Rozhledy matematicko-fyzikální

Miroslav Virius

Funkcionální programování přichází

Rozhledy matematicko-fyzikální, Vol. 85 (2010), No. 4, 48–55

Persistent URL: http://dml.cz/dmlcz/146383

Funkcionální programování přichází*

Miroslav Virius, FJFI ČVUT, Praha

Abstract. The article examines the elements of functional programming which appear in new versions of existing programming languages. It is discussed whether this paradigma will replace object-oriented programming in near future.

Úvod

Objektově orientované programování se od druhé poloviny devadesátých let stalo nejen zcela převládajícím, ale i vyžadovaným přístupem. V poslední době však do moderních programovacích jazyků začínají pronikat prvky funkcionálního programování a objevují se i nové funkcionální jazyky. Je to předznamenání nástupu tohoto paradigmatu jako hlavního proudu? Funkcionální programování není novou záležitostí; prvním významnějším pokusem v tomto směru byl programovací jazyk LISP (1958), který se v různých obměnách používá dodnes. Donedávna však platilo, že v hlavním proudu programování se tento přístup neuplatňuje.

Základní rysy funkcionálního a imperativního programování

Imperativní programování vyjadřuje program jako posloupnost kroků; vyjádření programu je podobné zápisu algoritmu používanému např. při výuce programování. Datové struktury, které se v imperativním programování používají, mohou být – a zpravidla jsou – měnitelné.

Imperativní programovací jazyky samozřejmě znají pojem funkce (ať už se jedná o „volnou“ funkci v klasickém strukturovaném programování, nebo o metodu v objektově orientovaném programování). Funkce je však jen jednou z programových konstrukcí a její použití je ve srovnání s funkcionálními programovacími jazyky poměrně omezené. Při volání funkce je zpravidla k dispozici pouze *striktní vyhodnocování* parametrů, při kterém se nejprve vyhodnotí parametry a ty se pak předají algoritmu implementovanému ve volané funkci. To znamená, že např. při zpracování výrazu $F(G(x))$ se nejprve vyhodnotí $G(x)$ a výsledek tohoto volání

* Práce na tomto příspěvku byla podporována grantem SGS 10/094.

se v dalším kroku předá jako skutečný parametr funkci F. Imperativní programovací jazyky také běžně dovolují, aby funkce měly vedlejší efekty.

Funkcionální programování je založeno na lambda-kalkulu, matematickém aparátu zavedeném ve 30. letech 20. století americkým matematikem A. Churchem (obr. 1), který umožňuje definovat funkci, popsat její aplikaci a vyjádřit rekurzivitu [1], [2].

Obr. 1. Alonzo Church (1903–1995)

Ve funkcionálním programování máme k dispozici vlastně pouze funkce; funkce lze samozřejmě volat, ale vedle toho funkce představují hodnotu, s níž lze pracovat. Funkce může vrátit jako výsledek jinou funkci, funkce lze skládat. Čistě funkcionální programovací jazyky neobsahují řídicí struktury známé z imperativních jazyků, jako jsou cykly.

Při volání se vedle striktního vyhodnocení může použít i *nestriktní vyhodnocení*, při kterém se funkci předají nevyhodnocené parametry. To znamená, že zápis $F(G(x))$ způsobí, že se funkci F předá $G(x)$ a to se dosadí na místo odpovídajícího formálního parametru v algoritmu, který F implementuje. (Podobným způsobem fungovalo „předávání parametru jménem“ v Algolu 60.) Dalším důležitým rysem je odložené vyhodnocování (lazy evaluation), při kterém se funkce nebo její část provede, až když je výsledek opravdu potřeba; do té doby se s ním v programu pracuje pouze symbolicky.

Funkcionálně orientované programovací jazyky obvykle nabízejí bohatší sadu datových struktur než imperativní jazyky. Datové struktury jsou ovšem zpravidla konstantní, tj. jejich obsah nelze po vytvoření měnit. To může vypadat na první pohled jako velké omezení: Chceme-li

změnit jednu hodnotu v seznamu, musíme vytvořit nový seznam, který bude kopií původního s jednou upravenou hodnotou. Na druhé straně ale přináší toto omezení zřejmé výhody: Není třeba starat se o platnost ukazatelů nebo iterátorů při odstranění údaje z datové struktury, konstantní datové struktury jsou bezpečné v prostředí paralelních výpočtů atd.

Odložené vyhodnocování umožňuje definovat i nekonečné datové struktury. Její prvky se definují pomocí funkce, která se zavolá, aby určitý prvek vytvořila, až když je ho třeba.

Prvky funkcionálního programování v současných jazycích

Nejspíš první vlaštovkou oznamující nástup funkcionálního programování do jazyků středního proudu bylo zavedení lambda-výrazů do jazyka C#. Další výrazný prvek, který se objevil v C#, je LINQ (Language-INtegrated Query). Poznamenejme, že tyto konstrukce nejsou v programování jako takovém novinkou; tou je jejich zavedení do standardu jednoho z hojně používaných jazyků (C# je podle indexu TIOBE v současné době na 6. místě [5]).

Lambda-výrazy

Lambda-výraz v C# umožňuje vytvořit anonymní delegát. Připomeňme si syntax: Nejprve uvedeme seznam parametrů, pak operátor `=>` a za ním výraz představující výsledek. Dokáže-li si překladač odvodit typy parametrů, nemusíme je uvádět. Například lambda-výraz pro funkci, která vrátí logickou hodnotu vyjadřující, zda je skutečný parametr větší než 10, může mít tvar:

```
x => x > 10
```

Má-li lambda-výraz více parametrů, uzavřeme je do závorek, a pokud chceme překladači napovědět typ některého z nich, zapíšeme ho před tento parametr, podobně jako v hlavičce „obyčejné“ funkce nebo delegátu:

```
(int x, string s) => s.Length > x
```

V lambda-výrazu bez parametrů uvedeme prázdné závorky, např.:

```
() => Metoda()
```

Lambda-výraz v C# představuje „hodnotu typu funkce“ a lze ho použít na místě, kde se očekává delegát. Je jasné, že lambda-výrazy nepřinášejí nic nového, pouze umožňují alternativní zápis a tím i jiný – možná srozumitelnější – pohled na řešený problém nebo jeho část.

Dotazovací jazyk LINQ

LINQ byl novinkou platformy .NET Framework verze 3.5. Jde o nástroj, který umožňuje psát dotazy (příkazy pro výběr dat z nějaké množiny), přičemž objektem dotazování mohou být datové kontejnery, databáze, dokumenty v XML atd. Skládá se ze tří základních součástí: *zdroje dat*, *popisu dotazu* a jeho *provedení*.

Zdrojem dat je vždy objekt. Může to být kontejner, i „obyčejné“ pole, objekt typu `XElement` v XML představující dokument v XML, objekt typu `DataContext` představující data z databáze atd.

Popis dotazu připomíná databázový dotazovací jazyk SQL, ve kterém je ovšem klauzule `SELECT` na konci. Obsahuje-li zdroj dat `zdroj` celá čísla a chceme-li z něj získat druhé mocniny všech v něm uložených čísel, seřazené podle velikosti od největší po nejmenší, napíšeme:

```
var dotaz = from numero in zdroj
       where numero % 3 == 0
       orderby numero descending
       select numero * numero;
```

Klíčové slovo `var` říká, že typ proměnné `dotaz` si má překladač odvodit sám. Za `from` následuje identifikátor proměnné, která bude sloužit jako parametr dotazu (představuje jednotlivou vybranou hodnotu); překladač si její typ opět odvodí.

Chceme-li data z dotazu získat, musíme projít proměnnou `dotaz` v cyklu `foreach`:

```
foreach(int i in dotaz)
{
    Zpracuj(i);
}
```

Poznamenejme, že tento příklad neukazuje všechny možnosti LINQ. Deklarativní zápis dotazu je ekvivalentní „imperativnějšímu“ zápisu:

```
IEnumerable<int> dotaz = zdroj.Where(num => num % 3 == 0)
                         .OrderByDescending(n => n)
                         .Select(x = >x*x);
```

založenému na volání metod; ostatně tímto způsobem – jako posloupnost volání metod – se výše uvedený dotaz v LINQ také překládá.

Hybridní jazyk F#

Součástí MS Visual Studia 2010 je mimo jiné i programovací jazyk F#. I když „F“ v názvu napovídá, že by mělo jít o funkcionální jazyk, jde jednoznačně o jazyk hybridní, který integruje možnosti jak funkcionálního, tak imperativního objektově orientovaného jazyka. Podle indexu TIOBE [5] se tento jazyk dostal na 40. místo, před jazyky, jako je Smalltalk, Prolog nebo Haslkel. Zde si tento jazyk velice stručně představíme. Podrobnější informace o něm můžete najít např. v [3] nebo [4].

Funkce

Podívejme se na příklad deklarace jednoduché funkce, která počítá druhou mocninu. Skladba této deklarace se neopírá o lambda-výrazy, vychází spíše ze zvyklostí imperativních jazyků. (S definicí založenou na lambda-výrazech se setkáme dále.):

```
let mocnina  n =
  let n1 = n * n
  n1
```

Ponechme stranou drobnosti, jako je vyjádření bloku odsazením nebo všemocné klíčové slovo `let`, a podívejme se na volání této funkce. Vedle zápisu

```
mocnina 2,
```

který v podstatě odpovídá zvyklostem imperativních jazyků (až na to, že argumenty nepíšeme do závorek), můžeme použít operátor zřetězení (pipelining), s jehož pomocí zapíšeme předchozí výraz takto:

```
2 |> mocnina
```

Půvab tohoto zápisu vynikne, jestliže zřetězíme více funkcí. Můžeme napsat např.:

```
let pom = text |> slova |> filtr (fun s -> s = "ne")
```

V tomto výrazu vezmeme proměnnou `text`, jež obsahuje znakový řetězec, a předáme ji funkci `slova`, která tento řetězec rozloží na jednotlivá slova a vytvoří z nich seznam. Tento seznam dále předáme funkci `filtr`, jež z něj vybere pouze slova vyhovující předanému predikátu. Takto vytvořený seznam řetězců „ne“ uložíme do proměnné `pom`. Všimněte si nepojmenované funkce deklarované pomocí klíčového slova `fun`, která je druhým argumentem funkce `filtr`; jde samozřejmě o lambda-výraz.

Datové struktury a datové typy

I když v deklaracích funkcí a proměnných běžně neuvádíme datové typy, je F# typový jazyk. Protože stojí nad prostředím .NET, máme v něm k dispozici všechny datové typy a všechny knihovny, které toto prostředí poskytuje. Vedle toho má své vlastní datové typy a datové struktury.

Pravděpodobně nejpoužívanějšími typy jsou n-tice (tuple), seznamy, pole a posloupnosti.

Skupina n hodnot různých typů oddělených čárkami a uzavřená v závorkách, např. `(x, 3, 3.14)`, s níž zacházíme v programu jako s celkem, je n-tice. (Kdybychom uzavřeli do závorek skutečné parametry při volání funkce, předávali bychom je jako jednu n-tici.)

Seznam zapíšeme v programu tak, že jeho položky uzavřeme do hranatých závorek, např. `[x, 5, text, 11]`.

Jak n-tice, tak seznam jsou generické typy a pro práci s nimi je k dispozici řada metod a operátorů. Například metoda `List.hd` vrátí hlavu (první prvek) seznamu, funkce `List.tl` vrátí seznam obsahující všechny prvky kromě hlavy. Operátor `::` připojí levý operand jako hlavu k seznamu, který je jeho pravým operandem.

Také posloupnosti (sequence) jsou generické. Posloupnost může obsahovat libovolné množství za sebou následujících hodnot. Příkladem posloupnosti může být:

```
seq{5I .. 1000000000000000I}
```

Přípona `I` zde označuje velká celá čísla (typ `bigint`). Základem práce s posloupnostmi je odložené vyhodnocování; členy posloupnosti se počítají, až když jsou opravdu potřeba. Implicitně mají posloupnosti krok 1; lze ovšem zadat i jiný krok a ve složených závorkách lze zapsat i cykly a podmíněné příkazy, jež určují jednotlivé prvky.

Poznamenejme, že posloupnost je zvláštním případem tzv. pracovního postupu (workflow), což je struktura, v níž definujeme postup, jak vypočítat její jednotlivé prvky.

Z dalších typů uvedeme typ `unit`, který má jedinou hodnotu vyjádřenou konstantou `()`; jde o analogii typu `void` z jazyka C. Generický typ `option` nabízí volbu mezi hodnotou typu, kterým je tento typ parametrizován, a speciální hodnotou `None`, jež vyjadřuje skutečnost, že hodnota není dána. (Jde vlastně o běžný datový typ doplněný o zvláštní hodnotu, která říká, že žádná hodnota nebyla zadána – podobně jako je tomu v databázích nebo u tzv. nulovatelných typů v C#.)

V jazyce F# je většina datových struktur (včetně jednoduchých proměnných) konstantní. Programátor ovšem může deklarovat a používat i měnitelné datové struktury.

Vzory

Pro rozhodování mezi větším počtem možností se používá výběr podle vzoru (pattern matching). Jeho základem je konstrukce `match` *výraz* `with`, za níž následují jednotlivé alternativy uvedené svislicí a vzorem. Pak následuje šipka a za ní výraz, který se provede, je-li odpovídající alternativa vybrána. Například funkce, která má sloužit jako filtr pro výběr některých webových adres, známe-li jejich url (řetězce) a čísla agentů, může mít tvar:

```
let filtr url agent =
    match (url, agent) with
    | "http://www.seznam.cz", _ -> true
    | _, 1 -> true
    | _ -> false
```

Zde znak podtržení představuje žolíka (wildcard) zastupujícího jakoukoli hodnotu. Jednotlivé alternativy se probírají postupně, takže např. v tomto příkladu se nejprve testuje, zda jde o adresu Seznamu s jakýmkoli agentem; v tom případě bude výsledek `true`. Pak se zkusí, zda jde o jakýkoli řetězec s agentem 1, a pokud ano, bude výsledek opět `true`. Cokoli jiného povede k výsledku `false`.

Vzory mohou být i aktivní – mohou používat různé *pohledy* na data, podle nichž se rozhoduje. To znamená, že s daty mohou před rozhodnutím proběhnout nějaké výpočty. Například jsou-li body reprezentovány v kartézských souřadnicích, můžeme definovat pohled na ně v polárních souřadnicích a ten si při rozhodování podle hodnoty bodu vyžádat.

Další možnosti

Nakonec se ve stručnosti zmíníme o ostatních (ne nutně funkcionálních) možnostech, které programovací jazyk F# poskytuje.

Najdeme v něm základní nástroje imperativního programování, tj. příkazy cyklu, příkazy pro větvení programu apod. nebo nástroje pro práci s výjimkami. Najdeme tu samozřejmě i běžné nástroje objektově orientovaného programování, které vyhovují pravidlům prostředí .NET – můžeme deklarovat třídy a struktury, můžeme odvozovat třídy od jiných tříd, můžeme deklarovat a implementovat rozhraní atd.

Mocným nástrojem pro jisté účely jsou *citace* (quotation). Označíme-li část kódu jako citaci pomocí značek `<@ a @>`, můžeme ji později získat mechanizmem podobným reflexi jako data a zpracovávat ji dalšími nástroji – např. pomocí analyzátoru lambda-výrazů, který je součástí knihoven jazyka F#.

Závěr

Zkušenosti ukazují, že některé nástroje vycházející z funkcionálně orientovaných jazyků se v imperativních jazycích těší značné oblibě (to se týká např. LINQ v C#). V čem spočívá jejich přitažlivost?

- Funkcionální programování používá jiné vyjadřovací prostředky než imperativní programování, a proto vede programátora k poněkud odlišnému pohledu na řešený problém.
- V některých případech je zápis pomocí prostředků funkcionálního programování podstatně přehlednější než tradiční imperativní zápis – typickým příkladem je použití lambda-výrazů pro krátké nepojmenované funkce nebo dotazovacího jazyka LINQ.
- Přitažlivost jazyka F# zjevně spočívá v jeho hybridnosti, v tom, že nabízí jak možnosti imperativního objektově orientovaného programování, tak i možnosti funkcionálního programování a umožňuje programátorům plynule mezi nimi přecházet.

Uvedené skutečnosti samozřejmě nelze považovat za signál nástupu funkcionálního programování jako takového, ukazuje ale, že se tento přístup dostává do povědomí širší programátorské veřejnosti.

Literatura

[1] Church, A.: An unsolvable problem of elementary number theory. *American Journal of Mathematics* **58** (1936), str. 354–363.

[2] Church, A.: *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[3] Syme, D., Granitz, A., Cisternino, A.: *F# Expert.* Apress, Berkeley. 2007.

[4] Virius, M.: Programovací jazyk F#. . In: *Objekty 2009*, Univerzita Hradec Králové, 2009, str. 254–263.

[5] TIOBE Programming Community Index for March 2010. `www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, (citace 21. březen 2010).