43. Práce v lambda kalkulu (demonstrace reprezentace čísel a pravdivostních hodnot a operací nad nimi).

Odkaz na video: SZZ: Lambda kalkul - definice pravdivostních hodnot a přirozených čísel

Video pokrývá témata: *Definice pravdivostních hodnot a přirozených čísel v lambda kalkulu. Pravdivostní hodnoty. Logické spojky. Ternární operátor. Komplikovanější definice pravdivostních hodnot a logických spojek. Poťouchlejší definice pravdivostních hodnot a logických spojek. Přirozená čísla. Operace následníka. Test na nulu. Operace předchůdce. Uspořádané dvojice. Sčítání. Odčítání. Násobení. Rekurze a pevný bod. Operátor pevného bodu. Dno. Rekurze - obecný návod a násobení. Rekurze - celočíselné dělení.*

Reprezentace pravdivostních hodnot v lambda kalkulu

Pomocí lambda výrazů je možné nadefinovat pravdivostní hodnoty **T** (*true*) a **F** (*false*) mnoha různými způsoby. Ukážeme si jich několik.

- **LET T = λ x y . x**
  - pravdivostní hodnotu *true* definujeme jako lambda abstrakci o dvou proměnných, která **vrací první argument a druhý zahazuje**
- **LET F = λ x y . y**
  - pravdivostní hodnotu *false* definujeme jako lambda abstrakci o dvou proměnných, která **vrací druhý argument a první zahazuje**

Pozn.: Občas zaznívají dotazy, proč definujeme pravdivostní hodnoty právě takto, jaký má zrovna tato definice vztah k fenoménům pravdy a nepravdy. Je to ale totéž jako se ptát, proč abstraktní ideje pravdy a nepravdy zastupujeme právě shlukem písmen "pravda" *a* "nepravda" *a ne nějakým jiným. Vztah mezi označovaným (koncept nepravdy) a označujícím (lambda výraz* (λ x y . y)*) je zde naprosto arbitrární. Podstatné je zde pro nás pouze to, aby v souladu s těmito definicemi bylo dále možné rozumně definovat operace pracující s pravdivostními hodnotami, ternární operátor apod.*

Výrokové spojky. Pomocí výše uvedených definic lze definovat libovolnou z 16 binárních výrokových spojek a unární negaci. Budeme se snažit splnit následující požadavky:

- výrokovou spojku definujeme jako **lambda abstrakci** přijímající očekávaný počet argumentů dle její arity
- počítáme s tím, že tato lambda abstrakce bude na vstup přijímat **libovolnou kombinaci pravdivostních hodnot** (tzn. v libovolném pořadí je tato lambda abstrakce aplikována na libovolné pravdivostní hodnoty, přičemž pomocí beta redukcí bude možné spojku pro tyto vstupy vyhodnotit)
- tato lambda abstrakce by měla fungovat tak, že pro dané vstupy **správně vyhodnotí danou výrokovou spojku** dle její definice ve výrokové logice
- výsledkem bude vždy opět lambda výraz odpovídající **T**, nebo **F** dle naší definice

Při konstrukci logické spojky v lambda kalkulu musíme počítat s tím, jak máme nadefinované hodnoty T a F. V našem případě T vrací první argument a druhý zahazuje, F vrací druhý argument a první zahazuje, tedy např. při konstrukci spojky AND *(konjunkce)* můžeme uvažovat takto:

- spojku AND definujeme jako LET AND = λ a b . a X Y, kde X Y jsou dosud neznámé lambda výrazy, které budeme hledat
- pokud **a = T**, pak:
  - v těle lambda abstrakce získáme **T X Y = (λ x y . x) X Y**
  - po provedení beta redukcí **(λ x y . x) X Y** $\rightarrow_\beta$ **(λ y . X) Y** $\rightarrow_\beta$ **X**
- pokud **a = F**, pak:
  - v těle lambda abstrakce získáme **F X Y = (λ x y . y) X Y**
  - po provedení beta redukcí **(λ x y . y) X Y** $\rightarrow_\beta$ **(λ y . y) Y** $\rightarrow_\beta$ **Y**
- pokud tedy **a = T**, pak **AND a b = X** a pokud **a = F**, pak **AND a b = Y**
- **zvolíme tedy X, Y tak, aby tento výsledek vyhovoval očekávanému výsledku spojky AND**
- vyhodnotíme **a AND b** postupně pro **pevně zvolené** hodnoty **a**:
  - **T AND b = b**, tedy **X = b**
  - **F AND b = F**, tedy **Y = F**
- konečně tedy získáváme **LET AND = λ a b . a b F**
  - **pokud je první argument AND zvolen jako T, vracíme druhý argument**
  - **pokud je první argument AND zvolen jako F, vracíme F**

Funkcionalitu si můžeme ověřit:

- **AND T T** = (λ a b . a b F) T T $\rightarrow_\beta$ (λ b . T b F) T $\rightarrow_\beta$ T T F = (λ x y . x) T F $\rightarrow_\beta$ (λ y . T) F $\rightarrow_\beta$ **T**
- **AND T F** = (λ a b . a b F) T F $\rightarrow_\beta$ (λ b . T b F) F $\rightarrow_\beta$ T F F = (λ x y . x) F F $\rightarrow_\beta$ (λ y . F) F $\rightarrow_\beta$ **F**
- **AND F T** = (λ a b . a b F) F T $\rightarrow_\beta$ (λ b . F b F) T $\rightarrow_\beta$ F T F = (λ x y . y) T F $\rightarrow_\beta$ (λ y . y) F $\rightarrow_\beta$ **F**

- **AND F F** = $(\lambda\ a\ b\ .\ a\ b\ F)\ F\ F \rightarrow_\beta (\lambda\ b\ .\ F\ b\ F)\ F \rightarrow_\beta F\ F\ F = (\lambda\ x\ y\ .\ y)\ F\ F \rightarrow_\beta (\lambda\ y\ .\ y)\ F \rightarrow_\beta$ **F**

Podobně nadefinujeme další spojky:

- **OR** *(disjunkce)*
  - očekáváme:
    - T OR b = T
    - F OR b = b
  - **LET OR = λ a b . a T b**
    - *všimněme si, že tato definice disjunkce není v normální formě, lze provést η-redukci a obdržet* **λ a b . a T b** = λ a . (λ b . a T b) $\rightarrow_\eta$ **λ a . a T**. *Ověřte si, že disjunkce bude fungovat i takto jako lambda abstrakce s jednou proměnnou, ovšem pro přehlednost zde tuto spojku necháme ve tvaru lambda abstrakce se dvěma proměnnými*
- **NOT** *(negace)*
  - očekáváme:
    - NOT T = F
    - NOT F = T
  - **LET NOT = λ a . a F T**
- **XOR** *(exkluzivní součet)*
  - očekáváme:
    - T XOR b = NOT b
    - F XOR b = b
  - **LET XOR = λ a b . a (NOT b) b**
- **EQ** *(ekvivalence)*
  - očekáváme:
    - T EQ b = b
    - F EQ b = NOT b
  - **LET EQ = λ a b . a b (NOT b)**
- ⇒ *(implikace)*
  - T ⇒ b = b
  - F ⇒ b = T
  - **LET ⇒ = λ a b . a b T**
- ⇐ *(zpětná implikace)*
  - T ⇐ b = T
  - F ⇐ b = NOT b
  - **LET ⇐ = λ a b . a T (NOT b)**
- **NAND** *(negace konjunkce)*

    - T NAND b = NOT b
    - F NAND b = T
    - **LET NAND = λ a b . a (NOT b) T**
- **NOR** *(negace disjunkce)*
    - T NOR b = F
    - F NOR b = NOT b
    - **LET NOR = λ a b . a F (NOT b)**
- *podobně by bylo možné definovat i zbylých 8 binárních logických spojek (negace implikace, negace zpětné implikace, identita prvního a druhého argumentu, negace identity prvního a druhého argumentu, tautologie, kontradikce), vhodné na procvičení*

Ternární operátor. Ternární operátor **(?:)** definujeme jako lambda abstrakci o třech proměnných. V případě první z nich očekáváme pravdivostní hodnotu. Pokud je tato pravdivostní hodnota T, vracíme druhý argument, jinak třetí argument

- **(?:)**
    - **LET (?:) = λ a b c . a b c**
        - **(?:) T A B** = (λ a b c . a b c) T A B $\rightarrow_\beta$ (λ b c . T b c) A B $\rightarrow_\beta$ (λ c . T A c) B $\rightarrow_\beta$ T A B = (λ x y . x) A B $\rightarrow_\beta$ (λ y . A) B $\rightarrow_\beta$ **A**
        - **(?:) F A B** = (λ a b c . a b c) F A B $\rightarrow_\beta$ (λ b c . F b c) A B $\rightarrow_\beta$ (λ c . F A c) B $\rightarrow_\beta$ F A B = (λ x y . y) A B $\rightarrow_\beta$ (λ y . y) B $\rightarrow_\beta$ **B**
        - *všimněme si, že tento lambda výraz zdaleka není v normální formě, lze provést až dvě eta redukce* **λ a b c . a b c** $\rightarrow_\eta$ λ a b . a b $\rightarrow_\eta$ **λ a . a**, *přičemž ternární operátor při této definici T, F lze zavést i takto jako obyčejnou identitu jednoho argumentu*
            - (λ a . a) T A B $\rightarrow_\beta$ T A B = (λ x y . x) A B $\rightarrow_\beta$ (λ y . A) B $\rightarrow_\beta$ A
            - (λ a . a) F A B $\rightarrow_\beta$ F A B = (λ x y . y) A B $\rightarrow_\beta$ (λ y . y) B $\rightarrow_\beta$ B
        - *pro jednoduchost budeme místo* ***(?:) x y z*** *psát* ***x ? y : z***

Exotičtější definice pravdivostních hodnot v lambda kalkulu

Ukažme několik další definic pravdivostních hodnot:

- **LET T = λ x y . x y**
    - pravdivostní hodnotu *true* definujeme jako lambda abstrakci o dvou proměnných, která **ponechává argumenty v původním pořadí**
    - *tuto definici ze upravit pomocí η-redukce λ x y . x y $\rightarrow_\eta$ λ x . x, ovšem pro přehlednost budeme používat se dvěma argumenty*
- **LET F = λ x y . y x**

- pravdivostní hodnotu *false* definujeme jako lambda abstrakci o dvou proměnných, která **prohazuje pořadí těchto argumentů**

Na tyto definice klademe podobné podmínky jako na předchozí definice. K definici logických spojek ale musíme přistupovat odlišně, což si ukážeme na příkladu spojky AND:

- spojku AND definujeme jako LET AND = λ a b . a X Y, kde X Y jsou dosud neznámé lambda výrazy
- pokud **a = T**, pak **AND a b = X Y** a pokud **a = F**, pak **AND a b = Y X**
- my ovšem chceme, aby výsledkem byla jediná pravdivostní hodnota, nikoliv tato aplikace
- **X, Y** můžeme zadefinovat jako lambda abstrakce s jedním argumentem, který se vůbec nevystykuje v těle, tzn. **X = (λ s . X'), Y = (λ s . Y')**
- potom **X Y = (λ s . X') (λ s . Y') →β X'** a **Y X = (λ s . Y') (λ s . X') →β Y'**, tedy pomocí této přebytečné proměnné budeme moci zahodit vždy druhý z lambda výrazů v aplikacích **X Y** a **Y X**, přičemž současně obnažíme těla **X'**, **Y'** těchto lambda abstrakcí
- **zvolíme tedy X', Y' tak, aby tento výsledek vyhovoval očekávanému výsledku spojky AND**
- vyhodnotíme **a AND b** postupně pro **pevně zvolené** hodnoty **a**:
  - **T AND b = b**, tedy **X' = b**
  - **F AND b = F**, tedy **Y' = F**
- konečně tedy získáváme **LET AND = λ a b . a (λ s . b) (λ t . F)**
  - **pokud je první argument AND zvolen jako T, neprohodíme následující dva lambda výrazy, přičemž po další beta redukci zůstane pouze tělo prvního z nich, což bude b**
  - **pokud je první argument AND zvolen jako F, prohodíme následující dva lambda výrazy, přičemž po další beta redukci zůstane pouze tělo prvního z nich, což bude F**

Funkcionalitu si můžeme ověřit:

- **AND T T** = (λ a b . a (λ s . b) (λ t . F)) T T →β (λ b . T (λ s . b) (λ t . F)) T →β T (λ s . T) (λ t . F) = (λ x y . x y) (λ s . T) (λ t . F) →β (λ y . (λ s . T) y) (λ t . F) →β (λ s . T) (λ t . F) →β **T**
- **AND T F** = (λ a b . a (λ s . b) (λ t . F)) T F →β (λ b . T (λ s . b) (λ t . F)) F →β T (λ s . F) (λ t . F) = (λ x y . x y) (λ s . F) (λ t . F) →β (λ y . (λ s . F) y) (λ t . F) →β (λ s . F) (λ t . F) →β **F**
- **AND F T** = (λ a b . a (λ s . b) (λ t . F)) F T →β (λ b . F (λ s . b) (λ t . F)) T →β F (λ s . T) (λ t . F) = (λ x y . y x) (λ s . T) (λ t . F) →β (λ y . y (λ s . T)) (λ t . F) →β (λ t . F) (λ s . T) →β **F**
- **AND F F** = (λ a b . a (λ s . b) (λ t . F)) F F →β (λ b . F (λ s . b) (λ t . F)) F →β F (λ s . F) (λ t . F) = (λ x y . y x) (λ s . F) (λ t . F) →β (λ y . y (λ s . F)) (λ t . F) →β (λ t . F) (λ s . F) →β **F**

Porovnejme aktuální a předchozí definice AND:

- **LET AND = λ a b . a b F**
  - **předchozí definice**, T a F zahazují jeden z argumentů a druhý ponechávají
- **LET AND = λ a b . a (λ s . b) (λ t . F)**
  - **aktuální definice**, T a F pouze ponechávají pořadí argumentů nebo jej přehazují, je tedy nutné zajistit zahození jednoho z nich ve vlastní režii pomocí lambda abstrakce s nadbytečným parametrem

Další spojky definujeme podobným způsobem:

- LET NOT = λ a . a (λ s . F) (λ t . T)
- LET XOR = λ a b . a (λ s . NOT b) (λ t . b)
- LET EQ = λ a b . a (λ s . b) (λ t . NOT b)
- LET ⇒ = λ a b . a (λ s . b) (λ t . T)
- LET ⇐ = λ a b . a (λ s . T) (λ t . NOT b)
- LET NAND = λ a b . a (λ s . NOT b) (λ t . T)
- LET NOR = λ a b . a (λ s . F) (λ t . NOT b)
- LET (?:) = λ a b c . a (λ s . b) (λ t . c)
- ...

Samozřejmě je možné zavést pravdivostní hodnoty i výrazně divočeji:

- LET T = λ x y . x, LET F = λ x y . y x x
  - pravdivostní hodnotu *true* definujeme jako lambda abstrakci o dvou proměnných, která vrací první argument
  - pravdivostní hodnotu *false* definujeme jako lambda abstrakci o dvou proměnných, která prohazuje pořadí těchto argumentů a zdvojnásobí první argument
  - LET AND = λ a b . a b (λ s t . F)
- LET T = λ x y . x x x, LET F = λ x y . y y y y y y
  - pravdivostní hodnotu *true* definujeme jako lambda abstrakci o dvou proměnných, která ztrojnásobí první argument a druhý zahodí
  - pravdivostní hodnotu *false* definujeme jako lambda abstrakci o dvou proměnných, která zšestinásobí druhý argument a první zahodí
  - LET AND = λ a b . a (λ s_1 s_2 . b) (λ t_1 t_2 t_3 t_4 t_5 . F)
- *možných definic je nekonečně mnoho*

V dalším textu budeme pracovat s elementy T, F a s logickými spojkami, přičemž budeme předpokládat, že jsou vhodně definované a fungují, tzn. odhlédneme od konkrétní definice.

Reprezentace přirozených čísel v lambda kalkulu

Přirozená čísla definujeme podle následujícího schématu:

- **LET 0 = λ z n . n**
  - v těle lambda abstrakce se 0x vyskytuje symbol *z*
  - *z* je 0x aplikováno na *n*
  - *tato definice je α-ekvivalentní s původní definicí nepravdy F = (λ x y . y)*
- **LET 1 = λ z n . z n**
  - v těle lambda abstrakce se 1x vyskytuje symbol *z*
  - *z* je 1x aplikováno na *n*
- **LET 2 = λ z n . z (z n)**
  - zavedeme novou konvenci, při níž takový typ uzávorkování zprava s opakovanou aplikací téhož lambda výrazu můžeme zkrátit pomocí zkratky **z (z n) = z^2 n**
  - v těle lambda abstrakce se 2x vyskytuje symbol *z*
  - *z* je 2x aplikováno na *n*
- **LET 3 = λ z n . z (z (z n))**
  - 3 = λ z n . z^3 n
- **LET N = λ z n . z^N n** *(obecné přirozené číslo)*
  - v těle lambda abstrakce se Nkrát vyskytuje symbol *z*
  - *z* je Nkrát aplikováno na *n*

Elementární práce s čísly. Pro práci s čísly zavádíme funkci následníka S, test na nulu Z, funkci předchůdce P

- **LET S = λ x y m . x y (y m)**
  - mělo by platit S N = (N + 1), tedy aplikace S na N vrátí následníka N, přirozené číslo o 1 vyšší
  - lambda abstrakce *S* má tři parametry, budeme ji ovšem aplikovat na jediné přirozené číslo
  - zbylé dva parametry *y, m* zde poslouží jako hlavička výsledného přirozeného čísla, neboť přirozená čísla jsme definovali jako lambda abstrakce se dvěma parametry
  - po aplikaci *S* na přirozené číslo *N* obdržíme (λ **x** y m . **x** y (y m)) **N** →$_\beta$ λ y m . **N** y (y m)
  - samotné číslo *N* je lambda abstrakce se dvěma parametry, tedy jelikož uvnitř těla uvedené lambda abstrakce vidíme *N y (y m)*, dostane *N* pomocí *β*-redukcí do svého těla lambda výrazy *y, (y m)*, přičemž spotřebuje své parametry a obnaží své tělo
  - ukažme si to na příkladu čísla 2
    - (λ **x** y m . **x** y (y m)) **2** →$_\beta$ λ y m . **2** y (y m) = λ y m . **(λ z n . z (z n))** y (y m)

- lambda výraz *y* spotřebuje parametr *z* výrazu *N = (λ z n . z (z n))*, tedy nahradí výskyty *z* v přirozeném čísle *N* za *y*, čímž přizpůsobí výsledné číslo tomu, že v hlavičce má parametry *y, m*, nikoliv *z, n*, jak jsme definovali výše
  - λ y m . (λ **z** n . **z** (**z** n)) **y** (y m) →β λ y m . (λ n . **y** (**y** n)) (y m)
- lambda výraz *(y m)* spotřebuje parametr *n* z lambda výrazu *(λ n . y (y n))*, přičemž proměnnou *n* uvnitř těla nahradí za *y m*, čímž zvýší počet *y* aplikovaných na *n* o jedno
- dovnitř těla přirozeného čísla vnutíme tímto způsobem jedno nové *y*, čímž zvýšíme jeho hodnotu o 1
  - λ y m . (λ **n** . y (y **n**)) **(y m)** →β λ y m . y (y **(y m)**)
- můžeme provést dvě alfa konverze, abychom dostali lambda abstrakci s parametry *y, n*, která bude identická s výše definovanými přirozenými čísly
  - λ **y** m . **y** (**y** (**y** m)) →α λ **z m** . **z** (**z** (**z m**)) →α λ z **n** . z (z (z **n**)) = 3

- **LET Z = λ p . p (λ s . F) T**
  - mělo by platit Z N = T ⇔ N = 0, tedy funkce Z (test na nulu) vrací T (true) tehdy, pokud je dané vstupní číslo 0
  - jinak funkce vrací F (false)
  - lambda abstrakce *Z* má jeden parametr, budeme ji aplikovat na jediné přirozené číslo, u nějž testujeme, zda jde o nulu
  - po aplikaci *Z* na přirozené číslo *N* obdržíme (λ **p** . **p** (λ s . F) T) **N** →β **N** (λ s . F) T
  - číslo 0 je ekvivalentní s definicí F, tedy z následujících dvou argumentů zahodí první a druhý ponechá, tedy pokud N = 0, pak:
    - (λ **p** . **p** (λ s . F) T) **0** →β **0** (λ s . F) T = **(λ z m . m)** (λ s . F) T →$^2_\beta$ **T**
  - ostatní čísla mají ve svém těle z^N m, tedy vlastně z (z^{N-1} m), což je lambda aplikace dvou lambda výrazů
  - za každý výskyt *z* se po β-redukci dosadí (λ s . F), tzn. obdržíme (λ s . F) ((λ s . F)$^{N-1}$ m)
  - další β-redukce zahodí celou část ((λ s . F)$^{N-1}$ m) a obnaží F v (λ s . F)
  - pokud např. N = 3, pak:
    - (λ **p** . **p** (λ s . F) T) **3** →β **3** (λ s . F) T = (λ z m . z (z (z m))) (λ s . F) T →$^2_\beta$ (λ s . F) ((λ s . F) ((λ s . F) T)) →β **F**

- **LET P = λ x y m . x (λ g h . h (g y)) (λ u . m) (λ u . u)**
  - pokud N > 0, mělo by platit P N = (N - 1), tedy aplikace P na N vrátí předchůdce N, přirozené číslo o 1 nižší
  - pokud N = 0, mělo by platit P N = N, tedy předchůdce 0 je opět 0
  - lambda abstrakce *P* má tři parametry, budeme ji ovšem aplikovat na jediné přirozené číslo
  - zbylé dva parametry *y, m* zde poslouží jako hlavička výsledného přirozeného čísla, neboť přirozená čísla jsme definovali jako lambda abstrakce se dvěma parametry

- po aplikaci *P* na číslo *N* obdržíme (λ **x** y m . **x** (λ g h . h (g y)) (λ u . m) (λ u . u)) **N** $\rightarrow_\beta$ λ y m. **N** (λ g h . h (g y)) (λ u . m) (λ u . u)
- význam jednotlivých částí těla P:
  - **(λ g h . h (g y))**
    - tato lambda abstrakce přijímá dva argumenty
    - druhý argument vloží na začátek svého těla, za první přidá *y*, což je v tomto případě proměnná, jejíž počet ve výsledném čísle určuje hodnotu tohoto čísla
    - pro jednoduchost budeme místo této lambda abstrakce používat zkratku **K**
    - získáme-li během výpočtu tvar *K (K (λ u . m)) y*, po provedení β-redukcí obdržíme *y (K (λ u . m) y)*,
    - používání *K* tedy povede k hromadění proměnných *y* zleva, což odpovídá hodnotě výsledného přirozeného čísla (*y* zde odpovídá *z* v naší definici čísel, *např. 2 = (λ z n . z (z n))*)
    - použití *K* navíc vytvoří novou proměnnou *y*, která bude následně použitím dalšího *K* opět přidána doleva
      - **K** (K (λ u . m)) **y** $\rightarrow_\beta$ **y** (**K** (λ u . m) **y**) $\rightarrow_\beta$ y (**y** ((λ u . m) y))
    - poslední takto vytvořené *y* bude zahozeno pomocí *(λ u . m) y* $\rightarrow_\beta$ *m*, čímž efektivně dojde ke snížení vstupního čísla o 1
  - **(λ u . m)**
    - v těle této lambda abstrakce je *m*, což je parametr výsledném přirozeného čísla odpovídající *n* v původní definici (*např. 2 = (λ z n . z (z n))*)
    - pokud N > 0, tuto lambda abstrakci postupně dostaneme nejhlouběji do těla výsledného přirozeného čísla, kde zahodí jedno *y* (v naší definici přirozených čísel odpovídá *z*), čímž sníží číslo o 1, přičemž obnaží své tělo *m*
    - pokud N = 0, část K bude ihned zahozena a v těle zůstane (λ u . m) (λ u . u) $\rightarrow_\beta$ m
  - **(λ u . u)**
    - pokud N = 0, smyslem této identity bude obnažení m v těle lambda abstrakce (λ u . m), protože provedeme (λ u . m) (λ u . u) $\rightarrow_\beta$ m, tzn. zahozením této identity se zbavíme zbytečného parametru *u* v (λ u . m) a vybudujeme výsledné číslo 0 = λ y m . m
    - pokud N > 0, tato identita bude po použití prvního K umístěna zcela doleva v těle výsledného čísla:
      - λ y m. **K** (K^{N-1} (λ u . m)) **(λ u . u)** $\rightarrow_\beta$ λ y m. **(λ u . u)** (K^{N-1} (λ u . m)) y
    - Tato identita následně zmizí pomocí β-redukce:
      - λ y m. **(λ u . u)** (K^{N-1} (λ u . m)) y $\rightarrow_\beta$ λ y m. (K^{N-1} (λ u . m)) y

- při použití *K* běžně přemístíme zcela doleva proměnnou *y*, čímž budujeme výsledné přirozené číslo, ale při prvním použití *K* v těle dosud není žádné *y*, které by bylo posunuto na začátek, ještě nebylo pomocí žádného *K* vytvořeno, pročež první použití *K* doleva místo neexistujícího *y* umístí tuto identitu (λ u . u), která nic nezkazí, neboť bude ihned spotřebována

- ukázka pro N = 0:
  - P **0** =
  - (λ x y m . x K (λ u . m) (λ u . u)) **0** $\rightarrow_\beta$
  - λ y m . **0** K (λ u . m) (λ u . u) =
  - λ y m . **(λ z n . n)** K (λ u . m) (λ u . u) $\rightarrow_\beta$
  - λ y m . (λ n . n) (λ u . m) (λ u . u) $\rightarrow_\beta$
  - λ y m . (λ u . m) (λ u . u)
  - λ y m . m $\rightarrow^2_\alpha$
  - λ z n . n =
  - 0
- ukázka pro N = 1
  - P **1** =
  - (λ x y m . x K (λ u . m) (λ u . u)) **1** $\rightarrow_\beta$
  - λ y m . **1** K (λ u . m) (λ u . u) =
  - λ y m . **(λ z n . z n)** K (λ u . m) (λ u . u) $\rightarrow_\beta$
  - λ y m . (λ n . K n) (λ u . m) (λ u . u) $\rightarrow_\beta$
  - λ y m . **K** (λ u . m) (λ u . u) =
  - λ y m . **(λ g h . h (g y))** (λ u . m) (λ u . u) $\rightarrow_\beta$
  - λ y m . (λ h . h ((λ u . m) y)) (λ u . u) $\rightarrow_\beta$
  - λ y m . (λ u . u) ((λ u . m) y) $\rightarrow_\beta$
  - λ y m . (λ u . m) y $\rightarrow_\beta$
  - λ y m . m $\rightarrow^2_\alpha$
  - λ z n . n =
  - 0
- ukázka pro N = 4, *nyní bez rozepisování K a s vícenásobnými redukcemi*
  - P **4** =
  - (λ x y m . x K (λ u . m) (λ u . u)) **4** $\rightarrow_\beta$
  - λ y m . **4** K (λ u . m) (λ u . u) =
  - λ y m . **(λ z n . z (z (z (z n))))** K (λ u . m) (λ u . u) $\rightarrow^2_\beta$
  - λ y m . **K** (**K** (**K** (**K** **(λ u . m)**))) **(λ u . u)** $\rightarrow^2_\beta$

- $\lambda\ y\ m\ .\ (\lambda\ u\ .\ u)\ (K\ (K\ (K\ (\lambda\ u\ .\ m)))\ y) \rightarrow_\beta$
- $\lambda\ y\ m\ .\ K\ (K\ (K\ (\lambda\ u\ .\ m)))\ y \rightarrow^2_\beta$
- $\lambda\ y\ m\ .\ y\ (K\ (K\ (\lambda\ u\ .\ m))\ y) \rightarrow^2_\beta$
- $\lambda\ y\ m\ .\ y\ (y\ (K\ (\lambda\ u\ .\ m)\ y)) \rightarrow^2_\beta$
- $\lambda\ y\ m\ .\ y\ (y\ (y\ ((\lambda\ u\ .\ m)\ y))) \rightarrow_\beta$
- $\lambda\ y\ m\ .\ y\ (y\ (y\ m)) \rightarrow^2_\alpha$
- $\lambda\ z\ n\ .\ z\ (z\ (z\ n)) =$
- 3

Uspořádané dvojice a práce s nimi. Definujeme uspořádané dvojice (např. čísel) a funkce **fst**, **snd**, které po řadě vrací první, druhý element uspořádané dvojice.

- **LET (_,_) = λ a b p . p a b**
  - uspořádaná dvojice je lambda abstrakce se třemi proměnnými
  - chceme-li uspořádanou dvojici *naplnit hodnotami*, aplikujeme ji na dvě vybrané hodnoty
    - $(A,B) = (\lambda\ a\ b\ p\ .\ p\ a\ b)\ A\ B \rightarrow_\beta (\lambda\ b\ p\ .\ p\ A\ b)\ B \rightarrow_\beta \lambda\ p\ .\ p\ A\ B$
  - naplněná uspořádaná dvojice v tomto tvaru nám poskytuje určité benefity:
    - hodnoty A, B jsou uvězněné uvnitř těla lambda abstrakce $\lambda\ p\ .\ p\ A\ B$, takže je vizuálně zřejmé, že určitým způsobem patří k sobě
    - zbývající parametr *p* nám umožňuje s uspořádanou dvojicí dále pracovat, tuto dvojici můžeme aplikovat na jinou funkci *C*, která se po provedení β-redukce dostane před hodnoty *A*, *B* a bude možné ji na ně aplikovat
      - $(\lambda\ p\ .\ p\ A\ B)\ C \rightarrow_\beta C\ A\ B$
- **LET fst = λ v . v T**
  - aplikací fst na nějakou uspořádanou dvojici dostaneme tuto dvojici před T, aplikujeme ji na T
  - T se následně dostane do těla uspořádané dvojice a zahodí druhý element dvojice
  - **fst (A, B)** $= (\lambda\ v\ .\ v\ T)\ (A, B) \rightarrow_\beta (A, B)\ T = (\lambda\ p\ .\ p\ A\ B)\ T \rightarrow_\beta T\ A\ B = (\lambda\ x\ y\ .\ x)\ A\ B \rightarrow^2_\beta$ **A**
- **LET snd = λ v . v F**
  - aplikací snd na nějakou uspořádanou dvojici dostaneme tuto dvojici před F, aplikujeme ji na F
  - F se následně dostane do těla uspořádané dvojice a zahodí první element dvojice
  - **snd (A, B)** $= (\lambda\ v\ .\ v\ F)\ (A, B) \rightarrow_\beta (A, B)\ F = (\lambda\ p\ .\ p\ A\ B)\ F \rightarrow_\beta F\ A\ B = (\lambda\ x\ y\ .\ y)\ A\ B \rightarrow^2_\beta$ **B**

Základní aritmetické operace. Definujeme nyní sčítání, odčítání a násobení v naší reprezentaci čísel.

- **LET ADD = λ a b y m . a y (b y m)**
  - mělo by platit ADD A B = (A + B), tedy aplikace ADD na A a B vrátí součet A + B
  - lambda abstrakce *ADD* má čtyři parametry, budeme ji ovšem aplikovat jen na dvě přirozená čísla
  - zbylé dva parametry *y, m* zde poslouží jako hlavička výsledného přirozeného čísla, neboť přirozená čísla jsme definovali jako lambda abstrakce se dvěma parametry
  - po aplikaci *ADD* na přirozená čísl *A, B* obdržíme:
    - $(\lambda\ a\ b\ y\ m\ .\ a\ y\ (b\ y\ m))\ A\ B \rightarrow_\beta (\lambda\ b\ y\ m\ .\ A\ y\ (b\ y\ m))\ B \rightarrow_\beta \lambda\ y\ m\ .\ A\ y\ (B\ y\ m)$
  - samotné číslo *A* je lambda abstrakce se dvěma parametry, tedy jelikož uvnitř těla uvedené lambda abstrakce vidíme *A y (B y m)*, dostane *A* pomocí *β*-redukcí do svého těla lambda výrazy *y, (B y m)*, přičemž spotřebuje své parametry a obnaží své tělo
    - $\lambda\ y\ m\ .\ \mathbf{A}\ y\ (B\ y\ m) = \lambda\ y\ m\ .\ \mathbf{(\lambda\ z\ n\ .\ z^A\ n)}\ y\ (B\ y\ m) \rightarrow^2_\beta \lambda\ y\ m\ .\ y^A\ (B\ y\ m)$
  - podobnou logiku lze použít i v případě lambda výrazu *(B y m)*, tedy obdržíme:
    - $\lambda\ y\ m\ .\ y^A\ \mathbf{(B}\ y\ m) = \lambda\ y\ m\ .\ y^A\ \mathbf{((\lambda\ z\ n\ .\ z^B\ n)}\ y\ m) \rightarrow^2_\beta \lambda\ y\ m\ .\ y^A\ (y^B\ m)$
  - zřejmě $\lambda\ y\ m\ .\ y^A\ (y^B\ m) = \lambda\ y\ m\ .\ y^{A+B}\ m = A + B$
  - ukažme to na příkladu ADD 1 2:
    - **ADD** 1 2 =
    - **(λ a b y m . a y (b y m))** $1\ 2 \rightarrow^2_\beta$
    - $\lambda\ y\ m\ .\ 1\ y\ (2\ y\ m) =$
    - $\lambda\ y\ m\ .\ (\lambda\ z\ n\ .\ z\ n)\ y\ (2\ y\ m) \rightarrow^2_\beta$
    - $\lambda\ y\ m\ .\ y\ (2\ y\ m) =$
    - $\lambda\ y\ m\ .\ y\ ((\lambda\ z\ n\ .\ z\ (z\ n))\ y\ m) \rightarrow^2_\beta$
    - $\lambda\ y\ m\ .\ y\ (y\ (y\ m)) \rightarrow^2_\alpha$
    - $\lambda\ z\ n\ .\ z\ (z\ (z\ n)) =$
    - 3
- **LET SUB = λ a b . b P a**
  - mělo by platit SUB A B = A - B, pokud A >= B, tedy aplikace SUB na A a B vrátí rozdíl A - B
  - mělo by platit SUB A B = 0, pokud A < B, tedy aplikace SUB na A a B vrátí 0
  - při výpočtu SUB A B provedeme Bkrát P A, tedy Bkrát po sobě určíme předchůdce A
  - uvažujme o obecných hodnotách A, B:
    - $\text{SUB A B} = (\lambda\ a\ b\ .\ b\ P\ a)\ A\ B \rightarrow^2_\beta B\ P\ A = (\lambda\ z\ n\ .\ z^B\ n)\ P\ A \rightarrow^2_\beta P^B\ A$

        - tedy *B*krát aplikujeme funkci předchůdce na A
    - ukažme to na příkladu SUB 4 2:
        - **SUB** 4 3 =
        - **(λ a b . b P a)** 4 3 $\rightarrow^2_\beta$
        - 3 P 4 =
        - (λ z n . z (z (z n))) P 4 $\rightarrow^2_\beta$
        - P (P (P 4)) →
        - P (P 3) →
        - P 2 →
        - 1
- **LET MUL = λ a b . a (ADD b) 0**
    - mělo by platit MUL A B = A * B, tedy aplikace MUL na A a B vrátí součin A * B
    - při násobení a * b provádíme *a*krát součet *b* + *b* + *b* + … + *b* (*a* výskytů *b* v součtu)
    - jelikož naše definice přirozených čísel provádějí iterativně nějaký výpočet, můžeme jako základ našeho iterativního výpočtu použít *0* a opakovaně k ní přičítat *b*, tzn. $(ADD\ b)^a\ 0$
    - uvažujme o obecných hodnotách A, B:
        - MUL A B = (λ a b . a (ADD b) 0) A B $\rightarrow^2_\beta$ A (ADD B) 0 = (λ z n . z^A n) (ADD B) 0 $\rightarrow^2_\beta$ $(ADD\ B)^A$ 0
        - tedy aplikujeme funkci *přičtení B* na hodnotu 0 dohromady *A*krát, což odpovídá tomu, co očekáváme od násobení
    - ukažme to na součinu 2 * 3:
        - **MUL** 2 3 =
        - **(λ a b . a (ADD b) 0)** 2 3 $\rightarrow^2_\beta$
        - 2 (ADD 3) 0 =
        - (λ z n . z (z n)) (ADD 3) 0 $\rightarrow^2_\beta$
        - (**ADD** 3) (ADD 3 0) =
        - **(λ a b y m . a y (b y m))** 3 (ADD 3 0) $\rightarrow^2_\beta$
        - λ y m . 3 y ((ADD 3 0) y m) =
        - λ y m . (λ z n . z (z (z n))) y ((ADD 3 0) y m) $\rightarrow^2_\beta$
        - λ y m . y (y (y ((**ADD** 3 0) y m))) =
        - λ y m . y (y (y (((**λ a b y m . a y (b y m)**) 3 0) y m))) $\rightarrow^2_\beta$
        - λ y m . y (y (y ((λ y m . 3 y (0 y m)) y m))) $\rightarrow^2_\beta$
        - λ y m . y (y (y (3 y (0 y m)))) =
        - λ y m . y (y (y ((λ z n . z (z (z n))) y (0 y m)))) $\rightarrow^2_\beta$
        - λ y m . y (y (y (y (y (y (0 y m)))))) =

- λ y m . y (y (y (y (y ((λ z n . n) y m)))))) $\rightarrow^2_\beta$
- λ y m . y (y (y (y (y (y m))))) $\rightarrow^2_\alpha$
- λ z n . z (z (z (z (z n))))

Pevný bod a rekurze v lambda kalkulu

Ať f : A → A je zobrazení. Element a ∈ A nazveme **pevným bodem** zobrazení f, pokud **f(a) = a**.

V lambda kalkulu jsme schopni definovat **operátor pevného bodu Y**:

- **LET Y = λ f . (λ x . f (x x)) (λ x . f (x x))**
  - Y E je pevný bod výrazu E
  - pokud tedy k_E je pevným bodem E, tzn. E k_E = k_E, pak platí:
    - E k_E = k_E = Y E ⇒ Y E = E (Y E)
  - pro operátor pevného bodu by mělo platit **Y E = E (Y E)**
  - to ověříme:
    - **Y E** =
    - (λ f . (λ x . f (x x)) (λ x . f (x x))) E $\rightarrow_\beta$
    - **(λ x . E (x x)) (λ x . E (x x))** $\rightarrow_\beta$
    - E **((λ x . E (x x)) (λ x . E (x x)))** =
    - E (Y E)

Definujme rovněž lambda výraz **dno** (bottom) ⊥:

- LET ⊥ = Y T
  - základní vlastností dna je to, že na výstup produkuje samo sebe nehledě na to, na jaký výraz je aplikováno
  - to ověříme na příkladu:
    - ⊥ ADD (FST (1, 2)) (SUB 2 3) =
    - (Y T) ADD (FST (1, 2)) (SUB 2 3) =
    - (T (Y T)) ADD (FST (1, 2)) (SUB 2 3) $\rightarrow_\beta$
    - (λ y . (Y T)) ADD (FST (1, 2)) (SUB 2 3) $\rightarrow_\beta$
    - (Y T) (FST (1, 2)) (SUB 2 3) =
    - (T (Y T)) (FST (1, 2)) (SUB 2 3) $\rightarrow_\beta$
    - (λ y . (Y T)) (FST (1, 2)) (SUB 2 3) $\rightarrow_\beta$
    - (Y T) (SUB 2 3) =
    - (T (Y T)) (SUB 2 3) $\rightarrow_\beta$

        - (λ y . (Y T)) (SUB 2 3) →β
        - Y T =
        - ⊥
    - dno lze používat jako výstup aritmetických operací v okamžiku, kdy je výstup nedefinován (např. dělení nulou)

Pomocí operátoru pevného bodu a dříve uvedených výpočetních prostředků jsme schopni definovat jakoukoliv parciálně rekurzivní funkci. Můžeme postupovat podle následujícího schématu:

- chceme definovat parciálně rekurzivní funkci, zapíšeme ji rekurzivním způsobem v nějakém programovacím jazyce pouze za použití operací, které jsme dříve definovali, a bez přiřazování do proměnných
- vyhneme se explicitnímu použití smyček *while, for* a podobně, využijeme rekurzi
- můžeme použít i selekci, kterou v lambda kalkulu později převedeme na ternární operátor
- pro přehlednost selekci rozepisujme přísně pomocí if/else, ačkoliv by to šlo napsat elegantněji
    - *def MUL(a, b):*
        - *if(Z(a)):*
            - *return 0*
        - *else:*
            - *return ADD(b, MUL(P(a), b))*
    - zde například násobení vyjadřujeme jako opakované sčítání, při každém rekurzivním zavolání MUL snižujeme *a* o 1 pomocí funkce předchůdce
    - MUL voláme tak dlouho, dokud první parametr není 0, potom při vynoření z rekurze provedeme opakovaný součet
- nyní tento program přepíšeme na lambda výraz definující MUL
- tento lambda výraz bude lambda abstrakcí s daným počtem parametrů, kolik má funkce v uvedeném programu
- větvení nahradíme ternárním operátorem
    - pro jednoduchost a přehlednost místo *(?:) a b c* budeme psát *(a ? b : c)*
- definici nové funkce nejprve zapíšeme neformálně, bude závislá sama na sobě, **což není validní zápis**, je to pouze mezikrok
    - **LET MUL = λ a b . (Z a) ? 0 : (ADD b (MUL (P a) b)**
- této závislosti se zbavíme tak, že celý tento lambda výraz zabalíme do nové lambda abstrakce s jedním novým parametrem, kterým nahradíme výskyty cyklické závislosti na pravé straně rovnítka
- současně před tuto novou lambda abstrakci umístíme operátor pevného bodu Y
    - **LET MUL = Y (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))**
    - MUL 2 2 =
    - (Y (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) 2 2 =
        - *použijeme vlastnost pevného bodu Y E = E (Y E), zde Y E = MUL*

- (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) MUL 2 2 $\rightarrow^{3}_{\beta}$
- (Z 2) ? 0 : (ADD 2 (MUL (P 2) 2)) =
  - *vyhodnotíme ternární operátor, 2 není 0, tedy Z 2 = F, vracíme poslední část*
- ADD 2 (MUL (P 2) 2) =
  - *pro přehlednost budeme vyhodnocovat zevnitř*
- ADD 2 (MUL 1 2) =
- ADD 2 ((Y (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) 1 2)
  - *použijeme vlastnost pevného bodu Y E = E (Y E), zde Y E = MUL*
- ADD 2 ((λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) MUL 1 2) $\rightarrow^{3}_{\beta}$
- ADD 2 ((Z 1) ? 0 : (ADD 2 (MUL (P 1) 2))) =
- ADD 2 (ADD 2 (MUL (P 1) 2)) =
- ADD 2 (ADD 2 (MUL 0 2)) =
- ADD 2 (ADD 2 (Y (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) 0 2)) =
  - *použijeme vlastnost pevného bodu Y E = E (Y E), zde Y E = MUL*
- ADD 2 (ADD 2 ((Y (λ f a b . (Z a) ? 0 : (ADD b (f (P a) b)))) 0 2)) =
- ADD 2 (ADD 2 ((λ f a b . (Z a) ? 0 : (ADD b (f (P a) b))) MUL 0 2)) $\rightarrow^{3}_{\beta}$
- ADD 2 (ADD 2 ((Z 0) ? 0 : (ADD 2 (MUL (P 0) 2)))) =
- ADD 2 (ADD 2 0) =
- ADD 2 2 =
- 4

Podobným způsobem můžeme definovat další operace:

- **celočíselné dělení**
  - *def DIVPOS(a, b):*
    - *if Z(S(a) - b):*
      - *return 0*
    - *else:*
      - *return S(DIVPOS(SUB(a, b), b))*
  - *def DIV(a, b):*
    - *if !b:*
      - *return* $\perp$
    - *else:*
      - *return DIVPOS(a, b)*
  - odečítáme *b* od *a* tak dlouho, dokud nedospějeme k 0
  - pokud *(a + 1) - b > 0*, můžeme pokračovat v dělení
  - při každém odečtení k výsledku přičteme 1 pomocí funkce následníka
  - pokud *b = 0*, pak vracíme $\perp$
  - **LET DIVPOS = (λ a b . (Z (SUB (S(a) b))) ? 0 : (S (DIVPOS ((SUB a b) b))))**
  - **LET DIVPOS = Y (λ f a b . (Z (SUB (S(a) b))) ? 0 : (S (f ((SUB a b) b))))**

16/18

- **LET DIV = (λ a b . (Z b) ? ⊥ : DIVPOS a b)**

- **faktoriál**
  - *def FACT(n):*
    - *if Z(n):*
      - *return 1*
    - *else:*
      - *return MUL(n, FACT(P(n)))*
  - **LET FACT = (λ n . (Z n) ? 1 : (MUL n (FACT (P n))))**
  - **LET FACT = Y (λ f n . (Z n) ? 1 : (MUL n (f (P n))))**
- **zbytek po celočíselném dělení**
  - *def MOD(a, b):*
    - *if !b:*
      - *return ⊥*
    - *else:*
      - *return MODPOS(a, b)*
  - *def MODPOS(a, b):*
    - *if Z(S(a) - b):*
      - *return a*
    - *else:*
      - *return MODPOS(SUB(a, b), b)*
  - **LET MODPOS = (λ a b . (Z (SUB (S(a) b))) ? a : (MODPOS ((SUB a b) b)))**
  - **LET MODPOS = Y (λ f a b . (Z (SUB (S(a) b))) ? a : (f ((SUB a b) b)))**
  - **LET MOD = (λ a b . (Z b) ? ⊥ : MODPOS a b)**
- **test prvočíselnosti**
  - *def PRIME(n):*
    - *if(Z(P(n))):*
      - *return F*
    - *else:*
      - *PRIMEINNER(n, P(n))*
  - *def PRIMEINNER(n, m):*
    - *if(Z(P(m))):*
      - *return T*
    - *else:*
      - *if(Z(MOD(n, m))):*
        - *return F*
      - *else:*
        - *return PRIMEINNER(n, P(m))*
  - funkce PRIME testuje, zda *n* je rovno 0 nebo 1, pokud ano, nejde o prvočíslo
  - jinak spustí funkci PRIMEINNER s parametrem *n* a *n-1*, kde druhý parametr budeme při rekurzivním zanořování snižovat až k *1*
  - pokud je *n* dělitelné některým z těchto čísel, vracíme *F*, jinak *T*
  - **LET PRIMEINNER = (λ n m . (Z (P m)) ? T : ((Z (MOD n m)) ? F : PRIMEINNER n (P m)))**

- **LET PRIMEINNER = Y (λ f n m . (Z (P m)) ? T : ((Z (MOD n m)) ? F : f n (P m)))**
- **LET PRIME = (λ n . (Z (P n)) ? F : PRIMEINNER n (P n))**

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele kocotom.