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. Normální model vyhodnocování

verze z 19. května 2019

Vyhodnocovací proces v Lispu (a dalších jazycích, jako jsou Scheme, C, C++, Java, C# atd.) je založen na tom, že před aplikací funkce se vyhodnotí všechny argumenty. Tomuto způsobu vyhodnocování se říká *aplikativní model vyhodnocování*. Výjimku z tohoto pravidla tvoří jen několik speciálních operátorů a maker, jako třeba `if`, `and`, `or` apod. Mnoho funkcionálních programovacích jazyků ovšem používá jiný, tzv. *normální vyhodnocovací model*. Tím se budeme zabývat v této a další přednášce. Abychom vyhodnocovací modely lépe pochopili, ponoříme se do jejich teoretického popisu pomocí *λ-kalkulu*. Souběžně budeme problematiku ilustrovat na výrazech jazyke Scheme.

1 Vyhodnocování jako substituce

Když se zaměříme na vyhodnocovací proces jako takový a nebudeme se zajímat o volání externích funkcí (primitiv), můžeme si všimnout, že vyhodnocování výrazů lze chápat jako dosazování hodnot za parametry procedur. Například:

```
((lambda (x) x) M)
  -> M

((lambda (x y) x) M N)
  -> M

((lambda (x y) (x K)) ((lambda (x y) x) L M) N)
  -> ((lambda (x y) (x K)) L N)
  -> (L K)

((lambda (x) (L x x)) ((lambda (y z) y) M N))
  -> ((lambda (x) (L x x)) M)
  -> (L M M)
```

Poznámky.

1. Jednoduchou šipkou (`->`) zde značíme jeden krok vyhodnocení.
2. Symboly `K`, `L`, `M`, `N` označují už vyhodnocené výrazy nebo výrazy, které se vyhodnocují na sebe (například čísla, `L` by měla být procedura).

3. Ve třetím a čtvrtém příkladě by vyhodnocení mělo pokračovat dál, třeba i hodně složitě (záleží na funkci `L`).

Každý krok vyhodnocení proběhl tak, že se v těle nějakého λ-výrazu za jeho parametry dosadily argumenty. Takovému vyhodnocení říkáme *redukce*. V tomto teoretickém úvodu nebudeme žádný jiný způsob vyhodnocení uvažovat. Nebudeme tedy uvažovat ani o speciálních operátorech a makrech, ani o externích (primitivních) funkcích, u kterých neznáme jejich seznam parametrů a tělo.

Redukci výrazu můžeme přesněji popsat takto: redukovat lze složené výrazy (seznamy), na jejichž prvním místě je λ-výraz. Redukce se pak provede takto:

```
((lambda (x1 ... xn) M) K1 ... Kn)
  -> M[x1 := K1] ... [xn := Kn]
```

kde `M[x1 := K1] ... [xn := Kn]` znamená výraz `M`, ve kterém jsou všechny *volné výskyty* parametrů `x1 ... xn` nahrazeny výrazy `K1 ... Kn` (co je *volný výskyt* parametru vysvětlíme přesněji dále).

Ve třetím a čtvrtém uvedeném příkladě ovšem můžeme výrazy za parametry procedur také dosazovat v jiném pořadí, než v jakém by to dělal Scheme:

```
((lambda (x y) (x K)) ((lambda (x y) x) L M) N)
  -> (((lambda (x y) x) L M) K)
  -> (L K)

((lambda (x) (L x x)) ((lambda (y z) y) M N))
  -> (L ((lambda (y z) y) M N) ((lambda (y z) y) M N))
  -> (L M M)
```

Tento způsob je založen na principu neredukovat žádný výraz, dokud to není opravdu potřeba. Neredukované výrazy pouze dosazujeme za parametry procedur stejně, jako jsme to dřív dělali s hodnotami. Jde o variantu **líného vyhodnocování** a později se k ní ještě vrátíme.

V našich příkladech jsme líným vyhodnocováním došli ke stejnému výsledku, k jakému by došel Scheme. Obecně ale tento způsob vyhodnocení přináší dva významné rozdíly:

1. Někdy vede k nutnosti vyhodnotit jeden podvýraz dvakrát. V našich příkladech jsme museli dvakrát redukovat podvýraz `((lambda (x y) x) L M)`.

2. Jindy se zase podvýraz nemusí vyhodnotit nikdy, což může vést k tomu, že vyhodnocení, které by se ve Scheme zacyklilo, úspěšně skončí:

```
((lambda (x y) y) ((lambda (x) (x x)) (lambda (x) (x x))) M)
  -> M
```

První rozdíl by mohl znamenat neefektivní výpočet a v případě vedlejšího efektu dokonce neočekávané chování programu (kdyby třeba místo výrazu `((lambda (x y) x) L M)` byl tisk). Při použití čistě funkcionálního programovacího stylu, ve kterém není možný vedlejší efekt, by se ale vícenásobné vyhodnocení navenek neprojevilo. Tento styl také umožňuje kompilátorům přeložit program efektivně: jednou vypočítanou hodnotu si lze zapamatovat a podruhé ji už nepočítat, protože nahrazení výrazu jeho hodnotou nemá na výsledek žádný vliv. Odborně se této vlastnosti funkcionálních programů říká **referenční transparentnost**.

Podvýraz daného výrazu se nazývá *referenčně transparentní*, jetliže jeho nahrazení jeho hodnotou nezmění efekt (tj. hodnotu ani případný vedlejší efekt) celého výrazu.

Poznámka. V textu z minulé přednášky je zmínka o *generátorech*. Jestliže v proměnné `gen` máme uložen generátor (třeba přirozených čísel), pak lispový výraz `(funcall gen)` není referenčně transparentní, protože každé jeho vyhodnocení vrací jinou hodnotu. Nelze jej tedy jeho jednou získanou hodnotou nahradit.

Druhý rozdíl lze považovat za přednost a můžeme jej, jak ještě uvidíme, výhodně používat.

Pokud se ve vyhodnocovaném výrazu vyskytuje více podvýrazů, které lze redukovat, je otázkou **modelu vyhodnocování**, které podvýrazy a v jakém pořadí se budou redukovat. Abychom modely vyhodnocování lépe pochopili, vysvětlíme si stručně základy λ-kalkulu.

2 Výrazy λ-kalkulu

Teoretickým modelem pro funkcionální programovací jazyky je λ-kalkul, publikovaný poprvé ve 30. letech minulého století Alonzem Churchem. Ukážeme základní principy λ-kalkulu a jak souvisí s problematikou nastíněnou v minulé kapitole.

V λ-kalkulu se zabýváme *výrazy* (*termy*), konstruovanými následujícím způsobem. Výrazy jsou

1. *proměnné*, značené $x, y, \ldots, a, b, \ldots$ apod.
2. *abstrakce*, což jsou výrazy $(\lambda x.M)$, kde x je proměnná a M libovolný výraz,
3. *aplikace* $(M\,N)$, kde M a N jsou libovolné výrazy.

U uvedené abstrakce se proměnná x nazývá *parametr abstrakce* a výraz M *tělo abstrakce*.

Dvě abstrakce, které se liší pouze názvem parametru, považujeme za totožné. Například

$$(\lambda x.(xz)) \equiv (\lambda y.(yz))$$

Symbol $\equiv$ označuje, že jde o totožné výrazy. Budeme ho používat i dále.

U aplikace $(M\,N)$ se výraz M nazývá *hlava* a N *argument*.

Ve výrazech λ-kalkulu snadno rozpoznáme některé výrazy Scheme (a s drobným doplňkem i Lispu).

1. proměnné lze reprezentovat symboly `x`, `y`, ..., `a`, `b`, ... atd.
2. abstrakci reprezentujeme λ-výrazem `(lambda (x) M)`,
3. aplikaci složeným výrazem `(M N)` (v Lispu `(funcall M N)`).

(Kurzívou psané *`M`* a *`N`* zde znamená opět libovolný výraz jednoho ze tří uvedených typů.)

Pokud nerozumíte výrazům λ-kalkulu, můžete si je kdykoliv představit zapsané ve Scheme nebo Lispu.

Závorky použité v definici abstrakce a aplikace lze vynechat, pokud to nemění význam. Používáme přitom pravidlo, že u vícenásobné aplikace se postupuje zleva doprava:

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

a u abstrakce se za její tělo považuje všechno za tečkou až do konce celého výrazu nebo do pravé závorky:

$$\begin{aligned}
\lambda x.MNL &\equiv \lambda x.((M\,N)\,L),\\
\lambda x.M\lambda y.NL &\equiv \lambda x.(M\,(\lambda y.(N\,L))),\\
(\lambda x.M\lambda y.N)L &\equiv ((\lambda x.(M\,(\lambda y.N)))\,L).
\end{aligned}$$

Abstrakce popisují funkce (procedury) jednoho parametru. K popisu procedur více parametrů můžeme použít vnořené abstrakce:

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

Tělem výrazu $\lambda xy.M$ je tedy výraz $\lambda y.M$ a jeho tělem je výraz M.

Můžeme to chápat tak, že každou funkci s více parametry je možno aplikovat na jeden argument, výsledkem je funkce, která má o parametr méně (spotřebuje se první z původních parametrů). Tomu se v programovacích jazycích říká *currying*.

Pokud je proměnná parametrem abstrakce a současně se vyskytuje v jejím těle, hovoříme o *vázaném výskytu proměnné*. Ostatní výskyty proměnné se nazývají *volné*.

Pokud tedy má abstrakce tento tvar:

$$\lambda x.M,$$

pak všechny výskyty proměnné x ve výrazu M jsou vázané. V tomto výrazu:

$$x\,\lambda x.x$$

je první výskyt proměnné x volný a druhý vázaný. (Uvedení proměnné jako parametru abstrakce, tedy za znak λ, za výskyt nepovažujeme.)

Množinu všech volných proměnných výrazu M lze definovat takto:

- Je-li M proměnná x, pak množina volných proměnných výrazu M je $\{x\}$.
- Je-li M abstrakce $\lambda x.N$, pak její množina volných proměnných je rovna množině volných proměnných výrazu N bez proměnné x.
- Je-li M aplikace $(K\,L)$, pak je množina jejích volných proměnných rovna sjednocení množin volných proměnných výrazů K a L.

Množinu volných proměnných výrazu jsme popsali tak podrobně, protože ji budeme potřebovat v implementaci.

3 Redukce

Teď se můžeme podívat na to, jak se v λ-kalkulu výrazy upravují, tzv. **redukují**. Ukážeme si tzv. **β-redukci**, která je založena čistě jen na textovém nahrazení parametru abstrakce argumentem.

Jak už bylo nastíněno v první části, redukce je schopná simulovat vyhodnocovací proces ve Scheme. Abychom se soustředili čistě na vyhodnocovací proces, přijmeme nyní pro Scheme následující omezení:

1. Všechny používané výrazy Scheme budou jen proměnné, abstrakce a aplikace.
2. Výskyt proměnných, které nemají žádnou hodnotu, ve výrazu nám nevadí.

Ke druhému bodu: výraz $x\,y$ (po schemovsku `(x y)`) je legitimní výraz, i když neznáme hodnoty proměnných x a y (proměnné jsou ve výrazu volné). Výraz prostě neumíme dále zjednodušovat. Přestože vyhodnocení výrazu `(x y)` by ve Scheme mohlo vést k chybě. Můžeme si třeba představit, že výraz je podvýrazem jiného, většího výrazu. Obecně: všechny správně utvořené výrazy jsou platné, žádný nemůže vést k chybě.

Libovolný výraz tvaru $(\lambda x.M)N$ (tedy aplikace, jejíž hlavou je abstrakce) můžeme **redukovat** na výraz $M[x := N]$, což je značení výrazu M, ve kterém jsou **všechny volné výskyty proměnné** x nahrazeny výrazem N. Redukci zapisujeme jednoduchou šipkou, takže můžeme psát

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

Aplikace, jejíž hlavou je abstrakce, se nazývá stručně *redex* (redukovatelný výraz).

Jestliže výraz obsahuje jako svůj podvýraz redex, můžeme celý výraz zredukovat tak, že v něm zredukujeme ten redex. Redukci opět značíme šipkou:

$$(\lambda x.xx)MN \to MMN$$

Tady byl zredukován redex $(\lambda x.xx)M$, který je podvýrazem celého výrazu.

Někdy je při redukci potřeba přejmenovat parametr abstrakce. To si ukážeme za chvíli.

První příklady redukcí uvedené v předchozí části a zapsané ve Scheme můžeme teď zapsat takto:

$$(\lambda x.x)M \to M$$
$$(\lambda xy.x)MN \to (\lambda y.M)N \to M$$

M a N nyní ovšem mohou být libovolné jiné výrazy, nikoliv jen výrazy, které nejde dále redukovat, jak jsme říkali v první části.

Ve druhém příkladě si je třeba uvědomit, že $\lambda xy.x$ je zkratka pro $\lambda x.\lambda y.x$. Je to tedy abstrakce s parametrem x a tělem $\lambda y.x$. Proto při první redukci dosazujeme do výrazu $\lambda y.x$ za x (které se zde vyskytuje volně) výraz M. To je currying v λ-kalkulu.

Další příklady (opět převzaté z první části) ukazují, že pokud výrazy obsahují více redexů, lze je redukovat různými způsoby, protože si lze vybrat, v jakém pořadí redexy redukujeme. Jen některé z nich odpovídají tomu, jak by redukoval Scheme.

$$(\lambda xy.xK)((\lambda xy.x)LM)N \to (\lambda xy.xK)LN \to LK$$
$$(\lambda xy.xK)((\lambda xy.x)LM)N \equiv (\lambda xz.xK)((\lambda xy.x)LM)N$$
$$\to (\lambda z.(\lambda xy.x)LMK)N \to (\lambda xy.x)LMK \to LK$$

$$(\lambda x.Lxx)((\lambda yz.y)MN) \to (\lambda x.Lxx)((\lambda z.M)N) \to (\lambda x.Lxx)M \to LMM$$
$$(\lambda x.Lxx)((\lambda yz.y)MN) \to L((\lambda yz.y)MN)((\lambda yz.y)MN)$$
$$\to L((\lambda z.M)N)((\lambda yz.y)MN) \to LM((\lambda yz.y)MN)$$
$$\to LM((\lambda z.M)N) \to LMM$$

Ve druhém příkladě (tam, kde je symbol $\equiv$) jsme museli přejmenovat parametr y, aby nedošlo ke konfliktu názvů proměnných.

Výraz, který nelze dále redukovat, se nazývá *normální forma*. Je to výraz, který neobsahuje jako žádný svůj podvýraz redex. Příklady normálních forem:

$$x, \quad xx, \quad xyz, \quad \lambda x.x, \quad \lambda xyz.x, \quad x(\lambda y.xy)(\lambda z.xz)$$

Snažit se zredukovat výraz až na normální formu ovšem obvykle není to, co bychom chtěli. Většinou totiž není vhodné redukovat tělo abstrakce. Například u této abstrakce:

$$\lambda y.(\lambda x.xx)(\lambda x.xx)$$

redukovat tělo nechceme. Abstrakce odpovídá proceduře Scheme definované výrazem

```
(lambda (y) ((lambda (x) (x x))
             (lambda (x) (x x))))
```

Její aplikace by vedla k nekonečnému výpočtu, ale dokud proceduru neaplikujeme, nic se neděje, výpočet se nespustí. Ve většině případů (výjimky uvidíme) tedy platí, že **tělo abstrakce neredukujeme**.

Druhé pravidlo je poněkud ošidnější, protože se týká situací, které ve Scheme vyvolají chybu. Je založeno na principu, že pokud víme, že se redukcí nedostaneme k výsledku, protože neznáme hodnotu nějaké proměnné, nemá smysl redukci provádět. Argument následující aplikace (tedy výraz $(\lambda x.x)y$) tedy nebudeme redukovat, protože bychom ho stejně nemohli použít k další redukci:

$$x\,((\lambda x.x)y)$$

Ve Scheme:

```
(x ((lambda (x) x) y))
```

Pravidlo: **argument aplikace, jejíž hlava se neredukuje na abstrakci, neredukujeme**.

Když dáme tato dvě pravidla dohromady, zjistíme, že dále neredukovatelné výrazy jsou výrazy, kterým se říká *hlavová normální forma*. Jsou to tyto výrazy:

1. proměnné,
2. abstrakce,
3. aplikace, jejichž hlava je hlavová normální forma, která není abstrakce.

4 Vyhodnocovací modely

Pokud se ve výrazu vyskytuje více redexů, které chceme redukovat, musíme si jeden z nich k redukci vybrat. Který to bude, je věcí *redukční strategie*. V kontextu programovacích jazyků se také hovoří o *modelu vyhodnocení*. Tady si uvedeme dva nejdůležitější. (Ve zdrojovém kódu máte ještě třetí.) Cílem každého je převést výraz na hlavovou normální formu.

Aplikativní model vyhodnocení.

1. Pokud je výraz hlavová normální forma, redukce končí.
2. Pokud není, je to aplikace. Zredukujeme její hlavu.

3. Pokud je výsledný výraz hlavová normální forma, končíme.
4. Jinak je hlava abstrakce. Zredukujeme argument.
5. Pak provedeme β-redukci a pokračujeme od začátku s novým výrazem.

Tento vyhodnocovací model v principu používá Scheme, Lisp a mnohé další jazyky. Jsou samozřejmě drobné, ale nepodstatné rozdíly, například, že v některých případech může dojít k chybě.

Aplikativní model vyhodnocení nemusí dojít k hlavové normální formě, přestože ji výraz má. Místo toho vyhodnocování nikdy neskončí. (Příklad najdete mezi příklady tohoto textu.)

Normální model vyhodnocení.

1. Pokud je výraz hlavová normální forma, redukce končí.
2. Pokud není, je to aplikace. Zredukujeme její hlavu.
3. Pokud je výsledný výraz hlavová normální forma, končíme.
4. Jinak je hlava abstrakce. Provedeme β-redukci a pokračujeme od prvního bodu.

Tento vyhodnocovací model používají některé funkcionální programovací jazyky, například Haskell.

Pokud má redukovaný výraz hlavovou normální formu, redukce podle normálního modelu k ní vždy dojde. Pokud nemá, redukce nikdy neskončí.

Otázky a úkoly na cvičení

Kód k této části implementuje vyhodnocování výrazů Scheme podle principů λ-kalkulu (jazyk *Lazy Scheme*). Obsahem cvičení by kromě následujících příkladů mělo být hlavně testování interpretu a analýza jeho zdrojového kódu. Na tyto věci příště navážeme.

Ve všech příkladech si výrazy λ-kalkulu můžete přepsat do syntaxe Scheme.

1. Které z následujících výrazů jsou normální formy? A které jsou hlavově normální formy?

   a) x b) $x\,y$ c) $\lambda x.x$ d) $\lambda x.y$
   e) $x\,\lambda x.x$ f) $(\lambda x.x)\,y$ g) $x\,((\lambda x.x)\,y)$ h) $\lambda x.(\lambda y.x)z$

2. Výrazy v předchozím příkladě, které nejsou normální formy, redukujte na normální formy.
3. Totéž udělejte pro hlavově normální formy.