PRINCIPY FUNKCIONÁLNÍHO PROGRAMOVÁNÍ

JIŘÍ POLÁK
katedra počítačů fel čvut
karlovo n. 13, praha 2

CO ŘÍCI PŘEDEM

Tak jako i s jinými pojmy zachází se s termínem *funkcionální programování* dosti volně. V každém případě to je pojem, který souvisí s budoucností programování, a proto - jako to módní - lze ho slyšet z mnoha úst hlásajících budoucnost. Nebudeme zde snášet argumenty, proč budoucnost patří funkcionálnímu programování (a ovšem nejen jemu). Zaměříme se na postihnutí hlavních odlišností a charakteristických rysů a pokusíme se postihnout, jak se pohled na principy funkcionálního programování měnil.

NĚCO Z HISTORIE

Již v jednom z prvotních návrhu normy jazyka ALGOL-60 byla obsažena pasáž přímo podporující funkcionální programování, do pozdějších definic se již nedostala. Nicméně skutečná matematikú si stále vedla svou a tak se počátkem šedesátých let zrodil programovací jazyk LISP. Tehdy se zdůrazňovala maximálnost výpočtů a práce se seznamy. LISP se začal používat zejména v oblasti umělá inteligence - tedy v oblasti výzkumných projektů mající daleko ke komerčnímu použití.

Minulost funkcionálního programování je tedy spojena s jazykem LISP a jeho implementacemi. Samotný jazyk LISP se neustále vyvíjí a stává se základem architektury specializovaných počítačů (LISP MACHINE tj. Symbolics v USA,

pracovní stanice WINE a její procesor AI IP704 fy. Toshiba v Japonsku).

V evropském měřítku pocházejí nové funkcionální jazyky zejména z Velké Británie. Patří sem jazyky SASL, KRC a Miranda (D. Turner University of Kent), ML (R. Milner Edinburgh University), Orwell (P. Walder Oxford University), Hope (R. Burstall, D. MacQueen, D. Sanella Edinburhg). Za velmi perspektivní se považuje zejména jazyk Miranda. Na MIT v USA je propagován následovník LISPu - jazyk Scheme.

FUNKCIONÁLNÍ PROGRAMOVÁNÍ (VČERA)

Co to je včera v naší republice, nemusí být nutně včera na celém světě. Zdůrazňujeme zde, použitím slovíčka včera, že to, co mnohým může připadat jako hudba budoucnosti, je někde jinde již skutečně trošku zastaralé. Nebudeme zde tedy hovořit o programování v jazyce LISP, tomu je věnován speciální příspěvek, ale o jiných již existujících systémech funkcionálního programování.

Nejprve si ukážeme na příkladech možností moderních funkcionálních jazyků. Pozn. *ležatým písmem, tj. kurzívou* jsou v textu vyznačeny části programů a v příkladech komunikace pak označují části vypsané systémem.

Funkcionální systémy vyhodnocují výrazy:

```
F> 7 + 9
16
```

Možnost definovat funkci, zde jen pojmenovat hodnotu výrazu, je charakteristickým rysem funkcionálního programování. Po definici následuje použití právě definované funkce.

```
F> def square x = x * x
square
F> square (3 + 3)
122
```

Funkce *f* je definována pomocí dvou případů. *x > 10* a *otherwise* jsou tzv. stráže (angl. guards), které musí v rámci jedné definice funkce pokrývat všechny možné případy a přitom být vzájemně disjunktní. Stráž *otherwise* je pravdivá, pokud žádná jiná stráž není pravdivá. Hodnota funkce se určí z výrazu, jehož stráž je pravdivá. Na pořadí vyhodnocení stráží nezáleží. V definici *f* je použito také lokální deklarace hodnoty *a*.

```
?> def f x y = x + a if x > 10
             = x - a otherwise
               where a = square(y + 1)
f
```

Rozdíl mezi min a min' je v jejich typu, viz také dále.

```
?> min( x, y)  = ...
min
?> min' x y    = ...
min'
```

min má jeden argument, dvojici čísel. *min'* je funkce, která má za argument číslo a její hodnotou je funkce; takový převod funkcí více argumentů na jednoargumentové funkce vede k větší čitelnosti programů, protože nevyžaduje používání tolika závorek.

Základním strukturovaným datovým typem ve funkcionálních jazycích je seznam. Seznam může být určen výčtem jeho prvků nebo zápisem intervalu. Seznam se používá i k popisu množiny hodnot splňujících nějakou podmínky či podmínky. Takový příklad následuje.

```
?> [ square x | x ← [1,,10]; even x]
[4, 16, 36, 64, 100]
```

Nad seznamy pracují funkce vyšších řádů, viz *map* a *filter* dále.

```
?> def map f listx = [ f x | x ← listx ]
map
?> map square [9, 7, 0]
[81, 49, 0]
```

```
?> filter p listx  = [x | x ← listx; p x ]
filter
```

Následující definice funkce je opět pomocí výběru z několika možností, které ale nejsou rozpoznávány podle vyhodnocení stráží. Zde se používá mechanismu nazývaného *porovnávání vzorů*, angl. pattern či pattern matching. Pro vzory použité na levé straně rovností nicméně musí platit totéž, co pro stráže: pokrytí všech případů a vzájemná disjunktnost. Proto také nezáleží na pořadí uvedení rovností pro vzory.

```
?> def length []       = 0
      length(x:listx)  = 1 + length listx
length
?> length [a, b, g, k]
4
```

Tolik ukázky jednoduchých funkcionálních programů, v nichž se vůbec nevyskytují typy. Informace o typech argumentů funkcí je velmi důležitá pro efektivní provádění programů, tedy pro výpočet hodnot funkcí, tedy pro redukci na základní tvar. V některých funkcionálních systémech musí uživatel typy deklarovat, jiné - jako např. jazyk ML odvozují typy funkcí a jejich argumentů automaticky. Předchozí příklady by tedy byly doplněny následující typovou informací:

```
?>  7 + 9
16 :: num
?> def square x = x * x
square :: num → num
?>  square (3 + 9)
122 :: num
?>  def f x y = x + a if x > 10
               = x - a otherwise
                 where a = square(y + 1)
f :: num → (num → num)
?> min(x, y)  = ...
min :: (num, num) → num
?> min' x y    =
min' :: num → (num → num)
?> [ square x | x ← [1..10]; even x]
```

```
[4, 16, 36, 64, 100] :: [num]
?> def map f listx = [ f x | x ← listx ]
map :: (α → β) → [α] → [β]
?> map square [9, 7, 0]
[81, 49, 0] :: [num]
?> filter p listx  = [x | x ← listx; p x ]
filter :: (α → bool) → [α] → [α]
?> def length []        = 0
      length(x:listx) = 1 + length listx
length :: [α] → num
?> length [a, b, g, k]
4 :: num
```

Typy jsou určeny ke každému výrazu či funkci a navíc je použito typových proměnných (např. α) pro polymorfní funkce.

K dalším rysům existujících funkcionálních systému patří možnost používat nekonečné seznamy (někdy označované jako tzv. proudy - angl. streams) a možnost definovat nové datové struktury. Příklady následují:

```
?> [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
{ přerušeno}
?> stupeň ::= Celsius | Kelvin | Fahrenheit
stupeň :: new-type
```

Dodržení konzistentnosti sytému při zavedení nekonečných seznamů vede k některým problémům (např. vyhodnocení výrazu *[x | x ← [1..]; x < 10]* nevede k desetiprvkovému seznamu ale k zacyklení), a proto se spíše setkáme s přístupem, kdy nekonečný seznam je základní datovou strukturou a konečné seznamy jsou jen speciální případy, např. v jazyce LUCID. Definice uživatelských typů zase ztěžuje inferenci typů a také není dosud tak vyžadována.

CO PLATÍ VŽDY

Funkcionální program je výraz mající hodnotu, přičemž postup vyhodnocení jednotlivých elementů výrazu nehraje roli. Místo

o vyhodnocení výrazu se hovoří také o redukci výrazu na hodnotu.

Druhým aspektem funkcionálního programování je (kromě vyhodnocování výrazů) vytváření definic funkcí, které jsou pak ve výraze k vyhodnocení použity.

Funkcionální program se tedy skládá z definic funkcí a z výrazu, který popisuje řešení úlohy pomocí kombinace definovaných a systémových neboli primitivních funkcí. Hodnota výrazu je pak výsledkem programu.

Již zmíněnou, ale velmi podstatnou vlastností funkcionálních programů a funkcí je *referenční transparentnost* (angl. referential transparency). To znamená, že funkce nemají žádné vedlejší efekty a že každou část výrazu lze vždy nahradit jeho hodnotou či výrazem, který má tutéž hodnotu.

Poznámka: Vždy vlastně pracujeme jen s reprezentacemi hodnot, přičemž redukce (zjednodušení, vyhodnocení) hledá dále neredukovatelný tvar, tzv. kanonickou formu reprezentace hodnoty; např. *III * III*, *7 * 7*, *XLIX* jsou výrazy redukovatelné na hodnotu 49. Určité výrazy nemají hodnotu, např. *1/0*. Pro ně se zavádí speciální symbol ⊥.

Pro již zmíněné typy platí, že ve funkcionálních programech má každý výraz svůj typ (jedná se o tzv. ostré typování, angl. strong typing) a že špatně typově vytvořené výrazy nejsou dobře formovanými výrazy a tedy nejsou ani vyhodnotitelné. Kromě syntaktické analýzy programů je tedy nutná i typová analýza programů, po nichž teprve následuje vyhodnocení výrazů.

Funkce sama o sobě je nejdůležitější hodnotou ve všech systémech funkcionálního programování. Funkce je hodnotou se vším všudy, může být argumentem jiné funkce či výsledkem další funkce.

Skládání funkcí, funkce vyšších řádů a používání inverzních funkcí k prostým funkcím je přirozenou součástí možností funkcionálních programovacích systémů.

Ve funkcionálních systémech hraje důležitou roli striknost či nestriknost funkce. Jestliže $f\ \perp = \perp$, pak říkáme, že f je striktní. Funkce definovaná

```
?> def three x = 3
```

nemusí být striktní, protože výraz *three (1/0)* může být redukován na *3*. Často je právě nestriktnímu chápání podobných případů dávána přednost, a to z několika důvodů: snadnější dokazování vlastností funkcionálních programů (*2 + three x = 5* pro všechna *x*), jednodušší pravidla substituce. Nestriktnost umožňje definovat řídící struktury (např. *if p then x else y* je obvykle nestriktní v *x, y*) a umožňuje výběr z n-tic, jejichž některé elementy jsou $\perp$. Striknost a nestriktnost funkcí se projevuje i v možných postupech (jinak strategiích) vyhodnocování výrazů: hladové či líné vyhodnocování, angl. eager resp. lazy evaluation. Líné vyhodnocování odkládá vyhodnocení argumentů, hladové nejprve vyhodnocuje argumenty. Líné vyhodnocování je možné pro nestriktní funkce.

FUNKCIONÁLNÍ PROGRAMOVÁNÍ ZÍTRA

Představa, že jeden směr programování zvítězí a ostatní zatlačí do defenzivy je pro nejbližší budoucnost utopická. Mohlo by se tak stát jen v souvislosti se zásadní změnou

architektury technického vybavení počítače, která by si vyžádala a případně i prosadila odlišný programátorský přístup.

Proto se s funkcionálními systémy setkáme často jen v kombinaci s objektovým a/nebo logickým programováním.

Jazyk Common Lisp je nyní v USA normalizován a jedná se o typickou kombinaci funkcionálního a objektového přístupu. Ve Velké Británii je obdobným integrovaným systémem produkt mnohaletého univerzitního výzkumu na Sussex University, totiž POPLOG, který v sobě kombinuje funkcionální, logické a objektové principy a který je navíc snadno slučitelný (ve výsledném produktu) s klasickými imperativními systémy.

CO ŘÍCI NAKONEC

Tam, kde se prosadí inženýrský přístup k programování (tj. vytváření prototypů a teprve pak konečných produktů), a tam, kde se využijí možnosti k reprezentaci znalostí, najdeme a použijeme jako jeden z možných prostředků funkcionální jazyky, či integrované systémy nabízející jako jednu z možností funkcionální podsystém (nejčastěji na objektovém pozadí a v kombinaci s logickým či data-flow programováním).

Naučit se moderní funkcionální jazyk by zejména pro jedince s dobrým matematickým vzděláním nemělo být problém pro jejich blízkost matematickému jazyku.

Zvláštní místo mají funkcionální systémy vyrostlé z jazyka LISP ve Spojených státech v projektech (a to i komerčních) v oblasti umělé inteligence a zvláště ve znalostním inženýrství.

Literatura

Bird R Wadler P Introduction to Functional Programming. Prentice Hall, New York 1988

Michaelson G An Introduction to Functional Programming through Lambda Calculus, Addison-Wesley, New York, 1989

Henderson P Functional Programming - Application and Implementation, Prentice Hall, New York 1980

Abelsson H Sussman G J & Sussman J Structure and Interpretation of Computer Programs, MIT Press, Cambridge MA, 1985

Jones S. L. P. The Implementation of Functional Programming Languages, Prentice Hall, New York, 1987

Harper R Mitchell K Introduction to Standard ML, LFCS Edinburgh Univereity, 1987

Polák J Tvrdík P Neimperativní modely výpočtu a jejich implemantace, in sbornik semináře MOP'87; bude publikováno v Ročenka výpočetní techniky, SNTL Praha 1990

Polák J Jelínek I Müller K Programovací jazyky, skripte FEL ČVUT, Praha 1988

Polák J Koukolíková J Logické programování, skripta FEL ČVUT, Praha 1990

PŘÍLOHA

```
double x = x + x

double x = 2 * x

both f x = f x x
double   = both (+)

double   = (*2)

capitalise x = decode (offset + code x)  if islower x
             = x                         otherwise
               where offset = code 'A' - code 'a'

sqrt x = until cond improve x
         where cond y    = abs(y*y - x) < 0.0001
               improve y = (y + x/y)/2

until p f x = x                          if p x
            = until p f (f x)            otherwise

koreny (a, b, c) = (r1, r2)              if d >= 0
                   where  r1 = (-b + r)/(2 * a)
                          r2 = (-b - r)/(2 * a)
                          r  = sqrt d
                          d  = b*b - 4 * a * c

mezery n             = [' ' | _ <- [1..n]]

prvocislo p = (delitele p == [ 1, p])
              where delitele n = [d | d <- [1..n], n `mod` d == 0]

reverse []           = []
reverse (x:xs)       = reverse xs ++ [x]

reverse xs           = shunt [] xs
shunt ys []          = ys
shunt ys (x:xs)      = shunt (x:ys) xs

ones                 = 1 : ones
```