Funkcionální programování

Lambda kalkulus

Petr Pudlák

26. října 2011

Vyčíslitelnost

Následující formální systémy definují stejné třídy funkcí:

- Turingův stroj
  Imperativní programování
- Lambda kalkulus (Church)
  Funkcionální programování
- Rekurzivní funkce

Lambda kalkulus

Citát

The point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical no one will believe it.

– Bertrand Russell

Lambda kalkulus

(netypovaný)

Definition (Jazyk lambda kalkulu)

Množina lambda termů Λ je induktivně vybudována z (nekonečné spočetné) množiny proměnných V:

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (MN) \in \Lambda \\ x \in V; M \in \Lambda &\Rightarrow (\lambda x.M) \in \Lambda \end{aligned}$$

Úmluva: Aplikace funkcí asociujeme vlevo, lambda abstrakce vpravo. Například:

$$\begin{aligned} (\ldots(FM_1)\ldots M_n) &= FM_1\ldots M_n \\ (\lambda x_1.(\ldots(\lambda x_n.M)\ldots)) &= \lambda x_1.\lambda x_2.\ldots.\lambda x_n.M \\ &= \lambda x_1\ldots x_n.M \end{aligned}$$

Lambda kalkulus

příklady

Example

Příklady: x; xz; $\lambda x.xz$; $(\lambda x.xz)y$; $(\lambda y.(\lambda x.xz)y)w$.

Lambda kalkulus

Rovnost termů (neformálně): $(\lambda x.M)N \;=\; M[x := N]$

Funkce s více argumenty – Currying

Lambda kalkulus popisuje jen funkce s jedním argumentem?
Na funkce s více argumenty se mohu dívat jako na funkce vracející opět funkce.

Example

Mějme výraz závislý na dvou proměnných $f(x, y)$.
Definujme $F_x = \lambda x.f(x, y)$ a $F = \lambda y.F_x$. Pak

$$(Fx)y = ((\lambda x.\lambda y.f(x, y))x)y = f(x, y)$$

V souladu s konvencí vynecháme závorky: Fxy.

Tento převod vymyslel Moses Schönfinkel (1924). Je nazýván *currying*[1] podle matematika Haskella Curryho, který ho znovuobjevil a používal.

[1] Výslovnost viz. http://www.haskell.org/pipermail/haskell-cafe/2010-October/thread.html#84486

Kompatibilní relace na termech

Definition

Řekneme, že relace ρ na lambda termech je *kompatibilní*, jestliže

$$\begin{array}{rcl} M\rho N & \Rightarrow & (ZM)\rho(ZN) \\ & & (MZ)\rho(NZ) \\ & & (\lambda x.M)\rho(\lambda x.N) \end{array}$$

α-konverze

Definition

$$\lambda x.M \ \rightarrow_\alpha \ \lambda y.M[x := y]$$

Definujme $\equiv$ jako nejmenší kompatibilní ekvivalenci, která obsahuje $\rightarrow_\alpha$. Nadále považujme všechny termy $M \equiv N$ za stejné a podle potřeby je zaměňujme.

Volné proměnné

Definition (Volné proměnné)

Termu M přiřadíme množinu jeho *volných proměnných* takto:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

Ostatní proměnné vyskytující se v termu nazveme *vázané*.
Term M nazveme *uzavřený* nebo též *kombinátor* pokud $FV(M) = \{\}$.

Neformálně: x má vázané výskyty v termu M všude, kde x není „schované" pod lambda abstrakcí s x.

Substituce

Definition (Substituce)

Substitucí termu N za proměnnou x v termu M, značeno $M[x := N]$, definujeme (níže všude $x \neq y$):

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \\ (PQ)[x := N] &\equiv (P[x := N])(Q[x := N]) \\ (\lambda x.P)[x := N] &\equiv (\lambda x.P) \\ (\lambda y.P)[x := N] &\equiv (\lambda y.(P[x := N])) \end{aligned}$$

V posledním případě nesmí být y volná v N. Pokud je, nejprve y přejmenujeme.
Proč? Například $(\lambda x.y)[y := x] = \lambda x.(y[y := x]) = \lambda x.x$.

β–redukce

Definition (β-redukce a expanze)

Definujme na Λ tyto binární relace:

1.
   - $(\lambda x.M)N \to_\beta M[x := N]$
   - $M \to_\beta N \Rightarrow ZM \to_\beta ZN$, $MZ \to_\beta NZ$, $(\lambda x.M) \to_\beta (\lambda x.N)$
   - Přechod ve směru šipky nazýváme β-*redukce*.
   - Přechodu proti směru šipky říkáme β-*expanze*.
2. Relace $\twoheadrightarrow_\beta$ je reflexivní tranzitivní uzávěr relace $\to_\beta$.
3. Relace $=_\beta$ je reflexivní, tranzitivní a symetrický uzávěr relace $\to_\beta$ (čili ekvivalence).

β–redukce

Další pojmy

Definition

1. Termy tvaru $(\lambda x.M)N$ nazýváme *β-redexy*[a]. Term $M[x := N]$ pak nazveme *β-contractum*.
2. Term M *je β-normální forma* pokud nemá žádný redex jako podterm.
3. Term M *má β-normální formu* pokud existuje N, který je β-nf a $M =_\beta N$.

[a] Odvozeno z reducible expression.

η–redukce

Definition (η-redukce a expanze)

$$(\lambda x.Mx) \to_\eta M, \text{kde } x \text{ není volná v } M.$$

Vyjadřuje určitý princip extensionality: Funkce jsou si rovny, pokud pracují stejně na daném argumentu.

- Není tak podstatná jako β-redukce, většinou je spíše pomocná/optimalizační.
- V určitém smyslu duální k β-redukci (aplikace vně/uvnitř abstrakce).
- $M \to_\eta N$ neznamená, že $M =_\beta N$. Například $\lambda x.yx \to_\eta y$, ale přitom oba termy jsou v β-nf. Platí ale, že $(\lambda x.Mx)N \to_\beta MN$.
- η-redukce tedy umožňuje zjednodušovat některé funkce dříve, než jsou aplikovány na argument.
- η-expanze se používá v některých striktních jazycích pro oddálení vyhodnocení. Výraz se tak vyhodnotí teprve až je mu dán (obvykle prázdný) argument.
- Při interpretaci lambda termů jako programů nemusí mít η-ekvivalentní termy stejnou sémantiku: $\lambda x.Mx$ může skončit a přitom M samotné ne.

Church-Rosserova věta

Theorem (Church-Rosser)

Jestliže $M \twoheadrightarrow_\beta N_1$ a $M \twoheadrightarrow_\beta N_2$ pak existuje L takové, že $N_1 \twoheadrightarrow_\beta L$ a $N_2 \twoheadrightarrow_\beta L$. (Nakreslit obrázek.)

Důkaz.

Viz. [2] nebo [1]. □

Důsledky:

- Je-li $M =_\beta N$ pak existuje L takové, že $M \twoheadrightarrow_\beta L$ a $N \twoheadrightarrow_\beta L$.
- Má-li M β-nf N pak $M \twoheadrightarrow_\beta N$.
- Každý lambda term má nejvýše jednu β-nf.

Church-Rosserova věta

Další důsledky

- λ-kalkulus je konzistentní (syntaktický důkaz).
- Existují termy, které nemají β-nf. Například

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

- Může existovat mnoho strategií, jaké redexy kontrahovat při hledání normální formy.
  - Všechny úspěšné vedou ke stejné normální formě.
  - Ať začneme jakkoliv, můžeme se vždy ještě k nf dostat – nemůžeme „špatně odbočit" bez možnosti návratu.

Modely lambda kalkulů

- Vědělo se, že lambda kalkulus je konzistentní, ale dlouho se neznaly modely.
- Problém: Mohu napsat term $\lambda x.xx$. Jaký má mít model funkce aplikovaná sama na sebe?
- Scott vytvořil tzv. *Domain theory*, která dává sémantiku lambda kalkulům.

β-redukce – cvičení

Cvičení

Najděte nějaké termy M a N takové, že:

1. *N nemá nf ale MN má nf.*
2. *M nemá nf ale MN má nf.*

Použijte kombinátory K, I, Ω.

Silně a slabě normalizující systémy

Definition

Systém redukcí je *silně normalizující*, pokud pro každý term libovolná sekvence redukcí skončí normální formou.
Systém je *slabě normalizující*, pokud pro každý term existuje alespoň jedna sekvence redukcí skončí normální formou.

Poznámky:

- Netypovaný lambda kalkulus není ani silně ani slabě normalizující. Viz. např. $\Omega = (\lambda x.xx)(\lambda x.xx)$.
- Existují typované lambda kalkuly, které jsou silně/slabě normalizující.
- Silně/slabě normalizující systém nemůže být turingovsky úplný (protože všechny jeho funkce jsou totální).

Rekurze

Example

Zkusme definovat neformálně faktoriál:

$$f = \lambda n.\text{if}\,(=0n)1(*n(f(-n1)))$$

Lambda kalkulus takovou definici neumožňuje, funkce jsou anonymní, a tak nemohou referovat samy na sebe.

Zkusme provést β-expanzi a vyabstrahovat rekurzi:

$$\begin{aligned} f &= \underbrace{\lambda f' n.if\,(=0n)1(*n(f'(-n1)))}_{H} f \\ &= H\,f \end{aligned}$$

Potřebujeme nalézt způsob, jak vyřešit tuto rovnici – nalézt *pevný bod* termu H.

Rekurze

pokračování

Chceme najít funkci Y takovou, že

$$YH = H(YH),$$

pak můžeme definovat

$$f = YH$$

Pozorování: Term $\Omega = (\lambda x.xx)(\lambda x.xx)$ se redukuje sám na sebe.
Definujme

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Počítejme:

$$\begin{aligned} YH &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))H \\ &= (\lambda x.H(xx))(\lambda x.H(xx)) \\ &= H((\lambda x.H(xx))(\lambda x.H(xx))) \\ &= H(YH) \end{aligned}$$

Kterou cestou na rozcestí, kterou cestou jen se dát?

Na který mi kyne štěstí, na který mi hrozí pád?

Mám-li lambda term, jak ho redukovat, abych dostal normální formu? Které redexy vybírat?

Example

Buď $K = \lambda xy.x$, $I = \lambda x.x$ a $\Omega = (\lambda x.xx)(\lambda x.xx)$. Redukujme $KI\Omega$:

Začneme-li prvním redexem, tedy redukujeme

$$\begin{aligned} KI\Omega &= ((\lambda x.(\lambda y.x))I)\Omega \\ &\to_\beta (\lambda y.I)\Omega \\ &\to_\beta I \end{aligned}$$

Budeme-li vybírat stále poslední redex, dostaneme:

$$\begin{aligned} KI\Omega &= KI(\underbrace{(\lambda x.xx)(\lambda x.xx)}) \\ &\to_\beta KI((\lambda x.xx)(\lambda x.xx)) \\ &\quad \vdots \end{aligned}$$

Redukční strategie

Definition

Redukční strategie je mapa F z Λ do Λ taková, že

$$M \in \Lambda \Rightarrow M \twoheadrightarrow_\beta F(M)$$

Redukční strategie je:

jednokroková pokud $M \rightarrow_\beta F(M)$ pro všechna M, která nejsou v nf;

efektivní pokud ji lze „jednoduše" implementovat;

rekurzivní pokud je „implementována rekurzivně";

Redukční strategie
(pokračování)

Definition

Redukční strategie je:

normalizující pokud jestliže M má nf pak existuje n takové, že $F^n(M)$ je nf.

kofinální pokud pro $M \twoheadrightarrow_\beta N$ existuje n takové, že $N \twoheadrightarrow_\beta F^n(M)$.

Church-Rosserovská pokud je-li $M =_\beta N$ pak existují m, n takové, že $F^m(M) = F^n(N)$.

nekončící pokud pro term M s nekonečnou redukční posloupností je $F^n(M) \twoheadrightarrow_\beta F^{n+1}(M)$ je neprázdná redukce.

Redukční strategie

(věty)

Theorem

Platí:

1. *Každá Church-Rosserovská strategie je kofinální,*
2. *Každá kofinální strategie je normalizující.*

Naopak to neplatí.
Pro beta-redukce existuje:

- *efektivní kofinální strategie (Gross-Knuthova strategie),*
- *rekurzivní Church-Rosserovská strategie,*
- *efektivní nekončící strategie.*

Normalizační věta I

Normální pořadí redukcí

Theorem (Normalizační věta, též 2. Church-Rosserova věta)

Buď F_1 jednokroková strategie, která vždy redukuje nejlevější redex (pokud není term v nf).
F_1 je normalizující.

Důkaz.

Viz. [1]. □

Definition

Strategii F_1 (resp. její pořadí vyhodnocování) nazýváme *normální* (anglicky *normal order*).

Normalizační věta II

Normální pořadí redukcí

- Nejlevější redex je první, na který narazíme při preorder procházení stromu vytvořeného z termu.
- Argumenty se redukují teprve po jejich dosazení do funkce (líné vyhodnocování).

Aplikativní pořadí redukcí

Definition

Aplikativní strategie redukuje v každém kroku nejvnitřnější nejlevější redex.
Odpovídá postorder pocházení stromu.
Termy jsou redukovány (pokud možno) dříve, než jsou použity jako argumenty do funkcí.

- *Aplikativní pořadí*, nazývané také *striktní (strict)* nebo *eager*, je použito v podstatě ve všech imperativních jazycích i mnoha funkcionálních.
- Je jednodušší na implementaci a implementace je obvykle rychlejší.
- Není normalizující. Například term[2] $KI\Omega$ má normální formu I, ke které dospějeme normálním pořadí redukcí, ale ne aplikativním.
- Důsledek: Existují programy (termy), které jsou normalizující (skončí s daným výsledkem), který ale nelze nalézt aplikativním pořadím vyhodnocování.

[2] $K = \lambda xy.x$, $I = \lambda x.x$ a $\Omega = (\lambda x.xx)(\lambda x.xx)$.

Efektivita redukcí

- Redukujme term $(\lambda x.fxx)M$. Při aplikativním pořadí nejprve zredukujeme M a potom provedeme jeho dosazení. Při normálním pořadí ale nejprve substituujeme M do fxx, a pak redukujeme obě jeho kopie.
- Normální pořadí může být méně efektivní než aplikativní, a to zásadně!
- Řešení: Místo stromu použijeme pro reprezentaci termů orientovaný acyklický graf (DAG). Substituované podtermy sdílíme, aby se redukovaly jen jednou.
  S myšlenkou přišel [5] a optimální řešení (v určitém smyslu) vypracoval [3].

(Nakreslit příklad.)

Vyhodnocovací strategie obecně

Vyhodnocovací strategie lze rozdělit na tyto základní skupiny:

Call by value: Výraz v argumentu funkce je vyhodnocen předtím, než je volána funkce samotná (ale nemusí to být nutně zleva doprava jako je aplikativní pořadí).

Call by name: Výraz v argumentu je substituován do funkce, jak v lambda kalkulu – redukce stromů.

Call by need: Pouze odkaz na výraz v argumentu je substituován do funkce, takže výraz je vyhodnocen maximálně jednou – redukce grafů.

Call-by-future: Výraz v argument je vyhodnocován paralelně na pozadí. Ve chvíli, kdy funkce potřebuje hodnotu argumentu, synchronizuje se s vláknem na pozadí.
Při opuštění funkce je vlákno na pozadí ukončeno.

let výrazy

- Pokud programátor identifikuje (pojmenovává) shodné výrazy, můžeme maximálně využít výhody *call by need*.
- Možnost pojmenovávat výrazy vede k lepšímu a čitelnějšímu stylu.
- Syntaxe:
  let $x = N$ **in** M
  kde $x \notin FV(N)$.
- Sémanticky je to ekvivalentní $(\lambda x.M)N$.
  Tomuto překladu v lambda kalkulu nelze obecně odvodit/ověřit typ, proto je konstrukce *let* nutná.
- Umožňuje překladači výraz optimalizovat, protože víme, že se tato lambda abstrakce aplikuje právě jednou a právě zde.

Example

$$square1 = \lambda x \to (x - 1) * (x - 1) \quad \text{-- neefektivní}$$
$$square1 = \lambda x \to (\textbf{let } x' = x - 1 \textbf{ in } x' * x')$$

kombinované *let* výrazy

V jednom *let* výrazu lze obvykle kombinovat více deklarací:

let
  $x1 = N1$
  $x2 = N2$
  $x3 = N3$
  ...
in M

rekurzivní *let* výrazy

- Je mnohem účelnější zapisovat výrazy v rekurzivním stylu než pomocí kombinátoru pro rekurzi (Y).
- Syntaxe:
  let $x = N$ **in** M
  kde $x \in FV(N)$.
- Sémanticky je to ekvivalentní $(\lambda x.M)(Y(\lambda x.N))$, kde Y je kombinátor rekurze.
- Je možné kombinovat více deklarací, vzájemně rekurzivních.
- V některých funkcionálních jazycích se rozlišuje ne-rekurzivní *let* a rekurzivní *letrec*. Haskell používá jen *let*, sám si to přebere.

Je nutna celá normální redukce?

- V praktickém funkcionálním jazyce jsou výsledkem výpočtu data, ne lambda termy.
- Výsledek typu „fuknce přičítající 4" nám nepřináší žádnou informaci (obecně totiž stejně nelze poznat, co nějaká funkce dělá).
- Funkcionální jazyk proto má zabudované funkce a datové objekty (např. čísla), pro která jsou zabudovaná pravidla, jak funkce redukovat. Například $+\,3\,4$ se zredukuje na 7 apod. Cílem je, aby výsledkem programu byla nějaká datový objekt.
- Pokud je zřejmé, že redukce nějakého termu (nebo podtermu) nevede k datovému objektu, ale k nějaké částečně vyhodnocené funkci, není třeba pokračovat ve vyhodnocování.

Hlavní normální forma I

Head normal form

Definition (head normal form)

Každý lambda term lze zapsat ve tvaru

$$\lambda x_1 \ldots x_n . N M_1 \ldots M_k$$

kde N není aplikace, $n \geq 0$ a $k \geq 0$, a platí, že

1. pro nějaké $l \leq k$ je $N M_1 \ldots M_l$ redex.
   Nazveme ho *hlavní (head) redex*. A nebo,
2. pro žádné $l \leq k$ není $N M_1 \ldots M_l$ redex.
   Pak řekneme, že term je *hlavní normální forma (head normal form)*.

(Nakreslit obrázek.)

Hlavní normální forma II

- Pro klasický lambda kalkulus to znamená, že N je
  - proměnná (pak je term HNF), a nebo
  - NM_1 je β-redex.
- Pro funkcionální jazyk s dalšími zabudovanými funkcemi a datatovými typy je term v HNF také když N je datový objekt nebo zabudovaná funkce, která nemá dostatek argumentů pro vyhodnocení.

Hlavní normální forma – redukce

Definition

Redukční strategii, která vždy redukuje hlavní redex nazveme *hlavní*.

Theorem

Lambda term má HNF právě když se hlavní redukční strategie zastaví.

Důkaz.

Viz. *Curry's standardization theorem* v [1]. □

Definition

Term může mít více různých HNF. Tu, kterou získáme hlavní redukční strategií, nazveme *principiální (principal) HNF*.

Slabá hlavní normální forma I

Weak head normal form

Definition (weak head normal form)

Každý lambda term lze zapsat ve tvaru

$$NM_1 \ldots M_k$$

kde N není aplikace, $k \geq 0$, a platí, že

1. pro nějaké $l \leq k$ je $NM_1 \ldots M_l$ redex. Nazveme ho *slabý hlavní (weak head) redex*, a nebo
2. pro žádné $l \leq k$ není $NM_1 \ldots M_l$ redex.
   Pak řekneme, že term je *slabá hlavní normální forma (weak head normal form)*.

(Nakreslit obrázek.)

Slabá hlavní normální forma II

Weak head normal form

- Pro klasický lambda kalkulus to znamená, že N je
  - proměnná (pak je term WHNF), nebo
  - N je lambda abstrakce a $k = 0$ (term je také WHNF), a nebo
  - NM_1 je β-redex.
- Pro funkcionální jazyk s dalšími zabudovanými funkcemi a datatovými typy je term v WHNF také v případech, kdy N datový objekt nebo zabudovaná funkce, která nemá dostatek argumentů pro vyhodnocení, a nebo lambda abstrakce a $k = 0$.

Slabá hlavní normální forma – redukce

Definition

Redukční strategii, která vždy redukuje slabý hlavní redex nazveme *slabou hlavní*.

Theorem

Lambda term má WHNF právě když se slabá hlavní redukční strategie zastaví.

Důkaz.

Pokud je N je lambda abstrakce a $k = 0$, je tvrzení zřejmé.
V opačném případě splývá HNF a WHNF. □

Definition

Term může mít více různých WHNF. Tu, kterou získáme slabou hlavní redukční strategií, nazveme *principiální (principal) WHNF*.

Slabá hlavní normální forma – využití

- WHNF se většinou používá pro vyhodnocování výrazů ve funkcionálních jazycích.
- U WHNF se totiž vyhneme problému s nutností přejmenování proměnných α-konverzí, na rozdíl od HNF. (Funkcionální program musí být kombinátor. Když si na začátku přejmenujeme všechny proměnné tak, aby byly různé, při hledání WHNF se vyhneme konfliktu v názvech proměnných.)
- Zajišťuje maximální lenost vyhodnocování, u HNF by docházelo k nadbytečnému vyhodnocování neúplně aplikovaných funkcí.
- U jazyků se striktním vyhodnocováním, které vyhodnocují termy do WHNF, lze odložit vyhodnocení výrazu jeho zabalením do lambda abstrakce s nějakou nepoužitou proměnnou:

$$M' = \lambda x.M$$

Výraz je pak vyhodnocen teprve až na aplikujeme M' na cokoliv, např.

$$M'0 = (\lambda x.M)0$$

Shrnutí normálních forem

- NF neobsahuje žádné redexy.
- HNF může mít redexy v argumentech nejvíce vnější funkce.
- WHNF může mít redexy jak v argumentech nejvíce vnější funkce, tak kdekoliv pod lambda abstrakcemi.

Poznámky k (slabé) hlavní normální formě

- HNF má teoretické uplatnění v teorii lambda kalkulu [1]. Lze například dokázat za jakých podmínek má term HNF, že kontrakcemi hlavních redexů dospějeme k normální formě, atd.
- WHNF má především praktické uplatnění v implementaci funkcionálních jazyků [4].

Literatura I

[1] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. Vol. 103. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984. URL: http://mathgate.info/cebrown/notes/barendregt.php.

[2] Henk Barendregt et al. ?Lambda Calculi with Types? In: *Handbook of Logic in Computer Science*. Oxford University Press, 1992, pp. 117–309. URL: ftp://ftp.cs.kun.nl/pub/CompMath.Found/HBKJ.ps.Z.

[3] J. Lamping. ?An algorithm for optimal lambda calculus reduction? In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 16–30. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.2386&rep=rep1&type=pdf.

Literatura II

[4] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/papers.html.

[5] C.P. Wadsworth. ?Semantics and pragmatics of the lambda-calculus? PhD thesis. University of Oxford, 1971.

Normální formy

Cvičení

Najděte nějaké lambda termy M a N takové, že:

1. *N nemá nf ale MN má nf.*
2. *M nemá nf ale MN má nf.*
3. *M nemá hnf ale MN má nf.*
4. *M nemá whnf ale MN má nf.*

Zapište tyto termy jako programy v Haskellu. Pokud takové termy neexistují, dokažte to.

Páry a alternativy

- Implementujte páry pomocí lambda termů. Tedy funkci *pair* kterou lze aplikovat na 2 argumenty, a funkce *first* a *second* takové, že platí:
  $first\ (pair\ M\ N) \twoheadrightarrow_\beta M$, a
  $second\ (pair\ M\ N) \twoheadrightarrow_\beta N$
- Nápověda: typ *pair* buď $a \to b \to (a \to b \to c) \to c$.

Dvojitá rekurze

1. Implementujte Hofstadterovy mužské a ženské posloupnosti jako funkce na číslech.
2. Implementujte je tak, abyste použili rekurzivní *let* pouze s jednou rekurzivní deklarací, místo dvou.
3. Implementujte je pomocí kombinátoru Y.

Faktoriál na různé způsoby I

Faktoriál obyčejnou rekurzí:

```
factorial :: Integer → Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Tail-rekurzivně:

```
factorial :: Integer → Integer
factorial =
  let
    fc r 0 = r
    fc r n = fc (r * n) (n - 1)
  in fc 1
```

Faktoriál na různé způsoby II

Pomocí kombinátoru Y:

$$
\begin{aligned}
&\mathit{factorial1} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{factorial1} = \lambda n \to \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * \mathit{factorial1}\ (n-1) \\
&\mathit{factorial2} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{factorial2} = \\
&\quad \textbf{let} \\
&\qquad f = \lambda n \to \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * f\ (n-1) \\
&\quad \textbf{in } f \\
&y :: (a \to a) \to a \\
&y = \lambda f \to f\ (y\ f) \\
&\mathit{factorial3} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{factorial3} = y\ (\lambda f \to (\lambda n \to \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * f\ (n-1))) \\
&\qquad \text{-- (Integer -> Integer) -> (Integer -> Integer)}
\end{aligned}
$$

Faktoriál na různé způsoby III

Pomocí kombinátoru Y tail-rekurzivně:

```
y :: (a → a) → a
y f = f (y f)
factorial :: Integer → Integer
factorial =
  let
    f :: (Integer → Integer → Integer)
      → (Integer → Integer → Integer)
    f rec r 0 = r
    f rec r n = rec (r * n) (n - 1)
  in y f 1
```

Faktoriál na různé způsoby IV

Striktní jazyk – Scala – zacyklí se v kombinátoru *Y*

```
object Factorial3Bad
{
  def y[A](f: A => A): A =
    f(y(f));

  def frec(rec: Int => Int)(n: Int): Int =
    n match {
      case 0  => 1
      case m  => m * rec (m-1)
    }

  def factorial(n: Int): Int =
    y(frec)(n);

  def main(args: Array[String])
  {
    print("Factorial of 5: ");
    print(factorial(5));
    print("\n");
  }
}
```

Faktoriál na různé způsoby V

Striktní jazyk – Scala – simulace líného vyhodnocování pomocí η expanze

```
object Factorial3
{
  def y[A](f: (Unit => A) => A): (Unit => A) =
    Unit => f(y(f));

  def frec(rec: Unit => (Int => Int))(n: Int): Int =
    n match {
      case 0  => 1
      case m  => m * rec()(m-1)
    }

  def factorial(n: Int): Int =
    y(frec)()(n);

  def main(args: Array[String])
  {
    print("Factorial of 5: ");
    print(factorial(5));
    print("\n");
  }
}
```

Dvojitě vzájemná rekurze I

Viz. sekvence http://oeis.org/classic/A005379 a http://oeis.org/classic/A005378.
Klasická verze pomocí *let*.

```
female :: Integer → Integer
female =
  let
    f :: Integer → Integer
    f 0  = 1
    f x  = (x :: Integer) − m (f (x − 1))
    m :: Integer → Integer
    m 0 = 0
    m x = x − f (m (x − 1))
  in f
```

Dvojitě vzájemná rekurze II

Varianta s pouze jednou rekurzivní 'let' deklarací za pomoci páru:

```
female' :: Integer → Integer
female' =
  let
    f1 :: (Integer → Integer)
       → (Integer → Integer)
       → Integer → Integer
    f1 frec mrec 0   = 1
    f1 frec mrec x   = x − mrec (frec (x − 1))

    m1 :: (Integer → Integer)
       → (Integer → Integer)
       → Integer → Integer
    m1 frec mrec 0 = 0
    m1 frec mrec x = x − frec (mrec (x − 1))
```

Dvojitě vzájemná rekurze III

$$
\begin{aligned}
&(f, m) = (f1\ f\ m, m1\ f\ m) \\
&\textbf{in}\ f
\end{aligned}
$$

Varianta s pouze jednou rekurzivní 'let' deklarací za pomoci kombinátoru Y a páry implementovanými pomocí funkcí:

$$
\begin{aligned}
&y :: (a \to a) \to a \\
&y\ f \quad = f\ (y\ f) \\
&pair \quad = \lambda x\ y\ f \to f\ x\ y \\
&first \quad = \lambda p \to p\ (\lambda x\ y \to x) \\
&second = \lambda p \to p\ (\lambda x\ y \to y) \\
&female'' :: Integer \to Integer \\
&female'' = \\
&\quad \textbf{let} \\
&\quad\quad f1 :: (Integer \to Integer)
\end{aligned}
$$

Dvojitě vzájemná rekurze IV

```
      → (Integer → Integer)
      → Integer → Integer
  f1 frec mrec 0 = 1
  f1 frec mrec x = x − mrec (frec (x − 1))
  m1 :: (Integer → Integer)
      → (Integer → Integer)
      → Integer → Integer
  m1 frec mrec 0 = 0
  m1 frec mrec x = x − frec (mrec (x − 1))
    -- (f, m) = (f1 f m, m1 f m)
in first (y (λp → let
  f = first p
  m = second p
  in pair (f1 f m) (m1 f m)))
```