Informatyka w Edukacji, XVI
UMK Toruń, 2019

RACHUNEK LAMBDA DLA POCZĄTKUJĄCYCH

Tomasz Drab
Wydział Matematyki i Informatyki, Uniwersytet Wrocławski
tdr@cs.uni.wroc.pl; ii.uni.wroc.pl/~tdr

Abstract. This paper presents basics of lambda calculus as a simple programming language accessible for students of primary and secondary schools. It contains theoretical remarks and examples of computations, programs and syntax extensions. It discusses also decisions to be made when introducing the calculus to beginners.

1. Wstęp

Za początek historii rachunku lambda można uznać rok 1932, w którym Alonzo Church opublikował pierwszą pracę, w której rachunek ten został wykorzystany [1]. Niedługo później okazał się on być modelem obliczeń równoważnym maszynie Turinga, a w badaniach nad językami programowania wykorzystywany jest do dziś jako język bazowy.

Celem niniejszej pracy jest przedstawienie rachunku lambda jako prostego języka programowania dostępnego dla uczniów szkół podstawowych i średnich. Prezentacja jest bardzo bliska formie, w jakiej rachunek lambda był przedstawiany uczestnikom eliminacji do konkursu KOMA w 2018 roku [4]. Została ona uzupełniona o terminologię i własności teoretyczne opisane w literaturze [2] oraz dalsze przykłady programistyczne [3]. Zawiera także uwagi dotyczące różnych możliwych form, w jakich można prezentować rachunek początkującym. Systemy typów dla rachunku lambda, które mogą ułatwiać myślenie o nim, zostały jednak pominięte w celu podkreślenia jego prostoty i samodzielności jako systemu obliczeń.

2. Notacja i redukcja

Działanie rachunku lambda opiera się na pojęciu funkcji jako czegoś, czego wartość można obliczyć dla konkretnej danej. Następująca składnia pozwala zapisywać funkcje zadane wzorem bez konieczności nazywania ich:

nazwa:	lambda	parametr	kropka	wzór/„przepis”
zapis:	λ	*x*	.	(*x* + 3) · 2
czytanie:	funkcja, która przyjmuje	iks	i zwraca	iks plus trzy razy dwa

Tabela 1 Składnia funkcji lambda

Pozostaje przyjąć składnię oznaczającą wartość funkcji f dla argumentu x. Na lekcjach matematyki jest to zapis f(x). W tej pracy przyjmiemy notację f x, dzięki czemu później będziemy mogli zapisywać x f, kiedy wygodniej będzie podać najpierw argument, a następnie funkcję. W środowisku naukowym związanym z rachunkiem lambda zwykle pomija się symbol aplikacji zapisując jedynie f x, ale taki zapis może być nieintuicyjny dla początkujących.

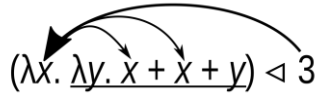
Możemy w ten sposób zapisać przykładowe obliczenia:

(λ*x*. (*x* + 3) · 2) ◃ 5 = (5 + 3) · 2 = 8 · 2 = 16

(λ*x*. λ*y*. *x* + *x* + *y*) ◃ 3 ◃ 4 = (λ*y*. 3 + 3 + *y*) ◃ 4 = 3 + 3 + 4 = 10

(λ*x*. *x* + *y*) ◃ 3 = 3 + *y*

(λ*x*. 5) ◃ 2 = 5



Rysunek 1 Schemat podstawiania argumentu do wzoru funkcji

Uproszczenie wyrażenia polegające na postawieniu argumentu do wzoru funkcji, tak jak powyżej, nazywa się β-redukcją. Będzie to jedyna operacja nazywana redukcją w tej pracy. Zgodnie ze schematem procedura redukcji jest następująca: przepisać wzór funkcji (podkreślony na schemacie) zastępując każde wystąpienie parametru (na schemacie „x”) argumentem (na schemacie „3”).

Działanie aplikacji funkcji do argumentu nie jest łączne, należy więc zwracać uwagę na kolejność wykonywania działań:

$x \triangleleft y \triangleleft z = (x \triangleleft y) \triangleleft z \neq x \triangleleft (y \triangleleft z)$

$x \triangleleft y \triangleleft z = z \triangleright (y \triangleright x) \neq z \triangleright y \triangleright x$

Wzór funkcji obejmuje wyrażenia najdalej, jak to możliwe:

$\lambda x.\ x \triangleleft x \ = \ \lambda x.\ (x \triangleleft x) \ \neq \ (\lambda x.\ x) \triangleleft x$

Popularnym skrótem notacyjnym jest skracanie ciągów lambd. Przykładowo zapis „λx. λy. λz. P" mógłby być skrócony do postaci „λxyz. P". Jednak w tej pracy ten skrót nie będzie stosowany.

Młodszym uczniom można przedstawić funkcje jako pudełka przekształcające argument na wartość:

$1 \triangleright$ [dodaj 3 i przemnóż przez 2] $\rightarrow \ (1 + 3) \cdot 2$

$5 \triangleright$ [dodaj 3 i przemnóż przez 2] $\rightarrow \ (5 + 3) \cdot 2$

$x \triangleright$ [dodaj 3 i przemnóż przez 2] $\rightarrow \ (x + 3) \cdot 2$

$x \triangleright$ [$\lambda x.\ (x + 3) \cdot 2$] $\rightarrow \ (x + 3) \cdot 2$

3. Postać normalna

Argumentami funkcji mogą być także inne funkcje. Przykładowo:

$(\lambda x.\ x \triangleleft x) \triangleleft \lambda y.\ y = (\lambda y.\ y) \triangleleft \lambda y.\ y = \lambda y.\ y$

$(\lambda x.\ \lambda z.\ x) \triangleleft (\lambda w.\ w) \triangleleft (\lambda x.\ \lambda y.\ y) = (\lambda z.\ \lambda w.\ w) \triangleleft (\lambda x.\ \lambda y.\ y) = \lambda w.\ w$

$(\lambda x.\ \lambda y.\ y \triangleleft x) \triangleleft 3 \triangleleft \lambda x.\ x + 4 = (\lambda y.\ y \triangleleft 3) \triangleleft \lambda x.\ x + 4 = (\lambda x.\ x + 4) \triangleleft 3 = 3 + 4 = 7$

Można także redukować zagnieżdżone wyrażenia:

$((\lambda x.\ x) \triangleleft 2) + ((\lambda y.\ y) \triangleleft 5) = 2 + ((\lambda y.\ y) \triangleleft 5) = 2 + 5 = 7$

$((\lambda x.\ x) \triangleleft 2) + ((\lambda y.\ y) \triangleleft 5) = ((\lambda x.\ x) \triangleleft 2) + 5 = 2 + 5 = 7$

Wyrażenie postaci „$(\lambda\_.\ \_\ ) \triangleleft$" można uprościć w jednym kroku redukcji. Takie wyrażenia nazywane są redeksami (ang. reducible expression). Wyrażenia, w których nie ma żadnych redeksów, nazywamy postaciami normalnymi. W powyższych pięciu przykładach wyrażenia zostały doliczone do postaci normalnej.

Zupełność rachunku lambda jako języka programowania pociąga za sobą konieczność wyrażania również nieskończonych obliczeń. Przyjmijmy dwie standardowe definicje:

$\omega = \lambda x.\ x \triangleleft x \qquad\qquad \Omega = \omega \triangleleft \omega$

Wyrażenie Ω jest przykładem wyrażenia, które nie ma postaci normalnej, ponieważ krok obliczeń prowadzi do jej wyjściowej postaci. Jest to przykład „zapętlenia się":

$\Omega \ = \ \omega \triangleleft \omega \ = \ (\lambda x.\ x \triangleleft x) \triangleleft \omega \ = \ \omega \triangleleft \omega \ = \ (\lambda x.\ x \triangleleft x) \triangleleft \omega \ = \ \omega \triangleleft \omega \ = \ \ldots$

Z twierdzenia Churcha-Rossera wynika, że każde wyrażenie ma co najwyżej jedną postać normalną – wynik, jeśli jest, jest tylko jeden.

Innym wnioskiem z twierdzenia jest, że jeśli wyjściowe wyrażenie ma postać normalną, zredukowane wyrażenie ma taką samą postać normalną – redukując wyrażenie nie da się „wejść w ślepą uliczkę".

Istnieje także tzw. normalna strategia redukcji, która ustala kolejność redukcji tak, że zawsze prowadzi do postaci normalnej, jeśli taka istnieje.

4. Konwersja

Funkcja, której wartością jest to, co dostała za argument, nazywa się identycznością i może zostać zapisana jako „$\lambda x.\ x$". Jednak takie samo znaczenie mają także inne wyrażenia:

identyczność = $\lambda x.\ x = \lambda y.\ y = \lambda z.\ z$

Taka zmiana nazw zmiennych nazywana jest *α-konwersją*. Może zostać wykorzystana do unikania zjawiska przesłaniania zmiennych:

$(\lambda x.\ \lambda y.\ x) \triangleleft (\lambda y.\ y) \triangleleft 5 = (\lambda y.\ \lambda y.\ y) \triangleleft 5 = \lambda y.\ y$

$(\lambda x.\ \lambda y.\ x) \triangleleft (\lambda y.\ y) \triangleleft 5 = (\lambda x.\ \lambda z.\ x) \triangleleft (\lambda y.\ y) \triangleleft 5 = (\lambda z.\ \lambda y.\ y) \triangleleft 5 = \lambda y.\ y$

Konwersja może być konieczna w szczególnych przypadkach, kiedy mogłoby dojść do przechwycenia zmiennej:

$\lambda x.\ (\lambda y.\ \lambda x.\ y) \triangleleft x = \lambda x.\ (\lambda y.\ \lambda z.\ y) \triangleleft x = \lambda x.\ \lambda z.\ x \neq \lambda x.\ \lambda x.\ x$

Problemy związane z α-równoważnością wyrażeń można rozwiązać przez zastosowanie w zapisie indeksów de Bruijna, ale zwykle nie są one używane w piśmie ze względu na obniżenie czytelności.

5. Arytmetyka

Wszystkie wyrażenia rachunku lambda budowane są za pomocą zmiennych, funkcji lambda i aplikacji jednego wyrażenia do drugiego. Te trzy reguły nie obejmują zatem wprowadzania do wyrażeń liczb. Liczby można jednak rozumieć jako ustalone wyrażenia. Jednym ze sposobów na wyrażenie arytmetyki liczb naturalnych jest kodowanie ich za pomocą liczebników Churcha:

$0 = \lambda f.\ \lambda x.\ x$ $2 = \lambda f.\ \lambda x.\ x \triangleright f \triangleright f$ $4 = \ldots$

$1 = \lambda f.\ \lambda x.\ x \triangleright f$ $3 = \lambda f.\ \lambda x.\ x \triangleright f \triangleright f \triangleright f$

W takim kodowaniu liczba naturalna *n* kodowana jest przez funkcję, która przyjmuje inną funkcję i argument, a jej wynikiem jest ten argument z *n*-krotnie nałożoną przyjętą funkcją.

Możemy zdefiniować funkcję obliczającą dla danego liczebnika liczebnik po niej następny (w zamierzeniu „nast ◃ 1 = 2"):

nast = λ*n*. λ*f*. λ*x*. (*n* ◃ *f* ◃ *x*) ▹ *f*

Przykład (w dolnym indeksie znaku równości podano uzasadnienie przejścia):

nast ◃ 1 $=_{\text{nast}}$ (λ*n*. λ*f*. λ*x*. (*n* ◃ *f* ◃ *x*) ▹ *f*) ◃ 1 $=_\beta$ λf. λx. (1 ◃ f ◃ x) ▹ f $=_1$

$=_1$ λf. λx. ((λf. λx. x ▹ f) ◃ f ◃ x) ▹ f $=_\beta$ λf. λx. ((λx. x ▹ f) ◃ x) ▹ f $=_\beta$

$=_\beta$ λf. λx. (x ▹ f) ▹ f = λf. λx. x ▹ f ▹ f $=_2$ 2

Możemy także zdefiniować funkcję dodającą dwa liczebniki:

dodaj = λ*n*. λ*m*. (*n* ◃ nast ◃ *m*)

W ten sposób możemy rozumieć zapisy „*n* + *m*" jako „dodaj ◃ *n* ◃ *m*". Wtedy:

2 + 2 $=_+$ dodaj ◃ 2 ◃ 2 $=_{\text{dodaj}}$ (λ*n*. λ*m*. (*n* ◃ nast ◃ *m*)) ◃ 2 ◃ 2 $=_\beta$

$=_\beta$ (λ*m*. (2 ◃ nast ◃ *m*)) ◃ 2 $=_\beta$ 2 ◃ nast ◃ 2 $=_2$ (λ*f*. λ*x*. *x* ▹ *f* ▹ *f*) ◃ nast ◃ 2 $=_\beta$

$=_\beta$ (λx. x ▹ *nast* ▹ nast) ◃ 2 $=_\beta$ 2 ▹ nast ▹ nast $=_{\text{nast}}$ 3 ▹ nast $=_{\text{nast}}$ 4

Podobnie można wyrazić mnożenie:

mnóż = λ*n*. λ*m*. (*n* ◃ (dodaj ◃ *m*) ◃ 0)

Przykład:

3 · 7 = mnóż ◃ 3 ◃ 7 = 3 ◃ (dodaj ◃ 7) ◃ 0 =

= 0 ▹ (dodaj ◃ 7) ▹ (dodaj ◃ 7) ▹ (dodaj ◃ 7) = 0 + 7 + 7 + 7 = 21

Kodowanie arytmetyki w rachunku lambda pociąga za sobą pewną trudność, która nie występuje w popularnych językach programowania. Dlatego należy podkreślić, że odsłanianie przed użytkownikami kodowania liczb nie jest konieczne. Wszelkie tego typu rozszerzenia rachunku można wprowadzać, nawet niejawnie, w sposób aksjomatyczny – w ten sposób posługiwaliśmy się liczbami w poprzednich sekcjach. Wyrażalność rozszerzenia w rachunku lambda stanowi zaś wtedy gwarancję poprawności takiego rozszerzenia.

6. Instrukcje warunkowe

Do rozważenia instrukcji warunkowych będziemy potrzebować wartości logicznych. Możemy je zdefiniować jako „tak = λ*x*. λ*y*. *x*" i „nie = λ*x*. λ*y*. *y*" (odpowiedniki angielskich „true" i „false"). Można przetestować ich własności obliczeniowe:

tak ◃ *A* ◃ *B* = (λ*x*. λ*y*. *x*) ◃ *A* ◃ *B* = (λ*y*. *A*) ◃ *B* = *A*

nie ◃ *A* ◃ *B* = *B* w analogiczny sposób.

Zatem wyrażenie warunkowe „jeśli *warunek* to *A* inaczej *B*" może być rozumiane po prostu jako „*warunek* ◃ *A* ◃ *B*". Dalej można zdefiniować operacje logiczne:

zaprzecz = λ*b*. *b* ◃ nie ◃ tak

oraz = λ*b*. λ*c*. *b* ◃ *c* ◃ nie lub = λ*b*. λ*c*. *b* ◃ tak ◃ *c*

Przykłady:

zaprzecz ◃ tak = (λb. b ◃ nie ◃ tak) ◃ tak = tak ◃ nie ◃ tak = nie

lub ◃ nie ◃ tak = (λb. λc. b ◃ tak ◃ c) ◃ nie ◃ tak = (λc. nie ◃ tak ◃ c) ◃ tak =

= nie ◃ tak ◃ tak = tak

Można także zdefiniować funkcję sprawdzającą, czy dany liczebnik jest zerem:

czy_zero = λ*n*. *n* ◃ (λ*b*. nie) ◃ tak

Przykładowo zapiszemy program, który zwraca liczbę 3, gdy dostanie 0, a liczbę 4, gdy dostanie inną liczbę, i przekażemy mu liczbę 5:

(λ*n*. czy_zero ◃ *n* ◃ 3 ◃ 4) ◃ 5 = czy_zero ◃ 5 ◃ 3 ◃ 4 =

= *5* ◃ (λ*b*. nie) ◃ tak ◃ 3 ◃ 4 = nie ◃ 3 ◃ 4 = 4

7. Struktury danych

Podobnie, jak liczby i wartości logiczne, można wyrazić w rachunku lambda potrzebne struktury danych. Definicję „para = λ*x*. λ*y*. λ*f*. *f* ◃ *x* ◃ *y*" można potraktować jako przykładowy konstruktor. Pozwala on na zebranie informacji z dwóch wyrażeń w jednym i korzystanie z nich przy pomocy operacji „tak" i „nie" (nawiasy dodane dla czytelności):

(para ◃ *A* ◃ *B*) ◃ tak = (λ*x*. λ*y*. λ*f*. *f* ◃ *x* ◃ *y*) ◃ *A* ◃ *B* ◃ tak =

= (λ*f*. *f* ◃ *A* ◃ *B*) ◃ tak = tak ◃ *A* ◃ *B* = *A*

(para ◃ *A* ◃ *B*) ◃ nie = *B*

W celu skrócenia zapisu można oznaczać postać normalną, do jakiej wylicza się wyrażenie „para ◃ *A* ◃ *B*”, przez „(*A*, *B*)”.

Większe krotki można kodować w podobny sposób lub składać je odpowiednio z zagnieżdżonych par (np. „(*A*, (*B*, *C*))” jako trójka *A*, *B* i *C*). W ten sposób można konstruować nie tylko listy (patrz sekcja 10.), ale także drzewa.

Zwykle pary obsługuje się jednak nie przez operacje „tak” i „nie”, a przez funkcje, które jako argument przyjmują krotkę i zwracają jej odpowiednią składową, ponieważ m.in. zapewnia to lepszą abstrakcję danych. Dla par byłyby to:

pierwszy = λ*p*. *p* ◃ tak drugi = λ*p*. *p* ◃ nie

pierwszy ◃ (*A*, *B*) = (λp. p ◃ tak) ◃ (*A*, *B*) = (*A*, *B*) ◃ tak = tak ◃ *A* ◃ *B* = *A*

W programowaniu funkcyjnym popularną składnią, uczytelniającą kod, są dopasowania wzorca. Korzystając jedynie częściowo z ich siły wyrazu można zaproponować, aby zapis „*P* ▹ λ*p*. (pierwszy ◃ *p*) ▹ λ*a*. (drugi ◃ *p*) ▹ λ*b*. *Q”* mógł być skracany do zapisu „*P* ▹ λ(*a*, *b*). *Q*”. Przykładowo:

(2, 7) ▹ λ(*a*, *b*). *a* = 2 ▹ λ*a*. (7 ▹ λ*b*. *a*) = 2

Takie „rozpakowanie” pary wykorzystamy w następnej sekcji.

8. Iteracja

Można powiedzieć, że liczebniki Churcha kodujące liczby naturalne odpowiadają pętli for, ponieważ powtarzają zadaną procedurę z góry ustaloną liczbę razy. Jeśli liczby są wprowadzone aksjomatycznie, można taką funkcjonalność ukryć pod nazwą „powtórz”:

powtórz ◃ 3 ◃ nast ◃ 2 =

= 2 ▹ nast ▹ nast ▹ nast = 3 ▹ nast ▹ nast = 4 ▹ nast = 5

W ten sposób można zapisać iteracyjny algorytm obliczania silni. W parze będzie przechowywany częściowo wyliczony wynik i następna liczba do domnożenia. W każdym powtórzeniu wynik jest mnożony przez drugi element pary, a druga liczba inkrementowana. W końcu z pary wyciągany jest pierwszy element — wynik:

silnia_iter = λ*n*. pierwszy ◃ (powtórz ◃ *n* ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (1, 1))

Przykład:

silnia_iter ◃ 5 =

= pierwszy ◃ (powtórz ◃ 5 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (1, 1)) =

= pierwszy ◃ (powtórz ◃ 4 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (1, 2)) =

= pierwszy ◃ (powtórz ◃ 3 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (2, 3)) =

= pierwszy ◃ (powtórz ◃ 2 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (6, 4)) =

= pierwszy ◃ (powtórz ◃ 1 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (24, 5)) =

= pierwszy ◃ (powtórz ◃ 0 ◃ (λ(*r*, *i*). (*r* · *i*, *i* + 1)) ◃ (120, 6)) =

= pierwszy ◃ (120, 6) = 120

9. Rekurencja

Jedną z najważniejszych cech języka programowania jest definiowanie funkcji rekurencyjnych. Można spróbować zdefiniować silnię w następujący sposób:

silnia = λ*n*. (czy_zero ◃ *n*) ◃ 1 ◃ (*n* · (silnia ◃ (*n* – 1)))

Jednak w powyższej definicji po prawej stronie równości występuje „silnia", która nie jest jeszcze zdefiniowanym wyrażeniem. Można jednak zdefiniować następujące wyrażenie:

krok_silni = λ*S*. λ*n*. (czy_zero ◃ *n*) ◃ 1 ◃ (*n* · (*S* ◃ (*n* – 1)))

Gdyby argumentem funkcji „krok_silni" zostałaby gotowa silnia, to otrzymalibyśmy również poprawną funkcję wyliczającą silnię. W pewnym sensie stanie się to dzięki wyrażeniu oznaczanemu „Y":

Y = λ*f*. (λ*x*. *f* ◃ (*x* ◃ *x*)) ◃ (λ*x*. *f* ◃ (*x* ◃ *x*))

Jego najważniejsza własność to następująca równość:

Y ◃ *F* = (λ*x*. *F* ◃(*x* ◃ *x*)) ◃ (λ*x*. *F* ◃ (*x* ◃ *x*)) =

= *F* ◃ ((λ*x*. *F* ◃(*x* ◃ *x*)) ◃ (λ*x*. *F* ◃ (*x* ◃ *x*))) = *F* ◃ (Y ◃ *F*)

Silnię można zatem zdefiniować tak, jak poniżej:

silnia = Y ◃ krok_silni = Y ◃ λ*S*. λ*n*. (czy_zero ◃ *n*) ◃ 1 ◃ (*n* · (*S* ◃ (*n* – 1)))

Otrzymane wyrażenie nie ma postaci normalnej, ale spełnia zamierzone na początku równanie:

silnia = Y ◃ krok_silni = krok_silni ◃ (Y ◃ krok_silni) =

= λ*n*. (czy_zero *n*) ◃ 1 ◃ (*n* · ((Y ◃ krok_silni) ◃ (*n* – 1))) =

= λ*n*. (czy_zero *n*) ◃ 1 ◃ (*n* · (silnia ◃ (*n* – 1)))

Zapis „*n*!” można w końcu definiować jako „silnia ◃ *n*”. Mimo tego, że funkcja „silnia” nie ma postaci normalnej, ma ją silnia wyliczona dla danej liczby naturalnej:

4! = (czy_zero ◃ 4) ◃ 1 ◃ (4 · (4 – 1)!) = nie ◃ 1 ◃ (4 · (4 – 1)!) = 4 · (4 – 1)! =

= 4 · 3! = 4 · 3 · 2! = 12 · 2! = 12 · 2 · 1! = 24 · 1 · 0! = 24 · 0! = 24 · 1 = 24

Podobnie, jak w przypadku liczebników Churcha, definiowanie funkcji rekurencyjnych za pomocą „Y” może być trudne. Ważniejsze od samego sposobu definiowania jest to, że definiowanie funkcji rekurencyjnych jest uzasadnioną operacją w rachunku lambda. Zatem, kiedy nie budzi to wątpliwości, za definicję można uznawać kluczowe równości dotyczące funkcji „silnia”:

0! = 1 $\qquad$ $n! = n \cdot (n-1)!$, $\quad$ gdy n nie jest 0

Przez kluczowe równości możemy zdefiniować wykorzystywaną wcześniej funkcję „powtórz” dla liczb naturalnych:

powtórz ◃ 0 ◃ *f* ◃ *x* = *x*

powtórz ◃ *n* ◃ *f* ◃ *x* = powtórz ◃ (*n* – 1) ◃ *f* ◃ (*f* ◃ *x*), $\quad$ gdy *n* nie jest 0

10. Listy

Rachunek lambda pozwala operować na listach, które w nauce programowania mogą odgrywać podobną rolę do tablic. Aby skonstruować dowolną listę, potrzeba listy pustej „[ ]” oraz możliwości tworzenia listy, której pierwszym elementem jest „*g*”, adalszą częścią listy jest lista „*r*”, co oznacza się przez „*g*::*r*”. Przykładowo listę elementów *a*, *b* i *c* (po lewej stronie równości wprowadzamy notację dla list) tworzymy następująco:

[*a*, *b*, *c*] = *a*::*b*::*c*::[ ]

Do operowania na listach potrzeba jeszcze funkcji sprawdzającej, czy dana lista jest pusta, oraz funkcji odtwarzających składowe listy. Ich kluczowe własności to:

czy_pusta ◃ [ ] = tak $\qquad$ czy_pusta ◃ *g*::*r* = nie

głowa ◃ *g*::*r* = *g* $\qquad$ ogon ◃ *g*::*r* = *r*

Powyższe wymagania również można spełnić definiując odpowiednie wyrażenia rachunku lambda (dla „[ ]”, „*g*::*r*”, „czy_pusta”, „głowa”, „ogon”). Jednak zakrycie ich przed użytkownikami pozwala na lepszą abstrakcję danych.

Przykładem działania na liście liczb jest obliczenie ich sumy. Zamierzamy zdefiniować funkcję o następujących własnościach:

suma ◃ [ ] = 0 $\qquad$ suma ◃ *g*::*r* = *g* + (suma ◃ *r*)

Funkcję taką można zdefiniować jako funkcję spełniającą dane równanie rekurencyjne (ponownie w nowy sposób wykorzystujemy dopasowanie wzorca):

suma ◃ *xs* = (czy_pusta ◃ *xs*) 0 (*xs* ▹ λ(*g*::*r*). *g* + suma ◃ *r*)

Przykład działania:

suma ◃ [1, 2, 4, 8] = 1 + suma ◃ [2, 4, 8] = 1 + 2 + suma ◃ [4, 8] =

= 1 + 2 + 4 + suma ◃ [8] = 1 + 2 + 4 + 8 + suma ◃ [ ] = 1 + 2 + 4 + 8 + 0 = 15

Standardową operacją działającą na listach jest także podmiana jej elementów zgodnie z podaną funkcją. Funkcja „podmiana" (propozycja zastąpienia angielskiego „map") powinna spełniać poniższe własności:

podmiana ◃ *f* ◃ [ ] = [ ]

podmiana ◃ *f* ◃ (*g*::*r*) = (*f* ◃ *g*)::(podmiana ◃ *f* ◃ *r*)

Podobnie jak w przypadku sumy wystarcza to do zdefiniowania funkcji „podmiana". W ten sposób można zwięźle wyrażać coraz bardziej złożone obliczenia:

podmiana ◃ (λ*n*. *n* + 3) ◃ [1, 2] = podmiana ◃ (λ*n*. *n* + 3) ◃ 1::2::[ ] =

= 4::(podmiana ◃ (λ*n*. *n* + 3) ◃ 2::[ ]) = 4::5::(podmiana ◃ (λ*n*. *n* + 3) ◃ [ ]) = [4, 5]

podmiana ◃ (λ*n*. *n* · *n*) ◃ [1, 2, 3, 4, 5] = [1, 4, 9, 16, 25]

11. Złożoność obliczeniowa

W powyższych ciągach równości wiele kroków obliczeń było pomijanych jako oczywiste. Jednak dokładniejsza analiza liczby kroków obliczeń pozwala wprowadzić podstawowe intuicje dotyczące złożoności obliczeniowej. Dla przykładu zdefiniujemy iteracyjną i rekurencyjną definicję ciągu Fibonacciego:

fib_iter = λ*n*. pierwszy ◃ (powtórz ◃ *n* ◃ (λ(*a*, *b*). (*b*, *a*+*b*)) ◃ (0, 1))

fib_rek = Y ◃ λ*F*. λ*n*. (czy_mniejsza ◃ *n* ◃ 2) ◃ *n* ◃ (*F* ◃ (*n* – 1) + *F* ◃ (*n* – 2))

Można zauważyć, że „fib_iter" dochodzi do postaci normalnej wykonując liniowo wiele β-redukcji względem argumentu. Natomiast „fib_rek", bez stosowania dodatkowych równości, wykonuje wykładniczo wiele β-redukcji względem argumentu.

12. Podsumowanie

Przedstawione w pracy przykłady prezentują rachunek lambda jako zupełny język programowania o prostych zasadach działania. Ta prostota daje szansę na naukę

programowania w ten sposób od momentu, w którym uczniowie potrafią pisać, czytać i liczyć. Liczne uproszczenia notacyjne i standardowe definicje pozwalają wygodniej prezentować bardziej zaawansowane tematy. Jednak określenie odpowiedniego momentu na ich wprowadzenie pozostaje w rękach nauczyciela.

Literatura

1. F. Cardone, J. R. Hindley, *History of Lambda-calculus and Combinatory Logic*, 2006.
2. H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantic*, North Holland, Amsterdam 1984.
3. H. Abelson, G. J. Sussman, J. Sussman, *Struktura i interpretacja programów komputerowych*, WNT, Warszawa 2002.
4. Konkurs Matematyczny KOMA, http://math.uni.wroc.pl/fmw/koma/konkurs-matematyczny-koma/, ostatni dostęp 7.05.2019 roku.