
Fakulta informatiky
Masarykovy univerzity

# Algoritmický postup otypování

## IB015 Neimperativní programování

**Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop**

Poslední modifikace: **23. září 2015**

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

# Algoritmický postup otypování

## 0.1 Úvod

Typování výrazů lze provádět v zásadě dvěma způsoby – *intuitivně* a *algoritmicky*.

U *intuitivního typování* neexistuje přesný postup. Snažíme se jenom na základě různých informací, které lze ze zadaného výrazu nebo funkce odvodit, zjistit požadovaný typ. To lze dělat představením si, co zadaný výraz dělá, například na konkrétních argumentech, určením typů vyskytujících se funkcí a jejich mentálním pospojováním do výsledného typu nebo zjednodušením výrazu tak, aby bylo typování jednodušeji proveditelné.

Tato metoda samozřejmě nese s sebou riziko chyby, nesprávného odhadu nebo jí nelze vůbec použít kvůli přílišné složitosti výrazu. Proto je vhodné ji používat jenom u jednodušších výrazů. Také při zjednodušování některých výrazů lze změnit jejich typ, což je zjevně nežádoucí. Jedná se zejména o výrazy obsahující funkci s polymorfním typem, z kterých po vyhodnocení dojde k zjednodušení výrazu a odstranění podvýrazů, které kladli omezení na původní tvar výrazu, například `tail [True]` $\rightsquigarrow$ `[]`.

V tomto textu se intuitivním typováním nebudeme dále zaobírat a soustředíme se na *algoritmický postup otypování*.

Popis celého tohoto postupu bude poměrně dlouhý, avšak na druhou stranu algoritmický postup typování bude vysvětlen podrobně (i na příkladech) a zmíní všechny případy, které mohou nastat.

## 0.2 Co je typem výrazu?

Tady připomeňme, co vlastně je (a co není) považováno za typ výrazu. Za typem výrazu považujeme nejspecifičtější typový výraz takový, který je kompatibilní (unifikovatelný) s každým možným konkrétním typem výrazu.

To znamená, že typ nemůžeme určit příliš obecně. Například neplatí `id :: a`, protože tento typ je příliš obecný a nevyjadřuje to, že `id` je funkce a vždy má argument.

Také příliš specifický typ není korektní, například `id :: Bool -> Bool`. Funkci `id` lze totiž aplikovat i na jiné argumenty než ty typu `Bool`.

Typ výrazu je jednoznačný až na:

- pojmenování proměnných,
- pořadí typových kontextů,
- případné uvedení dvou typových kontextů, kdy jeden implikuje druhý.

# 0.3 Unifikace typů

Předtím, než se pustíme do samotného algoritmického postupu otypování, je potřebné popsat *unifikace typů*. V průběhu typování se dostaneme do situace, kdy budeme mít dáno několik rovnic mezi typovými výrazy. Z nich lze zjistit informace o jednotlivých typových proměnných pomocí unifikace, tedy ztotožňování typů.

Postup unifikace je v jednodušších případech poměrně intuitivní. Třeba z typové rovnice

```
a -> (b, c) -> [[d]] = m -> n -> m
```

lze jednoduše nahlédnout, že `a = m = [[d]], n = (b, c)`.

Unifikaci typových výrazů lze provádět následovným způsobem, tzv. *strukturní indukcí*. Vždy posoudíme strukturu výrazů a podle toho, jakého jsou tvaru, je rozložíme na rovnice mezi podvýrazy, které opět tímto způsobem rekurzivně zpracujeme. Unifikace může selhat nebo být úspěšná. U úspěšné unifikace je výstupem dosazení za typové proměnné tak, aby se oba zadané výrazy ztotožnili.

1. `TyCon = TyCon`

   Pokud jsou oba typové výrazy stejné typové konstruktory bez argumentů (například `Bool, Int, ...`) a unifikace je triviálně úspěšná.

2. `TyCon x1 x2 xn = TyCon y1 y2 ... yn`

   Pokud jsou oba typové výrazy stejné typové konstruktory (`TyCon`) (automaticky pak mají i stejnou aritu), musíme postupně unifikovat typové argumenty: `x1 = y1, x2 = y2, ..., xn = yn`. Tento případ zahrnuje případy 3–5, které však pro přehlednost rozepíšeme speciálně. Sem také spadají i uživatelsky definované datové typy.

3. `(x1, x2) = (y1, y2)` nebo `(x1, x2, x3) = (y1, y2, y3)` nebo ...

   (`TyCon` $\in \{$ `(,)` , `(,,)` , ... $\}$) U všech případů $n$-tic dostáváme `x1 = y1, x2 = y2, ..., xn = yn`.

4. `[x] = [y]`

   (`TyCon = []`) Musí platit `x = y`.

5. `x1 -> x2 = y1 -> y2`

   (`TyCon = (->)`) Opět musí platit `x1 = y1, x2 = y2`. U funkčních typů se běžně vyskytuje více šipek. Snadno lze tedy unifikaci zobecnit na `x1 -> x2 -> ... -> xn = y1 -> y2 -> ... yn` $\implies$ `x1 = y1, x2 = y2, ..., xn = yn`. Co však v případě, že je na jedné straně více šipek než na druhé? Operátor šipka má asociativitu zprava, tedy `a -> b -> c` $\equiv$ `a -> (b -> c)`. To tedy znamená, že u dvou funkčních typů s různým počtem šipek postupujeme zleva a dáváme do rovnosti typové podvýrazy, až nám na jedné straně zůstane poslední podvýraz. Ten pak dáme do rovnosti se zbytkem z druhé strany: `x1 -> ... -> xm = y1 -> ... -> yn`.

6. `a = ...`

   Triviálně platí.

Několik poznámek:

- Pokud jsme ze získaných typových rovnic schopni dojít k rovnici, kde nějaká typová proměnná se rovná výrazu, který v sobě obsahuje tu stejnou proměnnou, znamená to, že výraz nelze otypovat. Na základě této rovnice je totiž možno zkonstruovat *nekonečný typ*. Například pokud máme `a = [a]`, pak také `a = [a] = [[a]] = [[[a]]] = ...`
- Uvedené body popisují případy, kdy je možné výrazy daného tvaru dál unifikovat v podvýrazech. Pokud nelze uplatnit ani jedno pravidlo, pak to znamená, že typové výrazy nelze unifikovat kvůli nekompatibilním typovým konstruktorům (`Bool = Int, [a] = (b, c), ...`
- Typové třídy unifikaci neovlivňují, jenom se propagují přes rovnosti, tj. je-li `a = b, Integral a => a`, pak také `Integral b => b`.

## 0.4 Postup

Postupem se zaměříme na otypování výrazu. Jak postup aplikovat na jiné konstrukce (funkce) najdete v části 0.5.

Nejprve stručně popíšeme jednotlivé kroky:

1. Každý výskyt funkce, konstanty a formálního argumentu otypujeme (nezávislé typové proměnné).
2. Určíme rovnosti vyplývající z aplikací funkcí na argumenty (aplikace vynucuje shodu typu formálního a skutečného parametru).
3. Rozepíšeme a zjednodušíme rovnosti na co nejjednodušší.
4. Vyjádříme všechny typové proměnné pomocí minimální množiny typových proměnných.
5. Určíme hledaný typ a dosadíme do něj vyjádření.
6. Zohledníme typové kontexty, jsou-li nějaké.
7. Volitelně převedeme typ do kanonického tvaru.

### 0.4.1 Otypování vstupů

Ve výrazu nalezneme všechny identifikátory (funkce, konstanty nebo formální argumenty) a každý(!) výskyt nezávisle otypujeme. Výnimkou jsou formální proměnné, které jsou definovány v hlavičce $\lambda$-abstrakce. Tj. například ve výrazu `head . head` nelze určit typ `head` jenom jednou, nýbrž je ho potřeba určit pro první a druhý výskyt. Pro přehlednost lze výskyty indexovat: $\texttt{head}_1$ `.` $\texttt{head}_2$. Nezávislé otypování zas znamená, že pokud jsme v type jedné funkce použili nějakou proměnnou, nesmíme jí použit v typu jiné funkce. Příkladem chybného závislého otypování výrazu $\texttt{head}_1$ `.` $\texttt{head}_2$ je $\texttt{head}_1$ `:: [a] -> a,` $\texttt{head}_2$ `:: [a] -> a`, protože bychom pak dostali falešnou rovnici `a = [a]`, která zabrání v otypování výrazu, avšak tento výraz otypovatelný je.

**Typování konstrukcí Haskellu** Známé funkce a konstanty (čísla, řetězce, ...) otypujeme na základě jejích typu.

Formální argumenty otypujeme typovou proměnnou.

$\lambda$-abstrakce `\x1 x2 ... xn -> body` otypujeme jako `type(x1) -> type(x2) -> ... -> type(xn) -> type(body)`. Typ `body` určíme z typu funkce (nebo operátoru), která je ve výrazu nejvíce vnější, přičemž z její funkčního typu odstraníme tolik prvních typových argumentů, na kolik argumentů je funkce aplikována.

Výraz tvaru `if cond then branch1 else branch2` otypujeme jako `b`, kde `cond :: a, branch1 :: b, branch2 :: c`. Do typových rovnic v kroku 0.4.2 přidáme `b = c, a = Bool`.

Výrazy s (levou nebo pravou) operátorovou sekcí otypujeme jako (`b -> c` nebo `a -> c`), kde `a -> b -> c` je typ operátoru a do typových rovnic v kroku 0.4.2 přidáme (`t = a` nebo `t = b`), kde `t` je typem „vestavěného“ operandu operátorové sekce.

U seznamových výrazů určíme typ jako `[a1]`, kde na místě `a1` bude typový výraz odpovídající typu prvního prvku výrazu (protože je stejný jako typ ostatních prvků). Stejnost typů jednotlivých prvků zajistíme přidáním rovnic `a1 = a2, a2 = a3, ... a(n-1) = an` v kroku 0.4.2.

### 0.4.2 Určení aplikačních rovností

Následně najdeme ve výrazu všechny aplikace funkcí na všechny jednotlivé argumenty. Aplikace totiž vynucuje, že typ formálního argumentu funkce se musí rovnat typu skutečného argumentu. Tedy pokud máme výraz `f x y z` a máme `f :: a -> b -> c -> d, x :: e1, y :: e2, z :: e3`, tak získáme tři aplikační rovnice: `a = e1, b = e2, c = e3`. Stejně postupujeme u binárních operátorů.

Typové třídy při určování aplikačních rovnic nebereme do úvahy – budeme je zohledňovat až na konci.

### 0.4.3 Rozepsání aplikačních rovností na elementární

Tento krok je obyčejně poměrně jednoduchý, avšak získané rovnice můžou být někdy poměrně složité (`a -> [[b]] -> d = (b, c) -> c -> a`, proto je může být třeba rozepsat tak, aby v každé rovnice byla alespoň na jedné straně typová proměnná. To lze dosáhnout unifikováním obou stran rovnic.

### 0.4.4 Minimální vyjádření typových proměnných

Typové proměnné budeme potřebovat při dosazování unifikací zjištěných informací do hledaného typu. Cílem minimalizace je zajistit, že v hledaném typu budou rovnající se typové proměnné vyjádřeny vždy jen pomocí jedné z nich. V opačném případě bychom nesprávně určili obecnější typ.

Pokud jsou typové rovnice jednoduché, je tento krok snadný a můžeme jej provést tak, že pro každou typovou proměnnou prozkoumáme rovnice a zjistíme nejvíce omezující typ, který na

základě nich pro ní vyplývá. Přitom recyklujeme typové proměnné vždy, když je to možné. Tento postup může být riskantní v tom, že bychom u obdobného příkladu typových rovnic jako je `a = [b], d = e, a = [[c]]` přehlídli fakt, že `b = [c]`.

U složitějších typových rovnic lze použít postup, který má velmi blízko k řešení soustavy lineárních rovnic pomocí matic (úprava na trojúhelníkový tvar a zpětné dosazování).

Ze sady původních typových rovnic vybereme vyjádření některé typové proměnné, poznačíme si jí na začátek seznamu a do vytvoříme novou sadu typových rovnic, kde každý výskyt této typové proměnné rozepíšeme podle vybrané rovnice. V případě, že dostaneme typovou rovnici, kde na ani na jedné straně není typová proměnná, rozložíme ji na menší rovnice dle unifikačního algoritmu. Pokud by vznikla rovnice, která má nekompatibilní typové konstruktory nebo vyjadřuje nekonečný typ, lze typování ukončit a vyhlásit otypovávaný výraz za typově nekorektní a neotypovatelný. Pokud se některá rovnice vyskytuje vícekrát, všechny její duplikáty odstraníme. Jinak tento postup opakujeme na vzniklou sadu typových rovnic, která je o jednu rovnici menší a další vyjádření typových proměnných přidáváme na konec. Takto postupujeme, až zůstane sada typových rovnic prázdná. Tento stav odpovídá u matic získaní trojúhelníkového tvaru.

Pak z konce seznamu vybereme typovou rovnici a typovou proměnnou, kterou vyjadřuje, dosadíme do všech rovnic v seznamu, kde se vyskytuje. Takto postupujeme až se dostaneme na začátek seznamu. Tento postup odpovídá u matic zpětnému dosazování proměnných a výsledkem je diagonální matice.

Po tomto posledním kroku máme vyjádřeny všechny typové proměnné minimálně.

### 0.4.5 Určíme hledaný typ

Na to, abychom minimální vyjádření měli jak použít, musíme určit vyjádření typu, který hledáme. To uděláme dle postupu pro určení typu $\lambda$-abstrakce, který jsme uvedli v části 0.4.1.

Následně za všechny typové proměnné v tomto výrazu dosadíme jejich minimální vyjádření.

### 0.4.6 Zohlednění typových kontextů

Pokud se při otypování vstupů vyskytli typové kontexty, nezapomeneme je na základě rovnic mezi typy propagovat mezi všechny typové proměnné. Do hledaného typu je však přidáme pouze když se v něm odpovídající typová proměnná vyskytne.

Volitelně lze odstranit ty typové kontexty, jež jsou implikovány jinými, například `Integral a ⇒ Num a` nebo `Num a ⇒ Eq a`.

### 0.4.7 Kanonický tvar

Tento krok je volitelný. Aby bylo kontrolování typu snadnější, lze typové proměnné přejmenovat tak, že když budeme přecházet hledaný typ zleva (neuvažujíce typový kontext), první použitá typová proměnná bude `a`, další nová typová proměnná `b`, pak `c`, atd.

## 0.5 Typování jiných případů

Předchozí postup lze použít pouze pro výrazy.

Pokud chceme typovat funkci a je definována pomocí jedné rovnosti, můžeme ji převést na $\lambda$-abstrakci: `f 1 _ x -> length x + 2` převedeme na `\1 _ x -> length x + 2` a získaný typ $\lambda$-abstrakce bude typem funkce `f`.

Pokud chceme otypovat funkci definovánu pomocí více rovností, lze tento problém převést na otypování seznamu odpovídajících $\lambda$-abstrakcí:

```
f True  x = 10
f False x = x - 1
```

převedeme na otypování `[\True x -> 10, \False x -> x - 1]`, načež získáme typ tvaru `[a]` a hledaným typem bude `f :: a`.