æ

Wszystko jest funkcją

Michał Skrzypczak[*]

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:

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

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 $h(y) = 12 \cdot (3y+1)$.

Jeszcze inny przykład:

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

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

O ile 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 $\lambda x.e$.
- Skorzystać z wartości argumentu x w ciele funkcji.
- Obliczyć wartość funkcji f dla jakiegoś argumentu y. Tę czynność nazywamy *aplikacją* (funkcji) i zapisujemy $f\ y$.
- Operować na jakiś typach danych, powiedzmy liczbach naturalnych.

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

```
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 λ-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 λ-wyrażeń:

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

Rachunek lambda został wprowadzony przez Alonzo Church'a i Stephen'a Cole Kleene'a, w latach trzydziestych zeszłego stulecia, z powodów, o których można poczytać na stronie X. Stanowi on formalną podstawę programowania funkcyjnego.

Pewnych rzeczy nam jeszcze brakuje. Często korzystaliśmy z formuły **if-then-else**, a nie ma odpowiadającej jej λ- 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:

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

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

$$\begin{aligned}&((0 \geq 1)\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =\\ &(\mathbf{false}\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =\\ &((\lambda e_1.\lambda e_2.e_2)\ (\lambda x.x + 1)\ (\lambda x.x + 2))\ 7 =\\ &(\lambda x.x + 2)\ 7 = 9\end{aligned}$$

Teraz formułę **if** b **then** e_1 **else** e_2 zapiszemy jako λ-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 λ-wyrażenia można też zdefiniować podstawowe funkcje logiczne:

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

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

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

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

Zauważamy, 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. Efekt ten uzyskamy za pomocą specjalnego – „magicznego" λ-wyrażenia, nazywanego operatorem punktu stałego:

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

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, już wcześniej wyszliśmy poza ścisłe, matematyczne znaczenie słowa „funkcja", nie troszcząc się na przykład o zdefiniowanie dziedziny: agrumentem funkcji $\lambda x.x$ może być równie dobrze liczba 25, funkcja $\lambda y.2y$ jak i jakikolwiek inny obiekt. W każdym wypadku wyrażenie $\lambda x.x$ ma sens i jego wartością jest dostarczony argument.

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.

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

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

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.