Funkcionální programování

Úvod

Petr Pudlák

5. října 2011

Funkcionální programování?

- Imperativní programování je založeno na příkazech, které mění stav.
- Funkcionální programování je založeno na vyhodnocování výrazů. Vyhýbá se změnám stavů a proměnlivým datům.

Funkcionální programování

Výhody

- Funkcionální jazyky umožňují vyšší míru abstrakce než imperativní jazyky, a snažší rozdělení do komponent. Například, opakující se postup lze často abstrahovat pomocí funkcí vyšších řádů, a tím vzniká srozumitelnější a deklarativnější kód.
- Často jsou funkcionální programy kratší srozumitelnější, než odpovídající imperativní programy. (Někde se uvádí 2x až 10x.)[1]
- Silné typové systémy umožňují větší kontrolu kódu při překladu.[2]
- Neměnitelná data umožňují překladači účinněji optimalizovat (nebo dokonce paralelizovat) výsledný kód.
- Líné vyhodnocování často výrazně urychlí některé algoritmy (příklad).

[1] Prý studie ukazují, že efektivita programátora je více méně stejná v množství napsaného kódu nezávisle na jazyce. Takže možnost kratšího kódu vede k větší produktivitě.

[2] Kdosi se vyjádřil, že když vám podaří zkompilovat program v Haskellu, tak už bude fungovat.

Funkcionální programování

Nevýhody

- Počítače jsou navrženy a optimalizovány na imperativní programy.
- Některé algoritmy jsou z principu imperativní a těžko se implementují ve funkcionálních jazycích. Např. hašovací tabulka.
- Výsledný kód je často méně efektivní (konstantním faktorem), zvlášť u jazyků s líným vyhodnocováním.
- ... a s tím souvisí složitější překlad a nutnost specializovaných optimalizací.

Funkce vyššího řádu

Funkce jsou objekty první třídy[3].

- Hodnoty jsou funkce s 0 argumenty.
- Funkce tedy mohou být použity jako parametry do jiných funkcí (kterým se někdy říká *funkce vyššího řádu*), nebo jako výsledky výpočtů.
- Lze tak snadno abstrahovat opakující se postupy.
  Například, naprostou většina operací na seznamech lze provést pomocí *foldr*.

$$subtractTwoFromList :: Num\ \alpha \Rightarrow [\alpha] \to [\alpha]$$
$$subtractTwoFromList\ xs = map\ (\lambda x \to x - 2)\ xs$$

[3] Doslovný překlad *first-class objects* do češtiny *prvotřídní objekty* zní poněkud divně.

Uzávěry

- Funkce mohou být vnořené.
- Vnořené funkce se mohou odkazovat na proměnné nadřazených funkcí.

$$\mathit{subtractNFromList} :: \mathit{Num}\ \alpha \Rightarrow \alpha \to [\alpha] \to [\alpha]$$
$$\mathit{subtractNFromList}\ v\ xs = \mathit{map}\ (\lambda x \to x - v)\ xs$$

- Funkce $\lambda x \to x - v$ se odkazuje na v.
- Takové vnořené funkce mohou „unikat" z kontextu.
- Na výraz $\mathit{subtractNFromList}\ 3$ se lze také dívat jako na funkci vyššího řádu, která z čísla vytváří funkci.
- Protože hodnoty mohou takto „unikat" z kontextu, je nutno mít odpovídající správu paměti.

Rekurze

- Rekurze je základní prvek funkcionálních jazyků umožňující iterovat výpočty.
  (Klasické cykly nejsou, vyhýbáme se změnám stavů.)
- Rekurzi lze nahradit specializovaným kombinátorem.
  To se hodí spíše pro teoretické aspekty, prakticky je efektivnější a čitelnější klasický rekurzivní zápis.
- Cykly jsou vyjádřeny pomocí *tail recursion*.
- TODO //Recursion is the root of computation since it trades description for time.//

Typové systémy

- Výrazům jazyka jsou staticky[4] přiřazovány typy, a to automaticky nebo programátorem.
- Výsledek výpočtu výrazu má stejný typ jako původní výraz.
- Typy zajišťují, že program nikdy nezhavaruje proto, že by přistupoval k datům nesprávným způsobem.
- Typy nám poskytují informaci o výsledku výpočtu před jeho vlastním provedením.
- Typické konstrukce typových systémů:
  - Typ pro funkce – funkce z typu a do typu b má typ $a \to b$.
  - Typy pro uživatelské datové struktury.
  - Parametrizované polymorfní typy (např. funkce počítající délku seznamu).
  - Ad-hoc polymorfismus (např. rovnost definovaná různě na různých typech).

[4] Staticky = při překladu, dynamicky = za běhu.

Typové systémy

Síla (expresivita) typových systémů

- Problém:
  - Slabé typové systémy přinášejí malý užitek.
  - Silné typové systémy neumožňují odvození typů, takže programátor musí explicitně uvádět typy výrazů.
- Kompromis – Hindley–Milnerův typový systém umožňuje jednoznačné odvození nejobecnějšího typu výrazu/funkce (princip podobný unifikaci), a přitom je dostatečně silný (polymorfismus).
- Od vzniku Hindley–Milnerova typového systému v 70-tých letech se začaly ve větší míře používat ve funkcionálních jazycích typové systémy odvozené z typovaných lambda kalkulů.
- Existují i systémy s tzv. závislými typy, které umožňují v typu postihnout jakýkoliv predikát.

Algebraické datové typy a rozpoznávání vzorů

- Z primitivních (a funkčních) typů konstruujeme nové typy pomocí (pojmenovaných) direktních součinů a součtů.
  (To vzdáleně odpovídá konstrukcím *struct* a *union* z C.)
- Typy mohou být rekurzivní.
- Hodnoty těchto typů pak můžeme rozpoznávat podle vzorů a zjišťovat tak hodnoty, z kterých byly zkonstruovány. U direktní součtů pak větvit výpočet.

Algebraické datové typy a rozpoznávání vzorů

Příklad

Example

Seznam je buďto prázdný, a nebo je to dvojice hodnota a zbytek seznamu.
Matematicky:

$$\text{List } a = \mu t \,.\, 1 \oplus (a \otimes t)$$

V Haskellu:

$$\textbf{data } \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

Součet čísel v seznamu můžeme zapsat:

$$\begin{aligned} &\mathit{sum} :: \mathit{List}\ \mathit{Int} \to \mathit{Int} \\ &\mathit{sum}\ \mathit{Nil} = 0 \\ &\mathit{sum}\ (\mathit{Cons}\ x\ xs) = x + \mathit{sum}\ xs \end{aligned}$$

Algebraické datové typy a rozpoznávání vzorů

Příklad – tail recursion

Example

```
sum2 :: List Int → Int
sum2 xs = s 0 xs
  where
    s r []          = r
    s r (Cons x xs) = s (r + x) xs
```

Parametr r nese hodnotu mezivýsledku.

Example

```
sum3 :: List Int → Int
sum3 = foldl (+) 0
```

Čistota

- Některé funkcionální jazyky umožňují funkcím, aby prováděly akce mimo výpočtu svého výsledku.
- Tyto akce se nazývají *vedlejší efekty* (side effects)[5].
- Jazyky, které neumožňují vedlejší efekty funkcí se nazývají *čisté* (pure).
- I v nečistých jazycích je vhodné psát kód tak, aby co největší část zůstala čistá – nečistý kód je obvykle náchylnější k chybám.

[5] Hlavní efekt funkce je výpočet výsledku.

Čistota

Neměnitelná data

- Čisté jazyky neumožňují měnit data.
  (Změna dat by byla vedlejším efekt.)
- Místo změny datové struktury se vytvoří struktura nová, upravená o potřebné změny.
- Původní struktura zůstane zachována.
  Pokud není využita, tak její nadbytečné části jsou zlikvidovány GC.
- Optimální algoritmus zachová maximum původní struktury.
- Například:
  - U vyvážených binárních stromů při přidání prvku vznikne jen $O(\log n)$ nových uzlů.
  - Speciální datová struktura pro konečné sekvence má amortizovanou složitost[6] $O(1)$ pro operace na koncích a $O(\log n)$ pro operace uvnitř.

[6] Průměrná složitost při sledu operací v nejhorším případě.

Čistota

Referenční transparentnost

- *Referenční transparentnost:* Výsledek čisté funkce závisí pouze na jejích argumentech.
- To umožňuje odvozovat a dokazovat vlastnosti programů vlastnosti.

Čistota

Vedlejší efekty versus čistota

- Programátor čas od času potřebuje i "nečisté" prostředky, jako například:
  - Komunikace s uživatelem.
  - Zapisovat/číst do souborů.
  - Komunikovat po síti.
  - Používat a měnit velká pole.
- V praxi se používají 2 přístupy:
  1. Dovolí se vedlejší efekty funkcí (OCaml, F#).
     To vyžaduje větší disciplínu programátora.
     Nejde dohromady s líným vyhodnocováním.
  2. Vedlejší efekty se popíší pomocí *monád* (Haskell).
     Tento přístup zachovává čistotu a je konzistentní s líným vyhodnocováním.
     Zároveň lze z kódu jasně vyčíst, kde se provádějí jaké "nečisté" akce.

Líné versus striktní vyhodnocování

- *Líné* vyhodnocování (Haskell).
  - Výraz se vyhodnocuje teprve tehdy, když je jeho hodnota potřeba.
  - Výraz se vyhodnocuje jen do té míry, kolik je v dané situaci nutné.
  - Místo hodnoty se funkcím v argumentu předává tzv. *thunk*[7], který říká, jak hodnotu spočítat.
  - Nejde dohromady s vedlejšími efekty – nevíme, kdy/zda se daná funkce vyhodnotí.
  - Například *length* $[1, 2 \uparrow 1000, \bot]$ bude vyhodnoceno na 3 v čase $O(3)$.
  - Nevýhoda: Nevyhodnocený strom *thunk*ů může někdy nabývat nadměrných rozměrů – vzniká zcela zbytečná paměťová zátěž. Řešení: možnost přinutit překladač k striktnímu vyhodnocování, případně automatická detekce.
- *Striktní* vyhodnocování (známé z imperativních jazyků) naopak nejprve vypočte hodnotu výrazu, než ji použije jako argument do funkce.

[7] SAMPA: /TVNk/

Srovnání funkcionálních jazyků

	Vyhodnocování	IO	Typování	Čistý
Haskell	líné	monády	statické	ano
Clean	líné	?	statické	ano
Miranda	líné	líné seznamy	statické	ano
ML	striktní	vedlejší efekty	statické	ne
Scheme	striktní	vedlejší efekty	dynamické	ne
Erlang	striktní	vedlejší efekty	dynamické[8]	ne
OCaml	striktní	vedlejší efekty	statické	ne
F#	striktní	vedlejší efekty	statické	ne
Scala	striktní[9]	vedlejší efekty	statické [10]	ne

[8] Možnost anotací pro překladač.

[9] Jsou ale určité možnosti jak simulovat líné

[10] Částečně i dynamické do míry, kterou poskytuje JVM.

Literatura I