Wszystko jest funkcją

Michał SKRZYPCZAK[*]

[*]student,
Wydział Matematyki,
Informatyki i Mechaniki,
Uniwersytet Warszawski

Nie każdy sposób programowania polega na wydawaniu komputerowi poleceń. Jak można inaczej konstruować algorytmy? W tym artykule przedstawimy przejście przez programowanie funkcyjne do rachunku lambda.

Na początek rozważmy prosty program, obliczający silnię danej liczby naturalnej n:

$wynik := 1$
while $n > 0$ **do**
  $wynik := wynik * n$
  $n := n - 1$

Dla każdego, kto ma pewne obycie z programowaniem, powyższy kod jest zrozumiałym i naturalnym sposobem obliczania $n!$. Taki sposób pisania programów nazywamy programowaniem imperatywnym.

Jednak jeśli się nad tym zastanowić, to postępujemy tu w sposób sztuczny. Po co mówić komputerowi, jakie ma wykonać **operacje**, by wyliczyć silnię – lepiej powiedzieć mu, czym silnia **jest**. Stąd rodzi się alternatywne podejście do programowania:

function $silnia(n) =$
  if $n = 0$ **then**
    1
  else
    $n * silnia(n - 1)$

Takie podejście nazywamy programowaniem deklaratywnym. Zamiast opisywać operacje i ich kolejność, definiujemy pojęcia matematyczne (liczby, funkcje, ...). Zauważmy, że zdefiniowana powyżej funkcja *silnia* nie *zwraca* wartości (na przykład poleceniem **return**), tylko *jest* wartością. Jeśli spojrzymy na matematyczną definicję $n!$, okaże się, że jest to właściwie to samo, tylko zapisane symbolami, zamiast słowami:

$$n! = \begin{cases} 1 & \text{gdy } n = 0, \\ n \cdot (n-1)! & \text{gdy } n \geqslant 1. \end{cases}$$

Jeszcze inny przykład:

let nwd a b $=$
  if $b = 0$ **then**
    a
  else
    if $a > b$ **then**
      nwd $(a - b)$ b
    else
      nwd a $(b - a)$

Szczególnym rodzajem programowania deklaratywnego jest programowanie funkcyjne. Zakładamy tam, że funkcje są równie dobrymi wartościami, jak na przykład liczby, a więc mogą być argumentami i wartościami innych funkcji. Możemy, na przykład, napisać:

let f x y $=$ $x \cdot y + 1$

definiując tym samym funkcję, która bierze argument x i zwraca funkcję o jednym argumencie y i wartości $xy + 1$. Takiej funkcji możemy podać argumenty:

let a $=$ f 2 7

co nazywamy *aplikacją*. W efekcie wartością a będzie 15. Możemy też podać tylko pierwszy argument:

let g $=$ f 3

(tzw. częściowa aplikacja), definiując tym samym nową funkcję g, której matematyczny zapis to $g(y) = 3y + 1$. Nic nie stoi na przeszkodzie, aby zdefiniować funkcję, której zadaniem jest składanie funkcji dostarczonych jako argumenty:

let compose f g $=$
  function x $\rightarrow$ f $(g\ x)$

Tutaj użyliśmy konstrukcji zwanej *funkcją nienazwaną*. Korzystając z poprzednich definicji, możemy, na przykład, napisać:

let h $=$ compose (**function** x $\rightarrow$ $12 \cdot x$) $(f\ 3)$

Jak widać, otrzymaliśmy w ten sposób funkcję $h(y) = 12 \cdot (3y + 1)$.

Jeśli dany język programowania funkcyjnego (oznaczmy go F) udostępnia rozsądny zbiór typów danych (liczby, listy, drzewa, ...), to wszystkie programy, jakie da się napisać w sposób imperatywny (na przykład w języku C czy Pascalu), da się również napisać funkcyjnie w F. Przykładem takiego języka funkcyjnego jest OCaml.

Warto w powyższych przykładach samodzielnie przeliczyć, jak działają zdefiniowane funkcje i czy zwracają dobre wyniki. Zauważmy, że definicja funkcji nwd jest rekurencyjna. Okazuje się, że za pomocą rekurencji można realizować wszelkie znane z programowania imperatywnego pętle.

Spróbujmy teraz wyekstrahować z programowania funkcyjnego samą jego istotę i przyjrzeć się bliżej, co naprawdę powinniśmy umieć zrobić, aby programować funkcyjnie:

- Zdefiniować funkcję, która dla argumentu x obliczy jakąś wartość e (zależną od x). Oznaczamy taką konstrukcję symbolicznie $\lambda x.e$.
- Obliczyć wartość funkcji f dla jakiegoś argumentu y. Tę czynność nazywamy *aplikacją* (funkcji) i zapisujemy $f\ y$.
- Operować na jakichś typach danych, powiedzmy, liczbach naturalnych.

Jest to po prostu jeszcze jedna notacja. Można napisać $f(x)(y) = x + y$, można:

$$\textbf{let f } x\ y\ =\ x + y,$$

a odpowiednie wyrażenie w rachunku lambda ma postać:

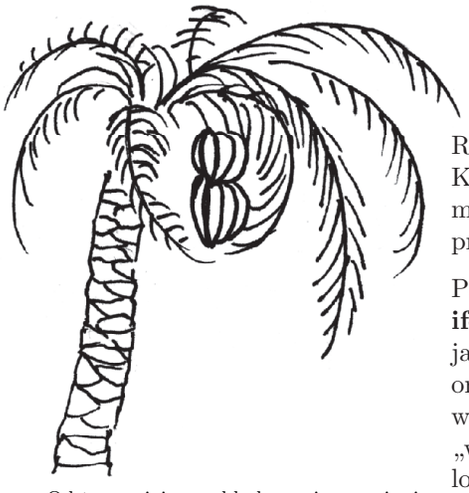
$$f\ =\ \lambda x.\lambda y.x + y.$$

Wyrażenia powstałe z powyższych konstrukcji będziemy nazywać lambda-wyrażeniami. Przykład najprostszego z tych wyrażeń to $\lambda x.x$. Oznacza ono funkcję identycznościową, która bierze jakiś argument i go zwraca. Wobec tego $(\lambda x.x)\ 25 = 25$.

Funkcjom z poprzedniej strony odpowiadają następujące, bardziej skomplikowane przykłady lambda-wyrażeń:

$$\begin{aligned}\textbf{f} &= \lambda x.\lambda y.x \cdot y + 1\\ \textbf{compose} &= \lambda f.\lambda g.\lambda x.f\ (g\ x)\\ \textbf{g} &= \textbf{f}\ 3\\ \textbf{h} &= \textbf{compose}\ (\lambda x.12 \cdot x)\ \textbf{g}\end{aligned}$$

Rachunek lambda został wprowadzony przez Alonzo Churcha i Stephena Cole'a Kleene'go w latach trzydziestych zeszłego stulecia, z powodów, o których będzie można poczytać w następnym numerze *Delty*. Stanowi on formalną podstawę programowania funkcyjnego.



Pewnych rzeczy nam jeszcze brakuje. Często korzystaliśmy z formuły **if-then-else**, a nie ma odpowiadającej jej lambda-konstrukcji. Zastanówmy się, jak mogłaby ona wyglądać. Instrukcja **if** ma trzy części: wyrażenie logiczne b oraz dwie klauzule e_1 i e_2. Wybór jednej z nich jest uzależniony od prawdziwości warunku b. A zatem warunek logiczny b mógłby być funkcją dwuargumentową, „wybierającą" jeden z dwóch argumentów do wykonania. Wtedy wartości logiczne powinny być zdefiniowane jako:

$$\begin{aligned}\textbf{true} &= \lambda e_1.\lambda e_2.e_1\\ \textbf{false} &= \lambda e_1.\lambda e_2.e_2\end{aligned}$$

Od tego miejsca zakładamy, że wyrażenia logiczne w języku arytmetyki, takie jak na przykład $a \leqslant 3$, „wyliczają się" do zdefiniowanych obok lambda-wyrażeń **true** i **false**. Na przykład:

$((0 \geqslant 1)\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =$
$(\textbf{false}\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =$
$((\lambda e_1.\lambda e_2.e_2)\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =$
$(\lambda x.x + 2)\ 7 = 9.$

Teraz formułę **if** b **then** e_1 **else** e_2 zapiszemy jako lambda-wyrażenie

$$b\ e_1\ e_2$$

W zależności od tego, czy b jest prawdą, czy nie, całe wyrażenie będzie równe e_1 albo e_2. W tym układzie jako lambda-wyrażenia można też zdefiniować podstawowe funkcje logiczne:

$$\begin{aligned}\textbf{or} &= \lambda b_1.\lambda b_2.b_1\ \textbf{true}\ b_2\\ \textbf{and} &= \lambda b_1.\lambda b_2.b_1\ b_2\ \textbf{false}\\ \textbf{not} &= \lambda b.b\ \textbf{false}\ \textbf{true}\end{aligned}$$

W tym momencie możemy już bardzo dużą klasę wartości (w tym funkcji) zapisać jako lambda-wyrażenia. Jednak brakuje nam rekurencji. Nie można przecież wstawić lambda-wyrażenia w nie samo, każde lambda-wyrażenie musi być skończone. Problem ten rozwiązujemy poprzez tzw. operator punktu stałego. Powiedzmy, że po raz kolejny chcemy zdefiniować funkcję silnia, tym razem jako lambda-wyrażenie. Możemy postąpić następująco.

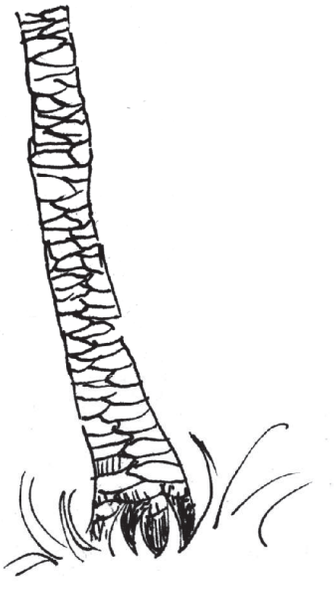
o pierwsze, definiujemy funkcję F, która otrzymawszy jako argument funkcję f czącą $n!$ dla $n = 0, 1, 2, \ldots, k-1$, zwróci funkcję liczącą $n!$ dla argumentów $= 0, 1, 2, \ldots, k$:

$$F = \lambda f.\lambda n.(n = 0)\ 1\ (n \cdot f\ (n-1)).$$

auważamy, że (niezależnie od tego, czym jest f):

- $(F\ f)\ n$, zwraca $n!$ dla $n = 0$,
- $(F\ (F\ f))\ n$, zwraca $n!$ dla $n = 0, 1$,
- $(F\ (F\ (F\ f)))\ n$, zwraca $n!$ dla $n = 0, 1, 2$,
- ...

Warto sprawdzić powyższe stwierdzenia dla kilku początkowych kroków. Gdyby udało nam się „nieskończenie wiele razy złożyć F", uzyskalibyśmy funkcję liczącą silnię dla *wszystkich* liczb naturalnych.



Rozwiązanie zadania M 1219.
Zauważmy, że $f(x) = (x+6)^2 - 6$. Zatem podstawiając $f(x)$ w miejsce x, uzyskujemy $f(f(x)) = (x+6)^4 - 6$. Kontynuując, otrzymujemy

$$f(f(f(f(x)))) = (x+6)^{32} - 6\,.$$

Zatem jedynymi rozwiązaniami danego równania są liczby $x = -6 \pm \sqrt[32]{6}$.

Obliczymy dla przykładu wartość $(\mathbf{Y}\ F)\ 3$:

$$\begin{aligned}(\mathbf{Y}\ F)\ 3 &= F\ (\mathbf{Y}\ F)\ 3\\ &= (3 \leqslant 1)\ 1\ (3 * ((\mathbf{Y}\ F)\ 2))\\ &= 3 * (F\ (\mathbf{Y}\ F)\ 2)\\ &= 3 * ((2 \leqslant 1)\ 1\ (2 * ((\mathbf{Y}\ F)\ 1)))\\ &= 3 * (2 * (F\ (\mathbf{Y}\ F)\ 1))\\ &= 3 * (2 * ((1 \leqslant 1)\ 1\ ((\mathbf{Y}\ F)\ 0)))\\ &= 3 * 2 * 1 = 6 = 3!\end{aligned}$$

Uwaga: „nieskończone złożenie” zostało oczywiście sprytnie ukryte. W wyrażeniu na $\mathbf{Y}$ występują bowiem funkcje (x oraz $\lambda x.\ F\ (x\ x)$), które są swoimi własnymi argumentami. Nie stwarza to jednak problemu – zwróćmy uwagę, że już wcześniej wyszliśmy poza ścisłe, matematyczne znaczenie słowa „funkcja”, nie troszcząc się, na przykład, o zdefiniowanie dziedziny: argumentem funkcji $\lambda x.x$ może być równie dobrze liczba 25, funkcja $\lambda y.2y$, jak i jakikolwiek inny obiekt. W każdym przypadku wyrażenie $\lambda x.x$ ma sens i jego wartością jest dostarczony argument.

Efekt ten uzyskamy za pomocą specjalnego – „magicznego” lambda-wyrażenia, nazywanego operatorem punktu stałego:

$$\mathbf{Y} = \lambda F.(\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))$$

Zauważmy, że

$$\begin{aligned}\mathbf{Y}\ F &= (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))\\ &= F\ ((\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)))\\ &= F\ (\mathbf{Y}\ F)\\ &= F\ (F\ (\mathbf{Y}\ F))\\ &= F\ (F\ (F\ (\mathbf{Y}\ F)))\\ &\ldots\end{aligned}$$

Czyli mamy to, czego szukaliśmy: $\mathbf{Y}\ F$ jest „nieskończenie wiele razy złożonym F”, więc szukaną funkcją silnia.

⋆ ⋆ ⋆

Podsumowując, zdefiniowaliśmy formalny system, w którym za pomocą zaledwie kilku konstrukcji możemy programować. Jednocześnie daje on poręczną notację służącą do definiowania i posługiwania się funkcjami. A dzięki swojej prostocie pozwala stosunkowo prosto wnioskować o własnościach tego języka programowania.

Typy i tautologie

Rachunek lambda w swych założeniach miał być systemem formalnym leżącym u podstaw całej matematyki. Dopiero gdy ten program zawiódł, wykorzystano powstały formalizm do zdefiniowania funkcji obliczalnych. Później okazało się, że po odpowiednim wzbogaceniu może służyć także swojemu pierwotnemu celowi. Przyjrzymy się teraz związkom rachunku lambda z logiką i rozumowaniem.

Wiemy już, że lambda-wyrażenie postaci $\lambda x.M$ reprezentuje pewną funkcję $f(x)$. Lambda-wyrażenie M należy wtedy interpretować jako definicję tej funkcji, czyli po prostu algorytm, który ją oblicza. W językach programowania zwykle nadajemy argumentom typy oraz określamy typ wyniku. W języku C napisalibyśmy np.

```
int f(int x) { return 5*x; }
```

definiując funkcję $f : \text{int} \to \text{int}$ (dla nieprogramistów: $\text{int} = \mathbb{Z}$).

W takim przypadku będziemy mówili, że *typem* funkcji f jest $\text{int} \to \text{int}$. Interesują nas tutaj tylko takie wyrażenia, w których wszystkie zmienne są związane z pewną lambdą, czyli na przykład $\lambda x.\lambda y.y\ x$, ale nie $\lambda y.y\ x$. Ponieważ lambda-wyrażenia są funkcjami, to powinniśmy móc im także nadać pewne typy.

Weźmy dla przykładu wyrażenie $\lambda x.x$, czyli funkcję identycznościową. Argument x w tym przypadku może być dowolnego typu. Nazwijmy ten typ α. Typ wyniku musi być taki sam jak typ zmiennej x, czyli też α. W takim razie typem wyrażenia $\lambda x.x$ jest po prostu $\alpha \to \alpha$.

Rozważmy nieco trudniejszy przykład $\lambda y.(\lambda x.x)$. Niech y będzie pewnego typu β, a x typu α. Wiemy już, że $\lambda x.x$ ma typ $\alpha \to \alpha$, więc $\lambda y.(\lambda x.x)$ będzie typu $\beta \to (\alpha \to \alpha)$.

Weźmy teraz wyrażenie $\lambda x.(\lambda y.y\ x)$. Jeśli x jest typu α, to y musi być funkcją, która przyjmuje argument typu α. Nie wiemy, jakiego typu jest wartość zwracana przez funkcję y, więc nazwiemy ten typ β. W takim razie y będzie typu $\alpha \to \beta$. W ten sposób możemy nadać wyrażeniu $\lambda x.(\lambda y.y\ x)$ typ

$$\alpha \to ((\alpha \to \beta) \to \beta).$$

Jako ćwiczenie Czytelnik może sprawdzić, że typem wyrażenia $\lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z)$ jest

$$(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)).$$

Nie każdemu lambda-wyrażeniu da się nadać typ. Na przykład wyrażenie $\lambda x.x\ x$ nie ma żadnego typu.

Zauważmy, że jeśli zinterpretujemy „$\to$” jako znak implikacji, to nasze typy staną się formułami logiki zdaniowej (czyli takiej, w której nie ma kwantyfikatorów $\forall$ i $\exists$). **Okazuje się, że wszystkie formuły, jakie mogą w ten sposób powstać, są tautologiami!** W szczególności nie istnieje lambda-wyrażenie np. typu $(\alpha \to \beta) \to \beta$, bo taka formuła nie jest tautologią. Lambda-wyrażenie danego typu τ możemy zatem traktować jako dowód prawdziwości formuły τ. Daje nam to ciekawy sposób dowodzenia, że dana formuła τ jest tautologią. Wystarczy znaleźć lambda-wyrażenie M typu τ. Co ciekawe, dla formuł zawierających koniunkcję $\wedge$, alternatywę $\vee$ oraz kwantyfikatory $\forall$ i $\exists$ też istnieją „rachunki lambda” o takiej własności, że każda formuła, która jest typem jakiegoś lambda-wyrażenia, jest też tautologią.

Sławomir KOLASIŃSKI