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

8. Normální model vyhodnocení

verze z 7. dubna 2020

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*, který je založen na myšlence líného vyhodnocení: výrazy jazyka jsou vyhodnocovány, až když je jejich hodnota potřeba. Tím se budeme zabývat v této a další přednášce. Abychom vyhodnocovací modely lépe pochopili, ponoříme se do jejich teoretického popisu pomocí *λ-kalkulu*, který budeme také ilustrovat na výrazech jazyka Scheme.

1 Vyhodnocování jako postupné substituce

Protože teď se budeme bavit o jazyce Scheme, vzpomeňme si, jak funguje jeho vyhodnocovací proces. Hlavní rozdíl proti Lispu je, že při vyhodnocování seznamu se jeho první prvek prostě vyhodnotí. Proto třeba výraz

```
((lambda (x) x) a)
```

se vyhodnotí na obsah proměnné `a`, protože se

1. nejdřív vyhodnotí `(lambda (x) x)` a dostane se procedura,
2. pak se vyhodnotí symbol `a`,
3. nakonec se procedura na hodnotu symbolu `a` aplikuje

(poslední bod by se dal podrobně rozebrat, nebudu to ale teď dělat).

Ve funkcionálních programovacích jazycích bývá použit jiný způsob vyhodnocování výrazů, založený na myšlence **líného vyhodnocování:** *výrazy se nevyhodnocují, dokud to není potřeba.* Tento výraz jazyka Scheme by se tedy vyhodnotil, aniž by se kdy počítal součet `(+ 2 2)`:

```
((lambda (x y) x)
 (+ 1 1)
 (+ 2 2))
```

Každý výpočet by se totiž odkládal až na nejzazší možnou dobu. Tady by se tedy nejprve „aplikovala“ funkce na *nevyhodnocené* výrazy `(+ 1 1)`, `(+ 2 2)`, takže bychom dostali výraz `(+ 1 1)`, který by se teprve vyhodnotil.

Můžeme si všimnout, že vyhodnocování výrazů lze tedy chápat jako postupné dosazování hodnot za parametry procedur. Například v klasickém Scheme:

```
((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 vyhodnocování mělo pokračovat dál, třeba i hodně složitě (záleží na proceduře `L`).

Všimněte si, že každý krok vyhodnocování v příkladech můžeme chápat tak, že se v těle nějakého λ-výrazu za jeho parametry dosadily (v klasickém Scheme vyhodnocené) argumenty. Takovému kroku říkáme v λ-kalkulu *redukce*. (Přesněji tento pojem vysvětlím později.)

(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.)

Například výraz `((lambda (x) x) M)` zredukujeme tak, že za parametr `x` výrazu `(lambda (x) x)` dosadíme v jeho těle argument `M`. Výraz

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

(je to samozřejmě tříprvkový seznam) jsme vyhodnocovali ve dvou redukcích: nejdřív jsme v podvýrazu `(lambda (x y) x)` za parametry `x` a `y` dosadili `L` a `M`, čímž jsme dostali výraz

```
((lambda (x y) (x K))
 L
 N)
```

Ve druhém kroku jsme pak za parametry `x` a `y` dosadili `L` a `N`. Tím jsme došli k (dále neredukovatelnému) výrazu `(L K)`.

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

```
((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)
```

Například v prvním případě jsme nejdřív dosazovali za parametry výrazu `(lambda (x y) (x K))` výrazy `((lambda (x y) x) L M)` a `N`, z nichž první jsme zatím neredukovali. To jsme udělali až ve druhém kroku.

Tento způsob je založen na principu neredukovat žádný výraz, dokud to není opravdu potřeba. Neredukované výrazy pouze dosazujeme za parametry procedur. Jde tedy o variantu **líného vyhodnocování**.

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 vícekrát. V posledním příkladě jsme museli dvakrát vyhodnocovat podvýraz `((lambda (x y) x) L M)`.
2. Jindy se zase podvýraz nemusí vyhodnocovat nikdy (s tím jsme se setkali už nahoře), 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
```

U druhého bodu byste se měli zastavit a krok po kroku si vyzkoušet, jak by vyhodnocení uvedeného výrazu probíhalo ve Scheme a jak líným způsobem vyhodnocení.

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 6. přednášky je zmínka o *iterátorech*. Jestliže v proměnné `iterator` máme uložen iterátor (třeba přirozených čísel), pak lispový výraz

```
(next iterator)
```

protože jej nelze nahradit jeho jednou získanou hodnotou (každé jeho vyhodnocení vrací jinou hodnotu).

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 lambda-kalkulu.

2 Výrazy λ-kalkulu

Teoretickým podkladem funkcionálních programovacích jazyků 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). Pokud pro vás není výraz λ-kalkulu srozumitelný, vždy si ho můžete na výraz Scheme (Lispu) převést. Při přípravě tohoto textu jsem zvažoval, zda vám vůbec tento jiný způsob zápisu ukazovat. Nakonec jsem se rozhodl, že ano, protože je mnohem stručnější a přehlednější než zápis ve Scheme (natož v Lispu).

Ve Scheme

1. lze **proměnné** 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.)

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) programovacího jazyka, které mají jeden parametr. 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.

Zápis $\lambda xy.M$ tedy nesmíme chápat doslova, protože abstrakce má vždy jen jeden parametr. Je to jen **zkratka** pro výraz $\lambda x.(\lambda y.M)$. Někdy říkáme, že druhý výraz je **kanonickým tvarem** výrazu prvního.

Můžeme tomu rozumět 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* a některé funkcionální jazyky (např. Haskell) ho podporují.

Kdyby Scheme podporoval currying, pak aplikace takové procedury

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

na jeden argument by nevedlo k chybě. Výsledkem aplikace

```
((lambda (x y) (+ x y))
 10)
```

by byla funkce jednoho parametru přičítající desítku, tedy funkce, kterou bychom mohli zapsat takto:

```
(lambda (y) (+ 10 y))
```

Při aplikaci by Scheme provedl **částečné dosazení**: dosadil by pouze za první parametr procedury, která má parametry dva.

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 a normální forma

Teď se můžeme podívat na to, jak se v λ-kalkulu výrazy 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 jsou schopné 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).

Tento výraz

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

je redex, protože je to aplikace – zapsána poctivě i se závorkami: $((\lambda x.xx)\,M)$ – a její hlava $\lambda x.xx$ je abstrakce. Redukce tohoto redexu je

$$(\lambda x.xx)M \to MM.$$

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 $\lambda x.\lambda y.x$, což je stále ještě zjednodušený zápis výrazu $\lambda x.(\lambda y.x)$. 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 výrazy vyhodnocoval 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:

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

4 Hlavová normální forma

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.**

Poznámka. Když v programovacím jazyce definujeme funkci, jistě nečekáme, že se její tělo bude vyhodnocovat předtím, než se funkce zavolá s nějakými argumenty, i když by formálně s výrazy v těle nějak manipulovat šlo. Redukovat tělo funkce se ovšem může pokusit kompilátor, když se snaží vytvořit rychlý program.

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 $(\lambda u.u)v$) tedy nebudeme redukovat, protože bychom ho stejně nemohli použít k další redukci:

$$x\,((\lambda u.u)v)$$

Ve Scheme:

```
(x ((lambda (u) u) v))
```

Redukce by totiž k ničemu nebyla: stejně bychom nemohli výpočet dokončit. Mělo by to smysl teprve až bychom se dozvěděli hodnotu symbolu `x` (tedy jaká procedura se bude aplikovat).

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

Když dáme tato dvě pravidla dohromady, zjistíme, že výrazy, které nechceme dále redukovat, 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.

5 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 použitého *modelu vyhodnocení* (také *strategie redukce*). Tady si uvedeme dva nejdůležitější. (Ve zdrojovém kódu máte ještě třetí.) Cílem každého je převést výraz na hlavovou normální formu.

V popisu dejte pozor na rozdíl mezi termíny *redukce* a *vyhodnocení* (v předchozích verzích souboru jsem v tom měl trochu zmatek). **Redukce** je jeden krok vyhodnocení, kdy se v aplikaci

Aplikativní model vyhodnocení.

Vyhodnocení výrazu E:

1. Pokud je E hlavová normální forma, vyhodnocení končí, výsledkem je výraz E.
2. Pokud není, je výraz E aplikace. Vyhodnotíme jeho hlavu, E s novou hlavou označíme F.
3. Pokud je F hlavová normální forma, končíme a výsledkem je F.
4. Jinak je hlava výrazu F abstrakce. Vyhodnotíme její argument, F s novým argumentem označíme G.
5. Pak provedeme redukci výrazu G a pokračujeme od začátku s novým výrazem.

Tento vyhodnocovací model v principu používá Scheme, Lisp a všechny běžné jazyky (kromě některých funkcionálních). Jsou samozřejmě drobné, ale nepodstatné rozdíly, například, že v některých případech může dojít k chybě (když je výraz proměnná).

Aplikativní model vyhodnocení nemusí dojít k hlavové normální formě, přestože ji výraz má. Místo toho vyhodnocování nikdy neskončí. Tedy: výpočet se může zacyklit, i když nemusí (příklad najdete mezi těmi, které jsem už uvedl).

Normální model vyhodnocení.

Vyhodnocení výrazu E:

1. Pokud je E hlavová normální forma, redukce končí a výsledek je E.
2. Pokud není, je to aplikace. Vyhodnotíme její hlavu, E s novou hlavou označíme F.
3. Pokud je F hlavová normální forma, končíme a výsledkem je F.
4. Jinak je hlava abstrakce. Provedeme 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. To je velký rozdíl proti aplikativnímu modelu: pokud se výpočet nemusí zacyklit, tak se nezacyklí.

Kód k této části implementuje vyhodnocování výrazů Scheme podle principů λ-kalkulu (jazyk *Lazy Scheme*).

Otázky a úkoly na cvičení

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) $(\lambda xy.x)\,y$ h) $x\,((\lambda x.x)\,y)$ i) $\lambda x.(\lambda y.x)z$

   j) $(\lambda xy.yx)\,((\lambda z.zz)\,y)$ k) $(\lambda xy.x)\,y\,((\lambda v.vv)\,(\lambda w.ww))$

2. Výrazy v předchozí úloze, které nejsou normální formy, redukujte na normální formy.

3. Totéž udělejte pro hlavově normální formy, a to jak aplikativním, tak normálním modelem vyhodnocení.

4. Doplňte interpret o funkci `ls-applicative-eval`, která realizuje aplikativní model vyhodnocení.