Závěr

V příštím pokračování se budeme věnovat dalším úpravám digitálního obrazu, například změně kontrastu, světlosti a podobně. Postupně se také dostaneme k obtížnějším tématům, jako je kupříkladu histogram a jeho využití v automatických úpravách obrazu.

Literatura

[1] *Gonzalez, R. C., Woods, R. E.*: Digital Image Processing. 3. vydání. Pearson Prentice Hall, 2008.

[2] *Huges, J. F. a kol.*: Computer Graphics. Principles and Practice. 3. vydání. Addison-Wesley, 2014.

[3] *Felkel, P., Sochor, J., Žára, J., Beneš, B.*: Moderní počítačová grafika. 2. vydání. Computer Press, 2005.

[4] *Martíšek, D.*: Matematické principy grafických systémů. Littera, 2002.

[5] Webové stránky Katedry informatiky PřF UP v Olomouci, sekce věnovaná spolupráci se středními školami https://www.inf.upol.cz/setkani-ucitelu-informatiky

Programování bez proměnných

JAN LAŠTOVIČKA

Přírodovědecká fakulta UP, Olomouc

Proměnné ve většině programovacích jazyků hrají zásadní roli: vkládají hodnoty na určitá místa výrazů, uchovávají dočasné výsledky a v neposlední řadě označují argumenty funkce. Článek představuje jazyk Joy, kde pojem proměnné neexistuje.

Joy je navíc čistě funkcionální jazyk. Přestože funkcionální jazyky mají bohatou historii, jejich prvky prosakují do moderních jazyků až poslední dobou. Jazyk Joy tak může poskytnout netradiční úvod do stále více populárnějších technik funkcionálního programování.

1. Historie

Odstraňováním proměnných z matematických tvrzení se na začátku dvacátých let minulého století zabýval ruský matematik Moses Schönfinkel. Jeho postupy dále rozvedl americký matematik Haskell Curry, který vzniklý formální systém pojmenoval *kombinatorická logika*. Významnou částí kombinatorické logiky je výpočetní model založený na několika abstraktních funkcích zvaných *kombinátory*. Ukázalo se, že každý myslitelný mechanický výpočet lze vyjádřit vhodným aplikováním kombinátorů.

Začátkem sedmdesátých let americký programátor Charles Moore představil jazyk Forth. Jazyk obešel použití proměnných zavedením jednoho univerzálního zásobníku, ze kterého programy získávají a kam ukládají hodnoty. Díky jednoduchému způsobu vykonávání programů našel jazyk uplatnění ve vestavěných systémech. Řídil například přistání sondy Philæ na kometě.

Australský filozof Manfred von Thun vytvořil začátkem dvacátého prvního století jazyk Joy[1)]. Jazyk vychází ze zjištění, že některé programy jazyka Forth lze chápat jako funkce, které mají zásobník na vstupu i na výstupu. Jazyk navíc umožňuje pracovat s programy jako s daty. Díky tomu lze v programech používat obdoby kombinátorů. Na rozdíl od kombinatorické logiky v jazyce Joy nové programy nevznikají aplikací funkcí, ale jejich skládáním.

2. Skládání programů

Ze všech možných čísel se zaměříme pouze na přirozená čísla včetně nuly. Uvažujme nejdříve programy, které číslu na vstupu výpočtem přiřazují číslo na výstupu. Takové programy můžeme přirozeně chápat jako funkce. Vezměme například program `succ`, který číslu přiřadí následovníka (číslo o jedno větší) a program `square`, který počítá čtverec (druhou mocninu) zadaného čísla.

Ze dvou libovolných programů A a B můžeme vytvořit program, který nejprve danému číslu N přiřadí programem A číslo M a poté číslu M přiřadí programem B číslo K. Říkáme, že program vznikl *složením* programů A, B a označíme jej prostě tak, že programy napíšeme za sebe: $A\ B$.

Například program `succ square` vzniklý složením `succ` a `square` přiřadí číslu N číslo $(N+1)^2$.

[1)] https://en.wikipedia.org/wiki/Joy_(programming_language)

Prázdným programem máme na mysli program, který neobsahuje žádný znak. Prázdný program libovolnému číslu přiřadí to samé číslo.

Jména programů (např. `succ` a `square`) řadíme mezi takzvané *atomické programy*.

Dva programy A, B považujeme za *rovné*, pokud libovolnému možnému vstupu přiřadí program A stejnou hodnotu jako program B. Rovnost programů označíme $A = B$.

Program `succ square` není rovný programu `square succ`. První jmenovaný přiřadí číslu jedna číslo $4 = (1 + 1)^2$, ale druhý číslo $2 = 1^2 + 1$.

Pokud skládáme programy A, B a C, je lhostejné, zda jsme nejprve složili A a B a výsledek složili s C, nebo jsme složili A s výsledkem složení B a C. V obou případech obdržíme programy, které nejprve použijí program A, poté program B a nakonec program C. Tedy zápis $A\ B\ C$ jejich složení je jednoznačný. Program `succ square succ` přiřazuje číslu N číslo $(N + 1)^2 + 1$.

Programy můžeme *vykonávat* jen s použitím jediného místa v paměti na uložení čísla. Vstupní číslo vložíme na místo, vykonáme program a jeho výsledek najdeme na místě. Prázdný program se vykoná triviálně tak, že neudělá vůbec nic. Vykonání neprázdného programu provedeme tak, že vykonáme postupně všechny jeho atomy. Vykonání atomu musí být definováno.

Pokud bychom dali na místo v paměti číslo jedna, pak by vykonání programu `succ square` znamenalo nejprve vykonání programu `succ`, které zvýší číslo na místě na dva, a poté vykonání programu `square`, které nahradí číslo na místě jeho čtvercem. Na místě se po skončení vykonávání nalézá číslo čtyři, které je tedy výstupem programu.

Pojmenováním libovolného programu rozšiřujeme jazyk. Nové jméno se stává atomickým programem a jeho vykonání probíhá tak, že se vykoná pojmenovaný program. Programům dáváme jména přirozeně použitím rovnítka. Například můžeme definovat:

```
f = succ square
```

Vykonání `f` pro číslo jedna na vstupu povede k vykonání programu `succ square` a tím pádem na výstupu dostaneme, jak již víme, číslo čtyři.

Jak zadat hodnotu vstupu jen za použití skládání programů? Stačí číslo N považovat za atomický program, který libovolnému číslu na vstupu přiřadí číslo N. Čísla lze tedy skládat s jinými programy. Výstup programu `1 succ` bude nehledě na číslo na vstupu vždy 2. Program `1 succ` předsta-

vuje hodnotu, kterou `succ` přiřadí číslu jedna. Dostáváme `1 succ` = 2.

Část programu můžeme nahradit jiným programem, který se části rovná. Přesněji, pokud A, B_1, B_2, C jsou programy, kde $B_1 = B_2$, pak:

$$A\ B_1\ C = A\ B_2\ C$$

Využitím předchozího pravidla odvodíme jakou hodnotu program `f` přiřadí číslu jedna:

$$\texttt{1 f} = \texttt{1 succ square} = \texttt{2 square} = \texttt{4}$$

3. Programy s více vstupy

Zbývá vyřešit, jak umožnit programům mít více vstupů. Výpočet sčítající dvě zadaná čísla můžeme považovat za program `+`, který na vstupu dostane dvojici čísel a jim přiřadí jejich součet. Program by dvojici čísel $(1, 2)$ přiřadil 3. Vstup a výstup programu ale nejsou stejného typu: vstup je dvojice čísel a výstup číslo. Aby bylo možné programy volněji skládat, potřebujeme sjednotit typ vstupu a výstupu.

Ukazuje se, že vhodným typem pro vstup a výstup programu je *zásobník*. Zásobník si představujeme jako uspořádanou n-tici hodnot (nazývaných *prvky* zásobníku), kde můžeme odebrat hodnotu ze začátku (říkáme také *vrcholu*) zásobníku a přidat hodnotu zase na vrchol zásobníku.

Zavedeme program `+` tak, aby zásobníku Z přiřadil zásobník, který vznikne tak, že se ze Z odeberou dvě vrchní čísla N, M a přidá jejich součet $N + M$. Pokud $(1, 2, 3)$ představuje zásobník, kde je na vrcholu 1, pak `+` přiřadí zásobníku $(1, 2, 3)$ zásobník $(3, 3)$. Volněji budeme říkat, že `+` nahrazuje dvě vrchní čísla na zásobníku jejich součtem.

Upravíme všechny naše programy tak, aby pracovaly se zásobníky. Prázdný program zásobník nezmění. Číslo přidá samo sebe na zásobník. Například program `1` přidá na vrchol zásobníku číslo jedna. Program `succ` nahradí číslo na vrcholu zásobníku jeho následovníkem a program `square` jeho čtvercem.

Vykonávání programů probíhá přesně tak, jak je popsáno výše, s tím rozdílem, že nyní na místě v paměti není číslo ale zásobník.

Program `1 2` přidá na vrchol zásobníku nejprve číslo jedna a poté číslo dva (to tedy bude na vrcholu zásobníku). Použití zásobníku umožnilo programům mít více výstupů.

Program `1 2` můžeme složit s programem `+` a vypočítat součet čísel jedna a dva: `1 2 +` = `3`.

Program `1 +` zvýší hodnotu na vrcholu zásobníku o jedna. Program `succ` lze nyní definovat jako `succ` = `1 +`.

Nahrazení vrchních tří čísel zásobníku jejich součtem provedeme programem `+ +`. Např. dostáváme:

$$\texttt{1 2 3 + +} = \texttt{1 5 +} = \texttt{6}$$

Zavedeme program `*` nahrazující dvě vrchní čísla zásobníku jejich součinem. Máme například `2 3 *` = `6`.

Můžeme pomocí `*` definovat program `square` počítající čtverec čísla? Jistě platí, že čtverec čísla tři získáme programem `3 3 *`. Pro výpočet čtverce libovolného čísla musíme nejprve zavést program `dup`, který znovu vloží na zásobník jeho vrchní prvek. Například `3 dup` = `3 3`. Nyní stačí položit `square` = `dup *`.

Hodnotu, kterou program `f` přiřadí jedničce, můžeme detailněji získat výpočtem:

$$\begin{aligned}\texttt{1 f} &= \texttt{1 succ square} = \texttt{1 1 + square} = \\ &= \texttt{2 square} = \texttt{2 dup *} = \texttt{2 2 *} = \texttt{4}\end{aligned}$$

Pro práci se zásobníkem si ukážeme další dva užitečné programy. Program `pop` odstraní ze zásobníku vrchní prvek a program `swap` prohodí dva vrchní prvky zásobníku. Můžete je zkusit využít k napsání programu, který odstraní druhý prvek ze zásobníku.

Víme, že u sčítání nezáleží na pořadí sčítanců. Konkrétněji pro libovolná čísla N, M je:

$$N\ M\ \texttt{+} = M\ N\ \texttt{+}$$

Bez odkazu na sčítance lze předchozí rovnost úsporněji zapsat takto:

$$\texttt{+} = \texttt{swap +}$$

Uměli byste (bez proměnných) rovností programů vyjádřit, že sčítání čísla samo se sebou dává jeho dvojnásobek?

4. Citované programy

Skutečnost, že u sčítání tří čísel N, M, K nezáleží na pořadí jejich součtů, můžeme vyjádřit rovností:

$$N\ M\ K\ \texttt{+ +} = N\ M\ \texttt{+}\ K\ \texttt{+}$$

Aby bylo možné odstranit z rovnosti odkazy na čísla N, M, K, potřebujeme mít schopnost vykonat `+` na druhý a třetí prvek zásobníku. Obecně chceme vykonat jistý program tak, aby neovlivnil první prvek zásobníku.

Uzavřením programu do hranatých závorek potlačíme jeho vykonání. Přesněji pro libovolný program A je `[A]` takzvaný *citovaný program*. Citovaný program považujeme za atomický program, jehož vykonání spočívá v tom, že se pouze (podobně jako číslo) vloží na zásobník.

Pro zrušení citace programu zavedeme program `dip`, který ze zásobníku odebere citovaný program `[`A`]` a libovolnou hodnotu V (číslo nebo citovaný program). Dále vykoná program A a poté vrátí na zásobník hodnotu V. Činnost programu `dip` lze také popsat rovností:

$$V\ \texttt{[}A\texttt{]}\ \texttt{dip} = A\ V$$

Nyní dostáváme

$$N\ M\ K\ \texttt{[+]}\ \texttt{dip}\ \texttt{+} = N\ M\ \texttt{+}\ K\ \texttt{+}$$

a nezávislost pořadí součtů můžeme úsporně formulovat takto:

$$\texttt{+}\ \texttt{+} = \texttt{[+]}\ \texttt{dip}\ \texttt{+}$$

Program `cons` vytváří citované programy. Pokud A je program a V hodnota, pak platí:

$$V\ \texttt{[}A\texttt{]}\ \texttt{cons} = \texttt{[}V\ A\texttt{]}$$

Zajímavé je, že `swap` můžeme nyní definovat programem `[] cons dip`. Poznamenejme, že `[]` je citace prázdného programu. Sami si můžete ověřit, že je definice správná.

5. Kombinátory

Programy rušící citace jiných programů se nazývají *kombinátory*. Program `dip` je tedy kombinátor. Představíme si nejjednodušší kombinátor `i`, který pouze zruší citaci programu; pro program A platí:

$$\texttt{[}A\texttt{]}\ \texttt{i} = A$$

Také můžeme říci, že program `i` odebere ze zásobníku citovaný program `[`A`]` a vykoná program A.

Můžete si sami zkontrolovat, že vykonávání programu `[dup i] dup i` nikdy neskončí.

Vhodnou úpravou předchozího programu lze psát rekurzivní programy. Jazyk však poskytuje několik kombinátorů, které psaní rekurze značně zjednodušují. Vezměme si kombinátor `primrec`, jehož chování vyjadřují rovnosti

$$0\ \texttt{[}A\texttt{]}\ \texttt{[}B\texttt{]}\ \texttt{primrec} = A$$

$$N\ \texttt{[}A\texttt{]}\ \texttt{[}B\texttt{]}\ \texttt{primrec} = N\ \texttt{dup 1 -}\ \texttt{[}A\texttt{]}\ \texttt{[}B\texttt{]}\ \texttt{primrec}\ B$$

kde N je nenulové číslo a A, B jsou libovolné programy.

Zkusíme si naprogramovat faktoriál čísla. Připomeňme, že faktoriál nuly je jedna a faktoriál nenulového čísla N je součin N a faktoriálu čísla $N-1$. Program počítající faktoriál definujeme:

```
[1] [*] primrec
```

Například dostáváme:

```
5 [1] [*] primrec
  = 5 dup 1 - [1] [*] primrec *
  = 5 4 [1] [*] primrec *
  ...
  = 5 4 3 2 1 0 [1] [*] primrec * * * * *
  = 5 4 3 2 1 1 * * * * * = 120
```

Můžete si vyzkoušet pomocí `primrec` napsat program, který spočítá součet čtverců všech čísel menších nebo rovných zadanému číslu.

Ukazuje se, že podobně jako v kombinatorické logice, lze každý program zapsat jen pomocí několika základních programů jejich citováním a skládáním. Za základní programy si můžeme vzít `dip`, `cons`, `dup` a `pop`.

Dokážete vyjádřit program `i` pomocí základních programů?

Jak bylo zmíněno v úvodu, kombinátorová logika je především matematická teorie. V připravovaném článku nabídneme formou jednoduché deskové hry úvod do teorie kombinátorů. Kartičky ve hře představují kombinátory a vhodné umístění kartiček na herní ploše jejich aplikaci.