Typy

- motto: *každý slušný programovací jazyk je typovaný*
- napriek tomu, mnoho beztypových jazykov sa teší veľkej obľube (Basic, PHP, Prolog, Smalltalk, Scheme, Python, Ruby, Yon, ...)
- typy obmedzujú programátora v písaní nezmyselných konštrukcií,
- ich úlohou je v čase kompilácie odhaliť chyby, ktoré by sa objavili v run-time (možno...ak by tou vetvou išli)
- typy obmedzujú vyjadrovaciu (event. výpočtovú) silu jazyka

Dnes bude:

- jednoduchý typovaný λ-calcul
  - type-checking a type-inference
  - unifikácia ako nástroj riešenia typových rovníc
- Curry-Howardov izomorfizmus
- zložitejšie typovacie systémy

Jednoduchý typovaný λ-calcul

Základná neformálna predstava o typoch:

- Typy:
  - základné: A, B, …
  - funkcionálne: t_1 Ÿ t_2

Pravidlá:

- Aplikácia: ak M:t_1Ÿ t_2, N:t_1, potom (M N):t_2
- Abstrakcia: ak x:t_1, M:t_2, potom (λx.M):t_1Ÿ t_2
- Konvencia: operátor funkčného typu Ÿ je asociatívny doprava
  - t_1 Ÿ t_2 Ÿ t_3 = t_1 Ÿ (t_2 Ÿ t_3)

Jednoduchý λ-calcul, F_1

Termy	t ::= x | λx.t | (t t) | n | +
Typy	α, β ::= Int | α Ÿ β

Definujeme reláciu ΓĐ t:α, znamená, že term t je v kontexte Γ typu α
Kontext Γ obsahuje informáciu o typoch premenných x:α, [(String,Typ)]

{x:α}:ΓĐ x:α [VAR]

$\frac{\{x:α\}:ΓĐ\ N:β}{ΓĐ\ (λx.N):αŸ\ β}$ [ABS]

$\frac{ΓĐ\ M:αŸ\ β,\ ΓĐ\ N:α}{ΓĐ\ (M\ N):β}$ [APPL]

ΓĐ n:Int [INT]

ΓĐ +:IntŸ IntŸ Int [PLUS]

S intuitívnou predstavou o typoch sa (bezhlavo) vrhnime do typovania výrazov

Tipujme typy

- $\lambda x.x$
  - $(\lambda x^{\alpha}.x^{\alpha})^{\beta}$, potom $\beta = \alpha$Ÿ α
- $\lambda xy.x$
  - $(\lambda x^{\alpha}\ y^{\beta}.x^{\alpha})^{\delta}$, potom $\delta = \alpha$Ÿ βŸ α
- $(\lambda x.x)y$
  - $(\lambda x^{\alpha}.x^{\alpha})^{\beta}\ y^{\delta}$, potom $\beta = \alpha$Ÿ α, $\delta = a$
  - $[(\lambda x^{\alpha}.x^{\alpha})^{\alpha\text{Ÿ}\ \alpha}\ y^{\alpha}]^{\alpha}$
- $\lambda xyz.(xz)(yz)$
  - $\lambda x^{\alpha}y^{\beta}z^{\delta}.[(x^{\alpha}z^{\delta})^{\eta}(y^{\beta}z^{\delta})^{\varepsilon}]^{\theta}$, $\alpha = \delta$Ÿ η, $\beta = \delta$Ÿ ε, $\eta = \varepsilon$Ÿ θ
  - $x{:}\alpha=\delta$Ÿ $(\varepsilon$Ÿ $\theta)$, $y{:}\beta = \delta$Ÿ ε, $z{:}\delta$
  - $(\delta$Ÿ $(\varepsilon$Ÿ $\theta))$Ÿ $(\delta$Ÿ $\varepsilon)$Ÿ δŸ θ, výsledok je typu θ
- $\lambda x.(x\ x)$
  - $\lambda x^{\alpha}.(x^{\alpha}\ x^{\alpha})^{\beta}$, potom $\alpha =^{?} \alpha$Ÿ β, nemá riešenie...

Sústava rovníc

Vlastnosti F_1

- niektoré λ-termy nie sú otypovateľné, Y= λf.(λx. f(x x)) (λx. f(x x)) :-) ☹
  - dôvod: podobne ako λx.(x x),
  - neformálne: sústava zodpovedajúcich typových rovníc nemá riešenie

- Church-Rosserova vlastnosť platí aj pre typovaný λ-kalkul ☺
  - typované λ-termy sú pomnožinou lambda termov

- typovaný λ-kalkul je silne normalizovateľný = noetherovský = neexistuje nekonečné odvodenie 😐

  dokázané až 1966-1968, a to >= 6x.

  Dôkaz: napr. v H. Barendregt – Lambda Calculus, Its Syntax and Semantics

- Dôsledok:

  žiaden operátor pevného bodu nie je otypovateľný, lebo X = F X = F (F X) = …

Normálne formy
(stupeň termu)

Cieľ: typovaný λ-term má normálnu formu = neexistuje nekonečné odvodenie

- **stupeň typu :**
  - pre základné: A, B, … je 1,
  - pre funkcionálne: α Ÿ β je 1+max(stupeň α, stupeň β)

- **stupeň redexu :**
  - $[(\boldsymbol{\lambda x^{\alpha}.M^{\beta}})^{\alpha\ \text{Ÿ}\ \beta}\ \mathbf{N}^{\alpha}]^{\beta}$ je stupeň typu (α Ÿ β)

- **stupeň termu M :**
  - 0 ak neobsahuje redex,
  - max.stupeň redexu v **M**

Lemma (stupeň po substitúcii/redukcii):

- **stupeň termu $M[x^{\alpha}:N]$ <= max(stupeň M, stupeň N, stupeň α)**
  - redexy v $M[x^{\alpha}:N]$ sú v M, resp. N, resp. ak v M je podterm niekde tvaru (x Q) a za N dosadíme N =λy.P, tak vznikne nový redex ((λy.P) Q)
  - α = β Ÿ η, Q::β, (x Q):: η, λy.P::α = β Ÿ η, y::β, P::η
  - stupeň (x Q) = stupeň(β Ÿ η) = stupeň ((λy.P) Q)

Normálne formy
(konečné odvodenie)

Veta: typovaný λ-term má normálnu formu

Nájdeme R je redex v M maximálneho stupňa, ktorý obsahuje len redexy ostro menších stupňov, t.j.

- je taký,
- stupeň M = stupeň R,
- R už neobsahuje redexy stupňa (stupeň R), … ale len menšie !

- Pri β-redukcii redexu R v M sa redexy
  - mimo R nezmenia,
  - vnútri R sa nahradia sa inými, ale stupeň termu M nevzrastie (viď lemma),
  - R zmizne a je nahradený redexami s menším stupňom.

Dôsledok: β-redukciou nevzrastie stupeň termu a klesne počet redexov max.stupňa

Takže dvojica (stupeň M, počet redexov stupňa M) klesá v lexikografickom usporiadaní...

Keďže je to dobre usporiadanie, nemôže existovať nekonečná postupnosť.

Type checking vs. inference

```
data LExp = LAMBDA String LExp |
        ID String |
        APL LExp LExp |
        CON String | CN Integer
        deriving(Show, Read, Eq)

data Type =
        TInteger |
        Tvar Integer |
        Type :Ÿ  Type
        deriving(Show, Read, Eq)
```

checkType :: LExp Ÿ Type Ÿ Bool

Problém (rozhodnuteľný):

- checkType (A B) T_2 = checkType A (T_1Ÿ T_2) && checkType B T_1
- nevieme, ako uhádnuť typ T_1 ???

typeInference :: LExp Ÿ Maybe Type = (Just Type | Nothing)

Problém (rozhodnuteľný) :

- inferType (λx.M) = T_1 Ÿ (inferType M)
- ako zistiť typ premennej x viazanej v λ-abstrakcii, teda T_1 ???

inhabitation :: Type Ÿ Maybe LExp = (Just LExp | Nothing)

Problém (rozhodnuteľný) :

- ako zistiť term M predpísaného typu ?

Anotácie premenných

(typovaný λ-kalkul podľa Churcha)

```
data LExp = LAMBDA String Type LExp |
            ID String |
            APL LExp LExp |
            CON String | CN Integer
            deriving(Show, Read, Eq)

data Type = TInteger |
             Type :Ÿ  Type
            deriving(Show, Read, Eq)
```

Termy

t ::= x | λx:α.t | (t t) | n | +

- ak x:T_1, M:T_2, potom (λx:T_1.M): T_1Ÿ T_2

```
type Context = [(String, Type)]                  -- premenná a jej typ
inferType :: Context Ÿ  LExp Ÿ  Maybe Type

inferType ctx (ID var) = lookup var ctx           -- [VAR]

inferType ctx (APL m n) = t2 where                -- [APPL]
    t1 :Ÿ  t2 = inferType ctx m      -- môže výjsť Nothing, potom propaguj
    t3 = inferType ctx n             -- Nothing aj do výsledku
    {t1 == t3}                       -- tieto dva typy musia byť rovnaké

inferType ctx (LAMBDA x t1 m) = t1 :Ÿ  inferType ((x,t1):ctx) m    -- [ABS]
```

Tento kód nie je v Haskelli, len ilustruje ideu

Typovaný λ-kalkul podľa Churcha

$$(\text{axiom}) \qquad \Gamma \vdash x : \sigma, \qquad \text{if } (x{:}\sigma) \in \Gamma;$$

$$(\rightarrow\text{-elimination}) \qquad \frac{\Gamma \vdash M : (\sigma \rightarrow \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\,N) : \tau};$$

$$(\rightarrow\text{-introduction}) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma.M) : (\sigma \rightarrow \tau)}.$$

{x:α}:ΓƊ x:α [VAR]

{x:α}:ΓƊ N:β [ABS]
ΓƊ (λx:α.N):αŸ β

ΓƊ M:αŸ β, ΓƊ N:α [APPL]
ΓƊ (M N):β

http://plato.stanford.edu/entries/type-theory-church/

Typovaný λ-kalkul
Church vs. Curry

Nech |M| je term M bez typových anotácií, t.j. zobrazenie $Church_{\lambda}$ Ÿ $Curry_{\lambda}$

- Ak ΓƊ M:α podľa Church, potom ΓƊ |M|:α podľa Curry.
- Ak ΓƊ M:α podľa Curry, potom existuje anotovaný M′ taký, že
  - ΓƊ M′:α podľa Church,
  - |M′| = M.

Typové premenné

(typovaný λ-kalkul podľa Curry)

```
data LExp = LAMBDA String LExp |
            ID String |
            APL LExp LExp |
            CON String | CN Integer
            deriving(Show, Read, Eq)
data Type = TInteger |
             Tvar Integer |
              Type :Ÿ  Type
            deriving(Show, Read, Eq)
```

- ak nepoznáme konkrétny typ, zavedieme typovú premennú,
- algoritmus zozbiera podmienky (rovnosti) pre typové premenné,
- ak máme šťastie, podarí sa nám ich vyriešiť,
- typové premenné: α, β, δ, η, δ, ε, θ budeme radšej indexovať

```
type Constraints = [(Type,Type)]                      -- zoznam rovností, eqs
inferType :: Context->LExp->Constraints-> (Type, Constraints)

inferType ctx (APL m n) eqs = t2, {t1 = t3}:eqs'' where          -- [APL]
                         t3Ÿ t2, eqs' = inferType ctx m eqs
                         t1, eqs'' = inferType ctx n eqs'
inferType ctx (LAMBDA x m) eqs = t1Ÿ t2, eqs'                     -- [ABS]
                         -- t1 je nová typová premenná
                         where t2,eqs' = inferType ((x,t1):ctx) m eqs

inferType ctx (ID var) eqs = lookup var ctx, eqs                  -- [VAR]
```

Dostaneme sústavu rovníc, ktorú riešime

Inferencia typu – príklad

λf.λa.λb.λc. c (f a) (f b):T, Ø, Ø

((c (f a)) (f b)):T4, f:T0, a:T1, b:T2, c:T3, {T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4}

(c (f a)):T5, (f b):T6, f:T0, a:T1, b:T2, c:T3, {T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4, T5=T6Ÿ T4}

c:T3, (f a):T8, (f b):T6, f:T0, a:T1, b:T2, c:T3, {T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4, T5=T6Ÿ T4, T3=T8Ÿ T5}

c:T3, f:T0, a:T1, f:T0, b:T2, f:T0, a:T1, b:T2, c:T3, {T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4, T5=T6Ÿ T4, T3=T8Ÿ T5, T0=T1Ÿ T8, T0=T2Ÿ T6}

Sústava rovníc:

{T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4, T5=T6Ÿ T4, T3=T8Ÿ T5, T0=T1Ÿ T8, T0=T2Ÿ T6}

{T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4, T5=T6Ÿ T4, T3=T8Ÿ T5, T0=T1Ÿ T8, T2=T1, T8=T6}

Riešenie:

T=T0Ÿ T1Ÿ T2Ÿ T3Ÿ T4 =(T1Ÿ T8)Ÿ T1Ÿ T1Ÿ (T8Ÿ T5)Ÿ T4=

(T1Ÿ T8)Ÿ T1Ÿ T1Ÿ (T8Ÿ T8Ÿ T4) Ÿ T4

Domáca úloha 1:

inferType :: [(LExp,Type)]->Context->Constraint->Int->Constraint

Inferencia typu – príklad 2

λxyz.(xz)(yz):T, Ø, Ø

λyz.(xz)(yz):T2, x:T1, {T=T1Ÿ T2}

λz.(xz)(yz):T4, y:T3, x:T1, {T=T1Ÿ T2, T2=T3Ÿ T4}

(xz)(yz):T6, z:T5, y:T3, x:T1, {T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6}

(xz):T7, (yz):T8, z:T5, y:T3, x:T1, {T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6, T7=T8Ÿ T6}

x:T9, z:T10, (yz):T8, z:T5, y:T3, x:T1, {T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6, T7=T8Ÿ T6,T9=T10Ÿ T7, T9=T1, T10=T5}

y:T11, z:T12, z:T5, y:T3, x:T1, {T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6, T7=T8Ÿ T6,T9=T10Ÿ T7, T9=T1, T10=T5, T11=T12Ÿ T8, T11=T3, T12=T5}

Sústava rovníc:

{T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6, T7=T8Ÿ T6,T9=T10Ÿ T7, T9=T1, T10=T5, T11=T12Ÿ T8,T11=T3, T12=T5}

{T=T1Ÿ T2, T2=T3Ÿ T4, T4=T5Ÿ T6, T7=T8Ÿ T6, T1=T5Ÿ T7, T3=T5Ÿ T8}

Riešenie:

T=T1Ÿ T2=T1Ÿ (T3Ÿ T4)=T1Ÿ (T3Ÿ (T5Ÿ T6)= (T5Ÿ T8Ÿ T6)Ÿ (T5Ÿ T8)Ÿ T5Ÿ T6

Inferencia typu – príklad 3

Y = λf.(λx. f(x x)) (λx. f(x x)):T, Ø, Ø

(λx. f(x x)) (λx. f(x x)):T2, f:T1, {T=T1Ÿ T2}

(λx. f(x x)):T3, (λx. f(x x)):T4, f:T1, {T=T1Ÿ T2 , T3=T4Ÿ T2}

(f(x x)):T6, (λx. f(x x):T4, f:T1,x:T5, {T=T1Ÿ T2 ,
T3=T4Ÿ T2,T3=T5Ÿ T6}

(x x):T7, (λx. f(x x):T4, f:T1,x:T5, {T=T1Ÿ T2 , T3=T4Ÿ T2,T3=T5Ÿ T6,
T1=T7Ÿ T6}

(λx. f(x x):T4, f:T1,x:T5, {T=T1Ÿ T2 , T3=T4Ÿ T2,T3=T5Ÿ T6,
T1=T7Ÿ T6,T5=T5Ÿ T7}

Unifikácia

- unifikácia je spôsob riešenia rovníc v rôznych teóriach
- najjednoduchším prípadom je syntaktická (prázdna, či Herbrandova teória)
- v našom prípade je problém zredukovaný na jediný funkčný symbol Ÿ

$f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n), C$	=» $t_1=s_1, \ldots, t_n=s_n, C$
$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m), C$	=» FAIL, C
$x=t, C$	=» $x=t, C[x:t]$ if $x \notin t$ [OCCUR CHECK]
$x=t, C$	=» FAIL if $x \in t$
$x=x, C$	=» C

!!! tento naivný algoritmus je exponenciálny

Príklad:

$t_1=g(t_2, t_2), t_2=g(t_3, t_3), t_3=g(t, t)$

$t_1=g(t_2, t_2), t_2=g(g(t, t), g(t, t)), t_3=g(t, t)$

$t_1=g(g(g(t, t), g(t, t)), g(g(t, t), g(t, t))), t_2=g(g(t, t), g(t, t)), t_3=g(t, t)$

Ak zovšeobecníme vstup pre $t_1..t_n$, výsledok bude exponenciálny od dĺžky vstupu

Unifikácia

- type Constraints = [(Type,Type)] -- [(Type=Type)]
- unify :: ConstraintsŸ Constraints

unify [] = [] -- Just []
unify (S=T:C′)
| S == T = unify C′
| S == t_i && not($t_i \in$ S) = unify(C′[t_i:S]) ++ [t_i=S]
| t_i == T && not($t_i \in$ T) = unify(C′[t_i:T]) ++ [t_i=T]
| S == S_1Ÿ S_2 and T == T_1Ÿ T_2 = unify(S_1=T_1:S_2=T_2:C′)
| otherwise = fail -- Nothing

- unify :: ConstraintsŸ Maybe Constraints

| S == t_i && not($t_i \in$ Free(S)) = Just ([t_i=S] : subst) where
Just subst = unify(C′[t_i=S])

Unifikácia

(odstránenie substitúcie)

unify::(Type,Type)Ÿ Maybe Constraints Ÿ Maybe Constraints

unify (_,_) Nothing = Nothing
unify (a1Ÿ b1,a2Ÿ b2) subst = subst2 where
subst1 = unify (a1,a2) subst
subst2 = unify (b1,b2) subst1

--dereferencia premennej miesto aplikacie substitúcie
unify (t_i,b) s@(Just subst) = unify (a,b) s where a = deref t_i subst
unify (a,t_i) s@(Just subst) = unify (a,b) s where b = deref t_i subst

--predpokladáme, že t_i je dereferencovaná a že platí occur check
unify (t_i,b) (Just subst) = Just ((t_i,b):subst) if t_i not in b
unify (a,t_i) (Just subst) = Just ((t_i,a):subst) if t_i not in b

– zabránenie vzniku cyklu medzi premennými
unify (t_i,t_j) s@(Just subst) | i < j = Just ((t_i , t_j):subst)
| j < i = Just ((t_j , t_i):subst)
| otherwise s

– otherwise
unify (_,_) _ = Nothing

Domáca úloha

Domáca úloha 1: rozpracujte inferType do podoby

```
inferType :: [(LExp,Type)]->Context->Constraint->Int->Constraint
použitie:          inferType t = inferType [(t,Tvar 0)] [] [] 1
```

Domáca úloha 2: rozpracujte unify do podoby

```
unify :: [(Type, Type)] -> Maybe Constraints ->Maybe Constraints
použitie:          unify constraint = unify constraint (Just [])
```

Demonštrujte spoločne:

```
typeExp t = case unify (inferType t) of
                Just subst -> lookup (T 0) subst
                Nothing -> Nothing
```

Typy a formule

- K = (λxy.x)^αŸ βŸ α
- S = λxyz.xz(yz)^(δŸ εŸ θ)Ÿ (δŸ ε)Ÿ δŸ θ

Hilbertov axiomatický systém:

A Ÿ (B Ÿ A)

((A Ÿ (BŸ C)) Ÿ ((AŸ B) Ÿ (AŸ C))

Modus ponens:

AŸ B A

B

K = (λxy.x)αŸ βŸ α
S = λxyz.xz(yz)(δŸ εŸ θ)Ÿ (δŸ ε)Ÿ δŸ θ

Curry-Howardov izomorfizmus

- S K K = I

S:(δŸ εŸ θ)Ÿ (δŸ ε)Ÿ (δŸ θ)
K:αŸ βŸ α
K:űŸ ɣ Ÿ ű

(δŸ εŸ θ)= (αŸ βŸ α)

- δ= α, ε=β

(δŸ ε)=(űŸ ɣ Ÿ ű)

- δ=ű, ε=ɣ Ÿ ű

δ=θ

S:(δŸ (ɣ Ÿ δ)Ÿ δ) Ÿ (δŸ (ɣ Ÿ δ)) Ÿ (δŸ δ)
K:(δŸ (ɣ Ÿ δ)Ÿ δ)
K:(δŸ (ɣ Ÿ δ))

K = (λxy.x)αŸ βŸ α
S = λxyz.xz(yz)(δŸ εŸ θ)Ÿ (δŸ ε)Ÿ δŸ θ

Curry-Howardov izomorfizmus

S:(δŸ (ɣ Ÿ δ)Ÿ δ) Ÿ (δŸ (ɣ Ÿ δ)) Ÿ (δŸ δ)
K:(δŸ (ɣ Ÿ δ)Ÿ δ)
K:(δŸ (ɣ Ÿ δ))

S:(AŸ (BŸ A)Ÿ A) Ÿ (AŸ (BŸ A)) Ÿ (AŸ A) K:(AŸ (BŸ A)Ÿ A)

SK:(AŸ (BŸ A)) Ÿ (AŸ A) K:(AŸ (BŸ A))

SKK:(AŸ A)

Polymorfický (druhorádový) λ-kalkul, System F_2 (Girard-Reynold)

V jednoducho-typovanom λ-kalkule doménou premenných sú funkcie,
v druho-rádovom λ-kalkule doménou premenných sú typy:

Typ

σ ::= Int | σ Ÿ σ | α | ∀α.σ   - všeobecne kvantifikovaný typ

- λx.x : ∀α.αŸ α
- NOT : ∀α.αŸ α
- K=TRUE (λx.λy.x) : ∀α.∀β.αŸ βŸ α, FALSE (λx.λy.y) : ∀α.∀β.αŸ βŸ β
- <u>0</u>,<u>1</u>,<u>2</u> (λf.λx.f(f x)),… : ∀α.(αŸ α)Ÿ αŸ α

Dôsledky:

- v takomto λ-kalkule vieme otypovať aj to, čo v F_1 nevieme (ω)
- inferencia v takomto kalkule je nerozhodnuteľný problém ☹, 1970-1994
- hierarchia F_1, F_2, F_3, … Ÿ F_ω
- dependent type (t:Type, t)

System F_2

Typy

σ ::= Int | σ Ÿ σ | α | ∀α.σ

{x:α}:ΓƊ x:α [VAR]

{x:α}:ΓƊ N:β
ΓƊ (λx.N):αŸ β [ABS]

ΓƊ M:αŸ β, ΓƊ N:α
ΓƊ (M N):β [APPL]

ΓƊ M:β
ΓƊ M:∀α.β [GEN] α not free in Γ

ΓƊ M:∀α.β
ΓƊ M:β[α:θ] [INST]

λx.x : ∀α.αŸ α
AND : ∀α.αŸ αŸ α
0,1,2,… : ∀t.(tŸ t)Ÿ (tŸ t)

Príklady v F_2

{x:α}D x:α	[VAR]	{x:Int}D x:Int	[VAR]
D(λx.x):αŸ α	[ABS]	D(λx.x):IntŸ Int	[ABS]
D(λx.x): ∀α.αŸ α	[GEN]		
D(λx.x): IntŸ Int	[INST]		

{x:∀α.αŸ α}D x:∀α.αŸ α [VAR]
{x:∀α.αŸ α}D x:(∀β.βŸ β)Ÿ (∀β.βŸ β) [INST] {x:∀α.αŸ α}D x:∀α.αŸ α [VAR]
{x:∀α.αŸ α}D (x x):∀α.αŸ α [APP]
D λx.(x x):(∀α.αŸ α)Ÿ (∀α.αŸ α) [ABS]

- podarilo sa nám otypovať výraz, ktorý v jednoducho-typovanom kalkule sa nedal
- dostali sme však typ (∀α.αŸ α)Ÿ (∀α.αŸ α), ktorý vnútri obsahuje kvantifikátory (deep type) na rozdieľ od tých, čo ich majú len na najvyššej úrovni, napr. ∀t.(tŸ t)Ÿ (tŸ t) – shallow type

Let polymorfizmus Hindley-Millner

chceme zakázať kvantifikáciu typov vnútri typového výrazu (zakázať deep type)

Typy

σ ::= ψ | ∀α.σ

ψ ::= Int | ψ Ÿ ψ | α

premenná viazaná λ výrazom nemôže byť polymofrického typu, napr.

f (λx.x) where f = λg.[… (g 1) …. (g True) …], alebo v Haskelli:

- idd = (\x->x)
- foo = f idd where f = \g->(if (g True) then (g 2) else (g 1))

Nahradíme pravidlo [GEN] pravidlom [LET]

Γ Ɗ M:β, Γ, x:∀α.β Ɗ N:δ [LET] α in β, α not free in Γ

Γ Ɗ let x=M in N : δ