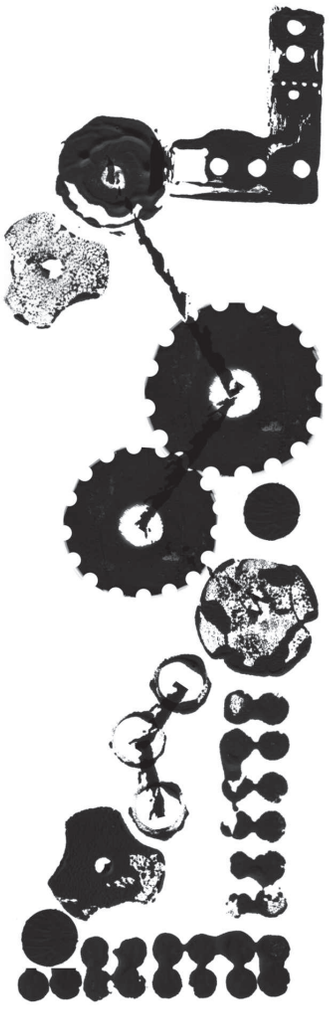
# Obliczalność, rachunek lambda i liczby naturalne

*Sławomir KOLASIŃSKI*[*]



Przypomnijmy, że w poprzednim numerze była mowa o lambda-wyrażeniach, czyli specjalnej notacji $\lambda x.M$ służącej do zapisu funkcji o argumencie $x$, wykonującej operację $M$. Na przykład zamiast

$$f(x)(y) = x + y$$

napisalibyśmy

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

Na początku XX wieku ludzie często zastanawiali się nad algorytmiczną rozwiązalnością różnych problemów. Nie było wtedy do końca jasne, co to znaczy, że jakiś problem jest „rozstrzygalny". Posługiwano się wtedy dość mętnym pojęciem istnienia „procedury o skończonej liczbie operacji". Intuicyjnie chodziło o to, czy istnieje powtarzalny ciąg czynności, który zawsze prowadzi do poprawnych wyników. Sformalizowanie pojęcia rozstrzygalności zajęło matematykom trochę czasu. W latach 30. XX wieku pojawiły się dwie koncepcje: definiowalność w rachunku lambda oraz obliczalność na maszynie Turinga. W 1937 roku Alan Turing udowodnił, że obie te koncepcje definiują tę samą klasę problemów. Warto tutaj zaznaczyć, że rachunek lambda pojawił się historycznie wcześniej niż maszyna Turinga. Ta ostatnia ma jednak bardziej mechaniczną naturę, co pozwoliło fizycznie ją zbudować. Pierwsze komputery były po prostu przeniesieniem abstrakcyjnej maszyny Turinga do świata rzeczywistego. Rachunek lambda jest za to bardziej matematyczną konstrukcją, której nie daje się tak prosto zrealizować w świecie fizycznym.

Te dwa modele, rachunek lambda i maszyna Turinga, zapoczątkowały dwa sposoby myślenia o programowaniu (paradygmaty). Pierwszy model jest abstrakcją programowania funkcyjnego, drugi imperatywnego. Niestety, wszystkie zbudowane dotychczas komputery działają jak maszyna Turinga, przez co imperatywny sposób myślenia jest dużo szerzej rozpowszechniony niż funkcyjny. Należy jednak pamiętać, że obydwa modele mają równie długą historię i obydwu należy się równie wiele uwagi.

Przedstawimy teraz pobieżnie sposób, w jaki można zdefiniować obliczalność w rachunku lambda. Po pierwsze, ustalmy, co chcemy obliczyć. Wszyscy wiemy, że w rzeczywistych komputerach wszystkie informacje reprezentowane są za pomocą jedynie dwóch liczb: zera i jedynki. Ponieważ pojemność pamięci naszych komputerów jest skończona, więc można zinterpretować całą zawartość pamięci jako bardzo, bardzo dużą liczbę naturalną zapisaną w systemie dwójkowym. Oznaczmy tę liczbę $n$. Wyobraźmy sobie, że uruchamiamy jakiś program (nazwijmy go $P$), który działa przez chwilę i się kończy. Nasz program pewnie zmienił zawartość pamięci komputera. Nową zawartość możemy znów zinterpretować jako liczbę naturalną $n'$. W ten sposób program $P$ może być rozumiany jako funkcja na liczbach naturalnych $P : \mathbb{N} \to \mathbb{N}$, taka że $P(n) = n'$. Znamy zatem odpowiedź na pytanie „Co chcemy obliczać?". Chcemy obliczać funkcje $f : \mathbb{N} \to \mathbb{N}$. Pozostaje pytanie „Co to znaczy, że funkcja $f$ da się obliczyć?". Odpowiedź jest prosta. Postulujemy, że funkcja $f$ jest *obliczalna*, jeśli istnieje lambda-wyrażenie $F$, które ją realizuje, czyli

$$\forall n, m \in \mathbb{N}\ (f(n) = m \iff F\ n = m).$$

W poprzednim numerze *Delty*, w artykule Michała Skrzypczaka *Wszystko jest funkcją*, wykazano, że każda funkcja, którą da się zaprogramować na komputerze, może być zapisana jako pewne lambda-wyrażenie. Było to jednak pokazane przy założeniu, że mamy do dyspozycji liczby naturalne oraz działania na nich. Wydaje się to dość naturalne, ale jeśli chcemy udowadniać coś o funkcjach obliczalnych, to dobrze byłoby zredukować do minimum liczbę przypadków, które musimy rozpatrzyć. W *czystym rachunku lambda* wyrażenie $M$ może być tylko postaci $M = P\ Q$, albo $M = \lambda x.P$, albo musi być zmienną $M = x$. Korzystając tylko z tych trzech możliwości, skonstruujemy liczby naturalne oraz operacje arytmetyczne $+$, $-$, $*$ i relacje $=$, $\leqslant$, $\geqslant$.

Liczby naturalne będziemy reprezentować za pomocą tzw. *numerałów Churcha*. Liczbę $n \in \mathbb{N}$ przedstawimy jako funkcję na funkcjach, która składa swój argument ze sobą $n$ razy. Spróbujmy wytłumaczyć to dokładniej. Numerał $\underline{n}$ to funkcja, która przyjmuje jako argument pewną funkcję $f : \mathbb{N} \to \mathbb{N}$. W wyniku daje nową funkcję $g = \underline{n}(f) : \mathbb{N} \to \mathbb{N}$, która jest $n$-krotnym złożeniem $f$,

[*]Instytut Matematyki, Uniwersytet Warszawski

czyli $g(x) = f^n(x)$. Symbolicznie można to zapisać następująco:

$$\underline{n}(f)(x) = f^n(x) = \underbrace{f(f(\ldots f}_{n}(x)\ldots)).$$

W rachunku lambda odpowiednie numerały będą miały postać

$$\begin{aligned}
\underline{0} &\equiv \lambda f.(\lambda x.x)\\
\underline{1} &\equiv \lambda f.f\\
\underline{2} &\equiv \lambda f.\lambda x.f\ (f\ x)\\
\underline{3} &\equiv \lambda f.\lambda x.f\ (f\ (f\ x)) \quad \text{itd.}
\end{aligned}$$

Na takich liczbach możemy zdefiniować operacje dodawania i mnożenia:

$$\begin{aligned}
\mathbf{add} &\equiv \lambda m.\lambda n.\lambda f.(\lambda x.(m\ f)\ (n\ f\ x))\\
\mathbf{mul} &\equiv \lambda m.\lambda n.\lambda f.m\ (n\ f)\,.
\end{aligned}$$

Sprawdźmy, jak to działa.

$$\begin{aligned}
(\mathbf{add}\ \underline{2}\ \underline{3})\ f\ x &= (\underline{2}\ f)\ (\underline{3}\ f\ x)\\
&= (\underline{2}\ f)\ (f\ (f\ (f\ x)))\\
&= f\ (f\ (f\ (f\ (f\ x)))) = \underline{5}\ f\ x\\
(\mathbf{mul}\ \underline{3}\ \underline{2})\ f\ x &= \underline{3}\ (\underline{2}\ f)\ x\\
&= (\underline{2}\ f)\ ((\underline{2}\ f)\ (\underline{2}\ f\ x))\\
&= (\underline{2}\ f)\ ((\underline{2}\ f)\ (f\ (f\ x)))\\
&= (\underline{2}\ f)\ (f\ (f\ (f\ (f\ x))))\\
&= f\ (f\ (f\ (f\ (f\ (f\ x))))) = \underline{6}\ f\ x
\end{aligned}$$

Anegdota głosi, że wpadł na swój genialny pomysł u dentysty, gdy był znieczulony gazem rozweselającym ($N_2O$) podczas usuwania zębów mądrości.

Z odejmowaniem nie pójdzie nam tak prosto. Czytelnik może sam spróbować zdefiniować operację poprzednika, czyli funkcję $pred : \mathbb{N} \to \mathbb{N}$, taką że $pred(x) = x - 1$, by przekonać się, że nastręcza to trudności. W istocie był to przez pewien czas otwarty problem. Rozwiązanie znalazł student Churcha, Stephen Cole Kleene. Do zrealizowania jego pomysłu będziemy potrzebowali dodatkowej struktury danych – pary.

$$\begin{aligned}
\mathbf{pair} &\equiv \lambda x.\lambda y.(\lambda f.f\ x\ y)\\
\mathbf{fst} &\equiv \lambda p.p\ (\lambda x.\lambda y.x)\\
\mathbf{snd} &\equiv \lambda p.p\ (\lambda x.\lambda y.y)\,.
\end{aligned}$$

W ten sposób $\mathbf{pair}\ A\ B$ koduje dwa lambda-wyrażenia $A$ oraz $B$ w jednym, a operacje **fst** i **snd** wybierają poszczególne współrzędne, na przykład:

$$\begin{aligned}
\mathbf{fst}\ (\mathbf{pair}\ A\ B) &= (\mathbf{pair}\ A\ B)\ (\lambda x.\lambda y.x)\\
&= (\lambda f.f\ A\ B)\ (\lambda x.\lambda y.x)\\
&= (\lambda x.\lambda y.x)\ A\ B = A.
\end{aligned}$$

Dla skrócenia notacji przez $\langle A, B\rangle$ będziemy oznaczać parę, czyli lambda-wyrażenie $\mathbf{pair}\ A\ B$. Wygodnie też będzie pisać nieformalnie $\lambda\langle p_1, p_2\rangle.M$, jeśli zakładamy, że argumentem definiowanej funkcji jest para. Możemy teraz skonstruować funkcje poprzednika oraz odejmowanie:

$$\begin{aligned}
\mathbf{pred} &\equiv \lambda m.\lambda f.\lambda x.\mathbf{fst}\ (m\ (\lambda p.\mathbf{pair}\ (\mathbf{snd}\ p)\ (f\ (\mathbf{snd}\ p)))\ (\mathbf{pair}\ x\ x)) \equiv\\
&\equiv \lambda m.\lambda f.\lambda x.\mathbf{fst}\ (m\ (\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ \langle x, x\rangle)\\
\mathbf{sub} &\equiv \lambda m.\lambda n.n\ \mathbf{pred}\ m\,.
\end{aligned}$$

Zauważmy, że nie mamy czegoś takiego jak liczby ujemne, więc $\mathbf{pred}\ \underline{0} = \underline{0}$.

Obliczmy dla przykładu $\mathbf{pred}\ \underline{3}$.

$$\begin{aligned}
\mathbf{pred}\ \underline{3}\ f\ x &= \mathbf{fst}\ (\underline{3}\ (\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ \langle x, x\rangle)\\
&= \mathbf{fst}\ ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\\
&\qquad ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ \langle x, x\rangle)))\\
&= \mathbf{fst}\ ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ \langle x, f\ x\rangle))\\
&= \mathbf{fst}\ ((\lambda\langle p_1, p_2\rangle.\langle p_2, f\ p_2\rangle)\ \langle f\ x, f\ (f\ x)\rangle)\\
&= \mathbf{fst}\ \langle f\ (f\ x), f\ (f\ (f\ x))\rangle = f\ (f\ x) = \underline{2}\ f\ x\,.
\end{aligned}$$

A oto jak porównywać liczby Churcha za pomocą wyrażenia warunkowego typu **if-then-else** oraz operatorów i wartości logicznych zapisanych za pomocą lambda-wyrażeń w sposób opisany we wspomnianym artykule:

$$\begin{aligned}
\mathbf{zero} &\equiv \lambda n.n\ (\lambda x.\mathbf{false})\ \mathbf{true}\\
\mathbf{le} &\equiv \lambda n.\lambda m.\mathbf{zero}\ (\mathbf{sub}\ n\ m)\\
\mathbf{ge} &\equiv \lambda n.\lambda m.\mathbf{zero}\ (\mathbf{sub}\ m\ n)\\
\mathbf{eq} &\equiv \lambda n.\lambda m.\mathbf{and}\ (\mathbf{le}\ n\ m)\ (\mathbf{ge}\ n\ m)
\end{aligned}$$

Jeśli dodamy do tego możliwość porównywania liczb (patrz margines), to zdefiniujemy pełną arytmetykę w czystym rachunku lambda. To już wystarczy, by wyrazić wszystko, co jesteśmy w stanie zaprogramować.