Funkcionální programování

IA014

Přednášky:	čtvrtek 14:00, B204
Konsultace:	pondělí od 16:30, B419
Zkoušky:	červen 2009

Literatura

- Anthony J. Field, Peter G. Harrison: *Introduction to Functional Programming*, Addison Wesley, 1988
- Greg Michaelson: *An Introduction to Functional Programming through Lambda Calculus*, Addison Wesley, 1989
- Simon L. Peyton Jones: *The Implementation of Functional Programming Languages*, Prentice Hall, 1987
- Simon L. Peyton Jones, David Lester: *Implementation Functional Languages: a tutorial*

Informace, odkazy

`http://www.fi.muni.cz/~libor/vyuka/IA014/`

Náplň kursu

- Netypovaný a typovaný lambda kalkul a jejich vlastnosti. Silná normalizace, Churchova-Rosserova vlastnost.
- Rekurse, věta o pevném bodě.
- Jazyk PCF a jeho sémantika.
- Typy. Parametrický polymorfismus. Problém otypování: nalezení a ověření typu. Otypování v různých typových systémech.
- Kombinátorový počet. Kombinátory S, K, I. Kombinátory B, C.

- Implementace funkcionálních jazyků.
  Překlad definic podle vzoru, stážených klauzulí, intensionálních seznamů.
- Grafová redukce. G-stroj. Superkombinátory, vynášení.
- Optimální redukce, plná lenost, plně líné vynášení.

- Imperativní prvky, vstup/výstup, ošetření výjimek, nedeterminismus, přepisovatelná pole, stav. Pokračování.
- Monády. Monadický datový typ pro vstup/výstup. Monadické kombinátory pro syntaktickou analýzu.
- Funkcionální datové struktury.
  Binární haldy, binomiální haldy, vyhledávací stromy.

Převod konstrukcí funkcionálního jazyka do rozšířeného lambda kalkulu

```
let length []     = 0
    length (_:t) = 1 + length t
    u = [1,2,3]
  in length u
```

definice podle vzoru $\rightarrow$ definice s parametry

```
let length s = if  null s  then  0  else  1 + length (tail s)
    u = [1,2,3]
  in length u
```

`if` *je funkce*

```
let length s = if  (null s)  0  (1 + length (tail s))
    u = [1,2,3]
  in length u
```

„syntaktický cukr“ (infixové operátory, seznamová syntax . . .)

```
let length s = if  (null s)  0  ((+) 1 (length (tail s)))
    u = (:) 1 ((:) 2 ((:) 3 []))
  in length u
```

definice funkce do tvaru jméno = lambda abstrakce

```
let length = λs. if  (null s)  0  ((+) 1 (length (tail s)))
    u = (:) 1 ((:) 2 ((:) 3 []))
  in length u
```

rekursivní definice $\rightarrow$ *použití kombinátoru pevného bodu*

```
let length = Y(λℓ λs. if  (null s)  0  ((+) 1 (ℓ (tail s)))
    u = (:) 1 ((:) 2 ((:) 3 [])))
  in length u
```

(lokální) definice s let $\rightarrow$ *výraz s abstrakcí a aplikací*

```
(λlength λu. length u)
   (Y(λℓ λs. if  (null s)  0  ((+) 1 (ℓ (tail s)))))
   ( (:) 1 ((:) 2 ((:) 3 [])))
```

konstanty $\rightarrow$ *simulace v čistém lambda kalkulu*

Netypovaný lambda kalkul

Syntax

Dána spočetná množina symbolů, tzv. proměnných $x_1, x_2, x_3, \ldots, x, y, z, t, u, v, \ldots$

Množina Λ lambda termů (výrazů) je definována induktivně:

- Každá proměnná je term
- M, N termy $\Rightarrow MN$ term (aplikace)
- x proměnná, M term $\Rightarrow \lambda x.M$ term (abstrakce)

Konvence:

$$M_1 M_2 M_3 \ldots M_n \equiv (\ldots((M_1 M_2) M_3)\ldots) M_n$$

$$\lambda x_1 x_2 \ldots x_n.M \equiv \lambda x_1 \lambda x_2 \ldots \lambda x_n.M \equiv \lambda x_1 (\lambda x_2 (\ldots (\lambda x_n.M)\ldots))$$

Def: *Podterm*

Je-li M term, je M svým podtermem.

Term MN má podtermy M, N.

Term $\lambda x.M$ má podterm M.

Má-li term M podterm N a term N má podterm P, pak P je také podtermem termu M.

Def: *Volné proměnné*

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

Def: *Vázané proměnné*

$$\begin{aligned} \mathrm{BV}(x) &= \emptyset \\ \mathrm{BV}(MN) &= \mathrm{BV}(M) \cup \mathrm{BV}(N) \\ \mathrm{BV}(\lambda x.M) &= \mathrm{BV}(M) \cup \{x\} \end{aligned}$$

Def: *Substituce*

P . . . term

x . . . proměnná

$[P/x] : \Lambda \to \Lambda$ . . . substituce

$$
\begin{aligned}
[P/x]x &= P \\
y \neq x \Rightarrow [P/x]y &= y \\
[P/x](MN) &= ([P/x]M)([P/x]N) \\
[P/x]\lambda x.M &= \lambda x.M \\
y \neq x, y \notin \mathrm{FV}(P) \Rightarrow [P/x]\lambda y.M &= \lambda y.[P/x]M \\
y \neq x, y \in \mathrm{FV}(P) \Rightarrow [P/x]\lambda y.M &= \lambda y_1.[P/x]([y_1/y]M),
\end{aligned}
$$

kde y_1 je čerstvá.

Lemma: Substituce se uplatní, jen je-li substituováno za proměnnou, která je v termu volná:

$$\forall P, M \in \Lambda\, \forall x \notin \mathrm{FV}(M)\,.\,[P/x]M = M$$

Důsledek: Uzavřené termy jsou invariantní vůči substitucím:

$$\forall M \in \Lambda\, \forall \text{ substituci } \sigma : \Lambda \to \Lambda\,.\,\mathrm{FV}(M) = \emptyset \Rightarrow \sigma M = M$$

Sémantika

Sémantiku lze zadat buď denotačně ($[\![\_]\!] : \Lambda \dashrightarrow D$, kde D je *doména* hodnot), anebo operačním stylem: na termech zavedeme relaci jednokrokové redukce $\rightsquigarrow \subseteq \Lambda^2$ a vytvoříme *přepisovací systém*.

β-redukce: $(\lambda x.M)N \rightsquigarrow_\beta [N/x]M$ $\qquad (\mathrm{FV}(N) \cap \mathrm{BV}(M) = \emptyset)$

η-redukce: $\lambda x.Mx \rightsquigarrow_\eta M$ $\qquad$ pro $x \notin \mathrm{FV}(M)$

Jednokroková redukce $\rightsquigarrow$ je pak kompatibilní uzávěr relace $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$ vzhledem k relaci podterm:

Pro všechny termy M, N, P a všechny proměnné x

$$
\begin{aligned}
M \rightsquigarrow_\beta N &\Rightarrow M \rightsquigarrow N \\
M \rightsquigarrow_\eta N &\Rightarrow M \rightsquigarrow N \\
M \rightsquigarrow N &\Rightarrow MP \rightsquigarrow NP \\
M \rightsquigarrow N &\Rightarrow PM \rightsquigarrow PN \\
M \rightsquigarrow N &\Rightarrow \lambda x.M \rightsquigarrow \lambda x.N
\end{aligned}
$$

Vícekroková redukce $\rightsquigarrow^*$ je reflexivním a transitivním uzávěrem jednokrokové redukce.

α-konverze

Systematické přejmenování vázaných proměnných $\sim_\alpha$

Výpočetní ekvivalence termů

$\approx \; = \; (\sim_\alpha \cup \rightsquigarrow \cup \rightsquigarrow^{-1})^*$

Redex, *redukt (kontraktum)*

Nechť M, N jsou termy, $M \rightsquigarrow N$, nechť existuje term U s jediným výskytem proměnné x tak, že $M = [P/x]U$, $N = [Q/x]U$, $P \rightsquigarrow_\beta Q$, resp. $P \rightsquigarrow_\eta Q$. Pak P je redex (β-redex resp. η-redex) v M, a Q je jeho redukt (β-redukt resp. η-redukt).

Fundovanost

Relace $\twoheadrightarrow \subseteq A^2$ je *fundovaná*, právě když neexistuje nekonečná posloupnost $(a_1, a_2, \ldots) \in A^\omega$ taková, že $a_1 \twoheadrightarrow a_2 \twoheadrightarrow \cdots$.

Normální forma prvku $a \in A$ je takový prvek $b \in A$, pro který platí $a \twoheadrightarrow^* b$

a současně neexistuje prvek $c \in A$, pro který by platilo $b \twoheadrightarrow c$.

Příklad: Term $(\lambda x \lambda y.y\, x)\; t\; u$ má normální formu $u\; t$.

Term $(\lambda y.u)\,((\lambda x.x\, x)(\lambda x.x\, x))$ má n. f. u, ale i nekonečnou redukční posloupnost.

Term $(\lambda x.x\, x)(\lambda x.x\, x)$ nemá normální formu.

Důsledek: Relace redukce $\rightsquigarrow$ v netypovaném lambda kalkulu není fundovaná.

Konfluence

Relace $\rightarrow \subseteq A^2$ je *konfluentní*, právě když $\rightarrow^* \circ \leftarrow^* \subseteq \leftarrow^* \circ \rightarrow^*$

Věta: Relace redukce $\rightsquigarrow$ v lambda kalkulu je konfluentní.

Důsledek: Každý term v Λ má nejvýše jednu normální formu.

Redukční strategie

Redukční strategie je částečné zobrazení $\varphi : \Lambda \dashrightarrow \Lambda$, které je definováno právě na reducibilních termech, a pro každý reducibilní term $M \in \Lambda$ platí $M \rightsquigarrow \varphi(M)$.

Nechť φ je redukční strategie. Pak *φ-redukční posloupnost* termu M je

buď konečná posloupnost $M_0, M_1, \ldots, M_k$ taková, že $M_0 = M$, M_k je n.f.,
$\forall i, 0 < i \leq k . M_i = \varphi(M_{i-1})$,

anebo nekonečná posloupnost $M_0, M_1, M_2, \ldots$ taková, že $M_0 = M$,
$\forall i > 0 . M_i = \varphi(M_{i-1})$.

Normální redukční strategie

Normální redukční strategie je definována:

$$\varphi_{\mathrm{n}}(M) = N \;\Leftrightarrow\; M \rightsquigarrow N \text{ v nejlevějším } \textit{vnějším} \text{ redexu}$$

Věta: Má-li term M normální formu N, pak existuje $k \in \mathbb{N}$ takové, že $N = \varphi_{\mathrm{n}}^{k}(M)$.

Tedy: může-li výpočet skončit, pak při normální strategii skončí.

Důkaz viz [Barendregt – Lambda Calculus]

Striktní redukční strategie

Striktní redukční strategie je definována:

$$\varphi_{\mathrm{s}}(M) = N \;\Leftrightarrow\; M \rightsquigarrow N \text{ v nejlevějším } \textit{vnitřním} \text{ redexu}$$

Věta: Má-li term M nekonečné redukční posloupnosti, pak $\varphi_{\mathrm{s}}^{k}(M)$ je definováno pro všechna $k \in \mathbb{N}$. Tedy: může-li se výpočet zacyklit, pak při striktní strategii se zacyklí.

Lambda term je v *hlavové normální formě* (*hnf*), právě když má tvar

$$\lambda x_1 \lambda x_2 \ldots \lambda x_k . v\ M_1\ \ldots\ M_m$$

kde $k \geq 0$,

$m \geq 0$,

v je proměnná nebo konstanta, a

$v\ M_1\ \ldots\ M_p$ není redex pro žádné $p \leq m$.

Tedy hnf je term, který tvoří proměnná či konstanta aplikovaná na příliš málo argumentů (případ $k = 0$), anebo lambda abstrakce, jejíž tělo není reducibilní (případ $k > 0$).

Slabá hlavová normální forma (*whnf*) je lambda term ve tvaru

$$v\ M_1\ \ldots\ M_n$$

kde $n \geq 0$,

v je proměnná, konstanta nebo lambda abstrakce, a

$v\ M_1\ \ldots\ M_p$ není redex pro žádné $p \leq n$.

Tedy whnf je term, který tvoří proměnná či konstanta aplikovaná na příliš málo argumentů, anebo lambda abstrakce.

Každá hnf je také whnf. Opak neplatí, např. $\lambda x.(\lambda y.y)z$ není hnf.

Poznámka: Prakticky používané systémy redukují do slabé hlavové normální formy, tj. neredukují v tělech lambda abstrakcí.

Kódování dat v čistém lambda kalkulu

Logické hodnoty a spojky

Uspořádané dvojice

Uspořádané trojice

Seznamy

Přirozená čísla a aritmetické operace

Stromy

Pravdivostní hodnoty

$$\begin{aligned} \text{True} &= \lambda x \lambda y.x \\ \text{False} &= \lambda x \lambda y.y \end{aligned}$$

Logické spojky

$$\begin{aligned} \text{not} &= \lambda y.y\ \text{False}\ \text{True} \\ \text{if} &= \lambda x \lambda y \lambda z.x\ y\ z \\ \text{and} &= \lambda u \lambda v.u\ v\ \text{False} \\ \text{or} &= \lambda u \lambda v.u\ \text{True}\ v \end{aligned}$$

Uspořádané dvojice

$$
\begin{aligned}
(M, N) &= \lambda z.z\ M\ N \\
\text{fst} &= \lambda p.p\ \text{True} \\
\text{snd} &= \lambda p.p\ \text{False}
\end{aligned}
$$

Uspořádané trojice

$$
\begin{aligned}
(M, N, P) &= \lambda z.z\ M\ N\ P \\
\text{proj31} &= \lambda t.t\ (\lambda x \lambda y \lambda z.x) \\
\text{proj32} &= \lambda t.t\ (\lambda x \lambda y \lambda z.y) \\
\text{proj33} &= \lambda t.t\ (\lambda x \lambda y \lambda z.z)
\end{aligned}
$$

Seznamy

$$
\begin{aligned}
[\,] &= \lambda z.z\ \mathsf{True}\ \mathsf{False}\ \mathsf{False} \\
(:) &= \lambda x \lambda y \lambda z.z\ \mathsf{False}\ x\ y \\
\mathsf{head} &= \lambda p.p(\lambda x \lambda y \lambda z.y) \\
\mathsf{tail} &= \lambda p.p(\lambda x \lambda y \lambda z.z) \\
\mathsf{null} &= \lambda p.p(\lambda x \lambda y \lambda z.x)
\end{aligned}
$$

Binární stromy s ohodnocenými uzly

$$
\begin{aligned}
\text{empty} &= \lambda g.g\ \text{True}(\lambda x.x)(\lambda x.x)(\lambda x.x) \\
\text{node} &= \lambda x\lambda y\lambda z\lambda g.g\ \text{False}\ x\ y\ z \\
\text{isempty} &= \lambda t.t\ (\lambda u\lambda x\lambda y\lambda z.u) \\
\text{root} &= \lambda t.t\ (\lambda u\lambda x\lambda y\lambda z.x) \\
\text{left} &= \lambda t.t\ (\lambda u\lambda x\lambda y\lambda z.y) \\
\text{right} &= \lambda t.t\ (\lambda u\lambda x\lambda y\lambda z.z)
\end{aligned}
$$

Kódování přirozených čísel pomocí Churchových číslovek

$$
\begin{aligned}
\bar{0} &= \lambda s\ z.z \\
\bar{1} &= \lambda s\ z.s\ z \\
\bar{2} &= \lambda s\ z.s\ (s\ z) \\
\bar{3} &= \lambda s\ z.s\ (s\ (s\ z)) \\
\bar{n} &= \lambda s\ z.s^n\ z \\
\mathsf{iszero} &= \lambda n.n\ (\lambda x.\mathsf{False})\ \mathsf{True} \\
\mathsf{succ} &= \lambda n.\lambda s\ z.n\ s\ (s\ z) \\
\mathsf{add} &= \lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z) \\
\mathsf{mult} &= \lambda m\ n.\lambda s\ z.m\ (n\ s)\ z \\
\mathsf{pow} &= \lambda m\ n.\lambda s\ z.n\ m\ s\ z \quad (= \lambda m\ n.n\ m)
\end{aligned}
$$

$$
\begin{aligned}
\text{succ } \bar{n} &\rightsquigarrow \lambda s\ z.\bar{n}\ s\ (s\ z) \\
&= \lambda s\ z.(\lambda s\ z.s^n\ z)\ s\ (s\ z) \\
&\rightsquigarrow \lambda s\ z.s^n\ (s\ z) \\
&= \overline{n+1}
\end{aligned}
$$

$$
\begin{aligned}
\text{add } \overline{m}\ \bar{n} &\rightsquigarrow \lambda s\ z.(\lambda s\ z.s^m\ z)\ s\ ((\lambda s\ z.s^n\ z)\ s\ z) \\
&\rightsquigarrow \lambda s\ z.(\lambda s\ z.s^m\ z)\ s\ (s^n\ z) \\
&\rightsquigarrow \lambda s\ z.s^m\ (s^n\ z) \\
&= \overline{m+n}
\end{aligned}
$$

$$
\begin{aligned}
\text{mult } \overline{m}\ \bar{n} &\rightsquigarrow \lambda s\ z.\overline{m}\ (\bar{n}\ s)\ z \\
&= \lambda s\ z.(\lambda s\ z.s^m\ z)((\lambda s\ z.s^n\ z)\ s)\ z \\
&\rightsquigarrow \lambda s\ z.(\lambda s\ z.s^m\ z)\ (\lambda z.s^n\ z)\ z \\
&= \lambda s\ z.(\lambda s\ z.s^m\ z)s^n\ z \\
&\rightsquigarrow \lambda s\ z.s^{mn}\ z \\
&= \overline{mn}
\end{aligned}
$$

$$
\begin{aligned}
\text{pow } \overline{m}\ \bar{n} &\rightsquigarrow \lambda sz.\bar{n}\ \overline{m}\ s\ z \\
&= \lambda sz.(\lambda sz.s^n\ z)(\lambda sz.s^m\ z)\ s\ z \\
&\rightsquigarrow \lambda sz.(\lambda sz.s^m\ z)^n\ s\ z \\
&\rightsquigarrow \lambda sz.(\lambda sz.s^m\ z)^{n-1}\ s^m\ z \\
&\rightsquigarrow \lambda sz.(\lambda sz.s^m\ z)^{n-2}\ s^{m^2}\ z \\
&\rightsquigarrow \lambda sz.(\lambda sz.s^m\ z)^{n-3}\ s^{m^3}\ z \\
&\ \ \vdots \\
&\rightsquigarrow \lambda sz.(\lambda sz.s^m\ z)\ s^{m^{n-1}}\ z \\
&\rightsquigarrow \lambda sz.s^{m^n}\ z \\
&= \overline{m^n}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.f\ (n\ a\ b) \\
a &= \lambda q.p\ (g\ q)\ (\mathsf{succ}(g\ q)) \\
b &= p\ \bar{0}\ \bar{0} \\
p &= \lambda x\ y\ z.z\ x\ y \\
f &= \lambda z.z\ \mathsf{True} \\
g &= \lambda z.z\ \mathsf{False}
\end{aligned}
$$

Věta o pevném bodě

Věta: Pro každý term F existuje term X takový, že $F\ X \approx X$.

Důkaz: Položme $\mathsf{Y} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$, $X = \mathsf{Y}\ F$.
Pak $FX \leftsquigarrow \circ \rightsquigarrow \circ \rightsquigarrow\ X$

Poznámka: Tedy každý term F má pevný bod $\mathsf{Y}\ F$. Termu Y se říká kombinátor pevného bodu.

Použití

Nechť M je term s volnými výskyty proměnné z a máme rekursivní definici tvaru $z = M$. Utvoříme term $F = \lambda z.M$, tj. rekursivně definovanou proměnnou z (volnou v M) vážeme abstrakcí, takže v F už volná není. Původní rekursivní definice je ekvivalentní zápisu $z \approx F\ z$, tj. definovaná proměnná z musí vyhovovat této ekvivalenci. Podle věty o pevném bodě je $z = \mathsf{Y}\ F = \mathsf{Y}(\lambda z.M)$

$$
\begin{aligned}
\textit{add} &\approx \lambda x \lambda y.\text{if iszero } x \text{ then } y \text{ else } \textit{add}\,(\text{pred } x)(\text{succ } y) \\
\textit{add} &\approx \big(\lambda f \lambda x \lambda y.\text{if iszero } x \text{ then } y \text{ else } f(\text{pred } x)(\text{succ } y)\big)\ \textit{add} \\
\textit{add} &= \mathsf{Y}\big(\lambda f \lambda x \lambda y.\text{if iszero } x \text{ then } y \text{ else } f\ (\text{pred } x)(\text{succ } y)\big) \\
\textit{mult} &= \mathsf{Y}\big(\lambda f \lambda x \lambda y.\text{if iszero } x \text{ then } 0 \text{ else } \textit{add}\, y\ (f(\text{pred } x)\, y)\big) \\
&= \mathsf{Y}\Big((\lambda f \lambda x \lambda y.\text{if iszero } x \text{ then } 0 \text{ else} \\
&\qquad \big(\mathsf{Y}(\lambda g\ \lambda u \lambda v.\text{if iszero } u \text{ then } v \text{ else } g\ (\text{pred } u)(\text{succ } v))\big)\ y \\
&\qquad (f(\text{pred } x) y)\Big) \\
\textit{fact} &= \mathsf{Y}\big(\lambda f \lambda n.\text{if iszero } n \text{ then } 1 \text{ else } \textit{mult}\, n\ (f(\text{pred } n))\big) \\
\textit{zeroes} &= \mathsf{Y}(\lambda s.(:)\ \bar{0}\ s)
\end{aligned}
$$

Věta o vícenásobném pevném bodě

Věta: Pro každých n termů $F_1, F_2, \ldots, F_n$ existuje n termů $X_1, X_2, \ldots, X_n$ tak, že

$$\begin{aligned} X_1 &\approx F_1\, X_1\, X_2\, \ldots\, X_n \\ X_2 &\approx F_2\, X_1\, X_2\, \ldots\, X_n \\ &\vdots \\ X_n &\approx F_n\, X_1\, X_2\, \ldots\, X_n \end{aligned}$$

Vícenásobná (vzájemná) rekurse

$$\begin{array}{lcl} X_1 & = & M_1 \\ & \vdots & \\ X_n & = & M_n \end{array}$$

$$\begin{array}{lcl} \mathbb{X} & \approx & (M_1, \ldots, M_n) \\ \mathbb{X} & \approx & \big(F_1\, X_1\, \ldots\, X_n,\, \ldots,\, F_n\, X_1\, \ldots\, X_n\big) \\ \mathbb{X} & \approx & \big(F_1(p_1\, \mathbb{X}) \ldots (p_n\, \mathbb{X}),\, \ldots,\, F_n(p_1\, \mathbb{X}) \ldots (p_n\, \mathbb{X})\big) \\ \mathbb{X} & \approx & \Big(\lambda z.\big(F_1(p_1\, z) \ldots (p_n\, z),\, \ldots,\, F_n(p_1\, z) \ldots (p_n\, z)\big)\Big)\, \mathbb{X} \end{array}$$

kde $\mathbb{X} = (X_1, \ldots, X_n), \quad F_i = \lambda x_1 \ldots x_n.[x_1/X_1, \ldots, x_n/X_n]M_i,$

p_j jsou selektory.

Podle Věty o (jednoduchém) pevném bodě je

$$\mathbb{X} = \mathsf{Y}\Big(\lambda z.\big(F_1(p_1\, z) \ldots (p_n\, z),\, \ldots,\, F_n(p_1\, z) \ldots (p_n\, z)\big)\Big)$$

Lambda kalkul s konstantami

Do jazyka přidáme nejvýše spočetně mnoho konstant $c_1, c_2, \ldots$

Každá taková konstanta je term.

Definici jedno- a vícekrokové redukce rozšíříme o *δ-redukci*:

Pro každou konstantu c_j zavedeme tzv. δ-pravidla tvaru

$$\frac{M_1 \rightsquigarrow N_1 \qquad \ldots \qquad M_k \rightsquigarrow N_k}{M \rightsquigarrow N}$$

Pokud přidané konstanty λ-definují vyčíslitelné funkce, tak se vyjadřovací síla takto rozšířeného lambda kalkulu nezvýší.

Příklad: Přidáme konstanty false, true, not a δ-pravidla

$$\frac{}{\text{not false} \rightsquigarrow \text{true}} \qquad \frac{}{\text{not true} \rightsquigarrow \text{false}} \qquad \frac{M \rightsquigarrow N}{\text{not } M \rightsquigarrow \text{not } N}$$

Vyčíslitelné funkce

Číselné funkce: $\mathbb{N}^n \dashrightarrow \mathbb{N}$ (prvního řádu)

Def: Primitivně vyčíslitelné jsou funkce

- $\mathsf{z}_n : \mathbb{N}^n \to \mathbb{N}, \ \ \forall x \in \mathbb{N}.\ \mathsf{z}_n(x) = 0$ (nulová)
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$ (následník)
- $\mathsf{proj}_{n,k} : \mathbb{N}^n \to \mathbb{N}, \ \ \mathsf{proj}_{n,k}(x_1, \ldots, x_n) = x_k$ (projekce)
- Jsou-li $g : \mathbb{N}^m \to \mathbb{N}$, $h_1, \ldots, h_m : \mathbb{N}^n \to \mathbb{N}$ primitivně vyčíslitelné, pak $f : \mathbb{N}^n \to \mathbb{N}$ taková, že
  $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$,
  je primitivně vyčíslitelná. (dosazení)

- Jsou-li $g\colon \mathbb{N}^n \to \mathbb{N},\ h\colon \mathbb{N}^{n+2} \to \mathbb{N}$ primitivně vyčíslitelné, pak $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ taková, že

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(y+1, x_1, \ldots, x_n) &= h(f(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n), \end{aligned}$$

  je primitivně vyčíslitelná. (primitivní rekurse)

- Každá primitivně vyčíslitelná funkce je *vyčíslitelná*.

- Je-li funkce $g\colon \mathbb{N}^{n+1} \dashrightarrow \mathbb{N}$ vyčíslitelná, pak $f\colon \mathbb{N}^{n+1} \dashrightarrow \mathbb{N}$ taková, že $f(y, x_1, \ldots, x_n) = \mu z.(g(z, x_1, \ldots, x_n) = y)$, je vyčíslitelná. ($\mu$-rekurse)

Lambda vyčíslitelnost

Def: Parciální funkce $f : \mathbb{N}^n \dashrightarrow \mathbb{N}$ je *λ-definovatelná*, právě když existuje term $F \in \Lambda$ takový, že pro všechny n-tice $(k_1, \ldots, k_n)$ z definičního oboru funkce f platí $F\ \bar{k_1}\ \bar{k_2}\ \ldots\ \bar{k_n} \approx \overline{f(k_1, k_2, \ldots, k_n)}$
a pro n-tice $(k'_1, \ldots, k'_n)$, v nichž funkce f není definována, term $F\ \bar{k'_1}\ \bar{k'_2}\ \ldots\ \bar{k'_n}$ nemá normální formu.

Poznámka: Z konfluence vyplývá, že když normální forma existuje, tak $F\ \bar{k_1}\ \bar{k_2}\ \ldots\ \bar{k_n} \rightsquigarrow^* \overline{f(k_1, k_2, \ldots, k_n)}$.

Věta: Každá parciální vyčíslitelná funkce je λ-definovatelná.

Důkaz:

(1) Každá totální vyčíslitelná funkce je λ-definovatelná:

nula

následník

projekce

dosazení: Jsou-li funkce $g\colon \mathbb{N}^m \to \mathbb{N}, h_1, \ldots, h_m\colon \mathbb{N}^n \to \mathbb{N}$ λ-definovatelné pomocí termů $G, H_1, \ldots, H_m$, pak funkce $f\colon \mathbb{N}^n \to \mathbb{N}$ taková, že $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$, je λ-definovatelná pomocí termu $F = \lambda x_1 \ldots x_n . G(H_1\ x_1\ \ldots\ x_n) \ldots (H_m\ x_1\ \ldots\ x_n)$.

primitivní rekurse: Jsou-li $g\colon \mathbb{N}^n \to \mathbb{N}, h\colon \mathbb{N}^{n+2} \to \mathbb{N}$ λ-definovatelné pomocí termů G, H, pak funkce $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ taková, že

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(y+1, x_1, \ldots, x_n) &= h(f(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n), \end{aligned}$$

je λ-definovatelná pomocí termu

$$F = \mathsf{Y}\big(\lambda f y x_1 \ldots x_n.\mathsf{if}\,(\mathsf{iszero}\ y)(G\ x_1\ \ldots\ x_n) \\ (H(f(\mathsf{pred}\ y)x_1\ \ldots x_n)(\mathsf{pred}\, y)x_1\ \ldots\ x_n))\big)$$

obecná (μ) rekurse: Je-li $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ λ-definovatelná pomocí termu G, pak funkce $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ totální a taková, že

$f(y, x_1, \ldots, x_n) = \mu z.(g(z, x_1, \ldots, x_n) = y)$,

je λ-definovatelná pomocí termu

$F = \lambda y x_1 \ldots x_n.\mathsf{Y}\big(\lambda h z.\mathsf{if}\,(\mathsf{eq}\ y\ (G\ z\ x_1\ \ldots\ x_n))\ z\ (h\ (\mathsf{succ}\ z))\big)\ \bar{0}$

(2) Každá parciální vyčíslitelná funkce je λ-definovatelná:

Podobně jako pro totální funkce, ale je třeba „nasimulovat striktní redukci“, aby termy, které striktně nenormalizují, byly převedeny na termy bez nf.

Věta: Každá λ-definovatelná funkce je vyčíslitelná:

Technické detaily důkazu pracné a zdlouhavé, idea prostá: Sestrojíme TM (nebo RAM, program v Haskellu, . . .), který pro každý term F, který λ-definuje funkci $f : \mathbb{N}^n \to \mathbb{N}$, a pro každou n-tici $(x_1, \ldots, x_n) \in \mathbb{N}^n$ interpretuje aplikaci $F\ \bar{x}_1\ \ldots\ \bar{x}_n$ pomocí normální redukční strategie.

Jednoduše typovaný lambda kalkul

Typy

Základní typ: ι

Jsou-li σ, τ typy, pak $\sigma \to \tau$ je typ.

Termy

Pro každý typ σ definujeme množinu Λ^σ termů typu σ:

proměnné $x^\sigma, x_1^\sigma, \ldots, y^\sigma, \cdots \in \Lambda^\sigma$

$\big($**konstanty** $c_1^\sigma, \ldots, c_k^\sigma \in \Lambda^\sigma\big)$

abstrakce $x^\sigma \in \Lambda^\sigma$ proměnná, $M \in \Lambda^\tau$, pak $\lambda x^\sigma.M \in \Lambda^{\sigma\to\tau}$

aplikace $M \in \Lambda^{\sigma\to\tau}$, $N \in \Lambda^\sigma$, pak $M\,N \in \Lambda^\tau$

Místo $M \in \Lambda^\sigma$ často píšeme $M : \sigma$.

Redukce $\rightsquigarrow$ se definuje stejně jako v netypovaném lambda kalkulu:

$$(\lambda x^{\sigma}.M)N \rightsquigarrow_{\beta} [N/x]M \qquad M \in \Lambda^{\tau}, N \in \Lambda^{\sigma}$$

$$\mathrm{FV}(N) \cap \mathrm{BV}(M) = \emptyset$$

$$\lambda x^{\sigma}.M\ x^{\sigma} \rightsquigarrow_{\eta} M \qquad \text{pokud } x^{\sigma} \notin \mathrm{FV}(M)$$

$$M \in \Lambda^{\sigma \to \tau}$$

Věta: Redukce v jednoduše typovaném lambda kalkulu je konfluentní.

Věta: Redukce v jednoduše typovaném lambda kalkulu je fundovaná.

Důkaz viz [H. Barendregt – Lambda Calculus, Its Syntax and Semantics].

Důsledek: V jednoduše typovaném lambda kalkulu *nelze* vyjádřit kombinátor pevného bodu Θ s vlastností

$$\forall F.\, \Theta F \rightsquigarrow^{*} F(\Theta F)$$

Normalizace v jednoduše typovaném lambda kalkulu (fundovanost redukce)

Věta: (o komutaci β-η)

$$\rightsquigarrow_\beta \circ \rightsquigarrow_\eta \subseteq \rightsquigarrow_\eta^* \circ \rightsquigarrow_\beta^*$$

Důkaz rozepsáním všech případů vzájemného zanoření β-redexů a η-redexů.

Věta: (o odložení η-redukcí)

$$\rightsquigarrow^* \subseteq \rightsquigarrow_\eta^* \circ \rightsquigarrow_\beta^*$$

Důkaz indukcí podle počtu η-redukcí následovaných β-redukcemi, užitím Věty o komutaci β-η.

Věta: Má-li term β-normální formu, pak má i normální formu.

Def: Stupeň typu

$$\delta(\iota) = 1$$
$$\delta(\sigma \to \tau) = \max\{\delta(\sigma), \delta(\tau)\} + 1$$

Def: Stupeň β-redexu

$$\delta\big((\lambda x^{\sigma}.M^{\tau})N^{\sigma}\big) = \delta(\sigma \to \tau)$$

Def: Stupeň termu $d(M)$ je 0, pokud term M neobsahuje β-redexy, jinak je $d(M)$ maximum ze stupňů všech β-redexů obsažených v termu M.

Poznámka: Pro β-redex R platí $\delta(R) \leq d(R)$.

Lemma: (o stupni substituce)

$$d([N/x^{\sigma}]M) \leq \max\{d(M), d(N), \delta(\sigma)\}$$

Důkaz: V termu $[N/x^{\sigma}]M$ jsou redexy

1. termu M (s N na místě x)
2. termu N (případně namnožené)
3. případně nové redexy tvaru $N\ Q$, pokud v M byl podterm $x\ Q$ a $N = \lambda y.P$

Věta: (o slabé normalizaci) Každý term jednoduše typovaného lambda kalkulu má normální formu

Důkaz: Nechť M je term a R je takový jeho redex maximálního stupně, který obsahuje redexy už jen nižších stupňů, tj.

(i) $\delta(R) = d(M)$

(ii) R neobsahuje redexy stupně $\delta(R)$

Při redukci termu M v redexu R:

- redexy vně R zůstanou
- redexy uvnitř R zůstanou, případně se i namnoží; ale dle Lemmatu o stupni substituce nevzroste $d(M)$
- redex R zmizí; je-li nahrazen jinými redexy, budou mít ostře nižší stupeň

Tedy: nevzroste stupeň termu a klesne počet redexů r_M stupně $d(M)$.

Neboli: přiřadíme-li každému termu M uspořádanou dvojici $p_M = (d(M), r_M)$, pak *při redukování vždy ve vnitřním redexu maximálního stupně* ostře klesá pořadí dvojice p_M v lexikografickém uspořádání.

Věta: (o silné normalizaci) Každý term jednoduše typovaného lambda kalkulu je silně normalizující, tj. redukce v jednoduše typovaném lambda kalkulu je fundovaná.

Důkaz: Zavedeme množiny R_σ pro každý typ σ:

$$\begin{aligned} R_\iota &= \{M :: \iota \mid \mathrm{SN}(M)\} \\ R_{\sigma\to\tau} &= \{M :: \sigma \to \tau \mid \forall N \in R_\sigma . M\, N \in R_\tau\} \end{aligned}$$

Vyšetřujeme následující implikace:

(i) $M \in R_\sigma \Rightarrow \mathrm{SN}(M)$

(ii) $M \in R_\sigma \,\&\, M \rightsquigarrow N \Rightarrow N \in R_\sigma$

(iii) M není abstrakce $\&\, \forall N.(M \rightsquigarrow N \Rightarrow N \in R_\sigma) \Rightarrow M \in R_\sigma$

PCF — příklad jednoduchého funkcionálního jazyka

Typy: základní typy Bool, Nat

pro každé dva typy σ, τ je $\sigma \to \tau$ typ

Místo $M \in \Lambda^\sigma$ píšeme $M :: \sigma$

Termy: pro každý typ σ máme spočetně mnoho proměnných $x_1^\sigma, x_2^\sigma, \ldots :: \sigma$

$x :: \sigma, M :: \tau \Rightarrow \lambda x.M :: \sigma \to \tau$

$M :: \sigma \to \tau, N :: \sigma \Rightarrow M\ N :: \tau$

n přirozené číslo $\Rightarrow \bar{n} :: \mathsf{Nat}$, $\mathsf{True} :: \mathsf{Bool}$, $\mathsf{False} :: \mathsf{Bool}$

$\mathsf{succ} :: \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{pred} :: \mathsf{Nat} \to \mathsf{Nat}$, $\mathsf{iszero} :: \mathsf{Nat} \to \mathsf{Bool}$

pro každý typ σ je $\mathsf{if}_\sigma :: \mathsf{Bool} \to \sigma \to \sigma \to \sigma$

pro každý typ σ je $\mathsf{Y}_\sigma :: (\sigma \to \sigma) \to \sigma$

Redukce: $(\lambda x.M)N \rightsquigarrow [N/x]M$

$\mathsf{if}_\sigma\ \mathsf{True}\ M\ N \rightsquigarrow M$, $\mathsf{if}_\sigma\ \mathsf{False}\ M\ N \rightsquigarrow N$

$M \rightsquigarrow N \Rightarrow \mathsf{if}_\sigma\ M\ P\ Q \rightsquigarrow \mathsf{if}_\sigma\ N\ P\ Q$

$\mathsf{succ}\ \bar{n} \rightsquigarrow \overline{n+1}$, $\mathsf{pred}\ \overline{n+1} \rightsquigarrow \bar{n}$, $(\mathsf{pred}\ \bar{0} \rightsquigarrow \mathsf{pred}\ \bar{0})$

$M \rightsquigarrow N \Rightarrow \mathsf{pred}\ M \rightsquigarrow \mathsf{pred}\ N$, $\mathsf{succ}\ M \rightsquigarrow \mathsf{succ}\ N$

$\mathsf{iszero}\ M \rightsquigarrow \mathsf{iszero}\ N$

$\mathsf{iszero}\ \bar{0} \rightsquigarrow \mathsf{True}$, $\mathsf{iszero}\ \overline{n+1} \rightsquigarrow \mathsf{False}$

$\mathsf{Y}_\sigma\ M \rightsquigarrow M(\mathsf{Y}_\sigma\ M)$

$M \rightsquigarrow N \Rightarrow M\ P \rightsquigarrow N\ P$

Def: Program v PCF je uzavřený term typu Bool nebo Nat (základního typu).

Věta: PCF je vnitřně deterministický, tj. pro každý term M existuje nejvýše jeden term N takový, že $M \rightsquigarrow N$.

Důkaz indukcí podle délky termu M, rozepsáním jednotlivých případů (tvarů termu).

Důsledek: Redukce v PCF je současně redukční strategií.

Věta: Je-li M program v PCF různý od $\bar{n}$, True, False, pak ho lze redukovat právě jedním způsobem.

Důsledek: Je-li M program v PCF, pak nastane právě jedna z následující možností:

$$M \rightsquigarrow^* \bar{n} \qquad \text{pro nějaké } n \in \mathbb{N}$$

$$M \rightsquigarrow^* \text{True}$$

$$M \rightsquigarrow^* \text{False}$$

$$M \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \dots$$

Syntaktická rozšíření PCF

Lokální definice konstant

$$\text{let } x = M \text{ in } N \qquad \equiv (\lambda x.N)M$$

nebo obecněji funkcí

$$\text{let } f\ x_1\ \ldots\ x_k = M \text{ in } N \qquad \equiv (\lambda f.N)(\lambda x_1 \ldots x_k.M)$$

Rekursivní definice konstant a funkcí

$$\text{letrec } f\ x_1\ \ldots\ x_k = M \text{ in } N \qquad \equiv (\lambda f.N)\big(\mathsf{Y}(\lambda f \lambda x_1 \ldots x_k.M)\big)$$

Uspořádané dvojice

Konstruktor $(,)_{\sigma\tau}$ $\qquad (M, N) \equiv (,)_{\sigma\tau}\ M\ N$

Selektory $\mathsf{fst}_{\sigma\tau}$, $\mathsf{snd}_{\sigma\tau}$

δ-pravidla $M \rightsquigarrow P \Rightarrow (M, N) \rightsquigarrow (P, N)$

$N \rightsquigarrow P \Rightarrow (M, N) \rightsquigarrow (M, P)$ pro $M \not\rightsquigarrow$, tj. M tvaru

$x, y, \ldots, \lambda x.Q$, True, False, $\bar{0}, \bar{1}, \ldots,$

succ, pred, iszero, if_σ, Y_σ, if Q, if $Q\ Q'$

$\mathsf{fst}\,(M, N) \rightsquigarrow M$

$\mathsf{snd}\,(M, N) \rightsquigarrow N$

Definice podle vzoru

Def: Vzor v PCF je proměnná nebo True nebo False nebo $\bar{0}$ nebo $\bar{1}$ nebo ... nebo (p, q), kde p, q jsou vzory.

Syntaktická zkratka $\lambda(p_1, p_2).M \equiv \lambda z.(\lambda p_1 p_2.M)(\mathsf{fst}\ z)(\mathsf{snd}\ z)$

Lokální rekursivní definice

$$\begin{aligned} &\text{letrec} \quad f_1\ x_1\ \ldots\ x_n \quad = M_1 \\ &\qquad\qquad f_2\ x_1\ \ldots\ x_n \quad = M_2 \\ &\qquad\qquad\qquad\qquad \vdots \\ &\qquad\qquad f_m\ x_1\ \ldots\ x_n \quad = M_m \\ &\text{in } N \end{aligned}$$

$$\equiv \big(\lambda(f_1, \ldots, f_m).N\big)\big(\mathsf{Y}(\lambda(f_1, \ldots, f_m).(\lambda x_1 \ldots x_n.M_1, \ldots, \lambda x_1 \ldots x_n.M_m))\big)$$

Polymorfní lambda kalkul

Typy

- spočetná množina typových proměnných $\alpha_1, \alpha_2, \ldots$
- každá typová proměnná je typ
- jsou-li σ, τ typy, je také $\sigma \to \tau$ typ
- je-li α typová proměnná, σ typ, pak $\forall\alpha.\sigma$ je typ
- typy se vytvářejí volně

Poznámka: Girardův-Reynoldsův typový systém:

$$\sigma ::= \alpha \mid \sigma \to \sigma \mid \forall\alpha.\sigma$$

Hindleyho-Millnerův typový systém:

$$\sigma ::= \tau \mid \forall\alpha.\sigma$$

$$\tau ::= \alpha \mid \tau \to \tau$$

Termy

- každá proměnná x^σ je term typu σ
- $x :: \sigma$ proměnná, $M :: \tau$ term, pak $\lambda x.M :: \sigma \to \tau$ term
- $M :: \sigma \to \tau$, $N :: \sigma$ termy, pak $M\,N :: \tau$ je term
- $M :: \sigma$ term, pak $\Lambda\alpha.M :: \forall\alpha.\sigma$ (generalizace)
- $M :: \forall\alpha.\sigma$ term, τ typ, pak $M\tau :: [\tau/\alpha]\sigma$ (specializace)

Redukce

Zavádí se stejně jako v jednoduše typovaném lambda kalkulu, navíc B, H.

B-redukce: $(\Lambda\alpha.M)\tau \rightsquigarrow [\tau/\alpha]M$

H-redukce: $\Lambda\alpha.\,M\,\alpha \rightsquigarrow M$ pro $\alpha \notin \mathrm{FV}(M)$

Věta: Redukce v polymorfním lambda kalkulu je konfluentní a fundovaná.

Důkaz fundovanosti (SN) podal Dag Prawitz.

Def: Uzavřené typy – typové termy bez volných typových proměnných

Uzavřené hodnoty – ireducibilní uzavřené termy uzavřeného typu

n-ární typové konstruktory – typové termy s n volnými typovými proměnnými, nad nimiž je provedena typová abstrakce

Lemma:

Každý uzavřený term typu $\sigma \to \tau$ v normální formě má tvar $\lambda x.M$, kde $x :: \sigma$, $M :: \tau$.

Každý uzavřený term typu $\forall \alpha.\sigma$ v normální formě má tvar $\Lambda \alpha.M$, kde $M :: \sigma$.

Důkaz: obě tvrzení se dokazují naráz indukcí podle struktury termu.

Definice základních uzavřených typů, typových konstruktorů a jejich (uzavřených) hodnot v polymorfním lambda kalkulu

$$\mathsf{Bool} = \forall\alpha.\alpha \to \alpha \to \alpha$$

dvě uzavřené hodnoty:

$$\mathsf{True} = \Lambda\alpha\ \lambda x :: \alpha\ \lambda y :: \alpha.\ x$$

$$\mathsf{False} = \Lambda\alpha\ \lambda x :: \alpha\ \lambda y :: \alpha.\ y$$

$$\mathsf{not} = \lambda z :: \mathsf{Bool}.\ z\ \mathsf{Bool}\ \mathsf{False}\ \mathsf{True}$$

$$\mathsf{and} = \lambda u :: \mathsf{Bool}\ \lambda v :: \mathsf{Bool}.\ u\ \mathsf{Bool}\ v\ \mathsf{False}$$

$$\mathsf{or} = \lambda u :: \mathsf{Bool}\ \lambda v :: \mathsf{Bool}.\ u\ \mathsf{Bool}\ \mathsf{True}\ v$$

$$\mathsf{if} = \Lambda\alpha\ \lambda c :: \mathsf{Bool}\ \lambda x :: \alpha\ \lambda y :: \alpha.\ c\ \alpha\ x\ y$$

$$\mathsf{Nat} = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$$

spočetně mnoho uzavřených hodnot:

$$\bar{n} = \Lambda\alpha\ \lambda x :: \alpha\ \lambda y :: \alpha \to \alpha.\ \underbrace{y(y(\ldots(y(y}_{n\text{-krát}}\ x))\ldots))$$

$$\mathsf{succ} = \lambda n :: \mathsf{Nat}\ \lambda x :: \alpha\ \lambda y :: \alpha \to \alpha.\ y\ (n\ \alpha\ x\ y)$$

$$\mathsf{iszero} = \lambda n :: \mathsf{Nat}.\ n\ \mathsf{Bool}\ \mathsf{True}\ (\lambda z :: \mathsf{Bool}.\ \mathsf{False})$$

$$\sigma \times \tau = \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha$$

$$(M, N) = \Lambda\alpha\ \lambda x :: \sigma \to \tau \to \alpha.\ x\ M\ N, \text{ kde } M :: \sigma, N :: \tau$$

$$\mathsf{fst} = \Lambda\sigma\ \Lambda\tau\ \lambda p :: \sigma \times \tau.\ p\ \sigma\ (\lambda x :: \sigma\ \lambda y :: \tau.\ x)$$

$$\mathsf{snd} = \Lambda\sigma\ \Lambda\tau\ \lambda p :: \sigma \times \tau.\ p\ \sigma\ (\lambda x :: \sigma\ \lambda y :: \tau.\ y)$$

$$\mathsf{List}\ \sigma = \forall\alpha.\alpha \to (\sigma \to \alpha \to \alpha) \to \alpha$$

$$\mathsf{Nil} = \Lambda\sigma\ \Lambda\alpha\ \lambda x :: \alpha\ \lambda y :: \sigma \to \alpha \to \alpha.\ x$$

$$(:) = \Lambda\sigma\ \lambda u :: \sigma\ \lambda t :: \mathsf{List}\ \sigma\ \Lambda\alpha\ \lambda x :: \alpha\ \lambda y :: \sigma \to \alpha \to \alpha.\ y\ u\ (t\ \alpha\ x\ y)$$

Churchova vs. Curryho varianta polymorfního lambda kalkulu

Church – kalkul s explicitním typováním

$$
\begin{array}{rcl}
 & & x^{\sigma} \;::\; \sigma \\
M :: \tau & \Rightarrow & \lambda x :: \sigma.\, M \;::\; \sigma \to \tau \\
M :: \sigma \to \tau,\; N :: \sigma & \Rightarrow & M\, N \;::\; \tau \\
M :: \sigma & \Rightarrow & \Lambda\alpha.\, M \;::\; \forall\alpha.\sigma \\
M :: \forall\alpha.\sigma,\; \tau \text{ typ} & \Rightarrow & M\, \tau \;::\; [\tau/\alpha]\sigma
\end{array}
$$

Churchova vs. Curryho varianta polymorfního lambda kalkulu

Curry – kalkul s implicitním typováním

$$
\begin{array}{rcl}
 & & x :: \sigma \\
x :: \sigma,\ M :: \tau & \Rightarrow & \lambda x.\, M :: \sigma \to \tau \\
M :: \sigma \to \tau,\ N :: \sigma & \Rightarrow & M\ N :: \tau \\
M :: \sigma & \Rightarrow & M :: \forall \alpha.\sigma \\
M :: \forall \alpha.\sigma,\ \tau \text{ typ} & \Rightarrow & M :: [\tau/\alpha]\sigma
\end{array}
$$

Otypování

Def: Konečná množina $\Gamma = \{(x_1 :: \sigma_1), \ldots, (x_k :: \sigma_k)\}$ dvojic (proměnná :: typ) taková, že proměnné $x_1, \ldots, x_k$ jsou navzájem různé, se nazývá *typový kontext*.

Def: Nechť Γ je typový kontext, M je term, σ je typ. Zápis $\Gamma \vdash M :: \sigma$ se nazývá *otypování termu M v typovém kontextu Γ*.

Otypováním termu M rozumíme jeho otypování v prázdném typovém kontextu, tj. $\emptyset \vdash M :: \sigma$, kde σ je typ. Zapisujeme $\vdash M :: \sigma$.

Poznámka: Je zřejmé, že ne všechna otypování termu vyjadřují jeho skutečný typ. Za platná budeme považovat jen ta otypování, která lze odvodit z axiomů pomocí daných odvozovacích pravidel.

Def: Term M je *otypovatelný*, lze-li odvodit jeho otypování (v prázdném typovém kontextu). Typ z takového otypování je typem termu M.

Odvozovací pravidla pro Girardův-Reynoldsův typový systém

Axiom proměnné (VAR)

$$\frac{}{\Gamma, x :: \sigma \vdash x :: \sigma}$$

Axiomy konstant (CON)

$$\frac{}{\vdash 0 :: \mathsf{Nat}}, \quad \frac{}{\vdash \mathsf{False} :: \mathsf{Bool}}, \ldots$$

Zeslabení (W)

$$\frac{\Gamma \vdash M :: \sigma}{\Gamma, x :: \tau \vdash M :: \sigma} \qquad x \notin \mathrm{FV}(M)$$

Abstrakce (ABS)

$$\frac{\Gamma, x :: \sigma \vdash M :: \tau}{\Gamma \vdash \lambda x.M :: \sigma \to \tau}$$

Aplikace (APP)

$$\frac{\Gamma \vdash M :: \sigma \to \tau \qquad \Gamma \vdash N :: \sigma}{\Gamma \vdash M\ N :: \tau}$$

Generalizace (GEN)

$$\frac{\Gamma \vdash M :: \sigma}{\Gamma \vdash M :: \forall \alpha.\sigma} \qquad \alpha \notin FTV(\Gamma)$$

Specializace (SPEC)

$$\frac{\Gamma \vdash M :: \forall \alpha.\sigma}{\Gamma \vdash M :: [\tau/\alpha]\sigma}$$

Existují termy, které lze otypovat v Girardově-Reynoldsově typovém systému, ale nelze je otypovat v Hindleyho-Milnerově typovém systému.

$$\overbrace{(\lambda \underbrace{f}_{\forall\alpha.\alpha\to\alpha} .(\underbrace{f}_{\forall\alpha.\alpha\to\alpha} 0, \underbrace{f}_{\forall\alpha.\alpha\to\alpha} \mathsf{True}))}^{(\forall\alpha.\alpha\to\alpha)\to \mathsf{Nat}\times\mathsf{Bool}} \underbrace{(\lambda x.x)}_{\forall\alpha.\alpha\to\alpha} :: \mathsf{Nat} \times \mathsf{Bool}$$

Abychom mohli požívat výhod polymorfismu a přitom zachovali jednoduchý (Hindleyho-Milnerův) typový systém, rozšíříme kalkul o „let“ výrazy, které spojují abstrakci s aplikací:

Termy:

$x, y, \ldots$	proměnné
$c_1, c_2, \ldots, c_k$	konstanty
$M\ N$	aplikace
$\lambda x.M$	abstrakce
let $x = M$ in N	let výraz

Odvozovací pravidla pro Hindleyho-Milnerův typový systém a termy s let

Axiom proměnné (VAR)

$$\frac{}{\Gamma, x :: \sigma \vdash x :: \sigma}$$

Axiomy konstant (CON)

$$\frac{}{\vdash 0 :: \mathsf{Nat}}, \quad \frac{}{\vdash \mathsf{False} :: \mathsf{Bool}}, \ldots$$

Zeslabení (W)

$$\frac{\Gamma \vdash M :: \sigma}{\Gamma, x :: \tau \vdash M :: \sigma} \qquad x \notin \mathrm{FV}(M)$$

Abstrakce (ABS)

$$\frac{\Gamma, x :: \sigma \vdash M :: \tau}{\Gamma \vdash \lambda x.M :: \sigma \to \tau}$$

Aplikace (APP)

$$\frac{\Gamma \vdash M :: \sigma \to \tau \qquad \Gamma \vdash N :: \sigma}{\Gamma \vdash M\ N :: \tau}$$

Let-výraz (LET)

$$\frac{\Gamma \vdash N :: \sigma \qquad \Gamma, x :: \sigma \vdash M :: \tau}{\Gamma \vdash \text{let } x = N \text{ in } M :: \tau}$$

Generalizace (GEN)

$$\frac{\Gamma \vdash M :: \sigma}{\Gamma \vdash M :: \forall \alpha.\sigma} \qquad \alpha \notin \mathrm{FV}(\Gamma)$$

Specializace (SPEC)

$$\frac{\Gamma \vdash M :: \forall \alpha.\sigma}{\Gamma \vdash M :: [\tau/\alpha]\sigma} \qquad \tau \text{ typ}$$

$$\dfrac{\dfrac{\Gamma, x :: \forall\alpha.\sigma \vdash M :: \tau}{\Gamma \vdash \lambda x.M :: (\forall\alpha.\sigma) \to \tau}\text{ABS} \qquad \Gamma \vdash N :: \forall\alpha.\sigma}{\Gamma \vdash (\lambda x.M)N :: \tau}\text{APP}$$

$$\dfrac{\Gamma \vdash N :: \forall\alpha.\sigma \qquad \Gamma, x :: \forall\alpha.\sigma \vdash M :: \tau}{\Gamma \vdash \text{let } x = N \text{ in } M :: \tau}\text{LET}$$

$$(\lambda x.M)N \approx \text{let } x = N \text{ in } M$$

Robinsonův unifikační algoritmus

$\mathtt{mgu}\ \sigma\ \tau\ =$

jestliže $\sigma == \tau$

pak je výsledkem identická substituce

jinak: Nechť φ resp. ψ jsou typové podvýrazy v σ resp. τ na nejlevější nejvyšší pozici, na níž se σ liší od τ.

Jestliže φ a ψ jsou oba typové konstruktory,

pak typy σ a τ nejsou unifikovatelné (unifikátor je $\bot$)

jinak je-li φ typová proměnná a $\varphi \notin \mathrm{FV}(\psi)$,

pak unifikátor je $\mathtt{mgu}(\rho\sigma)(\rho\tau) \circ \rho$, kde $\rho = [\psi/\varphi]$,

je-li ψ typová proměnná a $\psi \notin \mathrm{FV}(\varphi)$,

pak unifikátor je $\mathtt{mgu}(\rho\sigma)(\rho\tau) \circ \rho$, kde $\rho = [\varphi/\psi]$;

pokud $\varphi \in \mathrm{FV}(\psi)$ nebo $\psi \in \mathrm{FV}(\varphi)$, pak typy σ a τ nejsou unifikovatelné (unifikátor je $\bot$).

Algoritmus otypování v kalkulu s let

typ $M = \mathsf{snd}(\text{typ'}(\emptyset, M))$

typ' $(\Gamma, M) = (\rho, \sigma)$ σ je nejobecnější typ termu M

ρ je substituce, taková, že $\rho\,\Gamma$ je nový typový kontext,

kde

(i) je-li M konstanta, položíme

$\rho = id$

$\sigma = \mathsf{type}\,M$

(ii) je-li $M = x$, položíme

$\rho = id$

pokud $(x :: \tau) \in \Gamma$, pak $\sigma = \tau$, jinak $\sigma = \alpha$, kde α je čerstvá

(iii) je-li $M = \lambda x.N$, položíme

$(\rho, \tau) = \text{typ'}(\Gamma \cup \{x :: \alpha\}, N)$, α je čerstvá

$\sigma = \rho(\alpha \to \tau)$

(iv) je-li $M = \text{let } x{=}N \text{ in } P$, položíme

$(\rho_1, \varphi) = \text{typ'}(\Gamma, N)$

$(\rho_2, \sigma) = \text{typ'}(\rho_1\, \Gamma \cup \{x :: \varphi\}, P)$

$\rho = \rho_2 \circ \rho_1$

(v) je-li $M = NP$, položíme

$(\rho_1, \tau) = \text{typ'}(\Gamma, N)$

$(\rho_2, \varphi) = \text{typ'}(\rho_1\Gamma, P)$

$\rho_3 = \mathtt{mgu}(\rho_2\tau)(\varphi \to \alpha)$, α čerstvá

$\rho = \rho_3 \circ \rho_2 \circ \rho_1$

$\sigma = \rho_3\alpha$

Příklad – typ termu `elem '*' "-*-"`

typ'($\emptyset$,`elem`)=($id, \alpha \to [\alpha] \to$ Bool)

typ' ($\emptyset$,`'*'`)=(id, Char)

typ' ($\emptyset$,`"-*-"`)=(id, [Char])

typ' ($\emptyset$, `elem '*'`) = $(\rho_3 \circ \rho_2 \circ \rho_1, \rho_3\beta)$, kde

$\rho_1 = id,\ \tau = \alpha \to [\alpha] \to$ Bool

$\rho_2 = id, \varphi =$ Char

$\rho_3 = [[\text{Char}] \to \text{Bool}/\beta]$

tj. $([[\text{Char}] \to \text{Bool}/\beta], [\text{Char}] \to \text{Bool})$

typ' ($\emptyset$, `elem '*' "-*-"`) = $(\rho_3 \circ \rho_2 \circ \rho_1, \rho_3\beta)$, kde

$\rho_1 = [[\text{Char}] \to \text{Bool}/\beta], \tau = [\text{Char}] \to \text{Bool}$

$\rho_2 = id, \varphi = [\text{Char}]$

$\rho_3 = \texttt{mgu}([\text{Char}] \to \text{Bool})([\text{Char}] \to \gamma) = [\text{Bool}/\gamma]$

tj. $([[\text{Char}] \to \text{Bool}/\beta, \text{Bool}/\gamma], \text{Bool}])$

Význam typování

- včasná detekce většiny programových chyb
- pro operace s hodnotami známých typů lze generovat efektivnější kód
- dokumentace

Použití v typovaných jazycích

- typ, který uživatel nedeklaruje, je odvozen a použit
- typ, který uživatel deklaruje je také odvozen a zkontroluje se, zda je stejný nebo obecnější než deklarovaný; deklarovaný je použit

Kombinátory S, K, I

Def: Kombinátorový term je term obsahující pouze kombinátory (S, K, I) a aplikace:

S, K, I jsou kombinátorové termy

jsou-li M, N, P kombinátorové termy, je S M N P kombinátorový term

jsou-li M, N kombinátorové termy, je K M N kombinátorový term

je-li M kombinátorový term, je I M kombinátorový term

Typy kombinátorů v typovaném kalkulu:

S $:: (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

K $:: \alpha \to \beta \to \alpha$

I $:: \alpha \to \alpha$

Def: Kombinátorová redukce

$$\mathsf{S}\ M\ N\ P \quad \rightsquigarrow_{\mathsf{S}} \quad M\ P\ (N\ P)$$

$$\mathsf{K}\ M\ N \quad \rightsquigarrow_{\mathsf{K}} \quad M$$

$$\mathsf{I}\ M \quad \rightsquigarrow_{\mathsf{I}} \quad M$$

$\rightsquigarrow$ je kompatibilní uzávěr relace $\rightsquigarrow_{\mathsf{S}} \cup \rightsquigarrow_{\mathsf{K}} \cup \rightsquigarrow_{\mathsf{I}}$

$\sim$ je reflexivní, symetrický a transitivní uzávěr relace $\rightsquigarrow$

Ekvivalence kombinátorů s λ-termy:

$\mathsf{S} \approx \lambda f \lambda g \lambda x.f\ x\ (g\ x)$

$\mathsf{K} \approx \lambda x \lambda y.x$

$\mathsf{I} \approx \lambda x.x$

Věta: $\sim \subseteq \approx$

Důkaz: překladem $\mathsf{S} \mapsto \lambda xyz.xz(yz)$, $\mathsf{K} \mapsto \lambda xy.x$.

Poznámka: Inkluze je dokonce vlastní, tj. $\sim \subset \approx$

Důkaz: $\mathsf{S}\,\mathsf{K}\,\mathsf{K} \not\sim \mathsf{S}\,\mathsf{K}\,\mathsf{S}$.

Přidání pravidla extensionality:

$$\frac{\forall P.M\ P\ \sim'\ N\ P}{M\ \sim'\ N} \qquad \frac{M\ \sim\ N}{M\ \sim'\ N}$$

Věta: $\sim' \subseteq \approx$

Důkaz indukcí přes počet použití pravidla extensionality.

Věta: Každý uzavřený λ-term M lze převést na $\approx$-ekvivalentní kombinátorový term N.

Důsledek: $\sim' = \approx$

Důkaz vyplývá z předchozích dvou vět a z transitivity $\approx$.

Kombinátorová báze je nejmenší (vzhledem k inkluzi) množina kombinátorů, jimiž lze vyjádřit všechny uzavřené λ-termy.

Věta: $\{\mathsf{S}, \mathsf{K}\}$ je kombinátorová báze.

Důkaz:

(i) Každý uzavřený term lze vyjádřit pomocí S, K, I

(ii) $\mathsf{I} \approx \mathsf{S}\,\mathsf{K}\,\mathsf{K}$

(iii) S nelze vyádřit pomocí K

(iv) K nelze vyádřit pomocí S

Příklad: Nechť X je kombinátor definovaný pravidlem $\mathsf{X}M \rightsquigarrow M\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}$.

Pak $\{\mathsf{X}\}$ je kombinátorová báze.

Důkaz:

$$\mathsf{X}\,\mathsf{X}\,\mathsf{X} \rightsquigarrow \mathsf{X}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{X} \rightsquigarrow \mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{X} \rightsquigarrow \mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{X} \rightsquigarrow \mathsf{K}\,\mathsf{K}\,\mathsf{X} \rightsquigarrow \mathsf{K}$$

$$\mathsf{X}(\mathsf{X}\,\mathsf{X}) \rightsquigarrow \mathsf{X}\,\mathsf{X}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow \mathsf{X}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow \mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow$$

$$\mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow \mathsf{K}\,\mathsf{K}\,\mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow \mathsf{K}\,\mathsf{S}\,\mathsf{K} \rightsquigarrow \mathsf{S}$$

Rozšíření kombinátorové sady – kombinátory B, C

$$\begin{aligned} \mathsf{B}\ M\ N\ P &\rightsquigarrow M\ (N\ P) \\ \mathsf{C}\ M\ N\ P &\rightsquigarrow M\ P\ N \end{aligned}$$

Věta: Pro libovolné termy M, N, P platí

$$\begin{aligned} \mathsf{S}(\mathsf{K}\ P)\ Q &\approx \mathsf{B}\ P\ Q \\ \mathsf{S}P\ (\mathsf{K}\ Q) &\approx \mathsf{C}\ P\ Q \\ \mathsf{S}(\mathsf{K}\ P)\ (\mathsf{K}\ Q) &\approx \mathsf{K}\ (P\ Q) \\ \mathsf{S}(\mathsf{K}\ P)\ \mathsf{I} &\approx P \end{aligned}$$

Vylepšení algoritmu pro převod lambda termů do kombinátorových termů

$$
\begin{array}{lcll}
\lambda x.x & \not\approx & \mathsf{I} & \\
\lambda x.M & \not\approx & \mathsf{K}\ M, & \text{když } x \notin \mathrm{FV}(M) \\
\lambda x.M\ N & \not\approx & \mathsf{S}(\lambda x.M)(\lambda x.N) & \\
\lambda x.M\ x & \not\approx & M, & \text{když } x \notin \mathrm{FV}(M) \\
\mathsf{S}\ (\mathsf{K}\ M)\ N & \not\approx & \mathsf{B}\ M\ N & \\
\mathsf{S}M\ (\mathsf{K}\ N) & \not\approx & \mathsf{C}\ M\ N & \\
\mathsf{S}(\mathsf{K}\ M)\ (\mathsf{K}\ N) & \not\approx & \mathsf{K}\ (M\ N) & \\
\mathsf{S}(\mathsf{K}\ M)\ \mathsf{I} & \not\approx & M &
\end{array}
$$

Superkombinátory

Motivace pro λ-lifting:

$$P = (\lambda x.(+)\ x((\lambda y.(*)\ x\ y)\ 3))\ 4$$
$$Q = (\lambda xy.(+)\ x(\ (*)\ x\ y)\ )\ 4\ 3$$

$P \sim' Q$, ale překlad Q do kombinátorů je snazší než překlad P.

Superkombinátory

Idea: Každému termu (funkcionálnímu programu) „ušít na míru“ jeho sadu kombinátorů – tzv. *superkombinátorů*. Přepisovací pravidlo pro každý superkombinátor lze implementovat jako podprogram v cílovém jazyce (strojovém nebo bytovém kódu).

Def: *Uzavřený* term $S = \lambda x_1 \lambda x_2 \ldots \lambda x_n . E$ je superkombinátor když

$n \geq 0$

E není λ-abstrakce

všechny λ-abstrakce v E jsou superkombinátory

Číslo n je tzv. *arita* superkombinátoru S.

Superkombinátorový redex je tvaru $S\ M_1\ M_2 \ldots M_n$.

Redukční pravidlo $S\ x_1\ x_2 \ldots x_n \rightsquigarrow E$ neobsahuje volné proměnné, takže ho lze vyjádřit pevným podprogramem v cílovém kódu (G stroj).

Def: Superkombinátory arity 0 se nazývají konstantní aplikativní formy (CAF).

Příklady superkombinátorů

$\alpha = 3$	$\alpha = 3$
$\beta = (+)\ 25$	$\beta = (+)\ 25$
$\gamma = \lambda x.x$	$\gamma\ x = x$
$\delta = \lambda x.(+)\ x\ 1$	$\delta\ x = (+)\ x\ 1$
$\varepsilon = \lambda x.(*)\ x\ x$	$\varepsilon\ x = (*)\ x\ x$
$\zeta = \lambda x \lambda y.(-)\ y\ x$	$\zeta\ x\ y = (-)\ y\ x$
$\eta = \lambda f.f(\lambda x.(*)\ x\ x)$	$\eta\ f = f\varepsilon$

Příklady nesuperkombinátorů

$\lambda x.y$	není uzavřený
$\lambda y.(-)\ y\ x$	není uzavřený
$\lambda f.f\ (\lambda x.f\ x\ 2)$	vnitřní abstrakce není superkombinátor (není uzavřená)
$\lambda x.x\ (\lambda y.y\ (\lambda z.z\ y))$	nejvnitřnější abstrakce není superkombinátor (není uzavřená)

Algoritmus převodu λ-termu do superkombinátorového termu

Algoritmus: SC

Vstup: term M

Výstup: dvojice $(D, \overline{M})$, kde D je množina definic superkombinátorů

$\overline{M}$ je superkombinátorový term ekvivalentní M

$SC(M) = T(\emptyset, M)$

$T(D, M) =$ jestliže M neobsahuje žádnou λ-abstrakci

pak (D, M)

jinak $T(D \cup \{\langle \alpha\ x_1 \ldots x_n\ x = N \rangle\}, M')$

kde $\lambda x.N$ je nevnitřnější abstrakce v M

$\{x_1, \ldots, x_n\} = \mathrm{FV}(\lambda x.N)$

M' vznikne z M náhradou abstrakce $\lambda x.N$ termem $\alpha\ x_1 \ldots x_n$

Poznámka: Krokům transformace T se říká „vynášení lambda“ (lambda-lifting).

Věta: Algoritmus převodu λ-termu do superkombinátorového termu konverguje a výsledný term $\overline{M}$ je (při definicích z D) ekvivalentní původnímu termu M.

Důkaz:

Konvergence – v každém kroku ubude λ-abstrakce, algoritmus končí, když v termu už žádné abstrakce nejsou.

Korektnost – každý krok zachovává ekvivalenci: $\lambda x.N \approx \lambda x.\alpha\ x_1 \ldots x_n\ x$ (protože „$\alpha\ x_1 \ldots x_n\ x = N$“ přidáváme do D), takže $\lambda x.N \approx \lambda x.\alpha\ x_1 \ldots x_n$ (po η-redukci)

Věta: Při převodu λ-termu M velikosti k do superkombinátorového termu $\overline{M}$ bude mít term $\overline{M}$ velikost nejvýše k a celková velikost všech termů v D bude v $O(k)$.

Důkaz:

1. term $\alpha\ x_1 \ldots x_n$ není delší než $\lambda x.N$, protože všechny volné proměnné $x_1, \ldots, x_n$ se v N vyskytují.
2. v každém kroku je přírustek k celkové velikosti nejvýše roven počtu volných proměnných termu N.

Modifikace vynášecího algoritmu na plně líné vynášení

Def: Podvýraz M abstrakce L je *volný*, když všechny proměnné v M jsou volné v L.

Maximální volný výraz v L je takový volný výraz, který není vlastním podtermem žádného výrazu volného v L.

Příklad: $\lambda x \lambda y.y + \underline{\text{sqrt } x}$

Aby bylo zaručeno, že po lambda-vynesení bude zachována plná lenost, je nutné vynášet místo volných proměnných maximální volné výrazy.

Datové struktury ve funkcionálním programování

Binomiální haldy

Def: *Binomiální strom* je (kořenový) strom obecné arity s uzly ohodnocenými klíči.

- binomiální strom stupně 0 je jednouzlový strom
- binomiální strom stupně $r + 1$ vznikne ze dvou binomiálních stromů t_1, t_2 stupně r tzv. spojením: strom t_1 se stane novým, nejlevějším, následníkem stromu t_2
- uspořádání klíčů na všech větvích binomiálních stromů je rostoucí

Každý binomiální strom je dán svým kořenen se stupněm, ohodnocením (klíčem) a seznamem následnických podstromů. Následnické podstromy jsou v pořadí s klesajícím stupněm.

```
data (Ord a) => BTree a  =  Node Int a [BTree a]
rank (Node r _ _)  =  r
root (Node _ x _)  =  x
```

Při spojování se haldové řazení klíčů udržuje tím, že se připojuje vždy strom s kořenem s větším klíčem pod strom s kořenem s menším klíčem. Spojované stromy mají vždy stejný stupeň.

```
link  ::  (Ord a) => BTree a -> BTree a -> BTree a
link t1@(Node r x1 s1) t2@(Node _ x2 s2)
                =  if  x1 <= x2  then  Node (r+1) x1 (t2:s1)
                                 else  Node (r+1) x2 (t1:s2)
```

Def: *Binomiální halda* je konečná posloupnost binomiálních stromů v pořadí rostoucích stupňů. Stupně všech stromů v binomiální haldě jsou navzájem různé.

```
type  BHeap a  =  [ BTree a ]
```

Lemma: Binomiální halda velikosti n obsahuje nejvýše $\lfloor \log_2(n+1) \rfloor$ binomiálních stromů.

Vkládání

```
insert  ::  (Ord a) => a -> BHeap a -> BHeap a
insert x h  =  insBT (Node 0 x []) h


insBT  ::  (Ord a) => BTree a -> BHeap a -> BHeap a
insBT t []  =  [t]
insBT t h@(t':s)
      =  case compare (rank t) (rank t') of
            LT  ->  t : h
            GT  ->  t': insBT t s
            EQ  ->  insBT (link t t') s
```

Věta: Cena vložení klíče do binomiální haldy velikosti n je v $(\log\ n)$.

Poznámka: Amortizovaná cena je dokonce v (1).

Slučování

```
merge  ::  (Ord a) => BHeap a -> BHeap a -> BHeap a
merge h []  =  h
merge [] h  =  h
merge h1@(t1:s1) h2@(t2:s2)
      =  case compare (rank t1) (rank t2) of
            LT  ->  t1 : merge s1 h2
            GT  ->  t2 : merge h1 s2
            EQ  ->  insBT (link t1 t2) (merge s1 s2)
```

Věta: Cena sloučení dvou binomiálních hald velikosti n je v $(\log\ n)$.

Nalezení a zrušení nejmenšího prvku

```
findMin  ::  (Ord a) => BHeap a -> a
findMin  =  root . fst . remMinT

deleteMin  ::  (Ord a) => BHeap a -> BHeap a
deleteMin h  =  merge (reverse s) s'
                where (Node _ _ s, s') = remMinT h
```

Pomocná funkce `remMinT` vyjme ze seznamu strom s nejmenším kořenem.

```
remMinT  ::  (Ord a) => BHeap a -> (BTree a, BHeap a)
remMinT [t]    =  (t,[])
remMinT (t:s)  =  if root t <= root t'  then  (t,s)  else  (t',t:s')
                  where  (t',s') = remMinT s
```

Věta: Cena nalezení i zrušení prvku v binomiální haldě velikosti n je v $(\log\ n)$.

Konstruktorové třídy

Funktory

```
class Functor f where
    fmap :: (a → b) → f a → f b

instance Functor Maybe where
    fmap g Nothing = Nothing
    fmap g (Just x) = Just (g x)

instance Functor [ ] where
    fmap = map

instance Functor IO where
    fmap g x  =  x  ≫=  return ∘ g
```

Axiomy funktorů

$$\begin{aligned} \mathit{fmap}\ \mathit{id} &= \mathit{id} \\ \mathit{fmap}\ (f \circ g) &= \mathit{fmap}\ f \circ \mathit{fmap}\ g \end{aligned}$$

Věta: Instance `Maybe`, `[]`, `IO` konstruktorové třídy Functor splňují oba axiomy funktorů.

Funktory s jednoprvkovou strukturou

```
class Functor f ⇒ PointedFunctor f where
    pure  :: a → f a
```

Instance konstruktorové třídy `Functor` jsou často také instancemi třídy `PointedFunctor`.

```
instance PointedFunctor Maybe where
    pure = Just

instance PointedFunctor [ ] where
    pure x = [x]

instance PointedFunctor IO where
    pure = return
```

Axiom funktorů s jednoprvkovou strukturou

$$\mathit{fmap}\ g \circ \mathit{pure} = \mathit{pure} \circ g$$

Věta: Instance `Maybe`, `[]`, `IO` konstruktorové třídy PointedFunctor tento axiom splňují.

Aplikativní funktory

Funktory umožňují vynést „obyčejné“ funkce na funkce na strukturách. Často je však užitečné vynášet funkce, které jsou samy hodnotami ve struktuře. To umožňuje konstruktorová třída `Applicative`.

```
class Functor f ⇒ Applicative f where
    pure  ::  a → f a
    (<*>) ::  f (a → b) → f a → f b
```

nebo alternativně

```
class PointedFunctor f ⇒ Applicative' f where
    (<*>) ::  f (a → b) → f a → f b
```

Operátor `(<*>)` funguje jako operátor aplikace funkce, ale uvnitř struktury (dané typovým konstruktorem `f`).

Axiom aplikativních funktorů

$$fmap = (<*>) \circ pure$$

Poznámka: Operátor *fmap* tedy lze rozložit na dvě jednodušší operace: vložení do struktury a aplikaci v rámci struktury.

Instance třídy Applicative

```
instance Applicative Maybe where
    pure              = Just
    Just g <*> Just x = Just (g x)
    _      <*> _      = Nothing
```

Seznamový typový konstruktor lze učinit instancí třídy aplikativních funktorů dvěma způsoby. První z nich uvažuje seznamy jako posloupnosti a při aplikaci bere jen hodnoty na stejných pozicích:

```
newtype ZipList a = ZL [a]
instance Applicative ZipList where
    pure = repeat
    ZL gs <*> ZL xs = ZL (zipWith id gs xs)
```

Například skalární součin dvou vektorů reprezentovaných seznamy lze zapsat

$$\mathit{sum}\,(\mathit{pure}\ (*)\ \texttt{<*>}\ \texttt{ZL}\,[2,3,4]\ \texttt{<*>}\ \texttt{ZL}\,[5,6,7])$$

Druhý způsob (přijatý v haskellovské knihovně `Control`) uvažuje seznamy jako multimnožiny – výsledků nedeterministických výpočtů. Při aplikaci bere všechny dvojice „(funkce,argument)“:

```
instance Applicative [ ] where
    pure x = [x]
    gs <*> xs = [ g x | g <- gs, x <- xs ]
```

Například součin dvou „nedeterministických hodnot“ lze zapsat

pure `(*) <*> [2,3] <*> [5,6]`

Monády

```
class Monad m  where
    return  :: a → m a
    (≫=) :: m a → (a → m b) → m b
    (≫) :: m a →  m b → m b
    fail :: String → m a

    p ≫ q  =  p ≫= const q
    fail  =  error
```

Poznámka: Operátor (≫) je zde přidán pro zjednodušení deklarací instancí a operátor `fail` je zde také navíc jen z praktických důvodů. Alternativní definice třídy `Monad` by mohla být

```
class Applicative m ⇒ Monad' m  where
    (≫=) :: m a → (a → m b) → m b
```

```
instance Monad Maybe where
    return = Just
    Nothing >>= k = Nothing
    Just x >>=  k = k x
    fail s  = Nothing

instance Monad [] where
    return  =  (:[])
    [] >>= f  =  []
    (x:s) >>= f  =  f x ++ (s >>= f)
    fail s  =  []

instance Monad IO where
    return  =  primretIO
    (>>=) = primbindIO
```

Monadické axiomy

$$
\begin{aligned}
\mathit{return}\ a \ggg\!= g &= g\ a \\
m \ggg\!= \mathit{return} &= m \\
m \ggg\!= (\lambda x.\ g\ x \ggg\!= h) &= (m \ggg\!= g) \ggg\!= h
\end{aligned}
$$

Věta: Instance `Maybe`, `[]`, `IO` konstruktorové třídy Monad splňují všechny tři monadické axiomy.

Poznámka: Zavedeme-li pomocný operátor,

$$(\texttt{>=>}) :: \texttt{Monad}\, m \Rightarrow (a \to m\, b) \to (b \to m\, c) \to (a \to m\, c)$$

$$f \texttt{ >=> } g \quad = \quad \lambda x.\, f\, x \gg\!= g$$

pak budou mít monadické axiomy tvar

$$\begin{aligned} \textit{return} \texttt{ >=> } g \quad &= \quad g \\ g \texttt{ >=> } \textit{return} \quad &= \quad g \\ g \texttt{ >=> } (h \texttt{ >=> } k) \quad &= \quad (g \texttt{ >=> } h) \texttt{ >=> } k \end{aligned}$$

a monadické funkce spolu sperátorem $(\texttt{>=>})$ budou tvořit monoid.

Monády lze, místo dvojice operací *return* a $(\gg\!=)$, alternativně definovat pomocí dvojice operací *fmap* a *join*:

```
fmap :: Monad m ⇒ (a → b) → (m a → m b)
fmap f m  =  m ≫= return ∘ f

join :: Monad m ⇒ m (m a) → m a
join z  =  z ≫= id
```

Příklad: V seznamové monádě je

join $=$ *concat*, *fmap* $=$ *map*.

Monády s nulou a aditivní monády

```
class Monad m  ⇒  MonadZero m  where
    mzero  ::  m a

class MonadZero m  ⇒  MonadPlus' m  where
    mplus  ::  m a → m a → m a
```

nebo alternativně (bez „mezitřídy“)

```
class Monad m  ⇒  MonadPlus m  where
    mzero  ::  m a
    mplus  ::  m a → m a → m a
```

```
instance MonadPlus Maybe where
    mzero  =  Nothing
    Nothing `mplus` y  =  y
    Just x `mplus` _  =  Just x

instance MonadPlus [] where
    mzero  =  []
    mplus  =  (++)
```

Axiomy aditivních monád

$$
\begin{aligned}
m \gg \mathit{mzero} &= \mathit{mzero} \\
\mathit{mzero} \gg m &= \mathit{mzero} \\
m \text{ `mplus` } \mathit{mzero} &= m \\
\mathit{mzero} \text{ `mplus` } m &= m
\end{aligned}
$$

Věta: Instance `Maybe` a `[]` konstruktorové třídy MonadPlus splňují axiomy aditivních monád.

Zbytek je nutné přesázet (chybějící mezery mezi jmény proměnných, řádkový zlom, fonty. . .)

Monadické kombinátory pro syntaktickou analýzu

```
newtype Parser a = P (String → [(a, String)])

papply (P p) =  p

instance Functor Parser where
    fmap f (P p) = P (λinp ↦~ [(f  v,out)|(v,out)←p  inp])

instance Monad Parser where
    return v  =  P (λinp ↦~ [(v,inp)])
    (P p) >>=  f  =  P (λinp ↦~ concat [papply (f v) out |(v,out
```

return je konstantní parser, který vždy uspěje (bez čtení vstupu).

$(\gg\!=)$ vytváří sekvence z parserů.

Aplikace parseru na řetězec

$$papply :: Parser\ a \to String \to [(a, String)]$$
$$papply\ (P\ p)\ s = p\ s$$

pak $p\ \gg\!=\ h$ lze vyjádřit pomocí $papply$:

$$p\ \gg\!=\ h = P(\lambda x \to concat[papply(h\ v)t | (v, t) \leftarrow papply\ p\ s])$$

Operátor „determinizace“

$$first :: Parser\ a \to Parser\ a$$
$$first(P\ p) = P(\lambda s \to case\ p\ s\ of$$
$$[\,] \to [\,]$$
$$x : t \to [x]$$

Deterministická alternativa

$$(+++) :: Parser\ a \to Parser\ a \to Parser\ a$$
$$p\ +++\ q = first(p\ `mplus`\ q)$$

Sekvenci parserů lze zapsat

$$
\begin{aligned}
& p_1 \gg\!= \lambda x_1 \rightarrow \\
& p_2 \gg\!= \lambda x_2 \rightarrow \\
& \ldots \\
& p_n \gg\!= \lambda x_n \rightarrow \\
& return\ (f\ x_1\ x_2\ \ldots\ x_n)
\end{aligned}
$$

nebo

$$
\begin{aligned}
& do\ x_1 \leftarrow p_1 \\
& \quad x_2 \leftarrow p_2 \\
& \quad \ldots \\
& \quad x_n \leftarrow p_n \\
& \quad return\ (f\ x_1\ x_2\ \ldots\ x_n)
\end{aligned}
$$

Poznámka: V jistých verzích Haskellu existovala tzv. monadická comprehension:

$$[f\ x_1\ x_2\ \ldots\ x_n | x_1 \leftarrow p_1, x_2 \leftarrow p_2 \ldots, x_n \leftarrow p_n]$$

Jednoduché parsery

Parser, který uspěje na libovolném znaku, pokud je na vstupu; na prázdném vstupu selže.

$$
\begin{aligned}
&item :: Parser\ Char \\
&item = P\ \lambda x \rightarrow case\ s\ of \\
&\qquad\qquad [\,] \rightarrow [\,] \\
&\qquad\qquad (x : t) \rightarrow [(x, t)]
\end{aligned}
$$

Jednoduché parsery

$$sat :: (Char \to Bool) \to Parser\ Char$$
$$sat\ p = do\ x \leftarrow item$$
$$\qquad\qquad if\ p\ x\ then\ return\ x$$
$$\qquad\qquad\qquad else\ mzero$$

$$char :: Char \to Parser\ Char$$
$$char = sat \circ (==)$$

$$digit, lower, upper, letter, alphaNum :: Parser\ Char$$
$$digit = sat\ isDigit$$
$$lower = sat\ isLower$$
$$upper = sat\ isUpper$$
$$letter = lower\ `mplus`\ upper$$
$$aplhaNum = letter\ `mplus`\ digit$$

$$
\begin{aligned}
&string :: String \to Parser\ String \\
&string\ "" = return\ "" \\
&string(x : s) = do\ char\ x \\
&\qquad\qquad\qquad\qquad string\ s \\
&\qquad\qquad\qquad\qquad return\ (x : s)
\end{aligned}
$$

Kombinátory opakovacích parserů

$$many1 :: Parser\ a \to Parser\ [a]$$
$$many1\ p = do\ x \leftarrow p$$
$$\qquad\qquad\qquad\qquad s \leftarrow many\ p$$
$$\qquad\qquad\qquad\qquad return\ (x : s)$$

$$many :: Parser\ a \to Parser\ [a]$$
$$many\ p = many1\ p\ +\!+\!+\ return\ [\,]$$

$$nat :: Parser\ Int$$
$$nat = many1\ digit\ \gg\!=\ return \circ read$$
$$int :: Parser\ Int$$
$$int = (char\ '-'\ \gg\ nat\ \gg\!=\ return \circ negate)\ +\!+\!+\ nat$$

$$ident :: Parser\ String$$
$$ident = do\{x \leftarrow Lower; s \leftarrow many\ alphanum; return\ (x : s)\}$$

$$spaces :: Parser\ ()$$
$$spaces = many1\ (sat\ isSpace)\ \gg\ return\ ()$$

$$comment :: Parser\ ()$$
$$comment = string\ ”--”\ \gg\ many(sat\ (/ =' \backslash n'))\ \gg\ return\ ()$$

$$junk :: Parser\ ()$$
$$junk = many(spaces\ +\!+\!+\ comment)\ \gg\ return\ ()$$

$$token :: Parser\ a \rightarrow Parser\ a$$
$$tokenp = do\{v \leftarrow p; junk; return =\ v\}$$

$$symbol :: String \to Parser\ String$$
$$symbol = token \circ string$$

$$identifier :: [String] \to Parser\ String$$
$$identifierks = token(do\ x \leftarrow ident$$
$$if\ x\ `elem`\ ks$$
$$then\ mzero$$
$$else\ return\ x$$