



Fakulta informatiky
Masarykovy univerzity

Převod do pointfree a pointwise

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Poslední modifikace: **26. ledna 2023**

Chyby, překlepy, nejasnosti a nejednoznačnosti prosím nahlašujte v diskusním fóru předmětu.

Převod do pointfree a pointwise

0.1 Úvod

Z hlediska uvádění formálních argumentů lze výrazy převést do dvou speciálních tvarů – *pointfree* a *pointwise*.

Pointfree tvar se vyznačuje tím, že neobsahuje žádnou λ -abstrakci, a tedy ani žádný formální argument. U jednodušších výrazů je tento tvar poměrně užitečný v pozici argumentu funkcí vyšších řádů, například `map`, `filter`, atd. Namísto `\x y -> x * y` lze psát `(*)` nebo namísto `\x -> map f (filter p x)` jenom `map f . filter p`. Odstraňuje se tím zbytečný šum a zápis je kratší. Výraz v *pointfree* tvaru také neobsahuje identifikátory formálních argumentů, které jsou leckdy arbitrárně označeny. Na druhou stranu, tento tvar je potřeba užívat s mírou, protože u složitějších výrazů může být na pochopení jen velmi stěží proniknutelný. Například jen málokdo pozná, že za

$$(. (,)) . (.) . (,)$$

se skrývá ekvivalentní zápis jednoduchého výrazu

$$\backslash x y z -> (x, (y, z))$$

Takovýto výraz je pak také náročnější na otypování.

Pointwise tvar je zas opakem *pointfree* – nejvíce vnější funkce musí mít přes λ -abstrakci uvedeny všechny formální argumenty, které může dostat. Tento tvar je explicitní v tom, kolik argumentů funkce přijímá a aplikace jsou v ní přímo viditelné.

Připomeňme, že *pointfree* a *pointwise* tvary jsou krajními tvary výrazů. Výraz nemusí být ani v jednom z nich, například výraz `\f g -> f . g` není v *pointfree*, ani *pointwise* tvaru, výraz `(.)` je v *pointfree* tvaru a `\f g x -> f (g x)` je v *pointwise* tvaru.

Úpravy do *pointfree*/*pointwise* tvaru, které lze provádět na výrazech, lze provádět i na funkcích (musí však být definována na jednom řádku). Tedy `fun x = f (g x)` lze obdobně jako `\x -> f (g x)` upravit na `fun = f . g`.

Zároveň, vzhledem na možnou variabilitu kroků, lze při převodech dospět k mírně odlišným výsledkům. Vzniklé výrazy však musí být ekvivalentní.

0.2 η -redukce (eta-redukce)

Základním nástrojem, který umožňuje převod do *pointwise* nebo *pointfree* tvaru, je η -redukce, někdy nazývaná také η -konverze. η -redukce je pravidlo, které říká, že výrazy `\x -> M x` a `M` jsou ekvivalentní, za předpokladu, že proměnná `x` se nenachází ve výrazu `M` *volná*. Volný výskyt proměnné je takový, který není vázán žádnou λ -abstrakcí ve výrazu. Opakem je *vázaný* výskyt. Pokud bychom totiž převedli výraz `\x -> (zipWith (+) x) x` na `zipWith (+) x`, je zjevné, že tato úprava není korektní. V dalším textu budeme mít pod výskytem proměnné automaticky na mysli volný výskyt.

Prakticky nám tato definice klade několik podmínek na to, kdy můžeme provést η -redukcí:

- redukovaná proměnná se musí nacházet v těle λ -abstrakce právě jednou,
- redukovaná proměnná se musí nacházet úplně na konci λ -abstrakce, tj. výraz musí být ve tvaru $\lambda x \rightarrow \dots x$, přičemž v něm musí existovat implicitní závorky $\lambda x \rightarrow (\dots) x$.

U implicitních závorek musí jít skutečně o nové závorky. Pokud bychom do výrazu $\lambda x \rightarrow 3 + x$ doplnili závorky na $\lambda x \rightarrow (3 +) x$, dostaneme ekvivalentní výraz s operátorovou sekcí, avšak třeba myslet na to, že tyto závorky nelze považovat za implicitní, nýbrž za součást syntaxe operátorové sekce. Totiž samotný výraz $3 +$ není korektní.

Pozor, η -redukované výrazy jsou sice funkčně ekvivalentní v tom smyslu, že kdekoli můžeme použít původní výraz, můžeme použít i η -redukováný výraz, nemusí však být typově ekvivalentní:

```
\x y -> x y :: (a -> b) -> a -> b
\x   -> x   :: a -> a           -- po eta-redukcí y
```

Všimněte si však, že typ η -redukováného výrazu je obecnější, než typ původního výrazu. Obecně, typ výrazu po η -redukcí je stejný, nebo obecnější než typ původního výrazu.

0.3 Převod do pointfree tvaru

Při tomto převodu opakujeme stejný postup. Dokud se ve výrazu nachází λ -abstrakce, odstraníme pomocí η -redukcí poslední formální argument. Tedy pokud máme λ -abstrakci $\lambda x y z \rightarrow \dots$, odstraňujeme postupně z , pak y a nakonec x . Na to, abychom mohli použít η -redukcí, musíme splnit podmínky uvedené v 0.2.

Předpokládáme, že odstraňovanou proměnnou máme ve výrazu v právě jednom výskytu (jinak viz 0.3.1). Cílem je pak dostat jí na konec těla λ -abstrakce. Toho lze dosáhnout různými úpravami přičemž vždy musíme mít na paměti cíl – dostat proměnnou na konec těla. Obecně můžeme narazit na několik situací, které popíšeme, a u každé uvedeme úpravu, která nás přiblíží k cíli.

Použitá notace: x – odstraňovaná proměnná, expr – výraz neobsahující proměnnou x , expr_x – výraz obsahující proměnnou x , \oplus – operátor.

- proměnná ve výrazu napravo od operátoru:

```
\x -> expr1 \oplus expr2_x
\x -> ((\oplus) expr1) expr2_x
```

nebo, pomocí operátorové sekce:

```
\x -> (expr1 \oplus) expr2_x
```

Příklad:

```
\x -> flip . const x
\x -> ((.) flip) (const x)
```

- proměnná ve výrazu nalevo od operátoru:

```
\x -> expr1_x ⊕ expr2
\x -> (flip (⊕) expr2) expr1_x
```

nebo, pomocí operátorové sekce:

```
\x -> (⊕ expr2) expr1_x
```

Příklad:

```
\a -> even a || any [True, False]
\a -> (flip (||) (any [True, False])) (even a)
```

- složená funkce

```
\x -> expr1 (expr2 x)
\x -> (expr1 . expr2) x
```

Příklad:

```
\f -> zipWith (flip f)
zipWith . flip
```

- proměnná před jinými parametry funkce

```
\x -> f expr1 expr2_x expr3 ... exprn
\x -> flip (f expr1) expr3 expr2_x ... exprn
```

Příklad:

```
\x -> take x [(1+), (2*)] 10
\x -> (flip take [(1+), (2*)]) x 10
```

nebo

```
\s -> take 2 s 10
\s -> flip (take 2) 10 s
```

- proměnná jako funkce

```
\x -> x expr1
\x -> id x expr1
```

Příklad:

```
\f -> f 1 10
\f -> id f 1 10
```

0.3.1 Úprava výrazu na jeden výskyt formální proměnné

Výše uvedený postup lze použít, pokud máme ve výrazu pouze jeden výskyt proměnné. V opačném případě je potřeba výraz upravit. K tomuto účelu je někdy zapotřebí přidávat do výrazu nové funkce (lze vystačit s `id`, `const`, `dist`, `flip`¹).

Pokud se nenachází ani jednou, použijeme funkci `const` na její doplnění. Majíce výraz `\x -> M`, kde `M` neobsahuje `x`, provedeme úpravu na `\x -> const M x`.

¹Funkce `dist` jako jediná ze jmenovaných není definována v standardní knihovně.

Poznámka: Funkce `dist` a její použití je nad rámec předmětu a nebude probíráno na přednášce ani zkoušeno. Její použití je zde zahrnuto pouze pro úplnost. V rozsahu předmětu můžete tedy předpokládat, že nikdy nebude vaší úlohou převádět do pointfree výrazy obsahující stejný argument v těle více než jednou.

Pokud se proměnná nachází v těle více než jednou, lze použít funkci `dist` (`dist f g x = f x (g x)`) a to tak, že výraz upravíme na tvar těla této funkce a následně ho nahradíme odpovídajícím výrazem s funkcí `dist`. Tuto úpravu opakujeme, dokud výraz neobsahuje právě jeden výskyt dané proměnné. Například `\x -> 1 <= x && x <= 10` převedeme následovně:

```
\x -> (&&) ((1<=) x) ((<=10) x)
\x -> ((&&) . (1<=)) x ((<=10) x)
\x -> dist ((&&) . (1<=)) (<=10) x
```

0.4 Převod do pointwise tvaru

Cílem převodu do pointwise tvaru je zvýraznění všech argumentů, na které se daný výraz vždy aplikuje. Výraz do pointwise tvaru dostaneme tak, že mu přidáváme argumenty a postupně ho zjednodušujeme vyhodnocováním tzv. *kombinátorových funkcí* (`id`, `const`, `flip`, `dist`, `.`, `...`), čímž upravíme výraz na přehlednější tvar a obvykle tím odhalíme potřebu dodání dalších argumentů, které třeba přidat.

Postup je následovný:

- Zjistíme, která funkce je zadaném výrazu nejvíce vnější.
- Pokud má tato funkce dostatek argumentů dle jejího typu, končíme.
- Pokud této funkci chybí nějaký argument, přidáme čerstvý (dosud nepoužitý) argument na konec hlavičky. Existující tělo uzavřeme do závorek a stejný argument přidáme i za tělo.
- Jde-li zároveň o kombinátorovou funkci, rozepíšeme ji dle definice (ale viz poznámka dále).

Při tomto postupu však třeba být také obezřetný. Vyhodnocení některých funkcí může způsobit ztrátu informací o typu výrazů. Například u výrazu `\x -> const True (not x)` dojde po vyhodnocení na `\x -> True` ke ztrátě informace o tom, že `x :: Bool`. V takovýchto případech ukončíme převod na pointwise tvar u výrazu `\x -> const True (not x)`, protože bychom jinak dospěli k (typově) neekvivalentnímu výrazu.

Rovněž je také možnost postupovat tímto algoritmem pro každou funkci nacházející se ve výrazu. Tedy `map (f . g)` lze upravit nejenom na tvar `\s -> map (f . g) s`, ale také na tvar

```
\s -> map (\y -> f (g y)) s
```