KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 2 ◇ poznámky k přednášce

9. Výpočty v normálním modelu vyhodnocení

verze z 14. dubna 2020

Tato přednáška je určena především motivovaným zájemcům. Pokud se něco z ní objeví na zkoušce u písemky, bude to považováno za „obtížný příklad“ pro ty, kteří chtějí dosáhnout vysokého počtu bodů. Navíc budu na písemku z této přednášky vybírat jen jednodušší příklady.

1 Úvod

Na minulé přednášce jsme hovořili o různých **modelech vyhodnocování**, tedy způsobech, jak jazyk vyhodnocuje výrazy, a to především, zda před aplikací funkce vyhodnotí argumenty, nebo ne. Řekli jsme, že ten model, který jsme zatím používali, je tzv. **aplikativní model vyhodnocování**. (Tedy až na výjimky, jako je vyhodnocování speciálních operátorů a maker. Makra můžeme ovšem chápat jako pouhé rozšíření syntaxe, takže z pohledu vyhodnocovacího modelu věc nepodstatnou.) **Líné vyhodnocování**, které nevyhodnocuje argumenty funkcí, je zase založeno na **normálním modelu**.

K popisu zdrojového kódu jsme používali jazyk λ-kalkulu, ve kterém jsme také podrobně definovali, co to je vyhodnocení, redukce, normální forma výrazu — všechno pojmy potřebné k přesnému popisu vyhodnocovacího modelu.

Na přednášce jsme se vůbec nezabývali výpočty jako takovými. Za základní stavební kameny výpočtů lze považovat:

- podmíněné vyhodnocení (jako např. speciální operátor `if` v Lispu) a logické operace
- datové struktury
- nějaký typ rekurze
- operace s přirozenými čísly

Těmito problémy se budeme zabývat teď.

2 Opakování z minula

Výrazy λ-kalkulu

(Pro zajímavost dodávám i syntax jazyka Haskell.)

	λ-kalkul	Scheme	Lisp	Haskell
proměnná	$x, y, z, \ldots$	`x, y, z, ...`	`x, y, z, ...`	`x, y, z, ...`
abstrakce	$(\lambda x.M)$	`(lambda (x) M)`	`(lambda (x) M)`	`\x M`
aplikace	$(M\ N)$	`(M N)`	`(funcall M N)`	`M N`

Zjednodušení zápisu:

- vynecháváme závorky, kde to lze
- aplikace více výrazů:

$$((M\ N)\ L) \equiv M\ N\ L$$

- vše za tečkou je tělo:

$$\lambda x.M\lambda y.NK \equiv \lambda x.(M\lambda y.(NK))$$

- currying:

$$\lambda xy.M \equiv \lambda x.\lambda y.M$$

β-redukce

Podvýrazy výrazu lze **redukovat** tak, že se v aplikaci, jejíž hlava je abstrakce, dosadí v těle této abstrakce za všechny volné výskyty parametru abstrakce argument aplikace. Výsledkem je takto upravené tělo. Redukci značíme jednoduchou šipkou:

$$(\lambda x.M)N \to M[x := N].$$

- Výraz $(\lambda x.M)N$ se nazývá *redex*. (Je to aplikace, jejíž hlavou je abstrakce.)
- Zápis $M[x := N]$ značí, že se **volné výskyty** proměnné x v M se nahradí N

Normální formy

Normální forma je výraz, který neobsahuje redex.

Hlavová normální forma je výraz, který může obsahovat redex, ale nechceme ho redukovat:

1. protože je v těle abstrakce (funkce),
2. protože je v argumentu aplikace, ale hlavu nelze redukovat na abstrakci.

Model vyhodnocení (redukční strategie)

je způsob, v jakém pořadí redukovat redexy, pokud je jich ve vyhodnocovaném výrazu více.

Aplikativní vyhodnocovací model.

(Vyhodnocení „zevnitř ven“.)

- U aplikace nejprve vyhodnotíme hlavu, pak (pokud je výsledek abstrakce) vyhodnotíme argument, pak β-redukujeme a opakujeme, dokud to jde.
- Jiné výrazy než aplikace vůbec nevyhodnocujeme.

Normální vyhodnocovací model. (Vyhodnocení „zvenku dovnitř“.)

- U aplikace nejprve vyhodnotíme hlavu, pak (pokud je výsledek abstrakce) β-redukujeme (s nevyhodnoceným argumentem) a opakujeme, dokud to jde.
- Jiné výrazy než aplikace vůbec nevyhodnocujeme.

Výpočty v λ-kalkulu

Teď se pustíme do slibovaného tématu: popisu toho, jak se dají v λ-kalkulu popsat základní stavební kameny výpočtů: podmíněné vyhodnocování a logika, datové struktury, rekurze, operace s přirozenými čísly. Výrazy λ-kalkulu a jejich redukce zdánlivě popisují jen výpočetní proces bez konkrétního obsahu, tj. bez konkrétních výpočtů. Uvidíme, že to není pravda a že v λ-kalkulu lze modelovat všechny prvky, které může obecný výpočet obsahovat.

Všechno, co na této přednášce probíráme teoreticky, můžete vyzkoušet na interpretu Scheme s líným vyhodnocováním, který máte k dispozici (o něm více později).

3 Logika

Než se pustíme do logiky, domluvíme se na jednou značení. Pokud při vyhodnocování výrazu uděláme víc redukcí v jednom kroku, zapíšeme to šipkou s hvězdičkou: $\to^*$. Takže můžeme napsat toto:

$$(\lambda pca.pca)LMN \to^* LMN,$$

jelikož

$$(\lambda pca.pca)LMN \to (\lambda ca.Lca)MN \to (\lambda a.LMa)N \to LMN$$

(kde $\to$ značí, jak víme, vždy jednu redukci).

Podívejme se na následující tři výrazy:

$$\begin{aligned}\texttt{TRUE} &:= \lambda xy.x\\ \texttt{FALSE} &:= \lambda xy.y\\ \texttt{IF} &:= \lambda pca.pca\end{aligned}$$

Symbolem „:=“ značím, že výrazu na pravé straně dávám pro stručnost a přehlednost název, který píšu na levou stranu. Kdykoli tedy napíšu `TRUE`, budu mít na mysli přesně výraz $\lambda xy.y$. Můžeme tedy napsat

$$\texttt{TRUE} \equiv \lambda xy.x$$

Pojďme teď normálním vyhodnocovacím modelem (tedy tím líným) vyhodnotit výrazy $\texttt{IF TRUE}\,MN$ a $\texttt{IF FALSE}\,MN$:

$$\begin{aligned}\texttt{IF TRUE}\,MN \equiv (\lambda pca.pca)\texttt{TRUE}\,MN \to^* \texttt{TRUE}\,MN \equiv (\lambda xy.x)MN \to^* M\\ \texttt{IF FALSE}\,MN \equiv (\lambda pca.pca)\texttt{FALSE}\,MN \to^* \texttt{FALSE}\,MN \equiv (\lambda xy.y)MN \to^* N\end{aligned}$$

Výsledky (výrazy M a N) by se samozřejmě mohly redukovat ještě dále. Tak by to bylo v normálním modelu vyhodnocení. V aplikativním bychom se tak daleko nedostali, protože už na začátku (např. ve výrazu $\texttt{IF TRUE}\,MN$) by bylo nutné nejprve vyhodnotit výrazy M a N.

V normálním vyhodnocovacím modelu tedy můžeme definovat `IF` **jako funkci**. V aplikativním to nejde, protože `if` jakožto funkce by vyhodnocovala všechny své argumenty, takže by nešlo o podmíněné vyhodnocení. Jako funkce můžeme definovat i odvozené logické operace. Například:

$$\texttt{AND} := \lambda xy.\texttt{IF}\;x\;y\;\texttt{FALSE}$$

Nyní můžeme vyzkoušet vyhodnotit třeba

$$\begin{aligned}\texttt{AND TRUE TRUE} &\equiv (\lambda xy.\texttt{IF}\;x\;y\;\texttt{FALSE})\;\texttt{TRUE TRUE}\\ &\to^* \texttt{IF TRUE TRUE FALSE} \to^* \texttt{TRUE}\end{aligned}$$

Sami si můžete vyzkoušet, na co se vyhodnocují výrazy:

$$\texttt{AND TRUE FALSE},\qquad \texttt{AND FALSE TRUE},\qquad \texttt{AND FALSE FALSE}.$$

4 Datové struktury: páry a seznamy

Jako základní datové struktury použijeme osvědčené páry a seznamy. A opět ukážeme, že je lze zkonstruovat čistě pomocí λ-kalkulu.

Párem s první složkou M a druhou N nazveme výraz $[M, N]$ definovaný takto:

$$[M, N] := \lambda z.zMN$$

Výrazy CONS, CAR a CDR definované takto:

$$\begin{aligned}\texttt{CONS} &:= \lambda xyz.zxy\\ \texttt{CAR} &:= \lambda p.p\,\texttt{TRUE}\\ \texttt{CDR} &:= \lambda p.p\,\texttt{FALSE}\end{aligned}$$

pak fungují podle očekávání, jak se můžeme přesvědčit:

$$\begin{aligned}&\texttt{CONS}\,MN \equiv (\lambda xyz.zxy)MN \to^* \lambda z.zMN \equiv [M,N]\\ &\texttt{CAR}[M,N] \equiv (\lambda p.p\,\texttt{TRUE})(\lambda z.zMN) \to (\lambda z.zMN)\,\texttt{TRUE} \to \texttt{TRUE}\,MN \to^* M\\ &\texttt{CDR}[M,N] \equiv (\lambda p.p\,\texttt{FALSE})(\lambda z.zMN) \to (\lambda z.zMN)\,\texttt{FALSE} \to \texttt{FALSE}\,MN \to^* N\end{aligned}$$

Seznamy můžeme definovat pomocí párů, jak jsme zvyklí. Abychom ovšem mohli pracovat i s konečnými seznamy, musíme zavést prázdný seznam a test, zda je daný seznam prázdný. To je vaším úkolem na cvičení. Nebudu to tady tedy dělat, jen se omezím na předpoklad, že máme výraz EMPTY reprezentující prázdný seznam a výraz EMPTY, který rozhoduje, zda je daný výraz pár, nebo prázdný seznam. Tedy pro libovolný pár P:

$$\texttt{EMPTY?}\,P \to^* \texttt{FALSE}$$

a

$$\texttt{EMPTY?}\,\texttt{EMPTY} \to^* \texttt{TRUE}$$

Příklady na seznamy uvidíme za chvíli.

V normálním vyhodnocovacím modelu jsou páry a seznamy automaticky „líné“. Seznamy v lenosti předčí i proudy, o kterých jsme se bavili v 6. přednášce, protože jednak seznam není vůbec vypočten, dokud ho nepotřebujeme, a jednak, i když už seznam máme, tak stále nemusí být vypočítána jeho složka *car* (to u proudů vždycky byla).

5 Rekurze

Rekurze je ovšem oříšek. Víme, že ji ale programovací jazyk nezbytně potřebuje, aby v něm bylo možné zapsat libovolný algoritmus. Z λ-kalkulu se nám tady hezky vyvíjí zvláštní programovací jazyk, ale bez možnosti vyjádřit rekurzi by nebyl plnohodnotný.

Když už mluvím o λ-lambda kalkulu jako o programovacím jazyku, musím zmínit, že tak o něm samozřejmě nikdo neuvažuje; λ-kalkul slouží ale jako model (teoretický základ) pro programovací jazyky, které opravdu existují (zejména samozřejmě funkcionální programovací jazyky).

Uvažme tento příklad. Chceme definovat výraz EVERY?, který když aplikujeme na konečný seznam obsahující pouze hodnoty TRUE a FALSE, tak se aplikace vyhodnotí na TRUE právě když všechny prvky seznamu jsou TRUE. Jinak se vyhodnotí na FALSE.

Představme si třeba tříprvkový seznam L, který obsahuje prvky `TRUE`, `FALSE` a `TRUE`. Tento seznam bychom definovali takto:

$$L \equiv \texttt{CONS}\,\texttt{TRUE}\,(\texttt{CONS}\,\texttt{FALSE}\,(\texttt{CONS}\,\texttt{TRUE}\,\texttt{EMPTY}))$$

A chceme, aby se výraz `EVERY?` L vyhodnotil na `FALSE`.

Seznamy jsou rekurzivní datová struktura. Kdyby výraz `EVERY?` měl být funkcí v nějakém programovacím jazyce, musel by být nezbytně definován jako rekurzivní funkce nebo pomocí nějaké rekurzivní funkce.

My bychom rádi hledaný výraz definovali takto:

$$\texttt{EVERY?} \equiv \lambda x.\texttt{OR}\,(\texttt{EMPTY?}\,x)\,(\texttt{AND}\,(\texttt{CAR}\,x)\,(\texttt{EVERY?}\,(\texttt{CDR}\,x)))$$

(Pozorně si to rozeberte, není to tak těžké; případně si to přepište do Scheme.)

Takto ale výraz definovat nemůžeme, protože výraz nejde zapsat pomocí sebe sama. (Uvědomte si, že symbol `EVERY?` v λ-kalkulu není, jen nám má sloužit jako pojmenování nějakého výrazu λ-kalkulu.)

Tato „definice“ hledaného výrazu `EVERY?` je tedy nesmyslná. Hledání ovšem nevzdáme a uvědomíme si, že není nutné, aby výraz `EVERY?` byl totožný s pravou stranou, stačí, aby se na ni vyhodnotil. Stačí tedy, aby splňoval tuto podmínku:

$$\texttt{EVERY?} \to^* \lambda x.\texttt{OR}\,(\texttt{EMPTY?}\,x)\,(\texttt{AND}\,(\texttt{CAR}\,x)\,(\texttt{EVERY?}\,(\texttt{CDR}\,x)))$$

Abychom zdůraznili podstatu problému, definujme pomocný výraz:

$$\texttt{\%EVERY?} \equiv \lambda f x.\texttt{OR}\,(\texttt{EMPTY?}\,x)\,(\texttt{AND}\,(\texttt{CAR}\,x)\,(f\,(\texttt{CDR}\,x)))$$

S použitím tohoto pomocného výrazu by se nám hodilo, aby

$$\texttt{EVERY?} \to^* \texttt{\%EVERY?}\,\texttt{EVERY?},$$

protože tak bude výraz `EVERY?` splňovat podmínku, kterou jsem napsal před chviličkou.

Teď si všimněme, že pokud najdeme výraz `Y` takový, že

$$\texttt{Y}f \to^* f\,(\texttt{Y}f),$$

pak máme vyhráno, neboť pro

$$\texttt{EVERY?} := \texttt{Y}\,\texttt{\%EVERY?}$$

dostaneme

$$\texttt{EVERY?} \equiv \texttt{Y}\,\texttt{\%EVERY?} \to^* \texttt{\%EVERY?}(\texttt{Y}\,\texttt{\%EVERY?}) \equiv \texttt{\%EVERY?}\,\texttt{EVERY?},$$

což je to, co jsme chtěli.

Chápete? Pokud ne, není to ostuda. Když se nad tím dostatečně dlouho zamyslíte, pochopíte to. Jedná se o jeden z opravdu podivných a současně kouzelných poznatků teorie programovacích jazyků, na který přijít nabylo jen tak. Jeden

z opravdu slavných informatiků Haskell Curry na něj přišel a navíc odhalil, že výraz `Y` existuje a má tvar

$$\texttt{Y} := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Výrazu se říká *Y-kombinátor* nebo *Curryho paradoxní kombinátor* a má tu vpravdě paradoxní vlastnost, že se jeho aplikace na jiný výraz postupnými redukcemi zvětšuje (místo aby se zmenšovala, neboli redukovala), a to tak, jak potřebujeme:

$$\begin{aligned}\texttt{Y}f &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\, f \to (\lambda x.f(xx))(\lambda x.f(xx)) \\ &\to f\,(\lambda x.f(xx))(\lambda x.f(xx)) \equiv f\,(\texttt{Y}f)\end{aligned}$$

Výraz `EVERY?` tedy opravdu můžeme definovat, jak jsme řekli:

$$\texttt{EVERY?} := \texttt{Y}\,\texttt{\%EVERY?}$$

Tímto trikem můžeme jakýkoli rekurzivní výpočet definovat výrazem λ-kalkulu.

Mimochodem, currying, o kterém jsme mluvili minule, je opravdu pojmenován podle příjmení Haskella Curryho. Jazyk Haskell se zase jmenuje podle jeho křestního jména. Curry má ještě prostřední jméno, podle kterého se jmenuje další programovací jazyk, ale ten není moc významný.

6 Čísla

Nyní již stručně a pro zajímavost ukážu, jak lze v λ-kalkulu definovat přirozená čísla a výpočty s nimi. Tento poznatek se přímo v programovacích jazycích nepoužívá, ale je zajímavý sám o sobě a navíc má dalekosáhlý teoretický význam (a to opravdu).

Čísla zavádíme jako výrazy (používáme výraz pro pár, definovaný dříve):

$$\begin{aligned}0 &:= \lambda x.x \\ 1 &:= [\texttt{FALSE}, 0] \\ 2 &:= [\texttt{FALSE}, 1] \\ &\vdots \\ n+1 &:= [\texttt{FALSE}, n]\end{aligned}$$

Následující výraz `ZERO?` pomáhá rozlišit, zda je číslo nula, nebo ne:

$$\texttt{ZERO?} := \texttt{CAR}$$

Opravdu:

$$\texttt{ZERO?}\,0 \equiv \texttt{CAR}\,\lambda x.x \equiv (\lambda p.p\,\texttt{TRUE})\,\lambda x.x \to (\lambda x.x)\,\texttt{TRUE} \to \texttt{TRUE}$$

A pro $n > 0$:

$$\texttt{ZERO?}\,n \equiv \texttt{CAR}\,[\texttt{FALSE}, n-1] \to \texttt{FALSE}$$

Každé přirozené číslo má svého následníka a každé kromě nuly i svého předchůdce. Jsou to dvě základní operace s čísly, pomocí kterých jdou definovat ostatní. Takto lze následníka a předchůdce zavést pro naše čísla:

$$\begin{aligned}\texttt{SUCC} &:= \lambda n.\texttt{CONS}\,\texttt{FALSE}\,n\\ \texttt{PRED} &:= \texttt{CDR}\end{aligned}$$

Takto nám to vychází pro libovolné n a pro $m > 0$:

$$\begin{aligned}\texttt{SUCC}\,n &\equiv (\lambda n.\texttt{CONS}\,\texttt{FALSE}\,n)\,n \to \texttt{CONS}\,\texttt{FALSE}\,n \to^* [\texttt{FALSE}, n] \equiv n+1\\ \texttt{PRED}\,m &\equiv \texttt{CDR}\,[\texttt{FALSE}, m-1] \to^* m-1\end{aligned}$$

Předchůdce nuly jako dárek navíc:

$$\texttt{PRED}\,0 \equiv (\lambda p.p\,\texttt{FALSE})\,0 \to 0\,\texttt{FALSE} \equiv (\lambda x.x)\,\texttt{FALSE} \to \texttt{FALSE}$$

Aritmetická operace sčítání čísel se dá definovat pomocí PRED, SUCC a rekurze:

$$\texttt{+} = \lambda ab.\texttt{IF}\,(\texttt{ZERO?}\,b)\,a\,(\texttt{+}\,(\texttt{SUCC}\,a)\,(\texttt{PRED}\,b))$$

Z předchozího už víme jak na to:

$$\begin{aligned}\texttt{\%+} &:= \lambda fab.\texttt{IF}\,(\texttt{ZERO?}\,b)\,a\,(f\,(\texttt{SUCC}\,a)\,(\texttt{PRED}\,b))\\ \texttt{+} &:= \texttt{Y}\,\texttt{\%+}\end{aligned}$$

Vychází to, co jsme potřebovali:

$$\texttt{+} \equiv \texttt{Y}\,\texttt{\%+} \to^* \texttt{\%+}\,(\texttt{Y}\,\texttt{\%+}) \equiv \texttt{\%+}\,\texttt{+} \to^* \lambda ab.\texttt{IF}\,(\texttt{ZERO?}\,b)\,a\,(\texttt{+}\,(\texttt{SUCC}\,a)\,(\texttt{PRED}\,b))$$

Ještě porovnávání čísel (relace „menší nebo rovno“):

$$\begin{aligned}\texttt{\%<=} &\equiv \lambda fab.\texttt{IF}\,(\texttt{ZERO?}\,a)\,\texttt{TRUE}\,(\texttt{IF}\,(\texttt{ZERO?}\,b)\,\texttt{FALSE}\,(f\,(\texttt{PRED}\,a)\,(\texttt{PRED}\,b)))\\ \texttt{<=} &\equiv \texttt{Y}\,\texttt{\%<=}\end{aligned}$$

A takto lze definovat nekonečný seznam přirozených čísel:

$$\begin{aligned}\texttt{\%NATURALS} &:= \lambda fn.\texttt{CONS}\,n\,(f\,(\texttt{SUCC}\,n))\\ \texttt{NATURALS} &:= \texttt{Y}\,\texttt{\%NATURALS}\end{aligned}$$

Seznam přirozených čísel počínaje nulou je pak dán výrazem

$$\texttt{NATURALS}\,0$$

7 Stručně k interpretu

K přednášce máte k dispozici novou verzi interpretu líného Scheme. Je stejná jako předchozí verze s jednou novinkou: kvůli přehlednosti je možné dávat výrazům jména, jak jsme dělali na této přednášce. Tak si budete moci vyzkoušet všechno, co jsme na přednášce probírali, i vaše řešení úloh. Podrobnosti k použití najdete ve zdrojovém kódu interpretu. Samotným zdrojovým kódem nové části interpretu se zabývat nemusíte.

Otázky a úlohy na cvičení

1. Definujte logické spojky `NOT` a `OR`.
2. Pomocí výrazu `<=` definujte test na rovnost čísel:

   $$\texttt{=}\, m\, n \to^* \texttt{FALSE}$$

   pokud se čísla m a n nerovnají,

   $$\texttt{=}\, n\, n \to^* \texttt{TRUE}$$

3. Definujte výraz pro rovnost čísel přímo, bez použití `<=`. Mělo by to vést k efektivnějším výpočtům.
4. Lze `PRED` definovat pomocí `SUCC`? A obráceně?
5. K práci se seznamy potřebujeme ještě prázdný seznam a predikát, který rozhoduje, zda je seznam prázdný. Navrhněte výrazy `EMPTY` a `EMPTY?` tak, aby

   $$\texttt{EMPTY?}\, \texttt{EMPTY} \to^* \texttt{TRUE}$$

   a pro libovolný pár P

   $$\texttt{EMPTY?}\, P \to^* \texttt{FALSE}$$

6. Lze vyřešit předchozí úlohu pro $\texttt{EMPTY} \equiv \texttt{FALSE}$?
7. Definujte operaci rozdílu čísel. Při odečítání většího čísla od menšího může být výsledkem nula.
8. Redukce výrazu `(+ 100 100)` trvá v interpretu velice dlouho. Zkuste přijít na to, proč.

   K řešení takových problémů je vhodné použít *profiler*. V LispWorks je jeden k dispozici.

9. V interpretu je alternativní funkce `ls-force`, která implementuje vyhodnocování výrazu tak dlouho, dokud není v normální formě. Redukuje tedy všechny redexy (i ty, které jsou v těle abstrakce). Vysvětlete, proč funkce není schopna vyhodnotit výraz `(+ 0 0)`.

10. Předpokládejme, že `NOT` je výraz, který realizuje negaci ve *zobecněné logice*, tedy

$$\texttt{NOT}\,\texttt{FALSE} \to^* \texttt{TRUE}$$

a

$$\texttt{NOT}\,M \to^* \texttt{FALSE}$$

pro libovolný výraz M, který se nevyhodnotí na `FALSE`. Z vlastnosti Y-kombinátoru

$$\texttt{Y}\,f \to^* f\,(\texttt{Y}\,f)$$

dostáváme

$$\texttt{Y}\,\texttt{NOT} \to^* \texttt{NOT}\,(\texttt{Y}\,\texttt{NOT})$$

Co z toho plyne?