1 Rachunek λ

Niech V będzie przeliczalnie nieskończonym zbiorem zmiennych przedmiotowych x, y, ... (indeksowanych być może liczbami naturalnymi). Elementy takiego zbioru będziemy nazywali λ-*zmiennymi*. Ponieważ V jest potencjalnie nieskończony, zastrzegamy sobie możliwość wybierania w razie potrzeby wcześniej nie użytej zmiennej.

Definicja 1. (Zbiór $\tilde{\mathbf{\Lambda}}$ pretermów) Zbiorem pretermów będziemy nazywali najmniejszy (w sensie mnogościowym) zbiór wyrażeń $\tilde{\mathbf{\Lambda}}$ taki, że:

(P1) Jeśli $x \in V$, to $x \in \tilde{\mathbf{\Lambda}}$.

(P2) Jeśli M, $N \in \tilde{\mathbf{\Lambda}}$, to $(M\,N) \in \tilde{\mathbf{\Lambda}}$.

(P3) Jeśli $x \in V$ i $M \in \tilde{\mathbf{\Lambda}}$, to $(\lambda x.\, M) \in \tilde{\mathbf{\Lambda}}$.

Definicję **??** można równoznacznie wyrazić przy pomocy notacji Backusa-Naura. Wówczas ma ona następującą, zwięzłą postać:

$$\tilde{\mathbf{\Lambda}} \leftarrow V \mid (\tilde{\mathbf{\Lambda}}\,\tilde{\mathbf{\Lambda}}) \mid (\lambda V.\, \tilde{\mathbf{\Lambda}})$$

Elementy $\tilde{\mathbf{\Lambda}}$ będziemy oznaczali literami L, M, N, P, Q, R i ich wariantami z górnymi lub dolnymi indeksami. Wyrażenia postaci **??** nazywamy *aplikacjami* M do N. Symbol λ występujący w **??** nazywamy λ-*abstraktorem*, zaś wyrażenia powstałe przez zastosowanie tej reguły to λ-*abstrakcje*. W wyrażeniu postaci $(\lambda x.\, M)$ preterm M jest w *zasięgu* λ-abstraktora, a zmienna x jest przez niego *związana*. Ponadto, będziemy stosowali następujące konwencje notacyjne:

– *najbardziej zewnętrzne nawiasy bedą pomijane,*
– *aplikacja wiąże lewostronnie; wyrażenia postaci* $(PQ)R$ *będą zapisywane w postaci* PQR*,*
– λ*-abstrakcja wiaże prawostronnie:* $\lambda x_1.\,(\lambda x_2.\, P)$ *zapisujemy* $\lambda x_1.\, \lambda x_2.\, P$*,*
– *następujące po sobie* λ*-abstrakcje postaci* $\lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, P$ *zapisujemy pod wspólnym* λ*-abstraktorem:* $\lambda x_1\, x_2\, \ldots\, x_n.\, P$.

Powiemy, że dwa λ-termy są *syntaktycznie równe*, jeśli rozumiane jako ciągi znaków są identyczne. Równość syntaktyczną będziemy oznaczali znakiem $\equiv$.

Przykład 1. Podajmy kilka przykładów λ-pretermów pogrupowanych ze względu na ich konstrukcję.

(P1): x, y, z.

(P2): $x\,x$, $y\,x$, $x\,(x\,z)$,
$(\lambda x.\,(xz))y$, $y\,(\lambda x.\,(x\,z))$, $(\lambda x.\, x)\,(\lambda x.\, x)$.

(P3): $\lambda x.\,(x\,z)$, $\lambda yz.\, x$, $\lambda x.\,(\lambda x.\,(x\,x))$.

Podwyrażenia λ-pretermu mogą być wzajemnie identyczne i występować wielokrotnie. Obserwację tę ujmuje następująca definicja.

Definicja 2. (Multizbiór Sub podtermów pretermu)

(1) $\mathrm{Sub}(x) = \{x\}$

(2) $\mathrm{Sub}(MN) = \mathrm{Sub}(M) \cup \mathrm{Sub}(N) \cup \{M\,N\}$

(3) $\mathrm{Sub}(\lambda x.\, M) = \mathrm{Sub}(M) \cup \{\lambda x.\, M\}$

Elementy multizbioru $\mathrm{Sub}(M)$ nazywamy *podtermami* M. Jeśli L jest podtermem M, ale $L \not\equiv M$, to L nazywamy podtermem *właściwym*.

Przykład 2. Podtermy wybranych λ-pretermów.

$$\begin{aligned}
&\text{(a)} \quad \mathrm{Sub}\,(\lambda x.\, x\,x) = \left\{(\lambda x.\, x\,x)^1,\ (x\,x)^1,\ x^2\right\} \\
&\text{(b)} \quad \mathrm{Sub}\,((\lambda x.\, x\,x)\ (\lambda x.\, x\,x)) = \\
&\qquad\quad = \left\{((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))^1,\ (\lambda x.\, x\,x)^2,\ (x\,x)^2,\ x^4\right\}
\end{aligned}$$

W powyższych przykładach użyliśmy standardowej notacji w górnym indeksie umieszczając krotność występowania elementu.

Definicja 3. (Zbiór FV zmiennych wolnych) Dla dowolnego pretermu M określamy zbiór $\mathrm{FV}(M)$ *zmiennych wolnych* w M w następujący sposób:

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.\, P) &= \mathrm{FV}(P) \setminus \{x\} \\
\mathrm{FV}(PQ) &= \mathrm{FV}(P) \cup \mathrm{FV}(Q)
\end{aligned}$$

Jesli $\mathrm{FV}(M) = \varnothing$, to mówimy, że M jest *domknięty* lub nazywamy M *kombinatorem.*

Przykład 3. (a) $\mathrm{FV}(\lambda x.\, x\,y) = \{y\}$

(b) $\mathrm{FV}(x\,(\lambda x.\, x\,y)) = \{x,\ y\}$

(c) $\mathrm{FV}(\lambda xyz.\, xy) = \varnothing$

Definicja 4. (Podstawienie) Dla dowolnych M, N $\in \tilde{\mathbf{\Lambda}}$ i $x \in V$ przez $N[x/N]$ oznaczamy rezultat podstawienia termu N za wszystkie wolne wystąpienia zmiennej x w M, o ile w rezultacie podstawienia nie zostaną związane żadne zmienne wolne występujące w N. W takim wypadku:

(S1) $x[x/N] = N$

(S2) $y[x/N] = y,$ o ile $x \not\equiv y$

(S3) $(PQ)[x/N] = P[x/N]\,Q[x/N]$

(S4) $(\lambda y.\,P)[x/N] = \lambda y.\,P[x/N]$, gdzie $x \not\equiv y$ i $y \notin \mathrm{FV}(N)$

(S5) $(\lambda x.\,P)[x/N] = \lambda x.\,P$

Lemat 1. *(O podstawieniu) Niech M, N, $L \in \tilde{\mathbf{\Lambda}}$ i niech ponadto $x \not\equiv y$ oraz $x \notin \mathrm{FV}(L)$. Wówczas*

$$M[x/N][y/L] \equiv M[y/L]\,[x/N\,[y/L]]\,. \tag{1}$$

Dowód. Dowód przebiega przez indukcję strukturalną względem M. Rozważmy następujące przypadki:

i) M jest zmienną. Wówczas:

  a. Jeśli $M \equiv x$, to obie strony (**??**) po podstawieniu są postaci $N[y/L]$.

  b. Jeśli $M \equiv y$, to ponieważ $x \not\equiv y$ i $x \notin \mathrm{FV}(M)$, po wykonaniu podstawienia po lewej stronie (**??**) otrzymujemy $M[x/N][y/L] \equiv L$. Ponieważ $x \notin \mathrm{FV}(L)$, to po wykonaniu podstawienia po prawej stronie widzimy, że obydwie strony są identyczne.

  c. Jeśli $M \equiv z$ i $z \not\equiv x$ oraz $z \not\equiv y$, to obydwie strony (**??**) sa identyczne.

ii) $M \equiv PQ$ dla pewnych P, $Q \in \tilde{\mathbf{\Lambda}}$. Wówczas korzystając z hipotezy indukcyjnej wnosimy, że

$$P[x/N][y/L] \equiv P[y/L]\,[x/N\,[y/L]]\,,$$
$$Q[x/N][y/L] \equiv Q[y/L]\,[x/N\,[y/L]]\,.$$

Mając na względzie **??** widzimy, że twierdzenie zachodzi i w tym przypadku.

iii) Jeśli $M \equiv \lambda z.\,P$ oraz $z \equiv x$ lub $z \equiv y$, to z **??** widzimy, że obydwie strony (**??**) sa identyczne. Przypuśćmy, że $z \not\equiv x$ i $z \not\equiv y$ i $z \notin \mathrm{FV}(L)$. Wówczas na podstawie hipotezy indukcyjnej mamy:

$$\begin{aligned}(\lambda z.\,P)[x/N][y/L] &= \lambda z.\,P[x/N][y/L] = \\ &= \lambda z.\,P[y/L][x/N[y/L]] = \\ &= (\lambda z.\,P)[y/L][x/N[y/L]].\end{aligned}$$

□

Wniosek 1. *Jesli $M[x/y]$ jest określone i $y \notin \mathrm{FV}(M)$, to $M[x/y][y/x]$ jest określone oraz $M[x/y][y/x] = M$.*

Dowód. Mając na uwadze Lemat **??** dowód przebiega przez indukcję strukturalną względem M. □

1.1 Wyrażenia λ

Na ogół chcielibyśmy utożsamiać pretermy, które różnią się wyłącznie zmiennymi związanymi, tak jak w przypadku wyrażeń $\lambda x.\, zx$ i $\lambda y. zy$. W takim wypadku powiemy o nich, że są swoimi *α-wariantami* lub że są ze sobą w relacji *α-konwersji.*

Definicja 5. (Relacja α-konwersji) Relacją $=_\alpha$ (α-konwersji) nazywamy najmniejszy w sensie mnogościowym praporządek na $\tilde{\mathbf{\Lambda}}$ taki, że

(α1) Jeśli $y \notin \mathrm{FV}(M)$ oraz $M[x/y]$ jest określone,
to $\lambda x.\, M \;=_\alpha\; \lambda y.\, M[x/y]$

(α2) Jeśli $M =_\alpha N$, to dla dowolnego $x \in V$ zachodzi $\lambda x.\, M =_\alpha \lambda x.\, N$

(α3) Jeśli $M =_\alpha N$, to dla dowolnego $Z \in \tilde{\mathbf{\Lambda}}$ zachodzi $MZ =_\alpha NZ$

(α4) Jeśli $M =_\alpha N$, to dla dowolnego $Z \in \tilde{\mathbf{\Lambda}}$ zachodzi $ZM =_\alpha ZN$

Przykład 4.

$$\begin{aligned}\lambda xy.\, x(xy) &\equiv \lambda x.\, (\lambda y.\, x(xy)) \\ &\equiv_\alpha \lambda x.\, (\lambda z.\, x(xz)) \\ &\equiv_\alpha \lambda v.\, (\lambda z.\, v(vz)) \\ &\equiv \lambda vz.\, v(vz).\end{aligned}$$

Wniosek 2. *Relacja $=_\alpha$ jest relacją równoważności.*

Dowód. Wystarczy, że pokażemy, że relacja $=_\alpha$ jest symetryczna. Dowód przebiega przez indukcję względem Definicji **??**. Rozważmy następujące przypadki:

i) Jeśli $M =_\alpha N$ w konsekwencji zwrotności $=_\alpha$, to $M \equiv N$, a zatem również $N \equiv M$. Stąd $N =_\alpha M$.

ii) Jeśli $M =_\alpha N$ w konsekwencji przechodniości $=_\alpha$, to istnieje $L \in \tilde{\mathbf{\Lambda}}$ takie, że $M =_\alpha L$ i $L =_\alpha N$. Wówczas z hipotezy indukcyjnej $N =_\alpha L$ i $L =_\alpha M$. Z przechodniości relacji $=_\alpha$ otrzymujemy spodziewaną tezę.

iii) Przypuśćmy, że $M =_\alpha N$ w konsekwencji **??** dla $M \equiv \lambda x.\, M'$ i $N \equiv \lambda y.\, M'[x/y]$. Ponieważ $x \notin \mathrm{FV}(M'[x/y])$, to ze względu na Wniosek **??** mamy, że $M'[x/y][y/x] = M'$. Zatem, na podstawie **??**:

$$\lambda y.\, M'[x/y] =_\alpha \lambda x.\, M'[x/y][y/x].$$

iv) Jeśli $M =_\alpha N$ w konsekwencji **??**, gdzie $M = \lambda x.\, M'$ i $N = \lambda x.\, N'$ dla $M' =_\alpha N'$, to z hipotezy indukcyjnej $N' =_\alpha M'$ i w konsekwencji **??** mamy, że $N =_\alpha M$.

v) Jeśli $M =_\alpha N$ w konsekwencji **??** dla $M \equiv M'Z$ i $N \equiv N'Z$ takich, że $M' =_\alpha N'$, to z hipotezy indukcyjnej oczywiście $N' =_\alpha M'$, a zatem z **??** $N =_\alpha M$.

vi) Jeśli $M =_\alpha N$ w konsekwencji **??**, to postępujemy jak w przypadku **??**. $\square$

Definicja 6. (Zbiór $\mathbf{\Lambda}$ λ-termów) Każdą klasę abstrakcji relacji $=_\alpha$ nazywamy λ-termem. Zbiór wszystkich λ-termów $\mathbf{\Lambda}$ to zbiór ilorazowy relacji α-konwersji:

$$\mathbf{\Lambda} = \left\{ [M]_{=_\alpha} \mid M \in \tilde{\mathbf{\Lambda}} \right\}$$

Konwencja. *Wprowadzamy następujące konwencje notacyjne:*

$$\begin{aligned}
&x = [x]_{=_\alpha},\\
&PQ = [M'N']_{=_\alpha}, \text{ gdzie } M = [M']_{=_\alpha} \text{ i } N = [N']_{=_\alpha},\\
&\lambda x.\, M = [\lambda x.\, M']_{=_\alpha}, \text{ gdzie } N = [N']_{=_\alpha}.
\end{aligned}$$

Na zbiór $\mathbf{\Lambda}$ przenoszą się pojęcia podtermu, zmiennych wolnych i operacji podstawienia definiowane uprzednio dla pretermów.

Definicja 7. (Multizbiór Sub podtermów λ-termu) Dla dowolnego λ-termu $M = [M']_{=_\alpha}$ okreslamy

$$\mathrm{Sub}(M) = \mathrm{Sub}(M'),$$

gdzie $\mathrm{Sub}(M')$ jest multizbiorem podwyrażeń pretermu M' zdefiniowanym w myśl Definicji **??**.

Definicja 8. (Zbiór zmiennych wolnych FV) Dla dowolnego λ-termu $M = [M']_{=_\alpha}$ określamy zbiór $\mathrm{FV}(M)$ *zmiennych wolnych* w M

$$\mathrm{FV}(M) = \mathrm{FV}(M'),$$

gdzie $\mathrm{FV}(M')$ jest zbiorem zmiennych wolnych pretermu M' zdefiniowanym w myśl Definicji **??**.

Definicja 9. (Podstawienie) Niech $M = [M']_{=_\alpha}$ i $N = [N']_{=_\alpha}$ i niech $M'[x/N']$ będzie określone w myśl Definicji **??**. Wówczas

$$M[x/N] = [M'[x/N']]_{=_\alpha}.$$

Operacja podstawienia wymaga jednak pewnej delikatności. Rozważmy następującą relację:

$$\lambda x.\, zx =_\alpha \lambda y.\, zy$$

Zauważmy, że traktując podstawienie w sposób naiwny, mamy, że $(\lambda x.\, zx)[z/x] \neq_\alpha (\lambda y.\, zy)[z/x]$, a więc tracimy pożądaną własność niezmienniczości α-konwersji względem podstawienia. Stąd w Definicji **??** wymóg, aby podstawienie nie prowadziło do uszczuplenia zbioru zmiennych wolnych. Alternatywnym rozwiązaniem jest określenie podstawienia, które wprowadzałoby do wyrażenia nową zmienną i prowadziło w konsekwencji do abstrahowania po wcześniej nie występujacych zmiennych:

$$(\lambda x.\,M)[y/N] = \lambda x'.\,M[x/x'][y/N],$$

w przypadku, gdy $x \not\equiv y$, gdzie $x' \notin \mathrm{FV}(M)$ i $x' \notin \mathrm{FV}(N)$. Rozstrzygnięcie takie przytacza się w [**Hindley:2008:LCI:1388400**]. Po uwzględneniu odpowiednich modyfikacji, Definicja **??** przyjmuje następującą postać:

Definicja 4'. *(Podstawienie')*

(S'1) $x[x/N] = N$

(S'2) $y[x/N] = y$, *o ile* $x \not\equiv y$

(S'3) $(PQ)[x/N] = P[x/N]\,Q[x/N]$

(S'4) $(\lambda x.\,P)[x/N] = \lambda x.\,P$

(S'5) $(\lambda y.\,P)[x/N] = \lambda y.\,P$, *jeśli* $x \notin \mathrm{FV}(P)$

(S'6) $(\lambda y.\,P)[x/N] = \lambda y.\,P[x/N]$, *gdzie* $x \in \mathrm{FV}(P)$ *i* $y \notin \mathrm{FV}(N)$

(S'7) $(\lambda y.\,P)[x/N] = \lambda z.\,P[y/z][x/N]$, *gdzie* $x \in \mathrm{FV}(P)$ *i* $y \in \mathrm{FV}(N)$

przy czym w **??** *wymagamy, aby zmienna z nie występowała wcześniej w termach N i P jako zmienna wolna, zaś dla* **??**-**??** *dodatkowo* $y \not\equiv x$.

Uwaga 1. Każde podstawienie $[x/N]$ jest funkcją z $\mathbf{\Lambda} \to \mathbf{\Lambda}$, gdzie $x \in V$ i $N \in \mathbf{\Lambda}$ są dowolnymi parametrami. Zbiór S podstawień ma strukturę monoidu z działaniem składania

$$M([x_2/N_2] \circ [x_1/N_1]) = (M[x_1/N_1])[x_2/N_2] \equiv M[x_1/N_1][x_2/N_2]$$

dla dowolnych $[x_1/N_1]$, $[x_2/N_2] \in S$, o ile S posiada element neutralny ι taki, że

$$M\iota = M, \text{ gdzie } [x/x] = \iota \text{ dla dowolnego } x \in V.$$

W literaturze znajdujemy mnogość propozycji, które w ten czy inny sposób starają się ułatwić rzeczywistą implementację podstawienia. Na szczególną uwagę zasługują tutaj tak zwane *indeksy de Bruijna.* Zaproponowana przez N. G. de Brujina w [**deBruijnIndices**] notacja eliminuje bezpośrednie występowanie symboli zmiennych w λ-termach, zastępując je liczbą naturalną wyrażającą głębokość zagnieżdżenia odpowiedniej λ-abstrakcji przez którą jest związana, przykładowo:

$$\lambda f.\,(\lambda x.\,(f(xx))\lambda x.\,(f(xx))) \equiv_{deBrujin} \lambda(\lambda 2(11))\lambda 2(11)$$

Historycznie wiąże się ta notacja z jego pracami nad systemem komputerowo wspomaganego dowodzenia twierdzeń AUTOMATH. Rozwiązanie takie, podobnie jak w przypadku tzw. logik kombinatorów, eliminuje konieczność utożsamiania termów przez α-konwersję, ale istotnie zmniejsza ich czytelność.

Szerszy komentarz dotyczący dotychczasowych prób uchwycenia operacji podstawienia można prześledzić w [**txa:alpha-draft**]. Nasze rozważania opierają się w tej materii przeważająco na [**Urzyczyn2006**]. Samo podejście do definiowania λ-termow przez operację α-konwersji nie jest powszechne w literaturze przedmiotu. Analogiczną konstrukcję należałoby powtarzać wprowadzając każdy kolejny system, dlatego w dalszej części tej pracy będziemy poprzestawali na nieformalnym traktowaniu wyrażeń danego systemu jako odpowiednich klas α-konwersji.

Definicja 10. (Podstawienie jednoczesne) Dla dowolnego $M \in \mathbf{\Lambda}$, ciągu λ-zmiennych $\vec{x}$ i ciągu λ-termów $\vec{N}$ określamy:

($\vec{s}1$) $x_i[\vec{x}/\vec{N}] = N_i$ dla $i \in \mathbb{N}$.

($\vec{s}2$) $y[\vec{x}/\vec{N}] = y$ o ile dla dowolnego $i \in \mathbb{N}$, $y \not\equiv x_i$.

($\vec{s}3$) $(PQ)[\vec{x}/\vec{N}] = P[\vec{x}/\vec{N}]Q[\vec{x}/\vec{N}]$

($\vec{s}4$) $(\lambda y.\, P)[\vec{x}/\vec{N}] = \lambda y.\, P[\vec{x}/\vec{N}]$, jeśli $y \neq x_i$ dla wszystkich $i \in \mathbb{N}$ i $y \notin \bigcup_{i\in\mathbb{N}} FV(N_i)$

Konwencja. *Jeśli* $N_i \equiv x_i$ *dla wszystkich poza skończenie wieloma* $i_1, i_2, \ldots, i_n \in \mathbb{N}$, *to* $[x_{i_1}/N_{i_1}, x_{i_2}/N_{i_2}, \ldots, x_{i_n}/N_{i_n}] \equiv [\vec{x}/\vec{N}]$.

Przykład 5. Zauważmy, że podstawienia w myśl Definicji **??** i Definicji **??** mogą, ale nie muszą, prowadzić do różnych rezultatów.

a) $(xy)[y/x][x/u] = uu,$
$(xy)[y/x, x/u] = ux.$

b) $(\lambda x.\, yx)[x/y][y/z] = \lambda x.\, zx,$
$(\lambda x.\, yx)[x/y, y/z] = \lambda x.\, zx.$

1.2 Redukcja

Sens obliczeniowy λ-termom nadajemy przez określenie na $\mathbf{\Lambda}$ operacji β- i η-redukcji. Pożądane jest, żeby operacje te wykonywane na podtermach pozostowały w *zgodzie* ze strukturą całego λ-termu.

Definicja 11. (Relacja zgodna) Relację binarną $\mathcal{R}$ na zbiorze $\mathbf{\Lambda}$ nazywamy *zgodną*, jeśli dla dowolnych M, N, $P \in \mathbf{\Lambda}$ zachodzą następujące warunki:

(c1) Jeśli $M\mathcal{R}N$, to $(\lambda x.\, M)\,\mathcal{R}\,(\lambda x.\, N)$ dla dowolnej λ-zmiennej x.

(c2) Jeśli $M\mathcal{R}N$, to $(MP)\,\mathcal{R}\,(NP)$.

(c3) Jeśli $M\mathcal{R}N$, to $(PM)\,\mathcal{R}\,(PN)$.

Przez *domknięcie relacji* $\mathcal{R}_1$ będziemy rozumieli najmniejszą (w sensie mnogościowym) relację $\mathcal{R}_2$ taką, że $\mathcal{R}_1 \subset \mathcal{R}_2$. Z pewnego rodzaju domknięciami, ze względu na ich szczególną rolę, wiążemy następującą notację:

(a) Przez $\mathcal{R}^+$ oznaczamy przechodnie domknięcie relacji $\mathcal{R}$.

(b) Przez $\mathcal{R}^*$ oznaczamy zwrotnie domknięcie relacji $\mathcal{R}^+$.

(c) Przez $=_\mathcal{R}$ oznaczamy symetryczne domknięcie relacji $\mathcal{R}^*$.

Dla lepszego zrozumienia powyższych operacji warto zauważyć, że **??** wyznacza praporzadek, który w odniesieniu do redukcji określonych na $\mathbf{\Lambda}$ można rozumieć jako graf skierowany (w przypadku $\mathbf{\Lambda}$ być może nieskończony) w którym krawędzie odpowiadają możliwym krokom obliczenia, zaś **??** – kongruencję, która znów w szczególnym odniesieniu do λ-termów, będzie dokonywała podziału w $\mathbf{\Lambda}$ ze względu na rezultat obliczenia.

Definicja 12. (β-redukcja) β-redukcją nazywamy najmniejszą (w sensie mnogościowym) zgodną na $\mathbf{\Lambda}$ relację binarną $\to_\beta$ taką, że

$$(\lambda x.\, M)N \to_\beta M[x/N].$$

β-redeksami bedziemy nazywali wyrażenia postaci $(\lambda x.\, M)N$, zaś rezultat ich β-redukcji w postaci termu $M[x/N]$ – *β-reduktem. Ciągiem* β-redukcji nazywamy skończony lub nieskończony ciąg postaci

$$M_0 \to_\beta M_1 \to_\beta \dots$$

Przykład 6. Oznaczmy $\mathrm{Y} = \lambda f.\,(\lambda x.\,(f(xx))\lambda x.\,(f(xx)))$ i niech F będzie dowolnym λ-termem. Wówczas otrzymujemy nieskończony ciąg redukcji postaci

$$\begin{aligned}
\mathrm{Y}F &\equiv (\lambda f.\,(\lambda x.\,(f(xx))\lambda x.\,(f(xx))))F \\
&\to_\beta (\lambda x.\,F(xx))\lambda x.\,F(xx) \\
&\to_\beta F((\lambda x.\,F(xx))\lambda x.\,F(xx)) \\
&\to_\beta F(F((\lambda x.\,F(xx))\lambda x.\,F(xx))) \\
&\to_\beta \dots
\end{aligned}$$

Definicja 13. (η-redukcja) η-redukcją nazywamy najmniejszą (w sensie mnogościowym) zgodną na $\mathbf{\Lambda}$ relację binarną $\to_\eta$ taką, że

$$\lambda x.\, Mx \to_\eta M, \text{ o ile } x \notin \mathrm{FV}(M).$$

η-redukcja pozwala na pominięcie niczego nie wnoszącej λ-abstrakcji. Operację odwrotną nazywamy *η-abstrakcją*, zaś λ-termy będące w którejkolwiek z tych relacji nazywamy *η-konwersami*. Operacja ta nie ma wpływu na rezultat obliczenia, jedynie optymializuje zapis λ-termów i stąd ma duże znaczenie stylistyczne w programowaniu funkcyjnym.

Przykład 7. $\lambda x.((+1)x) =_\eta (+1)$.

1.3 Kodowanie typów danych

Prosta składnia języka rachunku λ pozwala wyrazić zaskakująco wiele struktur danych reprezentując je i operacje na nich jako funkcje. Z tego powodu, stanowiąc inspirację dla wielu projektantów języków programowania, uchodzi za protoplastę rodziny języków funkcyjnych, chociaż bezpośrednio nie ma on praktycznego zastosowania w praktyce programistycznej. Rozwój tej legendy dobrze oddaje cykl klasycznych artykułów (tzw. *Lambda Papers*) zapoczątkowany przez dokumentację języka Scheme [**Sussman:1975:IEL:889230**].

Najpopularniejszym sposobem reprezentacji danych przez funkcje w rachunku λ oparty jest na kodowaniu liczb Peano za pomocą tzw. liczebników Churcha. Metoda ta, ze względu na wynikające zeń problemy natury złożonościowej [**Koopman:2014:CED:2746325.2746330**], ma obecnie wyłącznie walory edukacyjne, dlatego w dalszej cześci pracy pokażemy tzw. kodowanie Scotta. Jest ona interesująca ze względu na praktyczną możliwość reprezentacji algebraicznych typów danych (ADT[1]) znanych ze współczesnych języków funkcyjnych [**Jansen:2013:P9C:2941698.2941** pozwalając tym samym zaimplementować te konstrukcje na przykład w paradygmacie imperatywnym. Fakt, że każdy typ danych można zastąpić tym sposobem odpowiadającą mu funkcją, wskazuje na metodę konstruowania prostych języków funkcyjnych, ale przede wszystkim na uniwersalność rachunku λ jako języka przejściowego dla kompilatorów języków wysokiego poziomu [**PeytonJones:1992:IFL:129390**].

1.3.1 Algebraiczne typy danych

Algebraiczne typy danych są podstawowym środkiem współczesnych języków funkcyjnych do wyrażania struktur danych. Powstają one przy użyciu tzw. typów sumacyjnych i typów produktowych, jednak pojęcia te na gruncie formalnym będą szczegółowo omówione w późniejszej części pracy. Na potrzeby prezentacji poszczególnych kodowań wystarczą nam w tym rozdziale intuicje o ADT zbudowane na gruncie następujących definicji w języku Haskell:

```
data Boolean     = True
                 | False
```

[1]Skrót od angielskojęzycznego *Algebraic Data Types*; nie należy mylić z *Abstract Data Types*.

```
data Tuple a b   = Tuple a b
data Temperature = Fahrenheit Int
                 | Celsius Int
data Maybe a     = Nothing
                 | Just a
data Nat         = Zero
                 | Suc Nat
data List t      = Nil
                 | Cons t (List t)
```

Definicja typu rozpoczynają się od słowa kluczowego `data`[2] po którym występuje *konstruktor typu*. Na wzór notacji BNF, typy przyjmują jedną z *wartości* odzielonych znakiem "|". Każda z wartości składa się z *konstruktora wartości* i ewentualnie występujących po nim *parametrów typowych*. Zauważmy, że umożliwia to rekurencyjnie konstruowanie typów, tak jak w wypadku `Nat` i `List`.

Pokażemy, że algebraiczne typy danych możemy reprezentować w zwięzły sposób w rachunku λ bez typów. Przedstawione tutaj koncepcje w zaskakujący sposób przenoszą się do bardziej złożonych typowanych systemów rachunku λ.

1.3.2 Proste typy wyliczeniowe

Typy wyliczeniowe to typy, które reprezentują możliwe warianty przyjmowanej wartości. Przykładem takiego typu jest `Boolean` z dwoma wartościami: `True`, `False`. Praca z tego rodzaju typami wymaga mechanizmu dopasowywania wzorców (ang. *pattern-matching*), który pozwala na wybór częściowej definicji funkcji w zależności od zadanego konstruktora wartości. Ponieważ w rachunku λ wyrażenia nie mają typów (lub, przyjmując perspektywę systemów z typami: wszystkie wyrażenia mają jeden, ten sam typ), interesowało nas będzie nie bezpośrednie kodowanie typu, ale kodowanie mechanizmu, który odpowiada za dopasowywanie wzorców. Posłużmy się znowu przykładem z języka Haskell i określmy funkcję odpowiadającą wykonaniu instrukcji warunkowej:

```
if True  a b = a
if False a b = b
```

gdzie `True` i `False` są wartościami typu `Boolean`. Właśnie ze względu na nie, mechanizm dopasowywania wzorca wybiera odpowiednią implementację instrukcji warunkowej. Ten sam efekt osiągnęlibyśmy kodując `True` i `False` w rachunku λ w

[2]Dyskusja ta ma na celu wyłącznie ustalenie uwagi; świadomi jesteśmy niuansów związanych z określaniem synonimów typów lub definiowaniem typów przy pomocy słowa kluczowego `newtype`.

następujący sposób:

$$\text{True} \equiv \lambda ab.\, a$$
$$\text{False} \equiv \lambda ab.\, b$$

Wówczas funkcję `if` możemy reprezentować wyrażeniem $\text{if} \equiv \lambda cte.\, cte$ lub jego η-reduktem: $\lambda c.\, c$.

1.3.3 Pary w rachunku λ

Parą nazywamy każdy nierekurencyjny typ, który posiada jeden konstruktor wartości parametryzowany przez dwa typy. W takim wypadku potrzebujemy dwóch funkcji zwracających odpowiednio pierwszy i drugi element pary. Przykładem takiego typu jest `Tuple`. Mamy wówczas:

```
fst (Tuple a b) = a
snd (Tuple a b) = b
```

Tego rodzaju typy możemy reprezentować przez tak zwane *domknięcie* (ang. *closure*), czyli cześciową aplikację termu. Standardowym sposobem reprezentacji pary w rachunku λ jest:

$$\text{Tuple} \equiv \lambda abf.\, fab$$

Aplikując Tuple tylko do dwóch termów (*domykając* term Tuple) otrzymujemy reprezentację pary. Pozostały, trzeci argument f nazywamy *kontynuacją*, gdyż aplikując (Tuple x y) dla dowolnych $x, y \in \mathbf{\Lambda}$ do pewnego $f \in \mathbf{\Lambda}$, w konsekwencji x i y zostają zaaplikowane do f. Zauważmy, że wówczas reprezentacja `fst` i `snd` ma postać:

$$\text{fst} \equiv \lambda t.\, t(\lambda ab.\, a)$$
$$\text{snd} \equiv \lambda t.\, t(\lambda ab.\, b)$$

Przykład 8. Wprowadzone konstrukcje pozwalają nam na definicję skończonych (w sensie liczby konstruktorów) typów. Rozważmy następujące przykłady:

a) Konstruktory wartości typu `Maybe` możemy reprezentować przez

$$\text{Nothing} \equiv \lambda nj.\, n$$
$$\text{Just} \equiv \lambda anj.\, ja$$

Rozważmy następującą funkcję:

```haskell
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x
```

Odpowiadająca jej reprezentacja to

$$\text{maybe} \equiv \lambda b f t.\, t b (\lambda a.\, f a)$$

b) Rozważmy następującą funkcję

```haskell
fromTemperature :: Temperature -> Int
fromTemperature (Fahrenheit a) = a
fromTemperature (Celsius a) = a
```

Ustalając reprezentację konstruktorów `Fahrenheit` i `Celsius`:

$$\text{Fahrenheit} \equiv \lambda t f c.\, f t$$
$$\text{Celsius} \equiv \lambda t f c.\, c t$$

otrzymujemy reprezentację funkcji `formTemperature` postaci:

$$\text{fromTemperature} \equiv \lambda t.\, t (\lambda f.\, f)(\lambda c.\, c)$$

1.3.4 Typy wyliczeniowe

W ogólnym przypadku, mając nastepującą definicję ADT:

```haskell
data type_constructor t1 t2 ... tk = C1 t11 ... t1n1
                                   | C2 t21 ... t2n2
                                   ...
                                   | Cm tm1 ... tmnm
```

dla $m, n \in \mathbb{N}$, wiążemy z nią następującą reprezentację w rachunku λ:

$$\begin{aligned}
\mathrm{C}_1 &\equiv \lambda t_{11}\, t_{12} \ldots t_{1n_1}\, f_1\, f_2 \ldots f_m.\ f_1 t_{11} t_{12} \ldots t_{1n_1} \\
\mathrm{C}_2 &\equiv \lambda t_{21}\, t_{22} \ldots t_{2n_2}\, f_1\, f_2 \ldots f_m.\ f_2 t_{21} t_{22} \ldots t_{2n_2} \\
&\vdots \\
\mathrm{C}_\mathrm{m} &\equiv \lambda t_{m1}\, t_{m2} \ldots t_{mn_m}\, f_1\, f_m \ldots f_m.\ f_1 t_{m1} t_{m2} \ldots t_{mn_m}
\end{aligned}$$