# 6. Wykład 6: Rachunek $\lambda$: programowanie w rachunku $\lambda$, kombinator punktu stałego, kody de Bruijn'a.

6.1. **Programowanie w rachunku $\lambda$.** Chcąc rozwinąć rachunek $\lambda$ do pełnowartościowego języka funkcyjnego, musimy najpierw pokazać, w jaki sposób można zakodować często używane datatypy takie jak zmienne boole'owskie, liczby naturalne, czy listy w rachunku $\lambda$. Jako że wszystkie wartości w rachunku $\lambda$ są $\lambda$-abstrakcjami, wszystkie wzmiankowane datatypy również są zakodowane z $\lambda$-abstrakcjami. Gdy raz zobaczymy w jaki sposób można zakodować specyficzne datatypy, będziemy mogli używać ich jak wbudowanych datatypów.

6.1.1. *Zmienne boole'owskie Churcha.* Główną cechą charakteryzującą zmienne boole'owskie jest możliwość wyboru jednej z dwóch różnych opcji. Na przykład, boole'owska prawda dokonuje wyboru pierwszej z dwóch możliwych opcji, tak jak w wyrażeniu SML-a `if true then` $e_1$ `else` $e_2$`.` Tym samym wartości boole'owskie w rachunku $\lambda$, zwane tu wartościami boole'owskimi Churcha, zapisujemy następująco:

$$\begin{array}{rcl} \mathsf{tt} & = & \lambda t.\lambda f.t \\ \mathsf{ff} & = & \lambda t.\lambda f.f \end{array}$$

Dalej, konstrukcja warunkowa $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ jest definiowana następująco:

$$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \quad = \quad e e_1 e_2$$

Oto proste przykłady redukowania konstrukcji warunkowych przy użyciu strategii wywołania według nazwy:

$$\begin{array}{rcccccccc} \mathsf{if\ tt\ then}\ e_1\ \mathsf{else}\ e_2 & = & \mathsf{tt}e_1e_2 & = & \underline{(\lambda t.\lambda f.t)e_1}e_2 & \mapsto & \underline{(\lambda f.e_1)e_2} & \mapsto & e_1 \\ \mathsf{if\ ff\ then}\ e_1\ \mathsf{else}\ e_2 & = & \mathsf{ff}e_1e_2 & = & \underline{(\lambda t.\lambda f.f)e_1}e_2 & \mapsto & \underline{(\lambda f.f)e_2} & \mapsto & e_2 \end{array}$$

Operatory logiczne na zmiennych boole'owskich są zdefiniowane następująco:

$$\begin{array}{rcl} \mathsf{and} & = & \lambda x.\lambda y.xy\mathsf{ff} \\ \mathsf{or} & = & \lambda x.\lambda y.x\mathsf{tt}y \\ \mathsf{not} & = & \lambda x.x\mathsf{ff}\mathsf{tt} \end{array}$$

Jako przykład rozważmy następujące ciągi redukcji $e_1e_2$ gdy $e_1 \mapsto^* \mathsf{tt}$ oraz $e_1 \mapsto^* \mathsf{ff}$, odpowiednio, przy użyciu strategii wywołania według nazwy:

$$\begin{array}{rlcl} & \mathsf{and}e_1e_2 & & \mathsf{and}e_1e_2 \\ \mapsto^* & e_1e_2\mathsf{ff} & \mapsto^* & e_1e_2\mathsf{ff} \\ \mapsto^* & \mathsf{tt}e_2\mathsf{ff} & \mapsto^* & \mathsf{ff}e_2\mathsf{ff} \\ \mapsto^* & e_2 & \mapsto^* & \mathsf{ff} \end{array}$$

Ciąg po lewej stronie pokazuje, że jeśli zachodzi $e_1 \mapsto^* \mathsf{tt}$, to $\mathsf{and}e_1e_2$ oznacza taką samą wartość prawdy, co $e_2$. Ciąg po prawej stronie pokazuje, że jeśli zachodzi $e_1 \mapsto^* \mathsf{ff}$, to $\mathsf{and}e_1e_2$ wartościuje się do $\mathsf{ff}$ niezależnie od $e_2$.

6.1.2. *Pary uporządkowane.* Główną cechą charakteryzującą pary jest możliwość przechowania dwóch niepowiązanych wartości i odczytania każdej z nich wtedy, gdy jest to konieczne. Tym samym chcąc reprezentować parę $(e_1, e_2)$, musimy zbudować $\lambda$-abstrakcję, która zwraca $e_1$ lub $e_2$, gdy zastosujemy ją do $\mathsf{tt}$ i $\mathsf{ff}$, odpowiednio. Operatory rzutowania traktują parę jak $\lambda$-abstrakcję i stosują ją do $\mathsf{tt}$ albo $\mathsf{ff}$.

$$\begin{array}{rcl} \mathsf{pair} & = & \lambda x.\lambda y.\lambda b.bxy\mathsf{ff} \\ \mathsf{fst} & = & \lambda p.p\mathsf{tt}y \\ \mathsf{snd} & = & \lambda p.p\mathsf{ff} \end{array}$$

Jako przykład zredukujmy $\mathsf{fst}(\mathsf{pair}e_1e_2)$ z użyciem strategii wywołania według nazwy. Zauważmy, że para $e_1e_2$ wartościuje się do $\lambda b.be_1e_2$, które z kolei oczekuje wartości boole'owskiej $b$ celem wyboru $e_1$ lub $e_2$. Jeżeli podstawimy $\mathtt{tt}$ w miejsce $b$, to wówczas $e_1e_2$ zredukuje się do $e_1$.

$$\begin{array}{rcl} \mathsf{fst}(\mathsf{pair}e_1e_2) & \mapsto & (\mathsf{pair}e_1e_2)\mathtt{tt} \\ & \mapsto^* & (\lambda b.be_1e_2)\mathtt{tt} \\ & \mapsto & \mathtt{tt}e_1e_2 \\ & \mapsto^* & e_1 \end{array}$$

6.1.3. *Liczebniki Churcha.* Główną cechą charakteryzującą liczbę naturalna $n$ jest możliwość przekazania informacji, że daną czynność należy powtórzyć $n$ razy. W przypadku rachunku $\lambda$ liczba $n$ jest zakodowana jako $\lambda$-abstrakcja $\hat{n}$ zwana **liczebnikiem Churcha**, która bierze daną funkcję $f$ i zwraca $f^n = \underbrace{f \circ f \circ \ldots \circ f}_{n \text{ razy}}$. Zauważmy, że $f^0$ jest funkcją identycznościową $\lambda x.x$, ponieważ $f$ zastosowane 0 razy do swego argumentu $x$ niczego nie zmienia, tak więc $\hat{1}$ jest sama funkcją identycznościową $\lambda f.f$.

$$\begin{array}{rcccl} \hat{0} & = & \lambda f.f^0 & = & \lambda f.\lambda x.x \\ \hat{1} & = & \lambda f.f^1 & = & \lambda f.\lambda x.fx \\ \hat{2} & = & \lambda f.f^2 & = & \lambda f.\lambda x.f(fx) \\ \hat{3} & = & \lambda f.f^3 & = & \lambda f.\lambda x.f(f(fx)) \\ \vdots & & & & \\ \hat{n} & = & \lambda f.f^n & = & \lambda f.\lambda x.f(f(f\ldots(fx)\ldots)) \end{array}$$

Jeśli będziemy odczytywać $f$ jak $\mathtt{S}$, to wówczas $\hat{n}fx$ zwróci nam reprezentację liczby naturalnej $n$ znaną z wcześniejszych wykładów.

Zdefiniujmy teraz operacje arytmetyczne na liczbach naturalnych. Operacja dodawania $\mathsf{add}\hat{m}\hat{n}$ zwraca $\widehat{m+n}$, czyli $\lambda$-abstrakcję biorącą funkcję $f$ i zwracającą $f^{m+n}$. Ponieważ $f^{m+n}$ może być zapisane jako $\lambda x.f^{m+n}x$, skonstruujemy $\mathsf{add}$ następująco (celem odróżnienia liczb naturalnych, na przykład liczby $n$, od odpowiadających im postaci zakodowanych, na przykład $\hat{n}$, będziemy używać $\hat{m}$ i $\hat{n}$ jako zmiennych):

$$\begin{array}{rcl} \mathsf{add} & = & \lambda\hat{m}.\lambda\hat{n}.\lambda f.f^{m+n} \\ & = & \lambda\hat{m}.\lambda\hat{n}.\lambda f.\lambda x.f^{m+n}x \\ & = & \lambda\hat{m}.\lambda\hat{n}.\lambda f.\lambda x.f^m(f^nx) \\ & = & \lambda\hat{m}.\lambda\hat{n}.\lambda f.\lambda x.\hat{m}f(\hat{n}fx) \end{array}$$

Zauważmy, że $f^m$ otrzymujemy jako $\hat{m}f$ (i analogicznie dla $f^n$).

Operacja mnożenia może być zdefiniowana na dwa sposoby. Prostą sztuczką jest wykorzystanie równości $m \cdot n = \underbrace{m + m + \ldots + m}_{n \text{ razy}}$. Innymi słowy $m \cdot n$ otrzymujemy przez dodanie $m$ do zera dokładnie $n$ razy. Jako że $\mathsf{add}\hat{m}$ jest w założeniu funkcją dodającą $m$ do swego argumentu, zastosujemy $\mathsf{add}\hat{m}$ do $\hat{0}$ dokładnie $n$ razy celem otrzymania $\widehat{m \cdot n}$ – lub, równoważnie, zastosujemy $(\mathsf{add}\hat{m})^n$ do $\hat{0}$:

$$\begin{array}{rcl} \mathsf{mult} & = & \lambda\hat{m}.\lambda\hat{n}.(\mathsf{add}\hat{m})^n\hat{0} \\ & = & \lambda\hat{m}.\lambda\hat{n}.\hat{n}(\mathsf{add}\hat{m})\hat{0} \end{array}$$

Drugi pomysł na zdefiniowanie tej samej operacji (który w istocie może być prostszy do wymyślenia od podanego przed chwilą rozwiązania) polega na wykorzystaniu równania $f^{m \cdot n} = (f^m)^n = (\hat{m}f)^n = \hat{n}(\hat{m}f)$:

$$\mathsf{mult} = \lambda\hat{m}.\lambda\hat{n}.\lambda f.\hat{n}(\hat{m}f)$$

Operacja odejmowania jest trudniejsza do zdefiniowania niż poprzednie dwie. Przypuśćmy bowiem, że mamy zdefiniowaną funkcję poprzednika $\mathsf{pred}$ obliczającą poprzednik danej liczby naturalnej: $\mathsf{pred}\hat{n}$ zwraca $\widehat{n-1}$ jeśli $n > 0$ lub $\hat{0}$ w przeciwnym wypadku. Chcąc zdefiniować operację odejmowania $\mathsf{sub}\hat{m}\hat{n}$, która zwraca $\widehat{m-n}$ jeśli $m > n$ lub $\hat{0}$ w przeciwnym wypadku, musimy zastosować $\mathsf{pred}$ do $\hat{m}$ dokładnie $n$ razy:

$$\begin{aligned}\mathsf{sub} &= \lambda\hat{m}.\lambda\hat{n}.\mathsf{pred}^n\hat{m}\\ &= \lambda\hat{m}.\lambda\hat{n}.(\hat{n}\mathsf{pred})\hat{m}\end{aligned}$$

Pozostaje, oczywiście, problem zdefiniowania funkcji $\mathsf{pred}$. Zrobimy to przy użyciu pomocniczej funkcji $\mathsf{next}$, która dla danej pary $\mathsf{pair}\hat{k}\hat{m}$ ignoruje $\hat{k}$ i zwraca parę $\mathsf{pair}\hat{m}\widehat{m+1}$:

$$\mathsf{next} = \lambda p.\mathsf{pair}(\mathsf{snd}p)(\mathsf{add}(\mathsf{snd}p)\hat{1})$$

Można pokazać, że stosując $\mathsf{next}$ do pary $\mathsf{pair}\hat{0}\hat{0}$ dokładnie $n$ razy otrzymamy parę $\mathsf{pair}\widehat{n-1}\hat{n}$ jeśli $n > 0$ (przy użyciu pewnej strategii redukcyjnej):

$$\begin{aligned}\mathsf{next}^0(\mathsf{pair}\hat{0}\hat{0}) &\mapsto^* \mathsf{pair}\hat{0}\hat{0}\\ \mathsf{next}^1(\mathsf{pair}\hat{0}\hat{0}) &\mapsto^* \mathsf{pair}\hat{0}\hat{1}\\ \mathsf{next}^2(\mathsf{pair}\hat{0}\hat{0}) &\mapsto^* \mathsf{pair}\hat{1}\hat{2}\\ &\vdots\\ \mathsf{next}^n(\mathsf{pair}\hat{0}\hat{0}) &\mapsto^* \mathsf{pair}\widehat{n-1}\hat{n}\end{aligned}$$

Jako że poprzednikiem zera jest tak czy inaczej zero, pierwszy składnik $\mathsf{next}^n(\mathsf{pair}\hat{0}\hat{0})$ koduje poprzednik $n$. Tym samym operacja $\mathsf{pred}$ jest zdefiniowana następująco:

$$\begin{aligned}\mathsf{pred} &= \lambda\hat{n}.\mathsf{fst}(\mathsf{next}^n(\mathsf{pair}\hat{0}\hat{0}))\\ &= \lambda\hat{n}.\mathsf{fst}(\hat{\mathsf{next}}(\mathsf{pair}\hat{0}\hat{0}))\end{aligned}$$

6.2. **Kombinator punktu stałego.** Jako że siła wyrazu rachunku $\lambda$ jest taka sama jak maszyn Turinga, każda maszyna Turinga może być symulowana za pomocą pewnego wyrażenia w rachunku $\lambda$. W szczególności istnieją wyrażenia w rachunku $\lambda$, które odpowiadają maszynom Turinga, które się nie zatrzymują, jak również istnieją maszyny Turinga obliczające **funkcje rekursywne**.

Jest stosunkowo łatwo znaleźć wyrażenie, którego redukcja się nie zatrzymuje. Przypuśćmy bowiem, że chcemy znaleźć wyrażenie $\mathsf{omega}$ takie, że $\mathsf{omega} \mapsto \mathsf{omega}$. Jako że redukuje się do samego siebie, jego redukcja nigdy się nie skończy. Przepiszmy $\mathsf{omega}$ jako $(\lambda x.e)e'$ tak, aby $\beta$-redukcja mogła być zastosowana do całego wyrażenie $\omega$. Otrzymujemy wówczas:

$$\mathsf{omega} = (\lambda x.e)e' \mapsto [e'/x]e = \mathsf{omega}.$$

Równość $\mathsf{omega} = [e'/x]e = (\lambda x.e)e'$ sugeruje, że $e = e''$ dla pewnego wyrażenia $e''$ takiego, że $[e'/x]e'' = \lambda x.e$ (oraz $[e'/x]x = e'$):

$$\begin{aligned}\mathsf{omega} &= [e'/x]e\\ &= [e'/x](e''x) && \text{wobec } e = e''x\\ &= [e'/x]e''[e'/x]x\\ &= [e'/x]e''e'\\ &= (\lambda x.e)e' && \text{wobec } [e'/x]e'' = \lambda x.e\end{aligned}$$

Wobec $e = e''x$ oraz $[e'/x]e'' = \lambda x.e$ otrzymujemy $[e'/x]e'' = \lambda x.e''x$. Podstawiając $e'' = x$ w $[e'/x]e'' = \lambda x.e''x$ otrzymujemy $e' = \lambda x.xx$. Tym samym $\mathsf{omega}$ może być zdefiniowana jak następuje:

$$\begin{array}{rcll} \mathsf{omega} & = & (\lambda x.e)e' & \\ & = & (\lambda x.e''x)e' & \text{wobec } e = e''x \\ & = & (\lambda x.xx)e' & \text{wobec } e'' = x \\ & = & (\lambda x.xx)(\lambda x.xx) & \text{wobec } e' = \lambda x.xx \end{array}$$

Teraz możemy pokazać, że redukcja $\mathsf{omega}$ zdefiniowana jak wyżej nigdy się nie zatrzyma.

Zobaczymy teraz w jaki sposób możemy definiować funkcje rekursywne w rachunku $\lambda$. Zaczniemy od założenia, że mamy dany **konstruktor funkcji rekursywnej** $\mathsf{fun}fx.e$, który definiuje funkcję rekursywną $f$ o argumencie $x$ i zawartości $e$. Zauważmy, że zawartość $e$ może zawierać odnośniki do $f$. Naszym celem będzie pokazanie, że $\mathsf{fun}fx.e$ jest w istocie **słodzikiem syntaktycznym** w takim sensie, że może zostać zapisany jako już istniejące wyrażenie w rachunku $\lambda$ i tym samym jego dodanie nie zmienia siły wyrazu rachunku $\lambda$.

Jako roboczy przykład rozważmy funkcję silnia $\mathsf{fac}$:

$$\mathsf{fac} = \mathsf{fun}\ f\ n.\mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n(f(\mathsf{pred}\ n))$$

Semantycznie $f$ w zawartości powyższej funkcji oznacza definiowaną właśnie funkcję $\mathsf{fac}$. Na początek mechanicznie tworzymy $\lambda$-abstrakcję $\mathsf{FAC} = \lambda f.\lambda n.e$ z $\mathsf{fac} = \mathsf{fun}fn.e$:

$$\mathsf{FAC} = \lambda f.\lambda n.\mathsf{if}\ \mathsf{eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else}\ \mathsf{mult}\ n(f(\mathsf{pred}\ n))$$

Zauważmy, że $\mathsf{FAC}$ ma zupełnie inną charakterystykę niż $\mathsf{fac}$: podczas gdy $\mathsf{fac}$ bierze liczbę naturalną $n$ i zwraca inną liczbę naturalną, $\mathsf{FAC}$ bierze funkcję $f$ i zwraca inną funkcję (gdyby $\mathsf{fac}$ oraz $\mathsf{FAC}$ mogły mieć typy, $\mathsf{fac}$ miałby typ $\mathsf{nat} \to \mathsf{nat}$, natomiast $\mathsf{FAC}$ miałby typ $(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat})$).

Główny pomysł, na którym opiera się konstrukcja funkcji $\mathsf{FAC}$ polega na tym, że przy danej **częściowej** implementacji $f$ funkcji silnia, $\mathsf{FAC}f$ zwraca **poprawioną** implementację tej samej funkcji. Przypuśćmy bowiem, że $f$ poprawnie oblicza silnię każdej liczby naturalnej nie większej od $n$. Wówczas $\mathsf{FAC}f$ poprawnie obliczy silnię z każdej liczby naturalnej nie większej od $n+1$ – jest to więc istotna poprawa implementacji. Zauważmy też, że $\mathsf{FAC}f$ poprawnie oblicza silnię z zera, niezależnie od $f$. W szczególności nawet wtedy, gdy weźmiemy przekazującą możliwie najmniej informacji funkcja $f = \lambda n.\mathsf{omega}$ (która niczego nie robi, ponieważ nigdy niczego nie zwróci), $\mathsf{FAC}f$ poprawnie obliczy silnię z zera. Tym samym możemy sobie wyobrazić nieskończony ciąg funkcji $\mathsf{fac}_0, \mathsf{fac}_1, \ldots, \mathsf{fac}_i, \ldots$ zaczynający się od $\mathsf{fac}_0 = \mathsf{FAC}\lambda n.\mathsf{omega}$ i kolejno wykorzystuje równość $\mathsf{fac}_{i+1} = \mathsf{FAC}\mathsf{fac}_i$:

$$\begin{array}{lcccl} \mathsf{fac}_0 & = & & & \mathsf{FAC}\lambda n.\mathsf{omega} \\ \mathsf{fac}_1 & = & \mathsf{FAC}\mathsf{fac}_0 & = & \mathsf{FAC}^2\lambda n.\mathsf{omega} \\ \mathsf{fac}_2 & = & \mathsf{FAC}\mathsf{fac}_1 & = & \mathsf{FAC}^3\lambda n.\mathsf{omega} \\ \vdots & & & & \\ \mathsf{fac}_i & = & \mathsf{FAC}\mathsf{fac}_{i-1} & = & \mathsf{FAC}^{i+1}\lambda n.\mathsf{omega} \\ \vdots & & & & \end{array}$$

Zauważmy, że $\mathsf{fac}_i$ poprawnie oblicza silnię z każdej liczby naturalnej nie większej niż $i$. Wobec tego, jeśli $\omega$ oznacza nieskończoną liczbę naturalną (o której, w uproszczeniu, możemy myśleć jak o liczbie naturalnej większej od dowolnej liczby naturalnej), możemy wziąć $\mathsf{fac}_\omega$ jako poprawną implementację funkcji silnia $\mathsf{fac}$, a zatem $\mathsf{fac} = \mathsf{fac}_\omega$.

Kolejna ważna obserwacja, jaką poczynimy, jest następująca: mając daną poprawną implementację funkcji silnia $\mathsf{fac}$, $\mathsf{FAC}\mathsf{fac}$ zwraca nową poprawną implementację funkcji silnia. Innymi słowy, jeśli $\mathsf{fac}$ jest

poprawną implementacją funkcji silnia, to wówczas

$$\lambda n.\mathsf{if\ eq}\ n\ \hat{0}\ \mathsf{then}\ \hat{1}\ \mathsf{else\ mult}\ n(\mathsf{fac}(\mathsf{pred}\ n))$$

jest również poprawną implementacją funkcji silnia. Jako że obydwie funkcje są *de facto* identyczne w takim sensie, że zwracają taki sam wynik po zastosowaniu do dowolnego argumentu, możemy zapisać $\mathsf{fac} = \mathsf{FACfac}$. Jeśli podstawimy $\mathsf{fac}_\omega$ za $\mathsf{fac}$ w równaniu, otrzymamy $\mathsf{fac}_\omega = \mathsf{fac}_{\omega+1}$, co również ma sens, jako że $\omega \leq \omega + 1$ wobec definicji dodawania oraz $\omega + 1 \leq \omega$ wobec definicji liczby $\omega$.

Wydaje się zatem, że $\mathsf{FAC}$ zawiera wszelką informację potrzebną do obliczenia $\mathsf{fac} = \mathsf{fac}_\omega = \mathsf{FACfac}$. Istotnie, okazuje się, że $\mathsf{fac}$ otrzymujemy przez zastosowanie **kombinatora punktu stałego** $\mathsf{fix}$ do $\mathsf{FAC}$, czyli że $\mathsf{fac} = \mathsf{fixFAC}$, gdzie $\mathsf{fix}$ jest zdefiniowany następująco:

$$\mathsf{fix} = \lambda F.(\lambda f.F(\lambda x.fffx))(\lambda f.F(\lambda x.fffx))$$

Zakładamy tutaj użycie stratego wywołania według wartości. Dla strategii wywołania według nazwy upraszczamy $\lambda x.fffx$ do $ff$ i stosujemy następujący kombinator punktu stałego $\mathsf{fix}_{CBN}$:

$$\mathsf{fix}_{CBN} = \lambda F.(\lambda f.F(ff))(\lambda f.F(ff))$$

Aby zrozumieć jak działa kombinator punktu stałego $\mathsf{fix}$ musimy najpierw zrozumieć pojęcie **punktu stałego**. Punktem stałym funkcji $f$ będziemy nazywali wartość $v$ taką, że $v = f(v)$. Na przykład punktem stałym funkcji $f(x) = 2 - x$ jest 1, ponieważ $f(1) = 1$. Tak jak wskazuje sama nazwa, $\mathsf{fix}$ bierze funkcję $F$ (która sama zamienia funkcję $f$ na nową funkcję $f'$) i zwraca jej punkt stały. Innymi słowy, $\mathsf{fix}F$ jest punktem stałym $F$:

$$\mathsf{fix}F = F(\mathsf{fix}F)$$

Nieformalnie wyrażenie po lewej stronie zamienia się w wyrażenie po prawej stronie poprzez ciąg następujących kroków, w których używamy symbolu $\approx$ do podkreślenia braku rygoru formalnego, który nie jest dostatecznie uzasadniony samą tylko $\beta$-redukcją:

$$\begin{array}{rcll}
\mathsf{fix}F & \mapsto & g\ g & \text{gdzie } g = \lambda f.F(\lambda x.fffx) \\
& = & (\lambda f.F(\lambda x.fffx))g & \\
& \mapsto & F(\lambda x.ggx) & \\
& \approx & F(g\ g) & \text{ponieważ } \lambda x.ggx \approx gg \\
& \approx & F(\mathsf{fix}F) & \text{ponieważ } \mathsf{fix}F \mapsto gg
\end{array}$$

Możemy teraz wyjaśnić, dlaczego $\mathsf{fixFAC}$ zwraca implementację funkcji silnia. Zgodnie z naturą kombinatora punktu stałego $\mathsf{fix}$ mamy

$$\mathsf{fixFAC} = \mathsf{FAC}(\mathsf{fixFAC}).$$

Innymi słowy, $\mathsf{fixFAC}$ zwraca funkcję $f$ spełniającą warunek $f = \mathsf{FAC}f$, a więc dokładnie ten warunek, jaki musi spełniać $\mathsf{fac}$. Tym samym możemy traktować $\mathsf{fixFAC}$ równoważnie z $\mathsf{fac}$.

Inny sposób wyjaśnienia zachowania $\mathsf{fixFAC}$ jest następujący: przypuśćmy, że chcemy obliczyć $\mathsf{fac}n$ dla dowolnie dużej liczby naturalnej $n$. Ponieważ $\mathsf{fixFAC}$ jest punktem stałym $\mathsf{FAC}$, otrzymujemy następujące równanie:

$$\begin{array}{rcccl}
\mathsf{fixFAC} & = & & & \mathsf{FAC}(\mathsf{fixFAC}) \\
& = & \mathsf{FAC}(\mathsf{FAC}(\mathsf{fixFAC})) & = & \mathsf{FAC}^2(\mathsf{fixFAC}) \\
& = & \mathsf{FAC}^2(\mathsf{FAC}(\mathsf{fixFAC})) & = & \mathsf{FAC}^3(\mathsf{fixFAC}) \\
\vdots & & & & \\
& = & \mathsf{FAC}^n(\mathsf{FAC}(\mathsf{fixFAC})) & = & \mathsf{FAC}^{n+1}(\mathsf{fixFAC})
\end{array}$$

Kluczowa jest tu następująca obserwacja: $\mathsf{FAC}^{n+1}(\mathsf{fixFAC})$ poprawnie oblicza silnię dowolnej liczby naturalnej nie większej od $n$ **niezależnie** od tego, co robi $\mathsf{fixFAC}$. Jako że $\mathsf{fixFAC} = \mathsf{FAC}^{n+1}(\mathsf{fixFAC})$, widzimy

że fixFAC poprawnie oblicza silnię **dowolnej** liczby naturalnej. Innymi słowy, fixFAC robi dokładnie to samo, co fac.

Reasumując, chcąc zakodować funkcję rekursywną $\mathsf{fun}fx.e$ w rachunku $\lambda$, zaczynamy od $\lambda$-abstrakcji $F = \lambda f.\lambda x.e$, a następnie $\mathsf{fix}F$ **automagicznie** zwraca funkcję zachowującą się tak samo, jak $\mathsf{fun}fx.e$.

6.3. **Kody de Bruijna.** Jako że $\lambda$-abstrakcje mają w założeniu oznaczać funkcje matematyczne z formalnymi argumentami, nazwy zmiennych powinny być integralną częścią syntaksu rachunku $\lambda$. Na przykład, wydaje się nieuniknione, aby w użyć formalny argument, powiedzmy $x$, przy definiowaniu funkcji identycznościowej. Z drugiej strony wybór konkretnej nazwy dla zmiennej nie ma wpływu na znaczenie $\lambda$-abstrakcji. Na przykład $\lambda x.x$ oraz $\lambda y.y$ oznaczają taką samą funkcję identycznościową, aczkolwiek używają różnych nazw argumentów. W ogólności $\alpha$-konwersja pozwala nam na przepisanie dowolnej $\lambda$-abstrakcji do postaci nowej $\lambda - abstrakcji$ z inną nazwą zmiennej związanej. Obserwacja ta pozwala przypuszczać, że istnieje sposób reprezentowania zmiennych w rachunku $\lambda$ bez użycia konkretnych nazw zmiennych. Przykładem takiej beznazwowej interpretacji zmiennych są **kody de Bruijna.**

Główny pomysł wykorzystywany przez kody de Bruijna polega na przypisaniu każdej zmiennej liczby całkowitej, zwanej kodem de Bruijna, zamiast nazwy. Kody de Bruijna mogą być w szczególności ujemne, ale zajmiemy się tu tylko dodatnimi kodami. Z grubsza biorąc kod de Bruijna zlicza **$\lambda$-wiązania** takie jak $\lambda x$, $\lambda y$ oraz $\lambda z$ leżące pomiędzy daną zmienną a odpowiadającym jej jednoznacznie wyznaczonym $\lambda$-wiązaniem. Na przykład, $x$ w zawartości $\lambda x.x$ ma przypisany kod de Bruijna 0, ponieważ pomiędzy $x$ oraz $\lambda x$ nie występuje żadne $\lambda$-wiązanie. Z drugiej strony $x$ w zawartości $\lambda x.\lambda y.xy$ ma kod de Bruijna 1, ponieważ pomiędzy $x$ oraz $\lambda x$ występuje jeszcze $\lambda$-wiązanie $\lambda y$. Tym samym kod de Bruijna zmiennej określa jej względne położenie wobec odpowiadającego jej $\lambda$-wiązania. W konsekwencji widać, że taka sama zmienna może mieć przypisane różne wartości kodów de Bruijna w zależności od swego położenia.

Jako że wszystkie zmienne są teraz reprezentowane przez liczby całkowite , nie ma potrzeby wprowadzania *explicite* zmiennych w $\lambda$-abstrakcjach. W istocie jest to niemożliwe, ponieważ tej samej zmiennej mogą zostać przyporządkowane różne kody de Bruijna. Tym samym wyrażenia z kodami de Bruijna, albo – jak je będziemy nazywać – **wyrażenia de Bruijna,** są zdefiniowane indukcyjnie jak następuje:

$$\begin{array}{rrcl} \text{wyrażenie de Bruijna} & M & ::= & n|\lambda.M|M\ M \\ \text{kod de Bruijna} & n & ::= & 0|1|2|\ldots \end{array}$$

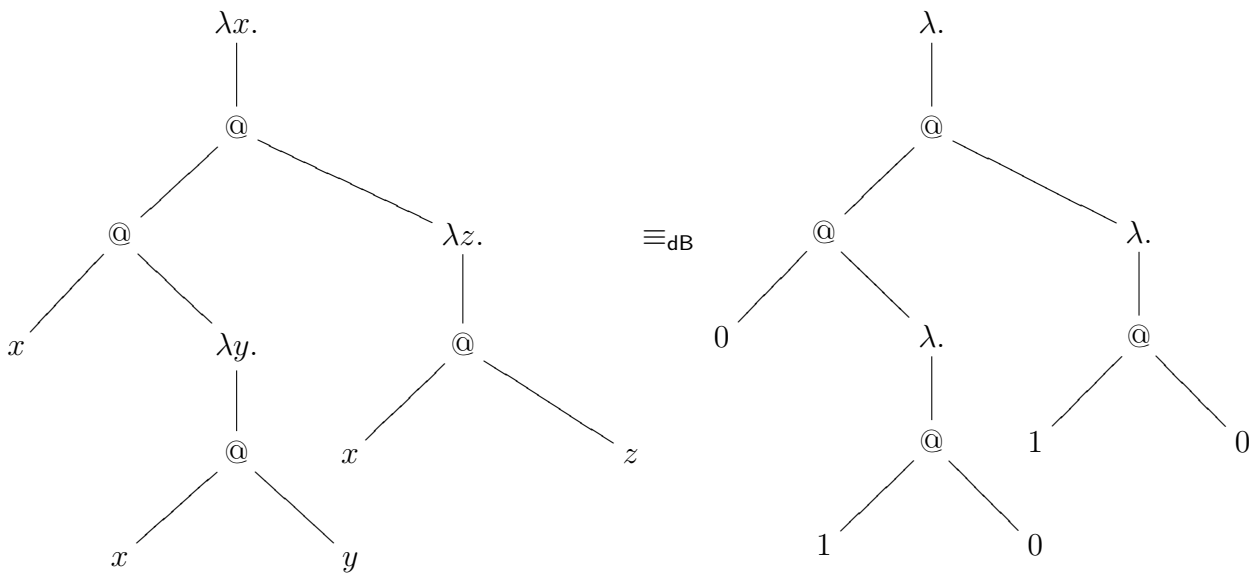
Dla wyrażeń de Bruijna będziemy używali metazmiennych $M$ i $N$, zaś dla kodów de Bruijna metazmiennych $m$, $n$ oraz $i$.

Będziemy pisali $e \equiv_{\mathsf{dB}} M$ dla oznaczenia, że zwykłe wyrażenie $e$ jest konwertowane do wyrażenia de Bruijna $M$. Czasami wystarczy dosłownie policzyć liczbę $\lambda$-wiązań pomiędzy poszczególnymi zmiennymi i odpowiadającymi im $\lambda$-wiązaniami, tak jak w następujących przykładach:

$$\begin{array}{rcl} \lambda x.x & \equiv_{\mathsf{dB}} & \lambda.0 \\ \lambda x.\lambda y.xy & \equiv_{\mathsf{dB}} & \lambda.\lambda.1\ 0 \\ \lambda x.x(\lambda y.xy) & \equiv_{\mathsf{dB}} & \lambda.0(\lambda.1\ 0) \end{array}$$

W ogólności, wszelako, konwertowanie zwykłego wyrażenia $e$ do wyrażenia de Bruijna wymaga jednak zinterpretowania $e$ jako obiektu o strukturze pewnego drzewka, nie zaś obiektu o strukturze liniowej. Jako przykład rozważmy wyrażenie $\lambda x.(x(\lambda y.xy))(\lambda z.xz)$. Licząc bezpośrednio $\lambda$-wiązania otrzymujemy wyrażenie de Bruijna $\lambda.(0(\lambda.1\ 0))(\lambda.2\ 0)$, w którym ostatnie wystąpienie zmiennej $x$ ma przypisany błędny kod de Bruijna 2, wynikający ze zliczenia $\lambda y$ oraz $\lambda z$. Intuicyjnie czujemy jednak, że ostatnie wystąpienie zmiennej $x$ musi mieć przypisany kod de Bruijna 1, ponieważ odpowiadające jej $\lambda$-wiązanie może być zlokalizowane z pominięciem $\lambda$-wiązania $\lambda y$. Tym samym prawidłowa metoda konwertowania wyrażenia $e$ do wyrażenia de Bruijna polega na zliczeniu $\lambda$-wiązań występujących pomiędzy daną

zmienną a odpowiadającym jej $\lambda$-wiązaniu występującym w reprezentacji $e$ jako drzewa. Na przykład $\lambda x.(x(\lambda y.xy))(\lambda z.xz) \equiv_{\mathsf{dB}} \lambda.(0(\lambda.1\ 0))(\lambda.1\ 0)$, zgodnie z poniższą ilustracją:




6.4. **Podstawianie.** Celem wykorzystania kodów de Bruijna do implementowania semantyki operacyjnej rachunku $\lambda$, będziemy potrzebować definicję podstawiania dla wyrażeń de Bruijna, z której, z kolei, będziemy w stanie wywieść definicję $\beta$-redukcji. Chcemy mianowicie zdefiniować $\sigma_0(M, N)$ tak, aby zachodził następujący związek:

$$\begin{array}{ccc} (\lambda x.e)e' & \mapsto & [e'/x]e \\ \equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\ (\lambda.M)N & \mapsto & \sigma_0(M, N) \end{array}$$

Innymi słowy, stosując $\lambda.M$ do $N$, lub podstawiając $N$ za zmienną związaną w $\lambda.M$, otrzymyjemy $\sigma_0(M, N)$ (znaczenie indeksu 0 w $\sigma_0(M, N)$ zostanie wyjaśnione później).

Zamiast zaczynać od kompletnej definicji $\sigma_0(M, N)$, zaczniemy od kilku prostych przykładów, które wykorzystamy do poprawienia pierwotnej naiwnej wersji definicji. Rozważmy następujący przykład, w którym redeks jest podkreślony:

$$\begin{array}{ccc} \lambda x.\lambda y.\underline{(\lambda z.xyz)(\lambda w.w)} & \mapsto & \lambda x.\lambda y.xy(\lambda w.w) \\ \equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\ \lambda.\lambda.\underline{(\lambda.2\ 1\ 0)(\lambda.0)} & \mapsto & \lambda.\lambda.10(\lambda.0) \end{array}$$

Zauważmy, że 0, które odpowiada zmiennej związanej $z$ w $\lambda$-abstrakcji $\lambda z.x\ y\ z$, jest zamieniona przez argument $\lambda.0$. Pozostałe kody 1 oraz 2 są zmniejszone o 1, ponieważ $\lambda$-wiązanie $\lambda z$ znika. Te dwie obserwacje prowadzą nas do następującej częściowej definicji $\sigma_0(M, N)$:

$$\begin{array}{rcll} \sigma_0(M_1M_2, N) & = & \sigma_0(M_1, N)\sigma_0(M_2, N) & \\ \sigma_0(0, N) & = & N & \\ \sigma_0(m, N) & = & m - 1 & \text{jeśli } m > 0. \end{array}$$

Aby zobaczyć jak definiujemy pozostały przypadek $\sigma_0(\lambda.M, N)$, rozważmy następujący przykład, w którym redeks jest podkreślony:

$$\begin{array}{ccc} \lambda x.\lambda y.\underline{(\lambda z.(\lambda u.x\ y\ z\ u))(\lambda w.w)} & \mapsto & \lambda x.\lambda y.(\lambda u.x\ y(\lambda w.w)u) \\ \equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\ \lambda.\lambda.\underline{(\lambda.(\lambda.3\ 2\ 1\ 0))(\lambda.0)} & \mapsto & \lambda.\lambda.(\lambda.2\ 1(\lambda.0)0) \end{array}$$

Zauważmy, że – inaczej niż w pierwszym przykładzie – 0 pozostaje nietknięte, ponieważ odpowiada do zmiennej związanej $u$ w $\lambda u.x\ y\ z\ u$, podczas gdy 1 odpowiada $z$ i tym samym jest zamienione przez $\lambda.0$. Powodem dla którego 1 jest teraz zastąpione przez $\lambda.0$ jest to, że kod de Bruijna $m$ poza $\lambda.M$ wskazuje na taką samą zmienną jak $m+1$ wewnątrz $\lambda.M$, to znaczy wewnątrz $M$. Ta obserwacja prowadzi nas do równania $\sigma_0(\lambda.M, N) = \lambda.\sigma_1(M, N)$, gdzie $\sigma_1(M, N)$ jest zdefiniowane jak następuje:

$$\begin{array}{rcll} \sigma_1(M_1M_2, N) & = & \sigma_1(M_1, N)\sigma_1(M_2, N) & \\ \sigma_1(0, N) & = & 0 & \\ \sigma_1(1, N) & = & N & \\ \sigma_1(m, N) & = & m-1 & \text{jeśli } m > 1. \end{array}$$

W powyższych dwóch przykładach zobaczyliśmy, że indeks $n$ w $\sigma_n(M, N)$ pełni rolę wskaźnika "granicznego": $m$ pozostaje bez zmian, jeśli $m < n$, jest zastępowane przez $N$, jeśli $m = n$ i jest zmniejszane o 1, jeśli $m > n$. Alternatywnie możemy spojrzeć na $n$ w $\sigma_n(M, N)$ jako na liczbę $\lambda$-wiązań otaczających $M$ tak, jak ilustruje poniższy wzór:

$$\sigma_0(\underbrace{\lambda.\lambda.\ldots\lambda.}_{n} M, N) = \underbrace{\lambda.\lambda.\ldots\lambda.}_{n} \sigma_n(M, N)$$

Poniższa definicja $\sigma_n(M, N)$ wykorzystuje $n$ jako wskaźnik graniczny i uogólnia związek pomiędzy $\sigma_0(\lambda.M, N)$ oraz $\lambda.\sigma_1(M, N)$:

$$\begin{array}{rcll} \sigma_n(M_1M_2, N) & = & \sigma_n(M_1, N)\sigma_n(M_2, N) & \\ \sigma_n(\lambda.M, N) & = & \lambda.\sigma_{n+1}(M, N) & \\ \sigma_n(m, N) & = & m & \text{jeśli } m < n, \\ \sigma_n(n, N) & = & N & \\ \sigma_n(m, N) & = & m-1 & \text{jeśli } m > n. \end{array}$$

Poniższy przykład łączy dwa podane powyżej:

$$\begin{array}{ccc} \lambda x.\lambda y.\underline{(\lambda z.(\lambda u.x\ y\ z\ u)(x\ y\ z))(\lambda w.w)} & \mapsto & \lambda x.\lambda y.(\lambda u.x\ y(\lambda w.w)u)(xy(\lambda w.w)) \\ \equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\ \lambda.\lambda.\underline{(\lambda.(\lambda.3\ 2\ 1\ 0)(2\ 1\ 0))(\lambda.0)} & \mapsto & \lambda.\lambda.(\lambda.2\ 1(\lambda.0)0)(1\ 0(\lambda.0)) \end{array}$$

Dzięki użyciu kodów de Bruijna możemy wyeliminować użycie $\alpha$-konwersji, ponieważ nigdy nie wystąpi konflikt pomiędzy zmiennymi związanymi. Mówiąc krótko, nie ma potrzeby zmieniania nazw zmiennych związanych, albowiem zmienne związane tak czy inaczej nie mają już nazw.

6.5. **Przesunięcia.** Jakkolwiek podana przed chwilą definicja $\sigma_n(M, N)$ będzie poprawnie funkcjonowała, gdy $N$ jest zamknięta, może jednak nie działać gdy $N$ będzie reprezentowało wyrażenie ze zmiennymi wolnymi. Mówiąc dokładniej, równość $\sigma_n(n, N) = N$ przestaje być prawdziwa gdy $n > 0$ i $N$ reprezentuje wyrażenie ze zmiennymi wolnymi. Rozważmy bowiem następujący przykład, w którym redeks został

podkreślony:

$$\begin{array}{ccc} \lambda x.\lambda y.\underline{(\lambda z.(\lambda u.z)z)(\lambda w.x\ y\ w)} & \mapsto & \lambda x.\lambda y.(\lambda u.\lambda w.x\ y\ w)(\lambda w.x\ y\ w) \\ \equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\ \lambda.\lambda.\underline{(\lambda.(\lambda.1)0)(\lambda.2\ 1\ 0)} & \mapsto & \lambda.\lambda.(\lambda.\lambda.3\ 2\ 0)(\lambda.2\ 1\ 0) \end{array}$$

Poprzednia definicja $\sigma_n(M,N)$ daje niewłaściwy rezultat, ponieważ $\sigma_1(1,\lambda.2\ 1\ 0)$ daje $\lambda.2\ 1\ 0$ zamiast $\lambda.3\ 2\ 0$:

$$\begin{array}{rcl} \underline{(\lambda.(\lambda.1)0)(\lambda.2\ 1\ 0)} & \mapsto & \sigma_0((\lambda.1)0,\lambda.2\ 1\ 0) \\ & = & \sigma_0(\lambda.1,\lambda.2\ 1\ 0)\sigma_0(0,\lambda.2\ 1\ 0) \\ & = & (\lambda.\sigma_1(1,\lambda.2\ 1\ 0))\sigma_0(0,\lambda.2\ 1\ 0) \\ & = & (\lambda.\lambda.2\ 1\ 0)(\lambda.2\ 1\ 0) \\ & \neq & (\lambda.\lambda.3\ 2\ 0)(\lambda.2\ 1\ 0) \end{array}$$

Aby zobaczyć, dlaczego w ogólności nie jest prawdziwy wzór $\sigma_n(n,N)=N$, przypomnjmy, że indeks $n$ w $\sigma_n(n,N)$ oznacza liczbę $\lambda$-wiązań otaczających kod de Bruijna $n$:

$$\sigma_0(\underbrace{\lambda.\lambda.\ldots\lambda.}_{n},N)=\underbrace{\lambda.\lambda.\ldots\lambda.}_{n}\sigma_n(n,N)$$

Wobec tego wszystkie kody de Bruijna w $N$ odpowiadające zmiennym wolnym muszą być przesunięte o $n$ po podstawieniu tak, aby poprawnie pomijały $n$ $\lambda$-wiązań otaczających kod de Bruijna $n$. Na przykład:

$$\sigma_0(\underbrace{\lambda.\lambda.\ldots\lambda.}_{n},m)=\underbrace{\lambda.\lambda.\ldots\lambda.}_{n}\sigma_n(n,m)=\underbrace{\lambda.\lambda.\ldots\lambda.}_{n}m+n$$

(tutaj przez $m+n$ rozumiemy pojedynczy kod de Bruijna dodający $m$ do $n$, nie zaś złożone wyrażenie de Bruijna składające się z $m$, $n$ oraz $+$).

Oznaczmy przez $\tau^n(N)$ przesunięcie o $n$ wszystkich kodów de Bruijna w $N$ odpowiadających zmiennym wolnym. Możemy teraz zastosować wzór $\sigma_n(n,N)=\tau^n(N)$ w miejsce $\sigma_n(n,N)=N$. Częściowa definicja $\tau^n(N)$ dana jest przez warunki:

$$\begin{array}{rcl} \tau^n(N_1N_2) & = & \tau^n(N_1)\tau^n(N_2) \\ \tau^n(m) & = & m+n \end{array}$$

Pozostały przypadek $\tau^n(\lambda.N)$ nie może być, niestety, zdefiniowany indukcyjnie w oparciu o $\tau^n(N)$ na przykład jako $\tau^n(\lambda.N)=\lambda.\tau^n(N)$. Powodem tego jest fakt, że wewnątrz $N$ nie każdy kod de Bruijna odpowiada zmiennej wolnej: 0 odpowiada $\lambda$-wiązanie w $\lambda.N$ i tym samym musi pozostać niezmienione.

Powyższa obserwacja sugeruje konieczność wprowadzenia jeszcze jednego indeksu "granicznego" pozwalającego rozpoznawać, czy dany kod de Bruijna odpowiada zmiennej wolnej, czy nie. Na przykład, jeśli wskaźnik graniczny dla $\lambda.N$ równy jest 0, powinien zwiększyć się o 1 wewnątrz $N$. Tym samym powinniśmy wprowadzić ogólniejszą notację $\tau_i^n(N)$ dla przesunięcia o $n$ wszystkich kodów de Bruijna wewnątrz $N$ odpowiadających zmiennym wolnym, gdzie kod de Bruijna $m$ w $N$ dla $m<i$ nie liczy się jako zmienna wolna. Formalnie definicja $\tau_i^n(N)$ będzie następująca:

$$\begin{array}{rcll} \tau_i^n(N_1N_2) & = & \tau_i^n(N_1)\tau_i^n(N_2) & \\ \tau_i^n(\lambda.N) & = & \lambda.\tau_{i+1}^n(N) & \\ \tau_i^n(m) & = & m+n & \text{jeśli } m\geq i \\ \tau_i^n(m) & = & m & \text{jeśli } m<i. \end{array}$$

Tym samym kompletna definicja $\sigma_n(M,N)$ będzie następująca:

$$
\begin{array}{rcll}
\sigma_n(M_1M_2, N) & = & \sigma_n(M_1, N)\sigma_n(M_2, N) & \\
\sigma_n(\lambda.M, N) & = & \lambda.\sigma_{n+1}(M, N) & \\
\sigma_n(m, N) & = & m & \text{jeśli } m < n, \\
\sigma_n(n, N) & = & \tau_0^n(N) & \\
\sigma_n(m, N) & = & m - 1 & \text{jeśli } m > n.
\end{array}
$$

Wracając do omawianego wcześniej przykładu widzimy, że teraz $(\lambda.(\lambda.1)0)(\lambda.2\ 1\ 0)$ zredukuje się poprawnie:

$$
\begin{array}{rcl}
\underline{(\lambda.(\lambda.1)0)(\lambda.2\ 1\ 0)} & \mapsto & \sigma_0((\lambda.1)0, \lambda.2\ 1\ 0) \\
& = & \sigma_0(\lambda.1, \lambda.2\ 1\ 0)\sigma_0(0, \lambda.2\ 1\ 0) \\
& = & (\lambda.\sigma_1(1, \lambda.2\ 1\ 0))\sigma_0(0, \lambda.2\ 1\ 0) \\
& = & (\lambda.\tau_0^1(1, \lambda.2\ 1\ 0))\tau_0^0(0, \lambda.2\ 1\ 0) \\
& = & (\lambda.\lambda.\tau_1^1(2\ 1\ 0))\lambda.\tau_1^0(2\ 1\ 0) \\
& = & (\lambda.\lambda.(2+1)(1+1)0)(\lambda.(2+0)(1+0)0) \\
& = & (\lambda.\lambda.3\ 2\ 0)(\lambda.2\ 1\ 0)
\end{array}
$$

Za każdym razem, gdy konwertujemy zwykłe wyrażenie $e$ ze zmiennymi wolnymi $x_0, x_1, \ldots, x_n$ do wyrażenia de Bruijna, możemy zamiast tego skonwertować $\lambda x_0.\lambda x_1.\ldots.\lambda x_n.e$, co efektywnie przypisze kody de Bruijna $0, 1, \ldots, n$ zmiennym $x_0, x_1, \ldots, x_n$, odpowiednio. Wówczas możemy myśleć o redukowaniu $e$ jak o redukowaniu $\lambda x_0.\lambda x_1.\ldots.\lambda x_n.e$, gdzie $n$ $\lambda$-wiązań jest ignorowanych. Tym sposobem możemy wykorzystywać kody de Bruijna do redukowania wyrażeń ze zmiennymi wolnymi.