \mu p. \sigma(p)$ :

- Kres górny (suma) ciągu aproksymacji skończonych.
- Najmniejszy „prefixpoint”:

$$\frac{\sigma(R) \subseteq R}{\mu \subseteq R}$$

$$\frac{f : \sigma(R) \rightarrow R}{\mathbf{Elim} f : \mu \rightarrow R}$$

## Najmniejszy punkt stały $\mu = \mu p. \sigma(p)$ :

- Kres górny (suma) ciągu aproksymacji skończonych.
- Najmniejszy „prefixpoint”:

$$\frac{\sigma(R) \subseteq R}{\mu \subseteq R}$$

$$\frac{f : \sigma(R) \rightarrow R}{\text{Elim } f : \mu \rightarrow R}$$

## Największy punkt stały $\nu = \nu p. \sigma(p)$ :

- Kres dolny (przecięcie) ciągu aproksymacji nieskończonych.
- Największy „postfixpoint”:

$$\frac{R \subseteq \sigma(R)}{R \subseteq \nu}$$

$$\frac{f : R \rightarrow \sigma(R)}{\text{Intro } f : R \rightarrow \nu}$$

Największy „postfixpoint”  $\nu = \nu p. \sigma(p)$

$$\frac{R \subseteq \sigma(R)}{R \subseteq \nu}$$

$$\frac{f : R \rightarrow \sigma(R)}{\text{Intro } f : R \rightarrow \nu}$$

Na przykład **Stream** =  $\nu p. \text{nat} \times p$ :

$$\frac{R \subseteq \text{nat} \times R}{R \subseteq \text{Stream}}$$

$$\frac{f : R \rightarrow \text{nat} \times R}{\text{Intro } f : R \rightarrow \text{Stream}}$$

Co to znaczy?

- Jeśli  $R \subseteq \text{nat} \times R$ , to każdy element  $R$  jest strumieniem.
- Funkcja  $R \rightarrow \text{nat} \times R$  wyznacza funkcję  $R \rightarrow \text{Stream}$ .

## Destruktory

Destruktory nie są pierwotne.

Definiujemy je przez dopasowanie:

```
CoInductive Stream : Set
  := Cons : nat -> Stream -> Stream.
```

```
Definition hd (s : Stream)
  := match s with Cons a t => a end.
```

```
Definition tl (s : Stream)
  := match s with Cons a t => t end.
```

```
Coq < Check hd.
```

```
hd : Stream -> nat
```

```
Coq < Check tl.
```

```
tl : Stream -> Stream
```

## Przykład Coquanda

Może być więcej niż jeden konstruktor.  
(Nie ma uniwersalnych destruktorów.)



## Przykład Coquanda

Może być więcej niż jeden konstruktor.  
(Nie ma uniwersalnych destruktorów.)

Proces typu  $P$  może wczytać liczbę i wypisać liczbę.

$in : (\mathbf{nat} \rightarrow P) \rightarrow P;$

$out : \mathbf{nat} \rightarrow P \rightarrow P.$

## Przykład Coquanda

Może być więcej niż jeden konstruktor.  
(Nie ma uniwersalnych destruktatorów.)

Proces typu  $P$  może wczytać liczbę i wypisać liczbę.

$in : (\mathbf{nat} \rightarrow P) \rightarrow P;$

$out : \mathbf{nat} \rightarrow P \rightarrow P.$

Na przykład coś takiego:

$in(\lambda n_1. out(0, in(\lambda n_2. out(n_1, out(1, in(\lambda n_3 \dots$

## Przykład Coquanda

Może być więcej niż jeden konstruktor.  
(Nie ma uniwersalnych destruktatorów.)

Proces typu  $P$  może wczytać liczbę i wypisać liczbę.

$in : (\mathbf{nat} \rightarrow P) \rightarrow P;$

$out : \mathbf{nat} \rightarrow P \rightarrow P.$

Na przykład coś takiego:

$in(\lambda n_1. out(0, in(\lambda n_2. out(n_1, out(1, in(\lambda n_3 \dots$

Można definiować procesy ko-rekurencyjnie:

$p = in(\lambda n. out(n + 1, p))$

## Przykład: definicje korekurencyjne:

```
CoFixpoint samezera : Stream := Cons 0 samezera.
```

```
CoFixpoint codrugieod : nat -> Stream :=  
fun n :nat => Cons n (codrugieod (n+2)).
```

```
Definition wszystkieparzyste := codrugieod 0.
```

```
Coq < Eval simpl in hd (tl (wszystkieparzyste)).  
= 2  
: nat
```

## Preedykaty koindukcyjne

Równość strumieni (ekstensjonalna):

$s = t$ , wtedy i tylko wtedy, gdy:

- $hd(s) = hd(t)$  oraz
- $tl(s) = tl(t)$

To jest *bisymulacja*.

## Preedykaty koindukcyjne

Równość strumieni (ekstensjonalna):

$s = t$ , wtedy i tylko wtedy, gdy:

- $hd(s) = hd(t)$  oraz
- $tl(s) = tl(t)$

To jest *bisymulacja*.

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
```

```
  eqst :
```

```
  hd s1 = hd s2 -> EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

Preedykat koindukcyjny:  
strumień składa się tylko z liczb parzystych.

```
CoInductive Sameparzyste : Stream -> Prop :=  
krok : forall x : Stream,  
even (hd x) -> Sameparzyste (tl x) -> Sameparzyste x.
```

Predykat koindukcyjny:  
strumień składa się tylko z liczb parzystych.

```
CoInductive Sameparzyste : Stream -> Prop :=  
krok : forall x : Stream,  
even (hd x) -> Sameparzyste (tl x) -> Sameparzyste x.
```

Dowód przez koindukcję:  
(wiemy, że zeroparz : even(0))

```
CoFixpoint samezera : Stream := Cons 0 samezera.
```

```
CoFixpoint Zeraparzyste : Sameparzyste samezera :=  
krok samezera zeroparz Zeraparzyste.
```

```
Coq < Zeraparzyste is corecursively defined
```



## Sens moralny

```
CoInductive Sameparzyste : Stream -> Prop :=  
krok : forall x : Stream,  
even (hd x) -> Sameparzyste (tl x) -> Sameparzyste x.
```

```
CoFixpoint Zeraparzyste : Sameparzyste samezera :=  
krok samezera zeroparz Zeraparzyste.
```

Predykat `Sameparzyste` to w istocie największy punkt stały operacji  $s \mapsto \text{even} \times s$ . W tym dowodzie korzystamy z reguły

$$\frac{R \subseteq \text{even} \times R}{R \subseteq \text{Sameparzyste}}$$

dla relacji  $R = \{\text{samezera}\}$

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

Sprawdzamy, że  $hd(s)$  jest liczbą parzystą.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

Sprawdzamy, że  $hd(s)$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  też należy do rodziny  $R$ .

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

Sprawdzamy, że  $hd(s)$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  też należy do rodziny  $R$ .

Z założenia koindukcyjnego wnioskujemy, że  $tl(s)$  składa się z samych liczb parzystych.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

Sprawdzamy, że  $hd(s)$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  też należy do rodziny  $R$ .

Z założenia koindukcyjnego wnioskujemy, że  $tl(s)$  składa się z samych liczb parzystych.

A zatem  $s$  też składa się tylko z liczb parzystych.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że każdy strumień z rodziny  $R$  składa się wyłącznie z liczb parzystych.

Weźmy dowolny strumień  $s \in R$ .

Sprawdzamy, że  $hd(s)$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  też należy do rodziny  $R$ .

Z założenia koindukcyjnego wnioskujemy, że  $tl(s)$  składa się z samych liczb parzystych.

A zatem  $s$  też składa się tylko z liczb parzystych.  
Twierdzenie udowodnione.



# Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

# Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

Sprawdzamy, że  $hd(\textit{samezera}) = 0$  jest liczbą parzystą.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

Sprawdzamy, że  $hd(\textit{samezera}) = 0$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  to też *samezera*.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

Sprawdzamy, że  $hd(\textit{samezera}) = 0$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  to też *samezera*.

Z założenia koindukcyjnego wnioskujemy, że strumień *samezera* składa się z samych liczb parzystych.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

Sprawdzamy, że  $hd(\textit{samezera}) = 0$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  to też *samezera*.

Z założenia koindukcyjnego wnioskujemy, że strumień *samezera* składa się z samych liczb parzystych.

A zatem strumień *samezera* składa się tylko z liczb parzystych.

## Kolokwialny dowód przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Weźmy więc ten strumień.

Sprawdzamy, że  $hd(\textit{samezera}) = 0$  jest liczbą parzystą.

Sprawdzamy, że  $tl(s)$  to też *samezera*.

Z założenia koindukcyjnego wnioskujemy, że strumień *samezera* składa się z samych liczb parzystych.

A zatem strumień *samezera* składa się tylko z liczb parzystych.  
Twierdzenie udowodnione.

## Bałamutny „dowód” przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.



## Bałamutny „dowód” przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Korzystaliśmy z założenia koindukcyjnego, że strumień  $tl(\textit{samezera}) = \textit{samezera}$  składa się z samych liczb parzystych.

## Bałamutny „dowód” przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Korzystaliśmy z założenia koindukcyjnego, że strumień  $tl(\textit{samezera}) = \textit{samezera}$  składa się z samych liczb parzystych.

Ale w takim razie teza wynika bezpośrednio z założenia koindukcyjnego!

## Bałamutny „dowód” przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Korzystaliśmy z założenia koindukcyjnego, że strumień  $tl(\textit{samezera}) = \textit{samezera}$  składa się z samych liczb parzystych.

Ale w takim razie teza wynika bezpośrednio z założenia koindukcyjnego!

Twierdzenie udowodnione !?

## Bałamutny „dowód” przez ko-indukcję

Chcemy udowodnić, że strumień *samezera* składa się wyłącznie z liczb parzystych.

Korzystaliśmy z założenia koindukcyjnego, że strumień  $tl(\textit{samezera}) = \textit{samezera}$  składa się z samych liczb parzystych.

Ale w takim razie teza wynika bezpośrednio z założenia koindukcyjnego!

Twierdzenie udowodnione!?

**Nie**, ten „dowód” nie jest **produktywny**

## Definicje produktywne

Dowód przez koindukcję, to też obiekt koindukcyjny: nieskończony term dowodowy zbudowany z konstruktorów właściwych dla dowodzonego predykatu koindukcyjnego (w tym przypadku `krok`).

## Definicje produktywne

Dowód przez koindukcję, to też obiekt koindukcyjny: nieskończony term dowodowy zbudowany z konstruktorów właściwych dla dowodzonego predykatu koindukcyjnego (w tym przypadku *krok*).

Definicja nieskończonego obiektu (np. strumienia) jest *produktywna*, jeśli definiowany obiekt redukuje się do *postaci kanonicznej* (z konstruktorem) i wszystkie jego składowe też.

## Definicje produktywne

Dowód przez koindukcję, to też obiekt koindukcyjny: nieskończony term dowodowy zbudowany z konstruktorów właściwych dla dowodzonego predykatu koindukcyjnego (w tym przypadku `krok`).

Definicja nieskończonego obiektu (np. strumienia) jest *produktywna*, jeśli definiowany obiekt redukuje się do *postaci kanonicznej* (z konstruktorem) i wszystkie jego składowe też.

Te definicje nie są produktywne:

```
CoFixpoint Bad : Stream := Bad.
```

```
CoFixpoint Dab : Stream := tl (Cons 0 Dab)
```

## Nieproduktywna definicja funkcji $F : \text{Stream} \rightarrow \text{Stream}$

```
CoFixpoint F : Stream -> Stream :=  
fun s : Stream =>  
Cons ((hd s)+(hd(tl s))) (F(F(tl(tl(s)))))
```

Czyli po ludzku:  $F(a :: b :: t) = (a + b :: F^2(t))$ .



## Nieproduktywna definicja funkcji $F : \text{Stream} \rightarrow \text{Stream}$

```
CoFixpoint F : Stream -> Stream :=  
fun s : Stream =>  
Cons ((hd s)+(hd(tl s))) (F(F(tl(tl(s)))))
```

Czyli po ludzku:  $F(a :: b :: t) = (a + b :: F^2(t))$ .

Co jest na drugim miejscu w strumieniu  $F(0 :: 1 :: 2 :: 3...)$ ?

## Nieproduktywna definicja funkcji $F : \text{Stream} \rightarrow \text{Stream}$

```
CoFixpoint F : Stream -> Stream :=  
fun s : Stream =>  
Cons ((hd s)+(hd(tl s))) (F(F(tl(tl(s)))))
```

Czyli po ludzku:  $F(a :: b :: t) = (a + b :: F^2(t))$ .

Co jest na drugim miejscu w strumieniu  $F(0 :: 1 :: 2 :: 3...)$ ?

Liczmy:

$$F(0 :: 1 :: 2 :: 3...) = 1 :: F(F(2 :: 3...)) =$$
$$1 :: F(2 + 3 :: F^2(4...)) = 1 :: F(2 + 3 :: F(4 + 5 :: F^2(6...)))$$

Nie można ewaluować  $F(2 + 3 :: F^2(4...))$ ,  
bo argument nie jest postaci  $a :: b :: t$ .

# Jak zagwarantować produktywność

Definicje korekurencyjne w Coqu muszą być *strzeżone* (guarded). Wywołania korekurencyjne są dozwolone tylko bezpośrednio pod konstruktorem czołowym. Definicja:

```
CoFixpoint F : Stream -> Stream :=  
fun s : Stream =>  
Cons ((hd s)+(hd(tl s))) (F(F(tl(tl(s)))))
```

jest niestrzeżona, bo **F** występuje pod **F**.

Gdzie drwa rąbią. . .

Ta definicja jest, owszem, produktywna:

$$F(a :: t) = (a + 1 :: F^2(t)).$$

Gdzie drwa rąbią. . .

Ta definicja jest, owszem, produktywna:

$$F(a :: t) = (a + 1 :: F^2(t)).$$

Ale Coq jej też nie lubi:

## Gdzie drwa rąbią...

Ta definicja jest, owszem, produktywna:

$$F(a :: t) = (a + 1 :: F^2(t)).$$

Ale Coq jej też nie lubi:

```
CoFixpoint F : Stream -> Stream :=  
fun s : Stream =>  
Cons ((hd s)+1) (F(F(tl(s)))).
```

Coq < Error:

Recursive definition of F is ill-formed.

In environment

F : Stream -> Stream

s : Stream

Nested recursive occurrences.

Recursive definition is:

```
"fun s : Stream => Cons (hd s + 1) (F (F (tl s)))".
```

# Dependent types

## Dependent types

**Principal idea:** Type  $array[n]$  depends on  $n : int$ .  
It is created by a type constructor (type family)

$$array : int \Rightarrow *$$



# Dependent types

**Principal idea:** Type  $array[n]$  depends on  $n : int$ .  
It is created by a type constructor (type family)

$$array : int \Rightarrow *$$

**Curry-Howard:** Constructor  $P : \tau \Rightarrow *$  is a predicate on  $\tau$ .

# Dependent types

**Principal idea:** Type  $array[n]$  depends on  $n : int$ .  
It is created by a type constructor (type family)

$$array : int \Rightarrow *$$

**Curry-Howard:** Constructor  $P : \tau \Rightarrow *$  is a predicate on  $\tau$ .

Adding all numbers in an array:

$$sum_n : array[n] \rightarrow int$$

# Dependent types

**Principal idea:** Type  $array[n]$  depends on  $n : int$ .  
It is created by a type constructor (type family)

$$array : int \Rightarrow *$$

**Curry-Howard:** Constructor  $P : \tau \Rightarrow *$  is a predicate on  $\tau$ .

Adding all numbers in an array:

$$sum_n : array[n] \rightarrow int$$

Parameterized by  $n$ :

$$sum : (n : int) \rightarrow array[n] \rightarrow int$$

$$sum : \forall n : int. array[n] \rightarrow int$$

# System $\lambda P$

Poziomy syntaktyczne:

- ▶ Termy, np.  $\lambda x^{\forall z:q. \alpha(z)} \lambda y^{p \rightarrow q} v^p. x(yv)$ ;
- ▶ Typy i konstruktory, np.  $\lambda x^q. \alpha x \rightarrow p$ ,  
 $(\forall z : q. \alpha(z)) \rightarrow (y : p \rightarrow q) \rightarrow (v : p) \rightarrow \alpha(yv)$ .
- ▶ Rodzaje, np.  $\alpha$  jest rodzaju  $q \Rightarrow *$ .

Nie ma „absolutnej” definicji termu, konstruktora, ani rodzaju.  
Wszystko zależy od otoczenia.

Np. to, czy  $\alpha x$  jest legalnym typem czy nie,  
zależy od rodzaju zmiennej  $\alpha$  i typu zmiennej  $x$ .

## Surowa składnia („raw terms”)

$$\kappa ::= * \mid (\Pi x:\phi \kappa);$$

$$\phi ::= \alpha \mid (\forall x:\phi \phi) \mid (\phi M) \mid (\lambda x:\phi \phi);$$

$$M ::= x \mid (MM) \mid (\lambda x:\phi M);$$

$$\Gamma ::= \emptyset \mid \Gamma, (x : \phi) \mid \Gamma, (\alpha : \kappa).$$

# Alternatywna notacja

- ▶  $(x : \tau) \Rightarrow \kappa$  zamiast  $\prod_{x:\tau}. \kappa$ ;
- ▶  $\tau \Rightarrow \kappa$ , gdy  $x$  nie jest wolne w  $\kappa$ ;
- ▶  $(x : \tau) \rightarrow \sigma$  zamiast  $\forall x:\tau. \sigma$ ;
- ▶  $\tau \rightarrow \sigma$ , gdy  $x$  nie jest wolne w  $\sigma$ .

## Wiązanie zmiennych

$$FV(*) = \emptyset, \quad FV(\alpha) = \{\alpha\}, \quad FV(x) = \{x\},$$

$$FV(EE') = FV(E) \cup FV(E'),$$

$$FV(\nabla x:\phi E) = FV(\phi) \cup (FV(E) - \{x\}),$$

gdzie  $\nabla \in \{\lambda, \Pi, \forall\}$ .

(Zakładamy alfa-konwersję)

# Alfa-konwersja: ćwiczenie

Niech  $x : p$  oraz  $P : p \rightarrow *$ .

Który z tych osądów jest poprawny?

▶  $\Gamma \vdash (\lambda x:Px. x) : \forall x:Px. Px$ ?

▶  $\Gamma \vdash (\lambda x:Px. x) : \forall y:Px. Px$ ?



# Alfa-konwersja: ćwiczenie

Niech  $x : p$  oraz  $P : p \rightarrow *$ .

Który z tych osądów jest poprawny?

▶  $\Gamma \vdash (\lambda x:Px. x) : \forall x:Px. Px$ ?

▶  $\Gamma \vdash (\lambda x:Px. x) : \forall y:Px. Px$ ?

Po przemianowaniu zmiennych:

▶  $\Gamma \vdash (\lambda y:Px. y) : \forall y:Px. Px$

## Alfa-konwersja: ćwiczenie

Niech  $x : p$  oraz  $P : p \rightarrow *$ .

Który z tych osądów jest poprawny?

▶  $\Gamma \vdash (\lambda x:P_x. x) : \forall x:P_x. P_x ?$

▶  $\Gamma \vdash (\lambda x:P_x. x) : \forall y:P_x. P_x ?$

Po przemianowaniu zmiennych:

▶  $\Gamma \vdash (\lambda y:P_x. y) : \forall y:P_x. P_x$  (czyli  $P_x \rightarrow P_x$ ).

## Alfa-konwersja: ćwiczenie

Niech  $x : p$  oraz  $P : p \rightarrow *$ .

Który z tych osądów jest poprawny?

▶  $\Gamma \vdash (\lambda x:P_x. x) : \forall x:P_x. P_x ?$

▶  $\Gamma \vdash (\lambda x:P_x. x) : \forall y:P_x. P_x ?$

Po przemianowaniu zmiennych:

▶  $\Gamma \vdash (\lambda y:P_x. y) : \forall y:P_x. P_x$  (czyli  $P_x \rightarrow P_x$ ).

Typ  $\forall x:P_x. P_x$ , czyli  $\forall y:P_x. P_y$ , jest nielegalny.

Bo wtedy  $P : P_x \Rightarrow *$ .

# Beta-redukcja

Podstawienie  $E[x := M]$ , gdzie  $M$  to surowy term, definiujemy jak zwykle.

Beta-redukcja:

$$(\lambda x:\tau. M)N \rightarrow_{\beta} M[x := N];$$

$$(\lambda x:\tau. \phi)N \rightarrow_{\beta} \phi[x := N].$$

## System $\lambda P$ : reguły

są w pliku

<https://www.mimuw.edu.pl/~urzy/Litt/wreczalt-lamP.pdf>

## System $\lambda P$ : reguły

są w pliku

<https://www.mimuw.edu.pl/~urzy/Litt/wreczalt-lamP.pdf>

dostępnym też z Moodlea.

## Metazmienne w regułach:

- ▶  $\kappa$  – rodzaj;
- ▶  $\tau, \sigma$  – typy;
- ▶  $\varphi$  – konstruktor;
- ▶  $M$  – term obiektowy;
- ▶  $x$  – zmienna obiektowa;
- ▶  $\alpha$  – zmienna konstruktorowa (typowa).

## Trzy rodzaje osądów

- ▶ Kind formation judgements of the form  $\Gamma \vdash \kappa : \square$ ,  
 (“ $\kappa$  is a kind in the environment  $\Gamma$ ”)
- ▶ Kinding judgements of the form  $\Gamma \vdash \varphi : \kappa$ ,  
 (“ $\varphi$  is a constructor of kind  $\kappa$  in  $\Gamma$ .”)
- ▶ Typing judgements of the form  $\Gamma \vdash M : \tau$ ,  
 (“ $M$  is a term of type  $\tau$  in  $\Gamma$ .”)



# Poprawne otoczenie

Otoczenie  $\Gamma$  jest *ciągami* deklaracji, a nie zbiorem.

*Poprawne otoczenie*: takie, które występuje w jakimś wyprowadzalnym osądzie.

Nie każdy ciąg deklaracji jest poprawnym otoczeniem, np. ciąg  $\{\alpha : p \Rightarrow *, x : p\}$  nie jest, bo najpierw trzeba zadeklarować  $p : *$ .

Można udowodnić taki fakt:

$\Gamma$  jest poprawne wtedy i tylko wtedy, gdy  $\Gamma \vdash * : \square$ .

# Formowanie rodzajów

$$\vdash * : \square \qquad \frac{\Gamma, x : \tau \vdash \kappa : \square}{\Gamma \vdash \prod_{x : \tau} \kappa : \square}$$

Wszystkie rodzaje są postaci  $\prod_{x_1 : \tau_1} \dots x_n : \tau_n. *$ ,  
czyli postaci  $(x_1 : \tau_1) \Rightarrow \dots \Rightarrow (x_n : \tau_n) \Rightarrow *$ .

Ale tu mogą być zależności:

$$(x_1 : \tau_1) \Rightarrow (x_2 : \tau_2(x_1)) \Rightarrow (x_3 : \tau_3(x_1, x_2)) \Rightarrow \dots \\ \dots \Rightarrow (x_n : \tau_n(x_1, \dots, x_{n-1})) \Rightarrow *$$

# Rodzajowanie

$$\frac{\Gamma \vdash \kappa : \square}{\Gamma, \alpha : \kappa \vdash \alpha : \kappa} \quad (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau \sigma) : *}$$

$$\frac{\Gamma \vdash \varphi : (\prod x : \tau \kappa) \quad \Gamma \vdash M : \tau}{\Gamma \vdash (\varphi M) : \kappa[x := M]}$$

$$\frac{\Gamma, x : \tau \vdash \varphi : \kappa}{\Gamma \vdash (\lambda x : \tau \varphi) : (\prod x : \tau \kappa)}$$

# Typowanie

$$\frac{\Gamma \vdash \tau : *}{\Gamma, x : \tau \vdash x : \tau} \quad (x \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash M : (\forall x : \tau \sigma) \quad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \sigma[x := N]}$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x : \tau M) : (\forall x : \tau \sigma)}$$

## Oslabianie (generyczne)

$$\frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \mathbf{ls} : \mathbf{ps}}{\Gamma, x : \tau \vdash \mathbf{ls} : \mathbf{ps}} \quad (x \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \kappa : \square \quad \Gamma \vdash \mathbf{ls} : \mathbf{ps}}{\Gamma, \alpha : \kappa \vdash \mathbf{ls} : \mathbf{ps}} \quad (\alpha \notin \text{Dom}(\Gamma))$$

# Konwersja

$$\frac{\Gamma \vdash \varphi : \kappa \quad \Gamma \vdash \kappa' : \square}{\Gamma \vdash \varphi : \kappa'} \quad (\kappa =_{\beta} \kappa')$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma' : *}{\Gamma \vdash M : \sigma'} \quad (\sigma =_{\beta} \sigma')$$

**Uwaga:** Sprawdzanie typu jest nieelementarne!

(Bo taka jest konwersja  $=_{\beta}$ .)

Ale rozstrzygalne, bo to jest rachunek w stylu Churcha.

# Przykłady

Niech  $\Gamma = \{\tau : *, \alpha : \tau \Rightarrow *, \beta : \tau \Rightarrow *\}$ .

- ▶ Term  $\lambda x^{\forall z:\tau. \alpha z \rightarrow \beta z} \lambda y^{\forall z:\tau. \alpha z} \lambda z^\tau. xz(yz)$  ma w  $\Gamma$  typ  $(\forall x:\tau. \alpha x \rightarrow \beta x) \rightarrow (\forall x:\tau. \alpha x) \rightarrow \forall x:\tau. \beta x$ .

# Przykłady

Niech  $\Gamma = \{\tau : *, \alpha : \tau \Rightarrow *, \beta : \tau \Rightarrow *\}$ .

- ▶ Term  $\lambda x^{\forall z:\tau. \alpha z \rightarrow \beta z} \lambda y^{\forall z:\tau. \alpha z} \lambda z^\tau. xz(yz)$  ma w  $\Gamma$  typ  $(\forall x:\tau. \alpha x \rightarrow \beta x) \rightarrow (\forall x:\tau. \alpha x) \rightarrow \forall x:\tau. \beta x$ .
- ▶ Term  $\lambda f^{\tau \rightarrow \tau} \lambda y^{\forall x:\tau. \alpha x \rightarrow \alpha(fx)} \lambda x^\tau \lambda z^{\alpha x}. y(fx)(yxz)$  ma typ  $\forall f:\tau \rightarrow \tau. (\forall x:\tau. \alpha x \rightarrow \alpha(fx)) \rightarrow \forall x:\tau. \alpha x \rightarrow \alpha(f^2(x))$ .



# Przykłady

Niech  $\Gamma = \{\tau : *, \alpha : \tau \Rightarrow *, \beta : \tau \Rightarrow *\}$ .

- ▶ Term  $\lambda x^{\forall z:\tau. \alpha z \rightarrow \beta z} \lambda y^{\forall z:\tau. \alpha z} \lambda z^\tau. xz(yz)$  ma w  $\Gamma$  typ  $(\forall x:\tau. \alpha x \rightarrow \beta x) \rightarrow (\forall x:\tau. \alpha x) \rightarrow \forall x:\tau. \beta x$ .
- ▶ Term  $\lambda f^{\tau \rightarrow \tau} \lambda y^{\forall x:\tau. \alpha x \rightarrow \alpha(fx)} \lambda x^\tau \lambda z^{\alpha x}. y(fx)(yxz)$  ma typ  $\forall f:\tau \rightarrow \tau. (\forall x:\tau. \alpha x \rightarrow \alpha(fx)) \rightarrow \forall x:\tau. \alpha x \rightarrow \alpha(f^2(x))$ .
- ▶ Term  $\lambda y^\tau \lambda f^{\Pi_{x:\tau} \alpha(x)} \lambda g^{\alpha(y) \rightarrow \tau}. f(g(fy))$  ma typ  $\forall y^\tau \forall f^{\Pi_{x:\tau} \alpha(x)} \forall g^{\alpha(y) \rightarrow \tau}. \alpha(g(f(y)))$ .

# Wzajemne uwikłanie

- ▶ Rodzaje nie występują w termach, konstruktorach i typach, ale:
- ▶ W termach i rodzajach mogą występować konstruktory i typy;
- ▶ W rodzajach, konstruktorach i typach mogą występować termy.

Na przykład w rodzaju  $(y : \tau) \Rightarrow \alpha(\lambda z^{\beta(y)} z) \Rightarrow *$  jest term  $\lambda z^{\beta(y)} z$ , a w nim typ  $\beta(y)$ .

# Poprawność

## Lemma

*Jeśli  $\Gamma \vdash \varphi : \kappa$ , to  $\Gamma \vdash \kappa : \square$ , a jeśli  $\Gamma \vdash M : \tau$  to  $\Gamma \vdash \tau : *$ .*

## Theorem

- ▶ *Jeśli  $\Gamma \vdash M : \sigma$  oraz  $M \rightarrow_{\beta} M'$ , to  $\Gamma \vdash M' : \sigma$ .*
- ▶ *Jeśli  $\Gamma \vdash \varphi : \kappa$  oraz  $\varphi \rightarrow_{\beta} \varphi'$ , to  $\Gamma \vdash \varphi' : \kappa$ .*
- ▶ *Jeśli  $\Gamma \vdash \kappa : \square$  oraz  $\kappa \rightarrow_{\beta} \kappa'$ , to  $\Gamma \vdash \kappa' : \square$ .*

# Wycieranie zależności

Wycieranie zależności z konstruktorów i typów:

- ▶  $\overline{\alpha} = \alpha$ ;
- ▶  $\overline{(\forall x : \tau)\sigma} = \overline{\tau} \rightarrow \overline{\sigma}$ ;
- ▶  $\overline{\varphi M} = \overline{\varphi}$ .

Wynikiem jest zawsze typ prosty.

# Wycieranie zależności

Wycieranie zależności z termów:

- ▶  $\bar{x} = x$ ;
- ▶  $\overline{MN} = \overline{M} \overline{N}$ ;
- ▶  $\overline{\lambda_{x:\tau}.M} = \lambda_{x:\bar{\tau}}.\overline{M}$ .

Wycieranie zależności z otoczenia:

- ▶  $\bar{\Gamma} = \{(x : \bar{\tau}) \mid (x : \tau) \in \Gamma\}$ .

# Wycieranie zależności

Wycieranie zależności z rodzajów (na razie niepotrzebne):

- ▶  $\bar{*} = 0$ ;
- ▶  $\overline{\Pi(x : \tau)\kappa} = \bar{\tau} \rightarrow \bar{\kappa}$ ;

Wynikiem jest typ prosty.

## Wycieranie zależności (term : typ)

**Fakt:** Jeśli  $\Gamma \vdash M : \tau$ , to  $\bar{\Gamma} \vdash \bar{M} : \bar{\tau}$

Na przykład:

- ▶ Term  $\lambda x^{\forall z:\tau. \alpha z \rightarrow \beta z} \lambda y^{\forall z:\tau. \alpha z} \lambda z^\tau. xz(yz)$  ma typ  $(\forall x:\tau. \alpha x \rightarrow \beta x) \rightarrow (\forall x:\tau. \alpha x) \rightarrow \forall x:\tau. \beta x$ .

# Wycieranie zależności (term : typ)

**Fakt:** Jeśli  $\Gamma \vdash M : \tau$ , to  $\bar{\Gamma} \vdash \bar{M} : \bar{\tau}$

Na przykład:

- ▶ Term  $\lambda x^{\forall z:\tau. \alpha z \rightarrow \beta z} \lambda y^{\forall z:\tau. \alpha z} \lambda z^\tau. xz(yz)$  ma typ  $(\forall x:\tau. \alpha x \rightarrow \beta x) \rightarrow (\forall x:\tau. \alpha x) \rightarrow \forall x:\tau. \beta x$ .
- ▶ Jego wytarciem jest  $\mathbf{S} = \lambda x^{\bar{\tau} \rightarrow \bar{\alpha} \rightarrow \bar{\beta}} \lambda y^{\bar{\tau} \rightarrow \bar{\alpha}} \lambda z^{\bar{\tau}}. xz(yz)$  typu  $(\bar{\tau} \rightarrow \bar{\alpha} \rightarrow \bar{\beta}) \rightarrow (\bar{\tau} \rightarrow \bar{\alpha}) \rightarrow \bar{\tau} \rightarrow \bar{\beta}$



# Wycieranie typów

## Wycieranie typów

- ▶  $|x| = x$ ;
- ▶  $|\lambda x:\tau.M| = \lambda x |M|$ ;
- ▶  $|MN| = |M||N|$ .

**Lemat:**  $|\overline{M}| = |M|$ .

# System $\lambda P$ à la Curry

**Def:** Beztypowy term  $M$  jest *typowalny* w  $\lambda P$ , gdy istnieją takie  $\Gamma$ ,  $N$ ,  $\tau$ , że  $\Gamma \vdash_{\lambda P} N : \tau$  oraz  $|N| = M$ .

**Fakt:** Term  $M$  jest typowalny w  $\lambda P$ , wtedy i tylko wtedy, gdy jest typowalny w typach prostych.

**Dowód:** Jeśli  $\Gamma \vdash_{\lambda P} N : \tau$ , to  $\bar{\Gamma} \vdash \bar{N} : \bar{\tau}$  w typach prostych w stylu Churcha. Zatem  $\bar{\Gamma} \vdash |\bar{N}| : \bar{\tau}$  czyli  $\bar{\Gamma} \vdash |N| : \bar{\tau}$ .

# Sprawdzanie typu à la Curry

**Fakt 1:** Term  $M$  jest typowalny w  $\lambda P$ , wtedy i tylko wtedy, gdy jest typowalny w typach prostych.

**Fakt 2:** Następujący problem jest nierozstrzygalny:

Dany beztypowy term  $M$  oraz  $\Gamma$  i  $\tau$ .

Czy istnieje taki term  $N$ , że  $\Gamma \vdash_{\lambda P} N : \tau$  oraz  $|N| = M$ ?

## Problem inhabitacji dla $\lambda P$

**Fakt:** Następujący problem jest nierozstrzygalny:

*Dane  $\Gamma$  i  $\tau$ . Czy istnieje taki term  $N$ , że  $\Gamma \vdash_{\lambda P} N : \tau$ ?*

# Problem inhabitacji dla $\lambda P$

**Fakt:** Następujący problem jest nierozstrzygalny:

*Dane  $\Gamma$  i  $\tau$ . Czy istnieje taki term  $N$ , że  $\Gamma \vdash_{\lambda P} N : \tau$ ?*

**Dowód:** W systemie  $\lambda P$  mieści się intuicjonistyczna logika pierwszego rzędu (dla  $\rightarrow$  i  $\forall$ , ale to wystarczy).

# Silna normalizacja

**Twierdzenie:** *System  $\lambda P$  ma własność silnej normalizacji.*

**Dowód:** Zrobimy redukcję do typów prostych.

Na razie mamy wycieranie zależności.

Jeśli  $\Gamma \vdash M : \tau$ , to  $\bar{\Gamma} \vdash \bar{M} : \bar{\tau}$

ale to za mało. Zamiast  $\bar{M}$  zrobimy  $M^\bullet$ .

# Translacja •

Termy  $\lambda P$  przerabiamy na termy Churcha w typach prostych.

- ▶  $x^\bullet = x$ ;
- ▶  $(MN)^\bullet = M^\bullet N^\bullet$ ;
- ▶  $(\lambda x:\tau. M)^\bullet = (\lambda z^0 \lambda x^{\bar{\tau}}. M^\bullet)_{\tau^\bullet}$ .

## O co tu chodzi?

Przy tej translacji nie gubią się redexy wewnątrz typu  $\tau$ .

# Translacja •

Konstruktory i typy też przerabiamy na termy:

- ▶  $\alpha^\bullet = x_\alpha$ , gdzie  $x_\alpha$  jest nową zmienną;
- ▶  $(\varphi M)^\bullet = \varphi^\bullet M^\bullet$ ;
- ▶  $(\lambda x:\tau. \varphi)^\bullet = (\lambda z^0 \lambda x^{\bar{\tau}}. \varphi^\bullet) \tau^\bullet$ ;
- ▶  $(\forall x:\tau. \sigma)^\bullet = y \tau^\bullet (\lambda x^{\bar{\tau}}. \sigma^\bullet)$ ,  
gdzie  $y : 0 \rightarrow (\bar{\tau} \rightarrow 0) \rightarrow 0$  jest nową zmienną.



# Translacja •

Dla dowolnego  $\Gamma$  definiujemy:

$$\begin{aligned}\bar{\Gamma}^+ = & \{x : \bar{\tau} \mid (x : \tau) \in \Gamma\} \cup \\ & \cup \{x_\alpha : \bar{\kappa} \mid (\alpha : \kappa) \in \Gamma\} \cup \\ & \cup \{y_\rho : 0 \rightarrow (\rho \rightarrow 0) \rightarrow 0 \mid \rho \text{ typ prosty}\}.\end{aligned}$$

**Fakt 1:** Jeśli  $\Gamma \vdash \varphi : \kappa$ , to  $\bar{\Gamma}^+ \vdash \varphi^\bullet : \bar{\kappa}$  w typach prostych.

**Fakt 2:** Jeśli  $\Gamma \vdash M : \tau$ , to  $\bar{\Gamma}^+ \vdash M^\bullet : \bar{\tau}$  w typach prostych.

# Silna normalizacja

**Fakt 3:** If  $M_1 \rightarrow_\beta M_2$  then  $M_1^\bullet \rightarrow_\beta^+ M_2^\bullet$ .

**Fakt 4:** Jeśli  $\varphi_1 \rightarrow_\beta \varphi_2$  to  $\varphi_1^\bullet \rightarrow_\beta^+ \varphi_2^\bullet$ .

**Twierdzenie:** System  $\lambda P$  ma własność silnej normalizacji.

**Dowód:** Nieskończona redukcja w  $\lambda P$

$$M_0 \rightarrow_\beta M_1 \rightarrow_\beta M_2 \rightarrow_\beta \dots$$

tłumaczy się na nieskończoną redukcję w  $\lambda_{\rightarrow}$ :

$$M_0^\bullet \rightarrow_\beta^+ M_1^\bullet \rightarrow_\beta^+ M_2^\bullet \rightarrow_\beta^+ \dots$$

# Rachunek konstrukcji

# Dependencies in lambda-calculi

**The origin:** Objects (terms) depend on terms:

$$\lambda x:\tau M^\sigma : \tau \rightarrow \sigma$$

# Dependencies in lambda-calculi

**The origin:** Objects (terms) depend on terms:

$$\lambda x:\tau M^\sigma : \tau \rightarrow \sigma$$

Pattern:  $(x : \tau^*) \rightarrow \sigma^* : *$  and rule:  $(*, *, *)$ .

(Arrow is a *constructor* of kind  $* \Rightarrow * \Rightarrow *$ .)

# Dependencies in lambda-calculi

**The origin:** Objects (terms) depend on terms:

$$\lambda x:\tau M^\sigma : \tau \rightarrow \sigma$$

Pattern:  $(x : \tau^*) \rightarrow \sigma^* : *$  and rule:  $(*, *, *)$ .

(Arrow is a *constructor* of kind  $* \Rightarrow * \Rightarrow *$ .)

**Polymorphism:** Objects depend on types (system **F**):

$$\Lambda \alpha : * M^\sigma : \forall \alpha \sigma$$

Pattern:  $(\alpha : *^{\square}) \Rightarrow \sigma^* : *$  and rule:  $(\square, *, *)$ .

# Dependencies in lambda-calculi

**Dependent types:** Types depend on objects (system  $\lambda\mathbf{P}$ ):

$$\lambda x:\tau \sigma(x)^* : \tau \Rightarrow *$$

Pattern:  $(x : \tau^*) \Rightarrow *^{\square} : \square$       and rule:  $(*, \square, \square)$

# Dependencies in lambda-calculi

**Dependent types:** Types depend on objects (system  $\lambda\mathbf{P}$ ):

$$\lambda x:\tau \sigma(x)^* : \tau \Rightarrow *$$

Pattern:  $(x : \tau^*) \Rightarrow *^{\square} : \square$  and rule:  $(*, \square, \square)$

**What is missing?**



# Dependencies in lambda-calculi

**Dependent types:** Types depend on objects (system  $\lambda\mathbf{P}$ ):

$$\lambda x:\tau \sigma(x)^* : \tau \Rightarrow *$$

Pattern:  $(x : \tau^*) \Rightarrow *^{\square} : \square$  and rule:  $(*, \square, \square)$

**What is missing?** Types depend on types (system  $\lambda\omega$ )

$$\lambda \alpha : * \sigma(\alpha) : * \Rightarrow *$$

Pattern:  $(\alpha : *^{\square}) \Rightarrow *^{\square} : \square$  and rule:  $(\square, \square, \square)$ .

## Polimorfizm wyższego rzędu: system $F_\omega$

**Example:** Why is  $(\lambda zy. y(z\mathbf{I})(z\mathbf{K}))(\lambda x. xx)$  untypable in  $\mathbf{F}$ ?

Because types we can assign to  $\mathbf{I}$  and  $\mathbf{K}$  differ too much:

$$\mathbf{I} : \forall p. p \rightarrow p \qquad \mathbf{K} : \forall p. p \rightarrow (q \rightarrow p).$$

There is no common pattern for these two types...

## Polimorfizm wyższego rzędu: system $F_\omega$

**Example:** Why is  $(\lambda zy. y(zI)(zK))(\lambda x. xx)$  untypable in  $F$ ?

Because types we can assign to  $I$  and  $K$  differ too much:

$$I : \forall p. p \rightarrow p \qquad K : \forall p. p \rightarrow (q \rightarrow p).$$

There is no common pattern for these two types...  
unless we write it this way:

$$I, K : \forall p. p \rightarrow \alpha(p).$$

# Polimorfizm wyższego rzędu: system $F_\omega$

**Example:** Why is  $(\lambda zy. y(z\mathbf{I})(z\mathbf{K}))(\lambda x. xx)$  untypable in  $\mathbf{F}$ ?

Because types we can assign to  $\mathbf{I}$  and  $\mathbf{K}$  differ too much:

$$\mathbf{I} : \forall p. p \rightarrow p \qquad \mathbf{K} : \forall p. p \rightarrow (q \rightarrow p).$$

There is no common pattern for these two types...  
unless we write it this way:

$$\mathbf{I}, \mathbf{K} : \forall p. p \rightarrow \alpha(p).$$

Here,  $\alpha$  is a constructor of kind  $* \Rightarrow *$ .

## Curry-style $F_\omega$

Let  $\alpha : * \Rightarrow *$ ,  $\varphi : (* \Rightarrow *) \Rightarrow (* \Rightarrow *)$ . Then:

$$\mathbf{K} : \forall p (p \rightarrow q \rightarrow p)$$

$$\mathbf{2} : \forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow \alpha(\alpha(p))))$$

$$\mathbf{2} : [\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow (\alpha(\alpha(p)))))] \rightarrow \\ [\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow (\alpha(\alpha(\alpha(\alpha(p)))))))]$$

$$\text{Therefore } \mathbf{22K} : \forall p (p \rightarrow q \rightarrow q \rightarrow q \rightarrow q \rightarrow p)$$

## Curry-style $F_\omega$

Let  $\alpha : * \Rightarrow *$ ,  $\varphi : (* \Rightarrow *) \Rightarrow (* \Rightarrow *)$ . Then:

$$\mathbf{K} : \forall p (p \rightarrow q \rightarrow p)$$

$$\mathbf{2} : \forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow \alpha(\alpha(p))))$$

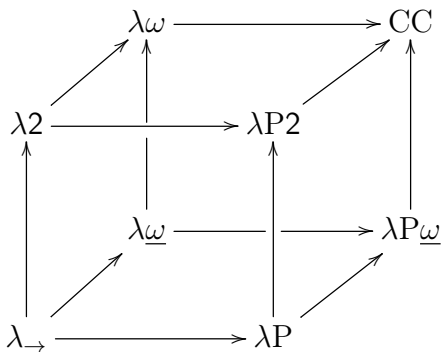
$$\mathbf{2} : [\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow (\alpha(\alpha(p)))))] \rightarrow \\ [\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow (\alpha(\alpha(\alpha(p)))))))]$$

$$\text{Therefore } \mathbf{22K} : \forall p (p \rightarrow q \rightarrow q \rightarrow q \rightarrow q \rightarrow p)$$

$$\mathbf{2} : \forall \varphi ([\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow \varphi(\alpha)(p))] \rightarrow \\ [\forall \alpha (\forall p (p \rightarrow \alpha(p)) \rightarrow \forall p (p \rightarrow (\varphi(\varphi(\alpha))(p))))])$$

$$\text{Therefore } \mathbf{222K} : \forall p (p \rightarrow q \rightarrow q \rightarrow \dots \rightarrow q \rightarrow q \rightarrow p)$$

# Barendregt's Cube: combine all dependencies



$\lambda_{\rightarrow}$  to typy proste,  $\lambda 2$  to system **F**,  $\lambda \omega$  to system **F <sub>$\omega$</sub>** .  
**CC** to rachunek konstrukcji

# Calculus of Constructions: Raw syntax

## Sorts:

$s ::= * \mid \square;$

## Kinds:

$\kappa ::= * \mid (\Pi x:\phi \kappa) \mid (\Pi x:\kappa \kappa);$

## Constructors:

$\phi ::= \alpha \mid (\forall x:\phi \phi) \mid (\forall \alpha:\kappa \phi) \mid (\phi M) \mid$   
 $(\phi\phi) \mid (\lambda x:\phi \phi) \mid (\lambda \alpha:\kappa \phi);$

## Terms:

$M ::= x \mid (MM) \mid (M\phi) \mid (\lambda x:\phi M) \mid (\lambda \alpha:\kappa M);$

## Environments:

$\Gamma ::= \emptyset \mid \Gamma, (x : \phi) \mid \Gamma, (\alpha : \kappa).$



# Rachunek konstrukcji: składnia uniwersalna

**Sorty:**  $s ::= * \mid \square$

**Pseudo-termny:**

$A ::= x \mid s \mid (AA) \mid (\lambda x:A A) \mid (\prod x:A A)$

**Redukcja:**

$(\lambda x:A. B)C \rightarrow_{\beta} B[x := C].$

## Rachunek konstrukcji: reguły

$$\text{(Ax)} \quad \emptyset \vdash * : \square$$

$$\text{(Var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A} \quad (x \notin \text{Dom}(\Gamma))$$

$$\text{(Prod)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\prod x:A. B) : s_2} \quad (s_1, s_2 \in \{*, \square\})$$

## Rachunek konstrukcji: reguły

$$\text{(Prod)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\Pi x:A. B) : s_2} \quad (s_1, s_2 \in \{*, \square\})$$

$$\text{(Abs)} \quad \frac{\Gamma, x:A \vdash B : C \quad \Gamma \vdash (\Pi x:A. C) : s}{\Gamma \vdash (\lambda x:A. B) : (\Pi x:A. C)}$$

$$\text{(App)} \quad \frac{\Gamma \vdash A : (\Pi x:B. C) \quad \Gamma \vdash D : B}{\Gamma \vdash (AD) : C[x := D]}$$

## Rachunek konstrukcji: reguły

$$\text{(Weak)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad (x \notin \text{Dom}(\Gamma))$$

$$\text{(Conv)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad (B =_{\beta} B')$$

# Rachunek konstrukcji: własności

- ▶ Poprawność redukcji:  
Jeśli  $\Gamma \vdash A : B$  i  $A \rightarrow A'$  to  $\Gamma \vdash A' : B$ .
- ▶ “Wzmacnianie”:  
Jeśli  $\Gamma, x : A, \Delta \vdash A : B$ ,  
oraz  $x$  nie jest wolne w  $\Delta, A$  ani w  $B$ ,  
to  $\Gamma, \Delta \vdash A : B$ .
- ▶ Jednoznaczność typu:  
Jeśli  $\Gamma \vdash A : B$  i  $\Gamma \vdash A : B'$ , to  $B =_{\beta} B'$ .
- ▶ Silna normalizacja: Jeśli  $\Gamma \vdash A : B$ , to  $A \in SN$ .

# Klasyfikacja w rachunku konstrukcji

## Fakt:

Jeśli  $\Gamma \vdash A : B$  to zachodzi dokładnie jeden przypadek:

- albo  $\Gamma \vdash B : *$  (czyli  $A$  jest obiektem typu  $B$ ),
- albo  $\Gamma \vdash B : \square$  (czyli  $A$  jest konstruktorem rodzaju  $B$ ),
- albo  $B = \square$  (czyli  $A$  jest rodzajem).

# Rachunek konstrukcji: podsumowanie

▶ Dwa *sorty*:  $*$  oraz  $\square$ ;

▶ Aksjomat  $*$  :  $\square$ ;

▶ Dozwolone są produkty postaci

$$(x : A^s) \rightarrow B(x)^t : t,$$

gdzie  $s$  i  $t$  są dowolnymi sortami;

▶ Te produkty są typami abstrakcji  $\lambda x:A^s. M^{B(x)}$ ;

▶ Wierzchołki kostki: ograniczenie w tworzeniu produktów.