

Lista 3 – Modele algorytmiczne

1 Wprowadzenie

Współczesne systemy informatyczne to skomplikowane układy, które co może dziwić bazują na bardzo prostych modelach algorytmicznych. Wśród wielu modeli algorytmicznych można wyróżnić:

- maszynę Turinga,
- programy licznikowe,
- maszynę o dostępie swobodnym.

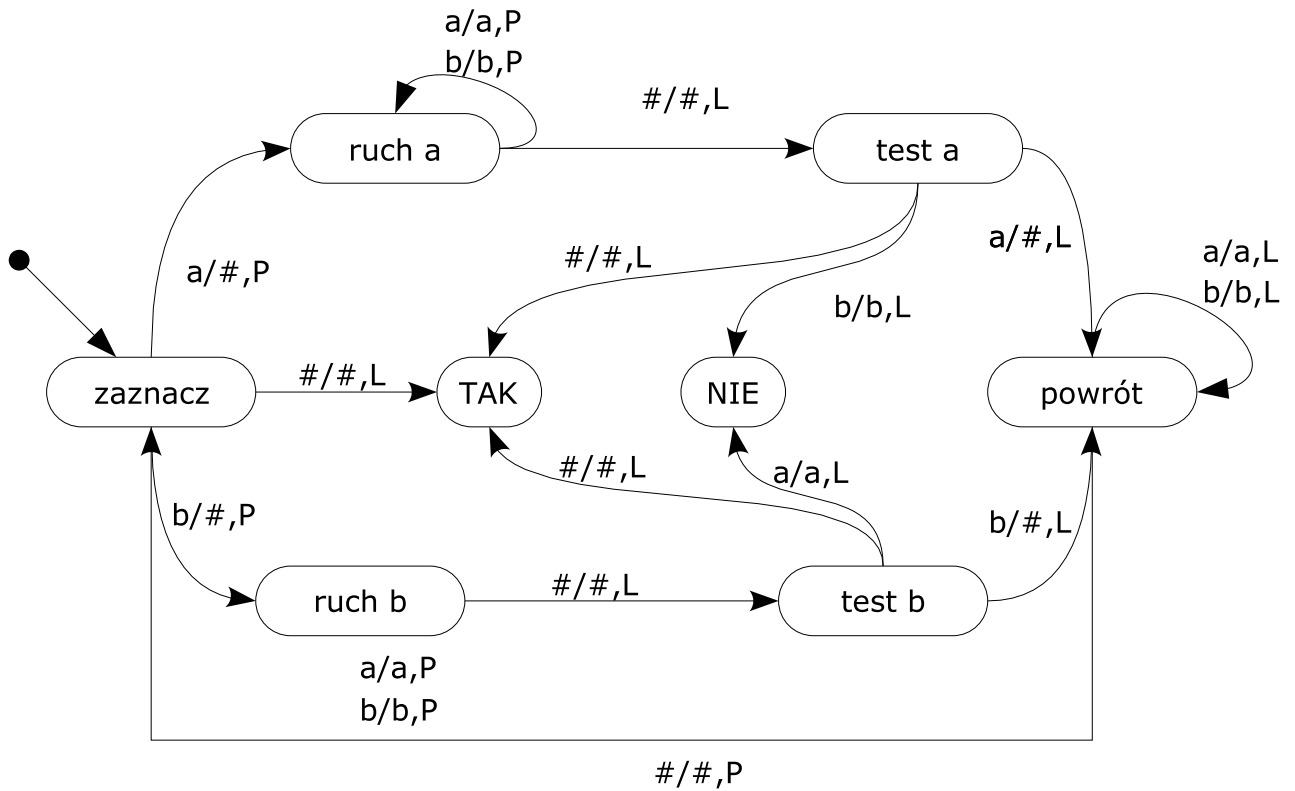
Istnieją także inne modele, jak np.: rachunek kombinatorów, rachunek lambda, czy model funkcji rekurencyjnych. Wszystkie wymienione modele są sobie równoważne, choć ich własności czasem predysponują dany model do pewnych zastosowań np.: maszyna RAM czy programy licznikowe, można traktować jako najprostsze języki programowania, natomiast maszyna Turinga jest bardziej ogólnym modelem, lepszym do opisywania ogólnych własności algorytmów.

1.1 Maszyna Turinga

Maszyna Turinga (a dokładniej deterministyczna maszyna Turinga) jest zbudowana z następujących elementów:

1. skończonego alfabetu symboli
2. skończonego zbioru stanów, z wyróżnionym stanem początkowym i końcowym
3. nieskończonej taśmy z zaznaczonymi kwadratami, z których każdy może zawierać pojedynczy symbol
4. ruchomej głowicy odczytująco/zapisującej, która może przesuwać się wzdłuż taśmy przesuując się na raz o jeden kwadrat
5. diagramu przejść między stanami, zawierającego instrukcje, które powodują, że zmiany następują przy każdym zatrzymaniu się

Rysunek 1 zawiera przykładowy diagram dla deterministycznej maszyny Turinga. Program ten sprawdza czy wyraz umieszczony na taśmie złożony tylko z liter a oraz b z obu stron oznaczony znakami $\#$ które reprezentują symbol pustego znaku jest palindromem. Określenie deterministyczna oznacza, że z każdego stanu do innego stanu istnieje tylko jedno przejście z tym samym warunkiem. Nie są dopuszczalne stany dla których występuje kilka przejść do różnych stanów lecz zawsze z takim samym warunkiem. Mówiąc wprost zawsze może być wykonana tylko jedna określona instrukcja.



Rysunek 1: Wykrywanie palindromów

Symbole jakie są stosowane w przypadku maszyn Turinga to zazwyczaj, cyfry, litery oraz znaki. W jednej kratce na taśmie znajduje się tylko jeden symbol. W przypadku liczb stosuje się system kodowania binarnego ale bardzo często wygodniej jest stosować kod jedynekowy. W tym przypadku liczba jedynek znajdujących się na taśmie określa liczbę całkowitą. Zero jest kodowane jedną jedyką natomiast dowolna inna liczba całkowita n jest reprezentowana przez dokładnie $n + 1$ kolejnych jedynek znajdujących się na taśmie.

Maszynę Turinga można zdefiniować w sposób bardziej formalny. Maszyna Turinga T jest całkowicie określona przez:

- alfabet zewnętrzny $\mathcal{A} = \{a_0, a_1, \dots, a_n\}$, gdzie dodatkowo $a_0 = 0, a_1 = 0$
- alfabet stanów wewnętrznych $\mathcal{Q} = \{q_0, q_1, \dots, q_m\}$
- program, czyli zbiór wyrażeń $T(i, j)$ gdzie $i = 1, \dots, m; j = 0, \dots, n$, każde wyrażenie może mieć jedną z następujących postaci:

$$q_i a_j \rightarrow q_k a_l, \quad q_i a_j \rightarrow q_k a_l R, \quad q_i a_j \rightarrow q_k a_l L$$

gdzie $0 \leq k \leq m, 0 \leq l \leq n$. Wyrażenia $T(i, j)$ nazywamy rozkazami.

Słowem maszynowym (konfiguracją) nazywamy słowo o postaci $Aq_k a_l B$, gdzie $0 \leq k \leq m, 0 \leq l \leq n$, A oraz B są słowami i mogą to być słowa puste w alfabecie \mathcal{A} . Będziemy też pisać a_i^x , jako skrót wyrażenia $\underbrace{a_i a_i \dots a_i}_{x \text{ razy}}$.

Dla danej maszyny T oraz słowa $M = Aq_i a_j B$. Przez M'_T oznaczmy słowo otrzymane z M według następujących reguł:

1. dla $i = 0$, $M'_T = M$
2. dla $i > 0$:
 - (a) jeśli $T(i, j)$ jest postaci $q_i a_j \rightarrow q_k a_l$, to $M'_T = A q_k a_l B$
 - (b) jeśli $T(i, j)$ jest postaci $q_i a_j \rightarrow q_k a_l R$, to:
 - (B_1) jeśli B nie jest słowem pustym to $M'_T = A a_l q_k B$
 - (B_2) jeśli B jest słowem pustym to $M'_T = A a_l q_k a_0$
 - (c) jeśli $T(i, j)$ jest postaci $q_i a_j \rightarrow q_k a_l L$, to:
 - (C_1) jeśli $A = A_1 a_s$ dla pewnych A_1 oraz a_s , to $M'_T = A_1 q_k a_s a_l B$
 - (C_2) jeśli A jest słowem pustym, to $M'_T = q_k a_0 a_l B$.

1.2 Programy licznikowe

Maszyna Turinga jest modelem matematycznym, choć z tego punktu widzenia lepszym modelem może się wydawać np.: rachunek lambda. Jednak maszyna Turinga nie najlepiej modeluje np.: języki programowania. Nieskomplikowanym modelem języków programowania jest maszyna licznikowa. Co może budzić zdziwienie, to model ten jest jeszcze bardziej prymitywny niż maszyna Turinga. Jest on zbudowany z zaledwie z czterech typów instrukcji. Pierwsza to przypisanie zera do zmiennej: $X \leftarrow 0$. Znak przypisania wyrażony za pomocą strzałki w lewo jest sprawą umowną można stosować operator przypisania z Pascala: $x:=0$; jak widać łącznie ze średnikiem.

Drugi typ instrukcji to zwiększenie wartości zmiennej o jedność i przypisanie wyniku do tej samej bądź innej zmiennej:

$$X \leftarrow Y + 1$$

Dopełnieniem tego typu instrukcji jest zmniejszenie wartości o jedność:

$$X \leftarrow Y - 1$$

Ostatnim typem instrukcji jest instrukcja skoku, jeśli zmienna jest równa zero: jeśli $X = 0$ skocz-do ETY. Można też umówić się, że jeśli w programie nie ma etykiety o podanej nazwie to program przerwie działanie. Koniec nastąpi również, gdy zostanie wykonana ostatnia instrukcja w programie.

Przykładowy program który oblicza iloczyn $X * Y$ przedstawia się następująco:

```

U ← 0
Z ← 0
A : jeśli X = 0 skocz-do L
    X ← X - 1
    V ← Y + 1
    V ← V - 1
B : jeśli V = 0 skocz-do A
    V ← V - 1
    Z ← Z + 1
    jeśli U = 0 skocz-do B

```

1.3 RAM – maszyna o dostępie swobodnym

Maszyna RAM (ang. random access machine) składa się z dwóch list, listy wejściowej (oznaczona jako LI) z której można tylko odczytywać wartości za pomocą instrukcji read. Druga lista jest listą wyjściową (określona symbolem LO) na którą można wyprowadzać dane za pomocą instrukcji write. W maszynie ram dostępna jest też pamięć (oznaczona jako $c(n)$) którą można modyfikować za pomocą funkcji read oraz store. W pamięci istotną rolę pełni komórka o indeksie zero która nazywa się akumulatorem. Operacje arytmetyczne oraz instrukcje skoku wykorzystują wartość jaka znajduje się w akumulatorze. W przypadku instrukcji skoku mówimy, że rejestr indeksu instrukcji (II – index instructions) zmienia swoją pozycję. Tabela 2 zawiera spis wszystkich dopuszczalnych instrukcji dla maszyny o dostępie swobodnym (pomiędzy instrukcje: SUB, MULT, DIV, bowiem przedstawiają się analogicznie jak dla instrukcji sumy ADD).

Lp.	Instrukcja	Znaczenie
1	LOAD =a	$c(0) \leftarrow a$
	LOAD n	$c(0) \leftarrow c(n)$
	LOAD *n	$c(0) \leftarrow c(c(n))$
2	STORE n	$c(n) \leftarrow c(0)$
	STORE *n	$c(c(n)) \leftarrow c(0)$
3	ADD =a	$c(0) \leftarrow c(0) + a$
	ADD n	$c(0) \leftarrow c(0) + c(n)$
	ADD *n	$c(0) \leftarrow c(0) + c(c(n))$
4	SUB =a	$c(0) \leftarrow c(0) - a$
5	MULT =a	$c(0) \leftarrow c(0) * a$
6	DIV =a	$c(0) \leftarrow c(0) \text{ div } a$
7	READ n	$c(0) \leftarrow LI_i$
	READ *n	$c(c(n)) \leftarrow LI_i$
8	WRITE =a	$LO_i \leftarrow a$
	WRITE n	$LO_i \leftarrow c(n)$
	WRITE *n	$LO_i \leftarrow c(c(n))$
9	JUMP ety	$II \leftarrow ety$
10	JFTZ ety	if $c(0) > 0$ then $II \leftarrow ety$
11	JMPZ ety	if $c(0) = 0$ then $II \leftarrow ety$
12	HALT	zatrzymanie pracy maszyny

Rysunek 2: Spis instrukcji maszyny RAM

Obliczanie wartości 5^n za pomocą maszyny o dostępie swobodnym ma następującą postać:

```

READ 1      ; odczytanie n
LOAD 1      ; i zapis do akumulatora
JMPZ L2     ; jeśli n=0 idziemy do L2
LOAD =5     ; podstawa
L1: STORE 2 ; wynik częściowy
LOAD 1      ; licznik pętli
SUB =1      ; zmniejszenie licznika
STORE 1     ; pętli
JMPZ L3

```

```

LOAD 2      ; wynik częściowy
MULT =5     ; jest mnożony przez pięć
JUMP L1     ; skok na początek pętli mnożącej
L2: WRITE =1 ; wynik gdy wykładnik równy zero
HALT
L3: WRITE 2  ; wynik gdy wykładnik większy od zera
HALT

```

Znak „=” oznacza, iż jest to wartość bezpośrednia, czyli LOAD =5 oznacza, że do akumulatora zostanie załadowana wartość 5.

2 Beztypowy rachunek lambda

2.1 Nieco historii

Rachunek lambda wraz z logiką kombinatoryczną powstał w latach trzydziestych dwudziestego wieku. Według autorów obydwa mechanizmy miały stanowić alternatywę dla teorii mnogości i stać się nowym podejściem w tłumaczeniu podstaw matematyki. Niestety, ostatecznie nie udało się zbudować odpowiednika teorii mnogości odwołując się do pojęcia funkcji. Jednakże sam rachunek lambda został zastosowany w teorii obliczeń. Definicja funkcji obliczalnej została podana w wcześniej w rachunku lambda, dopiero później podobną definicję funkcji obliczalnej podał Turing dla swojej koncepcji maszyny obliczeniowej.

Rachunek lambda posiada zastosowania nie tylko teoretyczne. Wraz z rozwojem języków programowania stał się istotnym narzędziem w semantyce. Pierwszym przykładem może być maszyna SECD Landina. Jest to uniwersalna maszyna zdolna do wykonywania dowolnego programu. Rachunek lambda jest również podstawą dla funkcyjnych języków programowania takich jak LISP, Scheme. Szczególnie w przypadku tego drugiego gdzie pojęcie procedury/funkcji jest bezpośrednio odnoszone do lambda-abstrakcji.

2.2 Składnia rachunku lambda

Upředzając wszelkie uwagi i rozważania definiujemy zbiór wszystkich termów:

Definicja 2.1 *Zbiór wszystkich lambda termów oznaczony jest przez: Λ .*

2.2.1 Gramatyka oraz notacja

Rachunek lambda oferuje niezwykle proste zasady budowy wyrażeń. W opisie BNF są to zaledwie trzy poniższe linijki gramatyki bezkontekstowej:

$$\begin{array}{l}
 1 \quad \langle \lambda - \text{wyr} \rangle ::= \langle \text{zmienna} \rangle \\
 2 \quad \quad \quad | (\langle \lambda - \text{wyr} 1 \rangle \langle \lambda - \text{wyr} 2 \rangle) \\
 3 \quad \quad \quad | (\lambda \langle \text{zmienna} \rangle . \langle \lambda - \text{wyr} \rangle)
 \end{array}$$

Linia pierwsza to definicja zmiennych (często określanych jako zmienne przedmiotowe). Zmienne nazywa się zwyczajowo małymi literami raczej z końca alfabetu łacińskiego np.: x,y,z ale też m,n,o,p. Linia druga to zapis operacji zastosowania (lub wprost zastosowanie) argumentu opisanego lambda wyrażeniem drugim do pierwszego wyrażenia. Lambda wyrażenia nazywane

są zwyczajowo dużymi literami jednak w wielu przypadkach gdy stanowią one nazwy konkretnych funkcji są pisane małymi literami. Dodatkowo aby podkreślić ich znacznie nazwy funkcji są pisane czcionką tłuścą oraz stosowane jest podkreślenie.

Ostatnia trzecia linia przedstawia tzw. lambda abstrakcję. Jest to operacja związania zmiennej (dość często mówi się, że jest to przyporządkowanie wyrażeniu pewnego argumentu) ze wszystkimi jej wystąpieniami do końca obszaru ograniczonego prawym nawiasem tego wyrażenia.

Pojęcie lambda-wyrażenia definiuje się również w sposób indukcyjny (podane w dalszej części definicje funkcji FV i BV posiadają podobną postać):

- zmienne np.: x, y, z są wyrażeniami;
- jeśli M i N są wyrażeniami to $(M N)$ także;
- jeśli M jest wyrażeniem a x to zmienną, to $(\lambda x.M)$ również jest wyrażeniem

Tak sformułowana składnia wymaga stosowania dużej ilości nawiasów co przy nawet względnie niewielkich wyrażeniach może powodować utratę czytelności. Dlatego stosuje się trzy dodatkowe reguły zwiększające czytelność oraz upraszczające zapis wyrażen:

- zastosowanie ma pierwszeństwo przed λ -abstrakcją, więc wyrażenie $\lambda x.x y$ w pełnej notacji ma postać $(\lambda x.(x y))$
- zastosowanie wiąże zawsze w lewą stronę: $x y z$ oznacza $((x y) z)$
- trzecia reguła dotyczy znaku kropki i abstrakcji zamiast wyrażenia $\lambda x_1.\lambda x_2.\lambda x_3.\dots \lambda x_n.M$ dopuszcza się zapis: $\lambda x_1 x_2 x_3 \dots x_n.M$

Stosowany jest jeszcze jeden skrót tzw. notacja wektorowa:

$$\lambda x_1 x_2 x_3 \dots x_i.M \text{ to } \lambda \vec{x}.M$$

Podobnie dla zastosowania:

$$M \vec{N} \text{ to } M N_1 N_2 \dots N_i$$

2.3 Długość wyrażenia lambda

Długość dowolnego wyrażenia oznacza się identycznie jak wartość bezwzględna: $|M|$. Stosuje się trzy reguły do wyznaczania długości wyrażen podstawowych. Długość zmiennej to:

$$|x| \stackrel{\text{def}}{=} 1$$

długość zastosowania jest równa długości poszczególnych wyrażen:

$$|(W_1, W_2)| \stackrel{\text{def}}{=} |W_1| + |W_2|$$

długością lambda abstrakcji jest suma jedności oraz długość wyrażenia W .

$$|(\lambda x.W)| \stackrel{\text{def}}{=} 1 + |W|$$

Przykład wyznaczenia długości lambda wyrażenia jest następujący:

$$\begin{aligned} |(\lambda x.xy)(\lambda z.x)(\lambda y.y)| &= |\lambda x.xy| + |\lambda z.x| + |\lambda y.y| \\ &= 1 + |xy| + 1 + |x| + 1 + |y| = \\ &= 1 + |x| + |y| + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 \end{aligned}$$

2.4 Zmienne wolne

Zmienne wolne to zmienne nie związane czyli takie dla których nie została zastosowana λ -abstrakcja. Funkcję która znajduje wystąpienia zmiennych wolnych określa się FV jest skrót od ang. słów *free variables*. Podobnie jak przy pojęciu długości również w zmiennych wolnych pojawiają się trzy definicje. Pierwsza wskazuje że zmienną wolną jest pojedyncze wystąpienie zmiennej:

$$FV[x] \stackrel{\text{def}}{=} \{x\}$$

Druga postać oznacza, że zmiennych wolnych należy szukać oddzielnie dla każdego wyrażania:

$$FV[W_1; W_2] \stackrel{\text{def}}{=} FV[W_1] \cup FV[W_2]$$

Trzecia definicja wyklucza ze zbioru zmiennych wolnych zmienną co do której zastosowano związanie:

$$FV[\lambda x.W] \stackrel{\text{def}}{=} FV[W] \setminus \{x\}$$

Przykład wyznaczania zmiennych wolnych:

$$\begin{aligned} FV[x(\lambda xy.xz)] &= FV[x] \cup FV[\lambda xy.xz] = \\ &= \{x\} \cup (FV[xz] \setminus \{x, y\}) = \\ &= \{x\} \cup ((FV[x] \cup FV[z]) \setminus \{x, y\}) = \\ &= \{x\} \cup ((\{x\} \cup \{z\}) \setminus \{x, y\}) = \{x, z\} \end{aligned}$$

Definicja 2.2 *Termem zamkniętym nazywamy term bez zmiennych wolnych: $FV(T) = \emptyset$. Term zamknięty określa się również mianem kombinatora.*

2.4.1 Zmienne związane

Analogicznie za pomocą podobnych wyrażeń jak dla FV można podać definicję zbioru zmiennych związanych. Tym razem stosuje się oznaczenie BV od ang. słów *bound variables*. Wolne wystąpienie zmiennej naturalnie nie należy do zbioru zmiennych związanych:

$$BV[x] = \emptyset$$

Wyznaczanie zmiennych związanych dla dwóch lambda wyrażeń czyli zastosowania identycznie jak przy zmiennych wolnych polega na oddzielnej analizie wyrażeń:

$$BV[W_1; W_2] \stackrel{\text{def}}{=} BV[W_1] \cup BV[W_2]$$

Zmienna związana lambda-abstrakcją naturalnie należy do zbioru zmiennych związanych:

$$BV[\lambda x.W] \stackrel{\text{def}}{=} BV[W] \cup x$$

2.4.2 Podtermy

Podtermy są wyznaczane przy pomocy funkcji S . Funkcja przyjmuje następującą postać podobną do definicji podanych wcześniej funkcji FV oraz BV :

- $S[x] \stackrel{\text{def}}{=} \{x\}$
- $S[(W_1 W_2)] \stackrel{\text{def}}{=} S[W_1] \cup S[W_2] \cup \{(W_1 W_2)\}$
- $S[(\lambda x.W)] \stackrel{\text{def}}{=} S[W] \cup \{(\lambda x.W)\}$

2.4.3 Zastąpienie (podstawienie)

Oznaczenie podstawienia pojawia się w dwóch postaciach: $M[N/x]$, co oznacza że w wyrażeniu M , w miejscu wystąpień zmiennej x zostanie wstawione wyrażenie N . Druga postać jest następująca: $M[x := N]$. W tej operacji trzeba jednak zwrócić uwagę na to, że nie podstawiamy tylko pod zmienne wolne ale i związane co może powodować kolizje w nazwach zmiennych. Należy w takim przypadku zmienić nazwy. Dokładna definicja uwzględniająca poszczególne przypadki jest następująca:

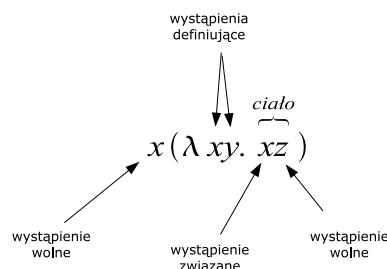
$$\begin{aligned}
 x[N/x] &= N \\
 y[N/x] &= y && y \neq x \\
 (PQ)[N/x] &= P[N/x]Q[N/x] \\
 (\lambda x.P)[N/x] &= \lambda x.P \\
 (\lambda y.P)[N/x] &= \lambda y.P && x \notin FV(\lambda y.P), x = y \\
 (\lambda y.P)[N/x] &= \lambda y.P[N/x] && y \neq x \in FV(P), y \notin FV(N) \\
 (\lambda y.P)[N/x] &= \lambda z.P[z/y][N/x] && y \neq x \in FV(P), y \in FV(N) \\
 &&& z \text{ jest nową zmienną}
 \end{aligned}$$

Przykład podstawienia, a gdy nie występuje kolizja jest trywialny:

$$((\lambda y.xy)x)[z/y] = ((\lambda z.xz)x)$$

Ten sam przykład, ale z kolizją wymaga skorzystania z ostatniej definicji podstawienia:

$$((\lambda y.xy)x)[y/x] = ((\lambda v.yv)y)$$



Rysunek 3: Przykładowe lambda wyrażenie

2.5 Redukcje i konwersje

2.5.1 Kongruencja λ -rachunku

Relację pomiędzy wyrażeniami w λ -rachunku nazywa się kongruencją. Relację oznacza się symbolem \sim i ma ona następujące własności:

- relacja \sim jest zwrotna, symetryczna i przechodnia (równoważność)
- jeśli $M \sim M'$ i $N \sim N'$ to $MN \sim M'N'$
- jeśli $M \sim M'$ to $\lambda x.M \sim \lambda x.M'$

2.5.2 Alfa-konwersja

Relacją alfa-konwersji (α -konwersji) jest najmniejsza relacja równoważności (oznaczana jako \equiv^α) w zbiorze wyrażeń spełniająca poniższy warunek:

$$\lambda x.M \equiv^\alpha \lambda y.[y/x]M \text{ dla dowolnego } y \notin FV[M]$$

O α -konwersji można też powiedzieć, że jest to operacja zmiany nazwy zmiennej. Co jest związane z następującym faktem: $M \equiv^\alpha M' \Leftrightarrow M \equiv M_0, M_1, M_2, \dots, M_n \equiv M'$ czyli M_{i-1} różni się od M_i (gdzie $i = 1, \dots, n$) tylko nazwą zmiennej związanej. Prawdziwe będą też następujące warunki: jeśli $M \equiv^\alpha M'$ to $\lambda x.M \equiv^\alpha \lambda x.M'$ oraz $MP \equiv^\alpha M'P \wedge PM \equiv^\alpha PM'$ dla dowolnego P .

Z tej definicji wynikają następujące własności α -konwersji:

- jeśli $M \equiv^\alpha N$ to $|M| = |N|$
- jeśli $M \equiv^\alpha N$ to $FV[M] = FV[N]$
- dla dowolnego M istnieje M' czyli $M \equiv^\alpha M'$ i w M' nie występują kolizje zmiennych

2.5.3 Beta-redukcja

Następną relacją w zbiorze Λ jest beta-redukcja (β -redukcja). Jest to najmniejsza relacja oznaczana przez \equiv^β o następujących własnościach:

- $(\lambda x.M)N \equiv^\beta M[N/x]$
- jeśli $M \equiv^\beta M'$ to $MN \equiv^\beta M'N$, $NM \equiv^\beta NM'$ oraz $\lambda x.M \equiv^\beta \lambda x.M'$

Relacja β -redukcja odgrywa najważniejszą rolę w przetwarzaniu wyrażeń w λ -rachunku ponieważ pozwala na zastosowanie argumentów do funkcji. Przykład gdy mamy wyrażenie $(\lambda x.1 + x + 2 + x)5$ to po zastosowaniu β -redukcji otrzymamy wyrażenie: $1 + 5 + 2 + 5$.

Ważnym pojęciem jest domknięcie przechodnio-zwrotne podanej powyżej relacji oznaczone jest jako:

$$\stackrel{\beta}{\leftarrow} \text{ bądź } \stackrel{\beta}{\rightarrow}$$

Wyrażenie w postaci $(\lambda x.M)N$ nazywa się β -redex'em (ang. redex – reducible expression) czyli jest to wyrażenie dla którego można zastosować relację β -redukcji. Gdy w termie M nie występuje żaden β -redex to nie zachodzi relacja $M \stackrel{\beta}{\equiv} N$. Oznacza to, że M jest w postaci β -normalnej (lub krócej w postaci normalnej).

Istnieją wyrażenia nie posiadające postaci normalnej np.:

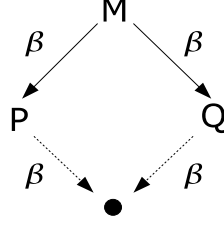
$$(\lambda x.xx)(\lambda x.xx) \stackrel{\beta}{\equiv} (\lambda x.xx)(\lambda x.xx) \stackrel{\beta}{\equiv} \dots$$

2.5.4 Twierdzenie Church'a-Rosser'a

W jednym termie może naturalnie znajdować się więcej niż jeden β -redex. Oznacza to, że dane wyrażenie można redukować na więcej niż jeden sposób. Nie oznacza to że można otrzymywać dwa różne ciągi ponieważ istnieje tw. Churcha-Rossera o następującej treści:

Twierdzenie 2.1 *Jeśli $P \stackrel{\beta}{\leftarrow} M \stackrel{\beta}{\rightarrow} Q$ to $P \downarrow_\beta Q$.*

Rysunek 4 pokazuje postać twierdzenia Churcha-Rossera jako romb. Jest to również postać dowodu. Jeśli M można zredukować na dwa sposoby i wiemy że P i Q są w postaci normalnej to istnieje T które powstała poprzez redukcję dokonane na drodze α -konwersji.



Rysunek 4: Uwaga do twierdzenia Church'a-Rosser'a w postaci rysunku

2.5.5 Eta-redukcja

Uzupełnieniem α -konwersji oraz β -redukcji jest η -redukcja. Oznaczana będzie symbolem $\stackrel{\eta}{\equiv}$ i spełnia następujące warunki:

- $\lambda x.M x \stackrel{\eta}{\equiv} M$ gdy $x \notin FV[M]$
- jeśli $M \stackrel{\eta}{\equiv} M'$ to $MN \stackrel{\eta}{\equiv} M'N$, $M \stackrel{\eta}{\equiv} M'$ to $NM \stackrel{\eta}{\equiv} NM'$ oraz $\lambda x.M \stackrel{\eta}{\equiv} \lambda x.M'$

Dość często spotyka się relację będącą sumą β oraz η redukcji (oznaczaną jako $\stackrel{\beta\eta}{\equiv}$).

2.6 Semantyka rachunku lambda

2.6.1 Semantyka aksjomatyczna (teoria równościowa)

Zakładamy że formuły M i N są termami rachunku lambda oraz $M = N$. Formalny system dowodzenia w postaci aksjomatów o przesłankach i regułach jest następujący:

$$\frac{}{\lambda \vdash M=N} (\rho) \quad \frac{\lambda \vdash M=N}{\lambda \vdash N=M} (\sigma) \quad \frac{\lambda \vdash M=N \quad \lambda \vdash N=P}{\lambda \vdash M=P} (\tau)$$

$$\frac{\lambda \vdash M=N}{\lambda \vdash QM=QN} (\mu) \quad \frac{\lambda \vdash M=N}{\lambda \vdash MQ=NQ} (\nu) \quad \frac{\lambda \vdash M=N}{\lambda \vdash \lambda x.M=\lambda x.N} (\xi)$$

$$\frac{}{\lambda \vdash (\lambda x.M)N=M[N/x]} (\beta) \quad \frac{x \notin FV[M]}{\lambda \vdash (\lambda x.Mx)=M} (\eta)$$

2.6.2 Semantyka operacyjna

Relacja β -redukcji dokonywana w jednym kroku jest zdefiniowana następująco:

$$(\lambda x.M)N \rightarrow_{\beta} M[N/x]$$

Podobnie operacja $\beta\eta$ -redukcji:

$$(\lambda x.Mx) \rightarrow_{\beta\eta} M \text{ gdy } x \notin FV[M]$$

2.6.3 Równoważność semantyk

Obydwie semantyki są sobie równe co przedstawia się w następujący sposób:

$$M =_{\beta} N \Leftrightarrow \lambda\beta \vdash M = N$$

$$M =_{\beta\eta} N \Leftrightarrow \lambda\beta\eta \vdash M = N$$

2.7 Siła wyrazu rachunku lambda

Rachunek lambda może wydawać się raczej prymitywnym narzędziem, gdzie można wyrażać tylko najprostsze, czy wręcz prymitywne elementy. Trudno spoglądając tylko na składnię i operacje jakie można w ramach rachunku wykonywać traktować λ -rachunek jako język programowania. Tym czasem, siła lambda rachunku leży w prostocie, bowiem za pomocą lambda abstrakcji i zastosowania można otrzymać w pełni funkcjonalny język programowania. Siła wyrazu lambda rachunku jest jednak znacznie większa i nieco odmienna ponieważ nie występuje w nim rekurencja typowa dla znaczącej większości języków programowania. Jednak rachunek oferuje operatory punktu stałego znakomicie rozwiązujące ten brak.

2.7.1 Kilka funkcji specjalnych

Wśród wielu oznaczeń na wyrażenia lambda rachunku niektóre są dość istotne więc zostaną w tym miejscu wymienione:

$$\begin{aligned}\underline{\mathbf{I}} &\stackrel{\text{def}}{=} \lambda x.x \\ \underline{\mathbf{K}} &\stackrel{\text{def}}{=} \lambda xy.x \\ \underline{\mathbf{S}} &\stackrel{\text{def}}{=} \lambda xyz.xz(yz) \\ \omega &\stackrel{\text{def}}{=} \lambda x.xx \\ \Omega &\stackrel{\text{def}}{=} \omega\omega\end{aligned}$$

Definicja funkcji o n argumentach zawsze równa zero (przy czym zero traktuje się tu w sposób tradycyjny) jest następująca:

$$\underline{\mathbf{Z}}_n \stackrel{\text{def}}{=} \lambda m_1 \dots m_n.0$$

Funkcja rzutu n -argumentowego na k -tą współrzędną:

$$\underline{\mathbf{\Pi}}_k^n \stackrel{\text{def}}{=} \lambda m_1 \dots m_n.m_k$$

2.8 Logika

Podstawowymi definicjami są wartości logiczne prawda i fałsz. Mają one zastosowania w większości wyrażen. I tak logiczną prawdę definiuje się w następujący sposób:

$$\underline{\mathbf{true}} \stackrel{\text{def}}{=} \lambda xy.x$$

Natomiast definicja logicznego fałszu różni się jednym detalem:

$$\underline{\mathbf{false}} \stackrel{\text{def}}{=} \lambda xy.y$$

Główna idea tych wyrażen to oddawanie jako wyniki pierwszego bądź drugiego argumentu. Jednak w połączeniu z odpowiednio dobranymi definicjami innych operatorów pojęcia te funkcjonują dokładnie tak jak należy. Następnym operatorem jest operator warunkowy, odpowiednik instrukcji warunkowej (selekcji):

$$\underline{\mathbf{if}} \stackrel{\text{def}}{=} \lambda bxy.bxy$$

Te trzy funkcji pozwalają na zdefiniowanie trzech podstawowych operatorów logicznych: not, and oraz or. Ich definicja przedstawia się następująco:

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

Ponieważ trudno na przysłowiowy pierwszy rzut oka uwierzyć, że podane wyrażenia istotnie funkcjonują poprawnie tych kilka poniższych przykładów powinno rozwiązać wątpliwości. Pierwszy przykład to instrukcja if z prawdziwym warunkiem:

$$\begin{aligned} \underline{\text{if true}} M N &\equiv (\lambda bxy.bxy) \underline{\text{true}} M N \\ &\stackrel{\beta}{\equiv} \underline{\text{true}} M N \stackrel{\beta}{\equiv} (\lambda xy.x) M N \stackrel{\beta}{\equiv} M \end{aligned}$$

Następny przykład związany z instrukcją if ale z fałszywym warunkiem jest następujący:

$$\begin{aligned} \underline{\text{if false}} M N &\equiv (\lambda bxy.bxy) \underline{\text{false}} M N \\ &\stackrel{\beta}{\equiv} \underline{\text{false}} M N \stackrel{\beta}{\equiv} (\lambda xy.y) M N \stackrel{\beta}{\equiv} N \end{aligned}$$

Przykład dla negacji jest następujący:

$$\begin{aligned} \underline{\text{not true}} &\equiv (\lambda b.\underline{\text{if}} b \underline{\text{false}} \underline{\text{true}}) \underline{\text{true}} \\ &\stackrel{\beta}{\equiv} \underline{\text{if true}} \underline{\text{false}} \underline{\text{true}} \stackrel{\beta}{\equiv} \underline{\text{false}} \end{aligned}$$

Operator and w opisie rachunku lambda działa w następujący sposób:

$$\begin{aligned} \underline{\text{and true false}} &\equiv (\lambda b_1 b_2.\underline{\text{if}} b_1 b_2 \underline{\text{false}}) \underline{\text{true}} \underline{\text{false}} \\ &\stackrel{\beta}{\equiv} \underline{\text{if true}} \underline{\text{false}} \underline{\text{false}} \stackrel{\beta}{\equiv} \underline{\text{false}} \end{aligned}$$

Następny przykład to operator lub:

$$\begin{aligned} \underline{\text{or true false}} &\equiv (\lambda b_1 b_2.\underline{\text{if}} b_1 \underline{\text{true}} b_2) \underline{\text{true}} \underline{\text{false}} \\ &\stackrel{\beta}{\equiv} \underline{\text{if true}} \underline{\text{true}} \underline{\text{false}} \stackrel{\beta}{\equiv} \underline{\text{true}} \end{aligned}$$

Wspólnym elementem jak widać w tych przykładach jest operator if oraz fakt, że wartości logiczne true oraz false pozwalają na selektywne wybieranie elementu. Odpowiednie ułożenie wartości logicznych powoduje, że stosowanie β -konwersji oznacza wybieranie właściwych wartości.

Nieco bardziej skomplikowaną formę przybierają operatory „jeśli ... to ...” oraz „wtedy i tylko wtedy” jeśli odrzuci się definicje prawdy i fałszu. Pierwszy z nich jest definiowany w następujący sposób:

$$\underline{\text{implies}} \stackrel{\text{def}}{=} \lambda u.\lambda v.(\lambda x.\lambda y.u(vxy)x)$$

Natomiast operator „wtedy i tylko wtedy” jest zdefiniowany tak:

$$\underline{\text{iff}} \stackrel{\text{def}}{=} \lambda u.\lambda v.(\lambda x.\lambda y.u(vxy)(vyx))$$

2.8.1 Pary, listy, krotki

W lambda rachunku można pokazać wiele definicji różnych pojęć. Jednym z przykładów są właśnie pary. Parę definiuje się w oparciu o warunek. Idea jak taka że istnieć będzie sposób na odwołanie się do pierwszego i drugiego elementu:

$$\underline{\text{pair}} \stackrel{\text{def}}{=} \lambda xyb.\underline{\text{if}} bxy$$

Odczytanie pierwszego lub drugiego elementu do zadanie funkcji **fst** i **snd** używają one jak widać wartości logicznych które jak wcześniej podano przekazują w wyniku odpowiednio pierwszy bądź drugi argument (czyli stanowią szczególną postać funkcji rzutu):

$$\begin{aligned}\mathbf{fst} &\stackrel{\text{def}}{=} \lambda p.p \mathbf{true} \\ \mathbf{snd} &\stackrel{\text{def}}{=} \lambda p.p \mathbf{false}\end{aligned}$$

Przykład pierwszy „wyciągnięcie” z pary pierwszego elementu:

$$\begin{aligned}\mathbf{fst}(\mathbf{pair} MN) &\equiv \lambda p.p \mathbf{true} (\mathbf{pair} MN) \\ &\stackrel{\beta}{=} \mathbf{pair} MN \mathbf{true} \equiv (\lambda xyb.\mathbf{if} bxy) MN \mathbf{true} \\ &\stackrel{\beta}{=} \mathbf{if} \mathbf{true} MN \stackrel{\beta}{=} M\end{aligned}$$

Przykład drugi „wyciągnięcie” z pary drugiego elementu:

$$\begin{aligned}\mathbf{snd}(\mathbf{pair} MN) &\equiv \lambda p.p \mathbf{false} (\mathbf{pair} MN) \\ &\stackrel{\beta}{=} \mathbf{pair} MN \mathbf{false} \equiv (\lambda xyb.\mathbf{if} bxy) MN \mathbf{false} \\ &\stackrel{\beta}{=} \mathbf{if} \mathbf{false} MN \stackrel{\beta}{=} M\end{aligned}$$

Krotka ta uogólnione pojęcie pary bowiem dopuszcza się jej dowolną wielkość:

$$(M_1, M_2, \dots, M_n) \stackrel{\text{def}}{=} \lambda x.x M_1, M_2, \dots, M_n$$

Wybór k-tego elementu tak zdefiniowanej struktury (n-elementów, wybieramy k-ty element) polega na wykorzystaniu funkcji rzutu:

$$\mathbf{sel}_k^n \stackrel{\text{def}}{=} \lambda p.p \mathbf{\Pi}_k^n$$

Na podstawie par można zdefiniować ogólną n-elementową listę bowiem jest ona zawsze reprezentowana w postaci par:

$$(E_1, E_2, E_3, \dots, E_n) \stackrel{\text{def}}{=} (E_1, (E_2, (E_3, (\dots) E_{n-1}) E_n))$$

Dla powyższego wyrażenia po zastosowaniu funkcji **fst** otrzymamy E_1 . Natomiast dla **snd** wynikiem będzie cała struktura rozpoczynająca się od wyrażenia E_2 . Oznacza to, że gdy chcemy się dostać do pierwszego elementu należy zastosować funkcję **fst** a ogólnie dla i-tego elementu wzór jest następujący:

$$\mathbf{adr} (E i) \stackrel{\text{def}}{=} \mathbf{fst}(\underbrace{\mathbf{snd}(\mathbf{snd}(\mathbf{snd}(\dots(\mathbf{snd} E))))}_{i-1 \text{ razy}})$$

2.8.2 Liczebniki Churcha

Liczby naturalne są reprezentowane w postaci tzw. liczebników Churcha. Wzór na dowolną naturalną liczbę jest następujący:

$$c_n \stackrel{\text{def}}{=} \lambda f x.f^n(x)$$

Zapis $f^n(x)$ należy rozumieć jako n-krotne zastosowanie funkcji f : $f(f(f(\dots(x)\dots)))$. Kilka pierwszych liczb naturalnych definiuje się tak:

$$\begin{aligned} \mathbf{0} &\stackrel{\text{def}}{=} \lambda f x. x \\ \mathbf{1} &\stackrel{\text{def}}{=} \lambda f x. f x \\ \mathbf{2} &\stackrel{\text{def}}{=} \lambda f x. f(f x) \\ \mathbf{3} &\stackrel{\text{def}}{=} \lambda f x. f(f(f x)) \\ \dots &\dots \dots \\ \mathbf{n} &\stackrel{\text{def}}{=} \lambda f x. f^n x \end{aligned}$$

Bardzo istotna uwaga: $\mathbf{1} =_{\beta\eta} \mathbf{I}$ ale $\mathbf{1} \neq_{\beta} \mathbf{I}$.

2.8.3 Następnik

Na podstawie liczebników Churcha można wyprowadzić pojęcie następnika i jest ono definiowane w następujący sposób:

$$\mathbf{succ} \stackrel{\text{def}}{=} \lambda n. \lambda f x. f(n f x)$$

To nie jest jedyna możliwa definicja następnika nieco inna jest następująca: $\lambda n f x. n f(f x)$ Następnik pozwala także definicję liczb naturalnych w sposób indukcyjny:

$$\begin{aligned} \mathbf{0} &\stackrel{\text{def}}{=} \lambda f x. x \\ \mathbf{1} &\stackrel{\text{def}}{=} \mathbf{succ} \mathbf{0} \\ \mathbf{2} &\stackrel{\text{def}}{=} \mathbf{succ} \mathbf{1} \\ \mathbf{3} &\stackrel{\text{def}}{=} \mathbf{succ} \mathbf{2} \\ \dots &\dots \dots \end{aligned}$$

Przykłady dla następnika są dość długie, nawet dla niewielkich wartości. Wyznaczenie wartości liczby 10 mogłoby zająć, co najmniej 15 linijek tekstu. Dlatego, zostanie pokazane rozwinięcie dla liczby 2.

$$\begin{aligned} \mathbf{2} f x &\equiv \mathbf{succ} \mathbf{1} f x \equiv \lambda n. \lambda f x. f(n f x) \mathbf{1} f x \\ &\stackrel{\beta}{=} (\lambda f x. f(\mathbf{1} f x)) f x \stackrel{\beta}{=} f(\mathbf{1} f x) \equiv f(\mathbf{succ} \mathbf{0} f x) \\ &\equiv f((\lambda n. \lambda f x. f(n f x)) \mathbf{0} f x) \stackrel{\beta}{=} f((\lambda f x. f(\mathbf{0} f x)) f x) \\ &\stackrel{\beta}{=} f(f(\mathbf{0} f x)) \equiv f(f((\lambda f x. x) f x)) \stackrel{\beta}{=} f(f x) \end{aligned}$$

Następny przykład pokazuje co zostanie wyznaczone dla następnika zero:

$$\begin{aligned} \mathbf{succ} \mathbf{0} &\equiv (\lambda n. \lambda f x. f(n f x) \mathbf{0} \stackrel{\beta}{=} \lambda f x. f(\mathbf{0} f x)) \\ &\equiv (\lambda f x. f((\lambda f x. x) f x)) \stackrel{\beta}{=} \lambda f x. f x \stackrel{\eta}{=} \lambda f. f \stackrel{\alpha}{=} \lambda x. x \equiv \mathbf{Id} \end{aligned}$$

2.8.4 Poprzednik

Zdefiniowanie poprzednika wymaga większego sprytu bowiem trzeba opisać fakt wywołania pewnej funkcji $n - 1$ razy mniej. Definicja pomocnicza funkcji h :

$$h \stackrel{\text{def}}{=} \lambda p. \mathbf{pair}(\mathbf{snd} p)(\mathbf{succ}(\mathbf{snd} p))$$

Pozwala na podanie czytelnej definicji poprzednika w następujący sposób:

$$\mathbf{pred} \stackrel{\text{def}}{=} \lambda n. \mathbf{fst}(n \ h \ (\mathbf{pair} \ \mathbf{0} \ \mathbf{0}))$$

Ponieważ zadaniem \mathbf{pred} jest zawsze wyciągnięcie pierwszego elementu z pary więc istotne są operacje dokonywane za pomocą funkcji pomocniczej h .

$$\begin{aligned} \mathbf{2} \ h \ (\mathbf{pair} \ \mathbf{0} \ \mathbf{0}) &\equiv (\lambda f x. f(fx)) \ h \ (\mathbf{pair} \ \mathbf{0} \ \mathbf{0}) \\ &\stackrel{\beta}{=} h(h \ (\mathbf{pair} \ \mathbf{0} \ \mathbf{0})) \\ &\stackrel{\beta}{=} h((\lambda p. \mathbf{pair}(\mathbf{snd} \ p)(\mathbf{succ} \ (\mathbf{snd} \ p)))(\mathbf{pair} \ \mathbf{0} \ \mathbf{0})) \\ &\stackrel{\beta}{=} h(\mathbf{pair} \ \mathbf{0} \ \mathbf{1}) \\ &\stackrel{\beta}{=} (\mathbf{pair}(\mathbf{snd}(\mathbf{pair} \ \mathbf{0} \ \mathbf{1}))(\mathbf{succ}(\mathbf{snd}(\mathbf{pair} \ \mathbf{0} \ \mathbf{1})))) \equiv \mathbf{pair} \ \mathbf{1} \ \mathbf{2} \end{aligned}$$

2.9 Działania

Z pomocą następnika oraz poprzednika można zrealizować podstawowe operacje arytmetyczne. I tak dodawanie jest zdefiniowane w następujący sposób:

$$\mathbf{add} \stackrel{\text{def}}{=} \lambda k n. k \ \mathbf{succ} \ n$$

W poniższym przykładzie nie jest rozwijana funkcja następnika tylko korzysta się bezpośrednio z definicji inaczej zapis straciłby na czytelności:

$$\begin{aligned} \mathbf{add} \ \mathbf{2} \ \mathbf{2} &\equiv (\lambda k n. k \ \mathbf{succ} \ n) \ \mathbf{2} \ \mathbf{2} \stackrel{\beta}{=} \mathbf{2} \ \mathbf{succ} \ \mathbf{2} \\ &\equiv (\lambda f x. f(fx)) \ \mathbf{succ} \ \mathbf{2} \stackrel{\beta}{=} \mathbf{succ}(\mathbf{succ} \ \mathbf{2}) \equiv \mathbf{succ} \ \mathbf{3} \equiv \mathbf{4} \end{aligned}$$

Operacja dodawania jest również przemienne:

$$\begin{aligned} \mathbf{add} \ \mathbf{2} \ \mathbf{0} &\equiv (\lambda k n. k \ \mathbf{succ} \ n) \ \mathbf{2} \ \mathbf{0} \stackrel{\beta}{=} \mathbf{2} \ \mathbf{succ} \ \mathbf{0} \\ &\stackrel{\beta}{=} (\lambda f x. f(fx)) \ \mathbf{succ} \ \mathbf{0} \stackrel{\beta}{=} \mathbf{succ}(\mathbf{succ} \ \mathbf{0}) \equiv \mathbf{succ}(\mathbf{1}) \equiv \mathbf{2} \end{aligned}$$

Zmiana kolejności argumentów nie spowoduje, że obliczenia nie będą poprawne:

$$\begin{aligned} \mathbf{add} \ \mathbf{0} \ \mathbf{2} &\equiv (\lambda k n. k \ \mathbf{succ} \ n) \ \mathbf{0} \ \mathbf{2} \stackrel{\beta}{=} \mathbf{0} \ \mathbf{succ} \ \mathbf{2} \\ &\stackrel{\beta}{=} (\lambda f x. x) \ \mathbf{succ} \ \mathbf{2} \stackrel{\beta}{=} \mathbf{2} \end{aligned}$$

Mnożenie w lambda rachunku przedstawia się następująco:

$$\mathbf{mult} \stackrel{\text{def}}{=} \lambda k n. k(\mathbf{add} \ n) \ \mathbf{0}$$

Przytoczony przykład, podobnie jak w przypadku ilustracji dodawania poszczególne operacje są podane bez pełnych rozwinięć następnika, podane zostały takie same argumenty:

$$\begin{aligned} \mathbf{mult} \ \mathbf{2} \ \mathbf{2} &\equiv (\lambda k n. k(\mathbf{add} \ n) \ \mathbf{0}) \ \mathbf{2} \ \mathbf{2} \stackrel{\beta}{=} \mathbf{2} \ (\mathbf{add} \ \mathbf{2}) \ \mathbf{0} \\ &\equiv (\lambda f x. f(fx)) \ (\mathbf{add} \ \mathbf{2}) \ \mathbf{0} \stackrel{\beta}{=} \mathbf{add} \ \mathbf{2}(\mathbf{add} \ \mathbf{2} \ \mathbf{0}) \\ &\stackrel{\beta}{=} \mathbf{2} \ \mathbf{succ} \ (\mathbf{add} \ \mathbf{2} \ \mathbf{0}) \stackrel{\beta}{=} \mathbf{succ}(\mathbf{succ}(\mathbf{add} \ \mathbf{2} \ \mathbf{0})) \\ &\stackrel{\beta}{=} \mathbf{succ}(\mathbf{succ}(\mathbf{succ}(\mathbf{succ} \ \mathbf{0}))) \equiv \mathbf{4} \end{aligned}$$

Sprawdzenie, czy podany argument jest zerem:

$$\mathbf{iszero} \stackrel{\text{def}}{=} \lambda n.n(\lambda x.\mathbf{false}) \mathbf{true}$$

Elementarny przykład sprawdzenia czy zero jest zerem według funkcji iszero:

$$\begin{aligned} \mathbf{iszero} \mathbf{0} &\equiv \lambda n.n(\lambda x.\mathbf{false}) \mathbf{true} \mathbf{0} \\ &\stackrel{\beta}{\equiv} \mathbf{0}(\lambda x.\mathbf{false})\mathbf{true} \stackrel{\beta}{\equiv} (\lambda f.x)(\lambda x.\mathbf{false})\mathbf{true} \stackrel{\beta}{\equiv} \mathbf{true} \end{aligned}$$

Funkcje: dodawania, mnożenia oraz następnika są ściśle ze sobą powiązane, każda nie jako zawiera w sobie poprzednią, widać to szczególnie, gdy ich definicje będzie się podawać jawnie bez żadnych skrótów. Uzupełnieniem jest definicja potęgowania:

$$\begin{aligned} \mathbf{succ} &\stackrel{\text{def}}{=} \lambda n.f x.f(n f x) \\ \mathbf{add} &\stackrel{\text{def}}{=} \lambda m n.f x.m f(n f x) \\ \mathbf{mult} &\stackrel{\text{def}}{=} \lambda m n.f x.m(n f) x \\ \mathbf{exp} &\stackrel{\text{def}}{=} \lambda m n.f x.m n f x \end{aligned}$$

2.10 Twierdzenie o punkcie stałym

W jednym z poprzednich punktów podano definicję funkcji Ω . Wyrażenie Ω nie posiada postaci normalnej. Stosowanie β -redukcji powoduje ciągle generowanie tego samego wyrażenia. Gdyby rozszerzyć definicję tej funkcji o możliwość wykonywania się innych czynności istniałaby możliwość cyklicznego wykonywania się jakiejś operacji. Odpowiada to pojęciu rekurencji bądź skoku albo pętli while. Funkcja o takich możliwościach przyjmuje następującą postać:

$$Y \stackrel{\text{def}}{=} \lambda W.(\lambda x.W(x x))(\lambda x.W(x x))$$

Odnoszą się do niej dwa następujące pojęcia:

Definicja 2.3 Punktem stałym λ -wyrażenia W jest wyrażenie V o następującej właściwości: $W V \stackrel{\beta}{\equiv} V$.

Definicja 2.4 Operatorem punktu stałego jest wyrażenie Y które przyporządkuje λ -wyrażeniom punkty stałe tych wyrażień: $Y V \stackrel{\beta}{\equiv} V (Y V)$

O tym że dowolne wyrażenie (np.: W) zastosowane do operatora Y spełnia równia punktu stałego można się przekonać wykonując następujące obliczenia $Y W$:

$$\begin{aligned} YW &= \underbrace{\lambda W.(\lambda x.W(x x))(\lambda x.W(x x))}_W W \\ &\stackrel{\beta}{\equiv} (\lambda x.W(x x))(\lambda x.W(x x)) \end{aligned}$$

Dalsze przekształcenia (β -redukcja) ostatniej części powyższego wzoru są następujące:

$$\begin{aligned} &(\lambda x.W(x x))(\lambda x.W(x x)) \\ &\stackrel{\beta}{\equiv} W \underbrace{(\lambda x.W(x x))(\lambda x.W(x x))}_W \equiv W(YW) \end{aligned}$$

W przekształceniu zostało wykorzystane poprzednie przekształcenia aby pokazać, że (YW) można zapisać też jako $(\lambda x.W(x x))(\lambda x.W(x x))$.

2.11 Ah ta silnia !

Na początek dwie proste definicje:

$$\underline{\text{sil}} \stackrel{\text{def}}{=} Y \underline{\text{SIL}}$$

Gdzie SIL to właściwa definicja:

$$\underline{\text{SIL}} \stackrel{\text{def}}{=} \lambda s. \lambda n. \underline{\text{if}}(\underline{\text{iszero}}\ n)\ 1\ (\underline{\text{mult}}\ n(s\ (\underline{\text{pred}}\ n)))$$

Samą definicję silni można przekształcać zgodnie z równaniem punktu stałego:

$$\underline{\text{sil}} \equiv Y\ \underline{\text{SIL}} \stackrel{\beta}{\equiv} \underline{\text{SIL}}\ (Y\ \underline{\text{SIL}}) \stackrel{\beta}{\equiv} \underline{\text{SIL}}\ \underline{\text{sil}}$$

Podane definicje pozwalają na wyznaczanie wartości tej rekurencyjnej funkcji opisaną za pomocą funkcji punktu stałego np.:

$$\underline{\text{sil}}\ 0 \stackrel{\beta}{\equiv} \underline{\text{SIL}}\ \underline{\text{sil}}\ 0 \stackrel{\beta}{\equiv} \underline{\text{if}}(\underline{\text{iszero}}\ 0)\ \underline{1}\ (\underline{\text{mult}}\ 0\ (\underline{\text{sil}}\ (\underline{\text{pred}}\ 0))) \stackrel{\beta}{\equiv} \underline{1}$$

Wyznaczenie wartości dla jedynki jest następujące:

$$\begin{aligned} \underline{\text{sil}}\ 1 &\stackrel{\beta}{\equiv} \underline{\text{SIL}}\ \underline{\text{sil}}\ 1 \stackrel{\beta}{\equiv} \underline{\text{if}}(\underline{\text{iszero}}\ 1)\ \underline{1}\ (\underline{\text{mult}}\ 1\ (\underline{\text{sil}}\ (\underline{\text{pred}}\ 1))) \\ &\stackrel{\beta}{\equiv} \underline{\text{mult}}\ 1\ (\underline{\text{sil}}\ (\underline{\text{pred}}\ 1)) \stackrel{\beta}{\equiv} \underline{\text{mult}}\ 1\ (\underline{\text{sil}}\ 0) \\ &\stackrel{\beta}{\equiv} \underline{\text{mult}}\ 1\ \underline{1} \stackrel{\beta}{\equiv} \underline{1} \end{aligned}$$

Natomiast wyznaczenie wartości dla dwójki wygląda tak:

$$\begin{aligned} \underline{\text{sil}}\ 2 &\stackrel{\beta}{\equiv} \underline{\text{SIL}}\ \underline{\text{sil}}\ 2 \stackrel{\beta}{\equiv} \underline{\text{if}}(\underline{\text{iszero}}\ 2)\ \underline{1}\ (\underline{\text{mult}}\ 2\ (\underline{\text{sil}}\ (\underline{\text{pred}}\ 2))) \\ &\stackrel{\beta}{\equiv} \underline{\text{mult}}\ 2\ (\underline{\text{sil}}\ (\underline{\text{pred}}\ 2)) \stackrel{\beta}{\equiv} \underline{\text{mult}}\ 2\ (\underline{\text{sil}}\ 1) \\ &\stackrel{\beta}{\equiv} \underline{\text{mult}}\ 2\ \underline{1} \stackrel{\beta}{\equiv} \underline{2} \end{aligned}$$

Zapis definicji lambda rachunku jak ogólnie sam lambda rachunek stał się wzorem dla języka Scheme. Definicja silni w tym języku przedstawia się następująco:

```
(define silnia
  (lambda (n)
    (if (= n 0) 1 (* n (silnia (- n 1))))))
```

```
(display "Silnia 5=")
(display (silnia 5))
```

Natomiast w Pascal ta funkcja przedstawia się następująco:

```
function silnia(n : longint) : longint;
begin
  if n=0 then
    silnia:=1
  else
    silnia:=n*silnia(n-1);
end;
```

3 Zadania

W zadaniach stosowane są następujące oznaczenia:

- A^* - zbiór wszystkich ciągów skończonych łącznie ze słowem pustym λ
- A^+ - zbiór wszystkich ciągów skończonych bez słowa pustego λ

W zadaniach, gdzie treść zadania jest poprzedzona skrótem (*form*) należy zastosować formalną definicję maszyny Turinga. W przypadku zadań dot. rachunku lambda, symbol λ oznacza naturalnie lambda abstrakcję.

1. Napisać program dla maszyny Turinga wykrywający palindromy.
2. Dla maszyny Turinga wykrywającej palindromy z rysunku 1, prześledzić sposób działania maszyny dla słowa $##abba##$ oraz dla słowa $##baba##$.
3. (form) Jaką funkcję $f(x)$ oblicza maszyna Turinga o następującym programie:

$$\begin{aligned}q_10 &\rightarrow q_20R \\q_11 &\rightarrow q_01 \\q_20 &\rightarrow q_01 \\q_21 &\rightarrow q_21R\end{aligned}$$

4. (form) Jaki typ funkcji oblicza poniższy program dla maszyny Turinga:

$$q_10 \rightarrow q_00$$

5. (form) Skonstruować maszynę Turinga, która prawidłowo obliczy tzn. maszyna zakończy swoje działanie przy spełnieniu odpowiedniego warunku: $f(x) = x + 1$.
6. (form) Napisać program dla maszyny Turinga, który oblicza wartość dla funkcji zerowej $o(x) = 0$. Dla dowolnego argumentu wartością funkcji $o(x)$ jest wartość zero.
7. (form) Skonstruować maszyny Turinga dla następujących funkcji:

(a) przeniesienie zera: $q_1001^x0 \mapsto q_001^x00$

(b) prawy skok: $q_1001^x0 \mapsto 01^xq_00$

(c) lewy skok: $01^xq_10 \mapsto q_001^x0$

(d) transpozycja: $01^xq_101^y0 \mapsto 01^yq_001^x0$

(e) podwojenie: $q_101^x0 \mapsto q_001^x01^x0$

(f) skok cykliczny: $q_101^{x_1}01^{x_2} \dots 01^{x_n}0 \mapsto q_001^{x_2} \dots 01^{x_n}01^{x_1}0$

(g) kopiowanie: $q_101^{x_1} \dots 01^{x_n}0 \mapsto q_001^{x_1} \dots 01^{x_n}01^{x_1} \dots 01^{x_n}0$

8. (form) Skonstruować maszynę Turinga, która prawidłowo oblicza funkcję rzutu:

$$\Pi_m^n(x_1, x_2, x_3, \dots, x_n) \quad \text{np. :} \quad \Pi_6^n(x_1, x_2, \dots, x_n) = x_6$$

Wykorzystać definicje funkcji z poprzedniego zadania oraz funkcję zerowania.

9. (form) Skonstruować maszyny Turinga, prawidłowo obliczające trzy następujące funkcje:

(a) $x + y$ – suma liczb całkowitych dodatnich

(b) $x - 1 = \begin{cases} 0, & \text{gdy } x = 0 \\ x - 1, & \text{gdy } x > 0 \end{cases}$

(c) $\text{sign } x = \begin{cases} 0, & \text{gdy } x = 0 \\ 1, & \text{gdy } x > 0 \end{cases}$

10. Napisać dwa programy dla maszyny Turinga obliczającej wartość funkcji sign:

$$\text{sign}(x) = \begin{cases} 0 & \text{jeśli } x = 0 \\ 1 & \text{jeśli } x > 0 \end{cases}$$

Jeśli, liczba x została zapisana w postaci dwójkowej oraz w postaci jedynekowej.

11. Opracować program dla maszyny Turinga kopiującej słowo wejściowe złożone z jedynek tzn. dla stanu początkowego w chcemy otrzymać ww .

12. Napisać program dla maszyny Turinga która będzie wykrywać, czy podane słowo należy do języka $L = 0^n 1^n$ dla $n \geq 1$. Oszacować złożoność obliczeniową.

13. Opracować maszynę Turinga która rozpoznaje czy słowo w należy do języka $L = 1^*01^*$. Odpowiedź maszyny powinna być następująca:

$$T(w) = \begin{cases} 0 & \text{jeśli } w \notin L \\ 1 & \text{jeśli } w \in L \end{cases}$$

14. Napisać program dla maszyny Turinga sprawdzającej, czy ciąg nawiasów został poprawnie zbudowany tzn. lewemu nawiasowi zawsze odpowiada prawy nawias.

15. Opracować programy dla maszyny licznikowej które wykonają następujące operacje:

(a) dodawanie liczb całkowitych

(b) odejmowanie liczb całkowitych

(c) mnożenie liczb całkowitych

(d) dzielenie liczb całkowitych

Wyznaczyć złożoność otrzymanych algorytmów.

16. Opracować program dla maszyny licznikowej który będzie realizował następujące operatory relacji dla dwóch zmiennych A i B :

(a) $A = B$

(b) $A \neq B$

(c) $A < B$

(d) $A > B$

Wykazać poprawność zastosowanych rozwiązań.

17. Wykorzystując utworzone w dwóch poprzednich zadaniach programy utworzyć program który będzie wyznaczać największy wspólny dzielnik dwóch liczb całkowitych dodatnich.
18. Opracować program dla maszyny RAM który obliczy wartość funkcji:

$$f(n) = n \frac{4n + 5}{2}$$

Wyznaczyć jego złożoność czasową.

19. Napisać program dla maszyny licznikowej który będzie wykonywał operację a^x .
20. Napisać program obliczający wartość silni dla maszyny o dostępie swobodnym. Wyznaczyć jego złożoność czasową.
21. Na taśmie wejściowej znajduje się liczba n bitowa liczba. Znacznikiem jej końca jest dowolna inna liczba poza naturalnie zerem bądź jednością. Napisać program dla maszyny RAM który sprawdzi, czy jest to liczba podzielna przez cztery.
22. Opracować program dla maszyny RAM który dla podanego ciągu wejściowego przepisze na ciąg wyjściowy tylko wielokrotności liczb 3. Ciąg wejściowy jest zakończony wartością zero.
23. Napisać program dla maszyny RAM który obliczy a^n .
24. Opracować program obliczający n -tą liczbę ciągu Fibonacciego dla maszyny RAM.
25. Napisać program na maszynę RAM który sprawdzi czy słowo w należy do języka o następującej postaci: $L = 1^*01^*$.
26. Obliczyć wartości logiczne dla funkcji **and**, **or**, **not** w ramach rachunku lambda.
27. Obliczyć następujące lambda wyrażenia (starając się podawać pełne rozwinięcia wyrażeń):

(a) **add 1 2**, **add 1 3**

(b) **sub 4 3**

(c) **mult 2 3**

28. Zdefiniować funkcję silnia **fact** i pokazać poszczególne β redukcje dla **fact 2**.
29. Zdefiniować funkcję **iszero** dla liczebników Churcha jako wyrażenie lambda rachunku określone w następujący sposób:

$$\begin{aligned} \mathbf{iszero}(0) &= \mathbf{true} \\ \mathbf{iszero}(n) &= \mathbf{false}, \text{ if } n \neq 0 \end{aligned}$$

30. Wykazać, iż dla dowolnych liczb naturalnych n, m :

(a) **add n m** $\stackrel{\beta}{\equiv}$ **n + m**

(b) **mult n m** $\stackrel{\beta}{\equiv}$ **n · m**

31. Zdefiniować liczby naturalne bez użycia liczebników Churcha, stosując za zero $(\lambda x.x)$, a kolejne liczby mogą być wyrażane za pomocą wartości fałszu logicznego.
32. Uwzględniając poprzednie zadanie udowodnić następujący lemat:

Lemat 3.1 *Istnieją wyrażenia lambda N, P, Z (następnik, poprzednik, test zera), takie że:*

$$\begin{aligned} S n &= n + 1 \\ P n + 1 &= n \\ Z 0 &= \text{true} \\ Z n + 1 &= \text{false} \end{aligned}$$

33. Udowodnić następujące twierdzenie:

Twierdzenie 3.1 *W beztypowym¹ rachunku lambda dowolny term F posiada punkt stały.*

34. Pokazać, że dla $\Omega \stackrel{\text{def}}{=} (\lambda x.xx)(\lambda x.xx)$ zachodzi:

$$\Omega \neq_{\beta\eta} \Omega\Omega.$$

4 Dalsze informacje

Poniższe pozycje odnoszą się do wszystkich list z ćwiczeniami z przedmiotu teoretyczne podstawy informatyki.

Literatura

- [1] David Harel: Rzecz o istocie informatyki Algorytmika, Edycja polska, Wydanie drugie, Wydawnictwa Naukowo-Techniczne 2000
- [2] Tomasz Bilski, Krzysztof Chmiel, Janusz Stokłosa: Zbiór zadań ze złożoności obliczeniowej algorytmów. Politechnika Poznańska 1992
- [3] Janusz Stokłosa: Zadania ze złożoności obliczeniowej algorytmów, Politechnika Poznańska 1989
- [4] L. Banachowski, Antoni Kreczmar: Elementy analizy algorytmów, Wydawnictwa Naukowo-Techniczne 1982
- [5] John E.Hopcroft, Jeffrey D.Ullman: Wprowadzenie do teorii automatów, języków i obliczeń. Wydawnictwo Naukowe PWN 2003
- [6] Mordechai Ben-Ari: Logika matematyczna w informatyce, Wydawnictwa Naukowo-Techniczne 2005
- [7] Christos H.Papadimitriou: Złożoność obliczeniowa, Wydawnictwa Naukowo-Techniczne 2002

¹Typ rachunku opisywany w tym dokumencie.

- [8] R.L. Graham, D.E. Knuth, O.Patashnik: Matematyka konkretna, Wydawnictwo Naukowe PWN 2002
- [9] Kenneth A.Ross, Charles R.B.Wright: Matematyka dyskretna, Wydawnictwo Naukowe PWN 2000
- [10] Piotr Wróblewski,: Algorytmy struktury danych i techniki programowania, Helion 1997
- [11] S. Sokołowski, Konstrukcja Języków Programowania, (notatki i slajdy do wykładu),2000/2001.
- [12] P. Urzyczyn, Rachunek Lambda, (notatki do wykładu),2002/2003.
- [13] M .Gordon, Introduction to Functional Programming, (notes for lectures).
- [14] J.C. Mitchell, Concepts in Programming Languages, Cambridge University Press,1988.
- [15] M.J.C. Gordon, Programming Language Theory and its Implementation, Cambridge University Press, 2002.