Rachunek Lambda i Języki Programowania

Małgorzata Biernacka

Instytut Informatyki UWr

Wykłady 1-2

Plan

Plan

Plan

Zastosowania rachunku lambda

- język symboliczny do formalizacji matematyki (Church)
- model obliczeniowy w teorii obliczalności, równoważny maszynom Turinga
- prototypowy język programowania – z typami; bez typów
- modelowy język programowania (semantyka, implementacja)
- język symboliczny do zapisu dowodów w logice, do formalizacji ekstrakcji programów z dowodów (izomorfizm Curry-Howarda)

Literatura

- H. P. Barendregt "The Lambda Calculus: Its Syntax and Semantics" (1984)
- Ch. Hankin "Lambda Calculi: A guide for computer scientists" (1994)
- P. Urzyczyn: materiały do wykładu "Rachunek lambda" (www.mimuw.edu.pl/~urzy)

Składnia

- Zakładamy, że dany jest nieskończony, przeliczalny zbiór V nazw zmiennych.
- Termami rachunku lambda nazywamy elementy najmniejszego zbioru spełniającego warunki:
  - każda zmienna jest termem
  - jeśli x jest zmienną i t jest termem, to $(\lambda x.t)$ jest termem
  - jeśli t i s są termami, to $(t\,s)$ jest termem
- Term postaci $(\lambda x.t)$ nazywamy lambda abstrakcją, a term postaci $(t\,s)$ aplikacją.
- Nawiasy można opuszczać wtedy, gdy nie powoduje to dwuznaczności, przyjmując zasady:
  - abstrakcja wiąże do prawej, np. $\lambda x.\lambda y.t \equiv \lambda x.(\lambda y.t) \equiv \lambda xy.t$
  - aplikacja wiąże do lewej, np. $r\,s\,t \equiv (r\,s)\,t$
- Abstrakcja $\lambda x.t$ reprezentuje funkcję `fun x -> t`; term t nazywamy ciałem abstrakcji (funkcji). Aplikacja $r\,s$ reprezentuje aplikację termu (funkcji) r do argumentu s.

Składnia, c.d.

- W skrócie zbiór termów zapisujemy używając notacji BNF:

$$t := x \mid \lambda x.t \mid t\ t$$

- Oznaczmy przez Λ zbiór termów rachunku lambda. Indukcyjną definicję tego zbioru możemy również zapisać używając reguł dedukcji:

$$\frac{x \in V}{x \in \Lambda}$$

$$\frac{x \in V \quad t \in \Lambda}{\lambda x.t \in \Lambda}$$

$$\frac{t \in \Lambda \quad r \in \Lambda}{t\ r \in \Lambda}$$

Zmienne wolne i związane

- Lambda abstrakcja $\lambda x.t$ wiąże zmienną x w termie t.
- Zmienne występujące w termie, które nie są związane przez żadną lambda abstrakcję, nazywamy zmiennymi wolnymi.
- Zbiór zmiennych wolnych $FV(\cdot)$ w danym termie definiujemy formalnie:

$$FV(x) = \{x\}$$

$$FV(\lambda x.t) = FV(t) \setminus \{x\}$$

$$FV(r\,s) = FV(r) \cup FV(s)$$

- Term t nazywamy zamkniętym (również: kombinatorem), jeśli $FV(t) = \emptyset$.
- Pojęcie zmiennej wolnej/związanej jest zależne od konkretnego wystąpienia tej zmiennej w termie, np. w termie $x\,(\lambda xy.x\,y)$ zmienna x występuje jako wolna i jako związana.

α-równoważność

- Kiedy dwa termy są równe (syntaktycznie)?
- Intuicyjnie, $\lambda x.x$ oznacza "tę samą funkcję"(a więc "ten sam term") co $\lambda y.y$.
- Przyjmuje się, że dwa termy są syntaktycznie równe, jeśli różnią się tylko nazwami zmiennych związanych.
- Jeśli dwa termy są równe, to jeden można otrzymać z drugiego przez przemianowanie zmiennych związanych (tzw. α-przemianowanie).
- Piszemy $t \equiv_\alpha s$ dla oznaczenia równości termów t i s modulo α-przemianowanie.
- Relacja α-równoważności jest relacją równoważności.
- Odtąd będziemy utożsamiać dwa termy α-równoważne, tj. będziemy operować na klasach abstrakcji relacji α-równoważności.

Konwencja Barendregta

(Założenie o zmiennych)

W dowolnym kontekście występowania termów rachunku lambda zakładamy, że żadna zmienna nie może występować jednocześnie jako zmienna wolna i zmienna związana. W razie konieczności dokonujemy α-przemianowania zmiennych związanych.

- Ćwiczenie: napisz funkcję weryfikującą, czy powyższa zasada jest zachowana dla danego termu, i w razie potrzeby sprowadzająca term do pożądanej postaci (zakładamy istnienie funkcji dostarczającej "świeżych" nazw zmiennych).

Podstawienie

- Obliczenia w rachunku lambda opierają sie na regule β-konwersji, która z kolei polega na metaoperacji podstawienia
- $s\ [t/x]$ oznacza term otrzymany przez zastąpienie wszystkich wolnych wystąpień zmiennej x w termie s przez term t
- Formalnie:

$$x\ [s/y] = x \qquad \text{jeśli } x \neq y$$

$$x\ [s/x] = s$$

$$(\lambda x.t)\ [s/y] = \lambda x.(t\ [s/y])$$

$$(r\ t)\ [s/x] = r\ [s/x]\ t\ [s/x]$$

- Gdybyśmy nie przestrzegali konwencji Barendregta, mogłoby dojść do błędnego podstawienia i związania zmiennej wolnej (variable capture), np. $(\lambda y.x\,y)\ [y/x]$ nie powinno być równe $\lambda y.y\,y$!

Podstawienie, c.d.

- W definicji podstawienia, w przypadku abstrakcji zapewnienie spełnienia konwencji można osiągnąć następująco:

$$(\lambda x.t)\ [s/y] = \begin{cases} \lambda x.(t\ [s/y]) & \text{jeśli } x \neq y \wedge x \notin FV(s) \\ \lambda x'.(t\ [x'/x])\ [s/y] & \text{w przeciwnym razie} \\ & \text{(x' - "świeża" zmienna)} \end{cases}$$

- "Świeża" zmienna to zmienna niewystępująca w dotychczas używanych termach

Lemat o podstawieniu

Lemat

Dla wszystkich termów s, t, u i dla wszystkich zmiennych x, y takich, że $x \neq y$ i $x \notin FV(u)$,

$$(s\ [t/x])\ [u/y] = (s\ [u/y])\ [(t\ [u/y])/x]$$

Dowód.

Indukcja po strukturze termu s. □

Warianty składni

- W definicji termu używaliśmy nazw zmiennych pochodzących z bliżej nieokreślonego, przeliczalnego zbioru V.
- W szczególności, jedynymi relacjami, jakie wprowadziliśmy na zbiorze V, są: relacja równości i jej dopełnienie.
- Można rozważać bardziej konkretne zbiory nazw zmiennych, które pozwalają na uproszczenie definicji, a w szczególności zapewnienie zachodzenia konwencji Barendregta i uniknięcie potrzeby generowania "świeżych" zmiennych.

Indeksy de Bruijna

- Przyjmujemy $V = \mathbb{N}$.
- Zbiór termów definiujemy następująco:

$$t := n \mid \lambda t \mid t\ t,$$

  gdzie $n \in \mathbb{N}$.
- Każde wystąpienie zmiennej n w termie oznacza odwołanie do n-tej obejmującej tę zmienną abstrakcji, licząc od środka termu
- Np. $\lambda 0$ odpowiada $\lambda x.x$, $\lambda\lambda 1$ odpowiada $\lambda x.\lambda y.x$.
- Notacja z indeksami de Bruijna jest najwygodniejsza w przypadku rozważania termów zamkniętych.

Indeksy de Bruijna, c.d.

- Znaczenie indeksom nadajemy przez odpowiednie zdefiniowanie operacji podstawienia:

$$(r\ s)\ [t/n] = (r\ [t/n])\ (s\ [t/n])$$

$$(\lambda r)\ [t/n] = \lambda(r\ [t/n+1])$$

$$m\ [t/n] = \begin{cases} m-1 & n > m \\ u_0^n(t) & n = m \\ m & n < m \end{cases}$$

$$u_i^n(r\ s) = (u_i^n(r))\ (u_i^n(s))$$

$$u_i^n(\lambda r) = \lambda(u_{i+1}^n(r))$$

$$u_i^n(m) = \begin{cases} m+n-1 & m \geq i+1 \\ m & m \leq i \end{cases}$$

Indeksy de Bruijna, c.d.

- W przypadku termów zamkniętych definicja podstawienia jest o wiele prostsza:

$$m\,[t/n] = \begin{cases} m & n < m \\ t & n = m \end{cases}$$

$$(\lambda r)\,[t/n] = \lambda(r\,[t/n+1])$$

$$(r\,s)\,[t/n] = (r\,[t/n])\,(s\,[t/n])$$

- t jest zamknięty
- zawsze $m \geq n$

Warianty składni - poziomy de Bruijna

- Przyjmując tę samą definicję termów, jak poprzednio:

$$t := n \mid \lambda t \mid t\ t,$$

  możemy określić wystąpienie zmiennej n odwrotnie niż poprzednio, wiążac je z n-tą lambda abstrakcją obejmującą dane wystąpienie zmiennej n, ale licząc od zewnątrz.
- Np. $\lambda\lambda 0$ oznacza $\lambda xy.x$.
- Plusy: ta sama zmienna w każdym miejscu jej wystąpienie ma ten sam "indeks" (przeciwnie do indeksów de Bruijna).
- Minusy: w przypadku podstawienia, wszystkie zmienne trzeba odpowiednio przenumerować.

Notacja mieszana

- W przypadku termów otwartych, wygodnie jest używać notacji, w której zmienne związane są reprezentowane jako indeksy de Bruijna, a zmienne wolne jako zmienne nazwane (abstrakcyjne).
- Zbiór termów definiujemy następująco:

$$t := n \mid x \mid \lambda t \mid t\ t.$$

- Więcej na ten temat np.: Xavier Leroy "A locally nameless solution to the POPLmark challenge".

Teoria $\lambda\beta$

- Rachunek lambda można przedstawić jako teorię, czyli zbiór formuł zamknięty ze względu na pewne reguły wnioskowania.
- Formułami są równości między termami $s = t$.
- System wnioskowania definiują reguły:

$$\frac{}{s = s} \qquad \frac{s = u}{u = s} \qquad \frac{r = s \quad s = u}{r = u}$$

$$\frac{}{(\lambda x.t)\, s = t\ [s/x]} \qquad \frac{s = r}{s\ t = r\ t}$$

$$\frac{s = t}{r\ s = r\ t} \qquad \frac{s = t}{\lambda x.s = \lambda x.t}$$

- Piszemy $\lambda\beta \vdash t = s$, jeśli istnieje dowód formuły $t = s$ w powyższym systemie dowodzenia.
- Dowodem nazywamy skończone drzewo wyprowadzenia danej formuły.

Spójność i rozstrzygalność teorii $\lambda\beta$

Z punktu widzenia zastosowań rachunku lambda, następujące pytania będą nas interesować:

- czy $\lambda\beta$ jest spójna, tj. czy istnieją termy zamknięte r, s takie, że formuła $r = s$ nie jest wyprowadzalna w $\lambda\beta$?
- czy $\lambda\beta$ jest zupełny, tj. czy dla każdej formuły $r = s$, zachodzi $\lambda\beta \vdash r = s$ lub $\lambda\beta + \{r = s\}$ jest niespójna?
- czy równość $=$ jest rozstrzygalna w $\lambda\beta$?

Przykład

- Niech $\mathbf{K} = \lambda xy.x$ i $\mathbf{I} = \lambda x.x$.
- Pokażemy, że $\lambda\beta \nvdash \mathbf{K} = \mathbf{I}$.
- $\mathbf{K} = \mathbf{I} \to \forall xy \mathbf{K}xy = \mathbf{I}xy$
- $\mathbf{K}xy = x$, $\mathbf{I}xy = xy$
- $\mathbf{K}xy = \mathbf{I}xy \to x = xy$
- Weźmy $x = \mathbf{I}$; wówczas $\mathbf{I} = \mathbf{I}y = y$
- Sprzeczność.

Konteksty

- Kontekstem nazywamy "term z dziurą"
- Dowolny kontekst jest izomorficzny z termem należacym do gramatyki:

$$C := [\,] \mid C[[\,]\ t] \mid C[t\ [\,]] \mid C[\lambda x.[\,]]$$

- Każda para (t, C) definiuje nowy term $C[t]$
- Krótki zapis ostatnich 4 reguł systemu wnioskowania dla $\lambda\beta$:

$$C[(\lambda x.r)\ s] = C[r\ [s/x]]$$

Przejrzystość referencyjna (referential transparency)

Lemat

Dla dowolnych termów r, s i dla dowolnego kontekstu C,

$$r = s \rightarrow C[r] = C[s].$$

Ekstensjonalność i η-redukcja

- W teorii $\lambda\beta$ równość $=$ jest intensjonalna, tzn. dwa termy są równe, jeśli są zapisem "tego samego algorytmu".
- Przykład: gdy $x \notin FV(r)$, $\lambda x.r\,x \neq r$, chociaż $(\lambda x.r\,x)\,s = r\,s$. (Ale jeśli $r = \lambda y.r'$ i $x \notin FV(r')$, to $\lambda x.r\,x = r$.)
- Jak osiągnąć ekstensjonalność?

  Można do systemu $\lambda\beta$ dodać odpowiedni aksjomat (regułę η), np.

$$\lambda x.r\,x = r, \quad \text{jeśli } x \notin FV(r)$$

  albo, równoważnie:

$$\frac{r\,x = s\,x}{r = s} \quad \text{jeśli } x \notin FV(r\,s).$$

Kombinatory punktu stałego

- Kombinatorem punktu stałego nazywamy term R taki, że dla dowolnego termu s mamy

$$R\,s = s\,(R\,s)$$

- $R\,s$ jest punktem stałym termu s.
- Przykłady:
  - Kombinator Curry'ego: $Y := \lambda f.(\lambda x.f\ (x\,x))\ (\lambda x.f\ (x\,x))$
  - Kombinator Turinga: $\Theta := A\,A$, gdzie $A := \lambda x.\lambda y.y\ (x\,x\,y)$.
  - Kombinator Klopa: $Y_k := LLLLLLLLLLLLLLLLLLLLLLLLLL$, gdzie $L = \lambda abcdefghijklmnopqstuvwxyzr.r\ (thisisafixedpointcombinator)$.

Przykład

- Jak skonstruować term spełniający pewne równanie stałopunktowe?
- Przykład: Znaleźć term t taki, że dla dowolnego s

$$t\,s = s\,t.$$

- t może być takie, że $t = \lambda y.y\ t$
- t jest punktem stałym termu $F = \lambda x.\lambda y.y\ x$ $(F\ t = \lambda y.y\ t = t)$
- Zatem $t := Y\ F$

Plan

Przepisywanie termów - podstawowe pojęcia

- Zbiór podtermów danego termu definiujemy jako najmniejszy zbiór spełniający warunki:
  - każdy term jest swoim podtermem
  - jeśli s jest podtermem u, to s jest podtermem $\lambda x.u$
  - jeśli s jest podtermem u, to s jest podtermem $u\,t$ oraz s jest podtermem $t\,u$
- Pojęciem redukcji na dowolnym zbiorze termów nazywamy relację binarną na tym zbiorze: $R \subseteq A^2$.
- Na podstawie pojęcia redukcji definiujemy relację redukcji jednokrokowej $\rightarrow_R \subseteq A^2$ jako kompatybilne domknięcie pojęcia redukcji, czyli najmniejszą relację zawierającą R i zamkniętą ze względu na konstruktory termów.
- Relacja $\rightarrow_R$ definiuje przepisanie termu za pomocą relacji (reguły) R.

β-redukcja

- W lambda rachunku definiujemy pojęcie β-redukcji jako relację R_β zadaną regułą:

$$(\lambda x.t)\, s \to_\beta t\ [s/x],$$

lub inaczej:

$$R_\beta = \{((\lambda x.t)\, s, t\ [s/x]) : s, t \in \Lambda; x \in V\}.$$

- Term postaci $(\lambda x.t)\, s$ nazywamy β-redeksem.
- Na podstawie pojęcia redukcji definiujemy relację redukcji jednokrokowej $\to_{R_\beta}$ (w skrócie $\to$):

$$\frac{(s,u) \in R_\beta}{s \to u} \qquad \frac{r \to s}{r\, u \to s\, u}$$

$$\frac{r \to s}{u\, r \to u\, s} \qquad \frac{r \to s}{\lambda x.r \to \lambda x.s}$$

β-redukcja, c.d.

- Relacja $\rightarrow$ definiuje jeden krok obliczenia.
- Dalej definiujemy relacje:
  - $\rightarrow^*$ jako zwrotne i tranzytywne domknięcie $\rightarrow$:

$$\frac{}{s \rightarrow^* s}$$

$$\frac{r \rightarrow s \quad s \rightarrow^* t}{r \rightarrow^* t}$$

  - $=_\beta$ jako zwrotne, symetryczne i tranzytywne domknięcie $\rightarrow$, czyli najmniejszą relację równoważności zawierającą R_β. Relację $=_\beta$ nazywamy β-konwersją.

- Okazuje się, że $=$ i $=_\beta$ definiują tę samą relację:

Twierdzenie.

$\lambda\beta \vdash t = s$ wtedy i tylko wtedy, gdy $t =_\beta s$. □

η-redukcja

- W podobny sposób możemy zdefiniować regułę η-redukcji:

$$\lambda x.r\, x \to_\eta r,$$

i regułę η-ekspansji

$$r \to_{\eta'} \lambda x.r\, x,$$

jeśli $x \notin FV(r)$.

Normalizacja

- Za pomocą relacji $\longrightarrow$ możemy generować ciąg termów, startując z danego termu t:

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow ...$$

- β-redukcja definiuje "minimalny"krok redukcji (odpowiada zastosowaniu funkcji do argumentu, a więc "minimalny"postęp obliczenia). W każdym kroku tworzenia ciągu $t, t_1, ...$ wykonujemy jedną β-redukcję.
- Mówimy, że term t jest w postaci normalnej, jeśli nie istnieje term s taki, że $t \longrightarrow s$, tj. gdy t nie zawiera żadnego β-redeksu.
- Mówimy, że term t jest postacią normalną termu s, jeśli s redukuje się do t w pewnej liczbie kroków, i t jest w postaci normalnej.
- Proces wyznaczania postaci normalnej termu nazwyamy normalizacją.

- Czy każdy term ma postać normalną?
- Czy wybór redeksu w każdym kroku jest jednoznaczny?

Normalizacja – przykłady

- $\lambda x.\lambda y.x\,y$ jest w postaci normalnej
- Niech $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$. Ω nie ma postaci normalnej; generuje nieskończony ciąg redukcji $\Omega \to \Omega \to \ldots$
- Niech $\mathbf{K} = \lambda xy.x$. Term $\mathbf{K}\,\mathbf{K}\,\Omega$ ma postać normalną, ale istnieje też nieskończony ciąg redukcji dla tego termu, w zależności od wyboru redeksu w każdym kroku.

Normalizacja – strategie

- Strategią redukcji nazywamy funkcję, która przyporządkowuje każdemu termowi niebędącemu w postaci normalnej pewien jego podterm, który jest redeksem.
- Mówimy, że dana strategia jest normalizująca, jeśli każdy term osiąga w tej strategii postać normalną.
- Mówimy, że term jest normalizowalny, jeśli ma postać normalną.
- Mówimy, że term jest silnie normalizowalny, jeśli nie istnieje nieskończony ciąg redukcji generowany przez ten term.
- Mówimy, że relacja redukcji jest słabo normalizująca, jeśli dla każdego termu istnieje strategia redukująca go do postaci normalnej.
- Mówimy, że relacja redukcji $\longrightarrow$ jest silnie normalizująca, jeśli każdy term jest w niej silnie normalizowalny.

β-redukcja nie jest ani silnie ani słabo normalizująca. (Silną normalizowalność można uzyskać wprowadzając odpowiednie typowanie termów, i rozważając tylko termy typowalne.)

Postać normalna

- Termy w postaci normalnej można scharakteryzować syntaktycznie.
- Term jest w postaci normalnej, jeśli należy do zbioru termów t_{nf} generowanych przez gramatykę:

$$t_{ne} := x \mid t_{ne}\ t_{nf}$$

i

$$t_{nf} := t_{ne} \mid \lambda x.t_{nf}.$$

- Inaczej :

$$t_{nf} := \lambda \vec{x}.y\ \vec{t_{nf}}$$

Własność Churcha-Rossera

- Mówimy, że relacja $\longrightarrow$ ma własność rombu, jeśli dla dowolnych termów t_1, t_2, t_3, dla których $t_1 \longrightarrow t_2$ i $t_1 \longrightarrow t_3$, istnieje term s taki, że $t_2 \longrightarrow s$ oraz $t_3 \longrightarrow s$.
- Mówimy, że relacja $\longrightarrow$ ma własność Churcha-Rossera, jeśli $\longrightarrow^*$ ma własność rombu, tj. jeśli dla dowolnych termów t_1, t_2, t_3, dla których $t_1 \longrightarrow^* t_2$ i $t_1 \longrightarrow^* t_3$, istnieje term s taki, że $t_2 \longrightarrow^* s$ oraz $t_3 \longrightarrow^* s$.

Twierdzenie (Twierdzenie)

Relacja $\longrightarrow_{R_\beta}$ ma własność Churcha-Rossera.

Szkic dowodu.

Definiujemy relację "równoległej redukcji", która ma własność rombu. Pokazujemy, że β-redukcja jest tranzytywnym domknięciem równoległej redukcji. [Barendregt '80] □

Własność Churcha-Rossera, c.d.

- Własność CR zapewnia istnienie co najwyżej jednej postaci normalnej dla każdego termu.
- Jeśli $s =_\beta t$, to istnieje term u taki, że $s \to^* u$ i $t \to^* u$.
- Dzięki własności CR, możemy pokazać spójność teorii $\lambda\beta$:

  Weźmy dwie różne (nie-β-równoważne) postaci normalne s i t. Załóżmy, że $\lambda\beta \vdash s = t$. Wówczas $s =_\beta t$, więc istnieje term u taki, że $s \to^* u$ i $t \to^* u$. Ponieważ s i t są w postaci normalnej, więc u musi być równe s i u musi być równe t. Sprzeczność. Zatem $\lambda\beta \nvdash s = t$.

Czołowa postać normalna

- Niech $\vec{r}$ oznacza wektor $r_1, r_2, \ldots, r_n$ dla pewnego $n \geq 0$.
- Każdy term można zapisać w postaci $\lambda\vec{x}.y\,\vec{r}$ albo $\lambda\vec{x}.((\lambda y.s)\,t)\,\vec{r}$.
- Term postaci

$$t_{hnf} := \lambda\vec{x}.y\,\vec{r}.$$

  nazywamy czołową postacią normalną
- W termie $\lambda\vec{x}.((\lambda y.s)\,t)\,\vec{r}$, redeks $((\lambda y.s)\,t)$ nazywamy redeksem czołowym.
- Term może mieć wiele czołowych postaci normalnych, np. $(\lambda x.x\,(\mathbf{I}\,\mathbf{I}))\,z$ ma dwie hnf: $z\,\mathbf{I}$ i $z\,(\mathbf{I}\,\mathbf{I})$.
- Kanoniczną czołową postacią normalną nazywamy tę z nich, którą osiągamy przez redukcję czołowego redeksu w każdym kroku.
- Słabą czołową postać normalną definiujemy następująco:

$$t_{whnf} := x\,\vec{t} \mid \lambda x.t$$

Standardyzacja

- Redeks nazywamy lewostronnym, jeśli jest redeksem położonym najbardziej na lewo w danym termie (tj., jego λ-abstrakcja zaczyna się najbardziej na lewo).
- Każdy redeks czołowy jest lewostronny, ale nie każdy redeks lewostronny jest czołowy – wtedy nazywamy go redeksem wewnętrznym.
- Oznaczamy:
  - $r \xrightarrow{l} s$, gdy $r \to s$ przez kontrakcję redeksu lewostronnego
  - $r \xrightarrow{h} s$, gdy $r \to s$ przez kontrakcję redeksu czołowego
  - $r \xrightarrow{i} s$, gdy $r \to s$ przez kontrakcję redeksu wewnętrznego

Standardyzacja, c.d.

Twierdzenie

Dla dowolnego termu r, jeśli r ma postać normalną s, to $r \overset{I}{\rightarrow}{}^{*} s$.

Szkic dowodu.

Indukcja po rozmiarze s. Pokazujemy, że dowolny ciąg redukcji w termie można symulować przez pewną liczbę redukcji czołowych, po których następują tylko redukcje wewnętrzne. Zatem $r \overset{h}{\rightarrow}{}^{*} s' \overset{i}{\rightarrow}{}^{*} s$. Jeśli $s = \lambda\vec{x}.y\ \vec{u}$, to $s' = \lambda\vec{x}.y\ \vec{u'}$, gdzie $u'_i \rightarrow^{*} u_i$. Stosując hipotezę indukcyjną, każde z u'_i redukuje się lewostronnie do u_i. Ustawiając je w odpowiedniej kolejności, mamy $s' \overset{I}{\rightarrow}{}^{*} s$, a zatem również z $r \overset{I}{\rightarrow}{}^{*} s$. □

Reguły delta

- W celu zwiększenia funkcjonalności rachunku lambda jako języka programowania, możemy rozszerzać składnię o stałe (moc obliczeniowa pozostaje ta sama)
- Następnie musimy dodać odpowiednie reguły redukcji modelujące znaczenie tych stałych
- Przykład – dodanie wartości logicznych i konstrukcji warunkowej:

$$t := \ldots \mid \mathtt{t} \mid \mathtt{f} \mid \mathtt{if}$$

- Pojęcie redukcji:

$$\mathtt{if}\ \mathtt{t}\ r\ s \to_\delta r$$

$$\mathtt{if}\ \mathtt{f}\ r\ s \to_\delta s$$

- Alternatywnie, `if` można zdefiniować jako "formę specjalną" (wtedy należy dodać odpowiednie reguły formujące nowe konteksty)

Reguły delta, c.d.

Twierdzenie (Mitschke)

Niech δ będzie stałą i R pewną n-arną relacją na zbiorze termów. Wprowadźmy regułę redukcji

$$\delta\, r \to_\delta s, \ \textit{gdy}\ R(r, s).$$

Relacja $\beta\delta$ jest CR, jeśli R jest zamknięta ze względu na $\beta\delta$-redukcję i podstawienie.

- Przykład: dodanie $\delta\, t\, t \to \delta_0$ do lambda rachunku zaburza własność CR (reguła nie spełnia warunków twierdzenia, bo $R(t, r) := t \equiv r$ nie jest zamknięta ze względu na β-redukcję)
- Każde rozszerzenie, w którym w termy r, s są zamknięte i w postaci normalnej, zachowują własność CR.

Plan

Wyrażalność w rachunku lambda

- Rachunek lambda jako język programowania jest zupełny w sensie Turinga.
- Z tezy Churcha wynika, że każdy algorytm da się zakodować w rachunku lambda.

Rekursja w rachunku lambda

Twierdzenie (Twierdzenie o punkcie stałym)

Dla każdego termu t, istnieje term x taki, że

$$x = t\,x.$$

Dowód.

$Y\,t$ definiuje punkt stały termu t: $Y\,t =_\beta t\,(Y\,t)$. Dla θ mamy nawet $\theta\,t \to^* t\,(\theta\,t)$. □

Operacje logiczne

- Stałe logiczne:

$$\lceil \texttt{true} \rceil = \lambda xy.x$$

$$\lceil \texttt{false} \rceil = \lambda xy.y$$

- Konstrukcja warunkowa:

$$\texttt{if a b c} = \begin{cases} \texttt{b} & \text{jeśli } \texttt{a=true} \\ \texttt{c} & \text{jeśli } \texttt{a=false} \end{cases}$$

$$\lceil \texttt{if} \rceil = \lambda xyz.x\ y\ z$$

- Sprawdzenie:

$$\lceil \texttt{if} \rceil \lceil \texttt{true} \rceil \lceil b \rceil \lceil c \rceil \rightarrow^* \lceil b \rceil$$

$$\lceil \texttt{if} \rceil \lceil \texttt{false} \rceil \lceil b \rceil \lceil c \rceil \rightarrow^* \lceil c \rceil$$

Pary

$$\lceil (a,\ b) \rceil = \lambda x.x\ \lceil a \rceil\ \lceil b \rceil$$

$$\lceil \pi_1 \rceil = \lambda x.x\ \lceil true \rceil$$

$$\lceil \pi_2 \rceil = \lambda x.x\ \lceil false \rceil$$

$$\lceil \pi_1 \rceil (\lceil (a,\ b) \rceil) \rightarrow^* \lceil a \rceil$$

$$\lceil \pi_2 \rceil (\lceil (a,\ b) \rceil) \rightarrow^* \lceil b \rceil$$

Numerały Churcha

- Liczby naturalne możemy reprezentować jako funkcje (numerały Churcha):

$$\lceil n \rceil = \lambda fx.f^n\, x,$$

  np. $\lceil 0 \rceil = \lambda fx.x$, $\lceil 1 \rceil = \lambda fx.f\, x$.

- Przykładowe operacje:

$$\lceil succ \rceil = \lambda nfx.f\ (nfx)$$

$$\lceil zero? \rceil = \lambda x.x\ (\lambda y.\lceil false \rceil)\ \lceil true \rceil$$

$$\lceil succ \rceil \lceil n \rceil \to^* \lceil n+1 \rceil$$

$$\lceil zero? \rceil \lceil 0 \rceil \to^* \lceil true \rceil$$

$$\lceil zero? \rceil \lceil n+1 \rceil \to^* \lceil false \rceil$$

Schemat reprezentacji

- Dla dowolnej sygnatury Σ, dla każdego jej sortu definiujemy różnowartościową funkcję $\lceil\cdot\rceil : T(\Sigma, V) \to \mathtt{NF}$ ($\mathtt{NF}$ – zbiór postaci normalnych w Λ)
- Definiujemy schemat dla reprezentacji funkcji definiowanych przez dopasowanie wzorca (pattern matching)
- Alternatywne kodowanie – z użyciem numerowania Gödla i reprezentacji numerałów

Schemat reprezentacji

- Dana jest algebra A nad sygnaturą Σ; sygnatura zawiera:
  - Zbiór sortów
  - Zbiór symboli (operacji) o określonych arnościach i sortach argumentów
- Przykład: sygnatura listy

$$\Sigma = \{A, L; \mathtt{nil}^{L}, \mathtt{cons}^{A \to L \to L}, head^{L \to A}, tail^{L \to L}\}$$

- Własności listy określają aksjomaty:

$$head(\mathtt{cons}\, a\, l) = a$$

$$tail(\mathtt{cons}\, a\, l) = l$$

- Pytanie: mając dany kod dla konstruktorów, jak zakodować operacje definiowanie przez aksjomaty?

Schemat reprezentacji

- Definiujemy $\lceil\cdot\rceil : \Sigma \to \mathtt{NF}$ jako funkcję różnowartościową w zbiór postaci normalnych rachunku lambda
- Weźmy dowolny sort S z sygnatury i ponumerujmy jego konstruktory: $c_1, \ldots, c_m$.
- Jeśli c_i jest konstruktorem sortu S o n argumentach, to

$$\lceil c\ t_1 \ldots t_n \rceil = \lambda x_1 \ldots x_m . x_i \lceil t_1 \rceil \ldots \lceil t_n \rceil$$

- Funkcje definiowane przez "dopasowanie wzorca"(pattern matching) kodujemy następująco:

$$\lceil \mathtt{match}\ t\ \mathtt{with}\ \mid c_1 \to t_1 \mid \ldots \mid c_m \to t_m \rceil =$$

$$\lceil t \rceil\ (\lambda x_1 \ldots x_{n_1} . \lceil t_1 \rceil) \ldots (\lambda x_1 \ldots x_{n_m} . \lceil t_m \rceil)$$

Schemat reprezentacji – przykład list

- Kodowanie konstruktorów:

$$\lceil \texttt{nil} \rceil = \lambda x_1 x_2 . x_1$$

$$\lceil \texttt{cons}\ a\ b \rceil = \lambda x_1 x_2 . x_2\ \lceil a \rceil\ \lceil b \rceil$$

- Kodowanie funkcji *head* i *tail* ($head(nil) = nil$ i $tail(nil) = nil$):

$$\lceil head \rceil = \lambda l . l\ \lceil nil \rceil\ (\lambda ab . a)$$

$$\lceil tail \rceil = \lambda l . l\ \lceil nil \rceil\ (\lambda ab . b)$$

Wyrażalność w językach programowania

Czy możemy zapisać powyższe definicje w funkcyjnych językach programowania?

- W językach funkcyjnych nie mamy dostępu do postaci wewnętrznej funkcji
- W językach typowanych nie wszystkie konstrukcje są syntaktycznie dopuszczalne

Przykłady (Ocaml)

- Definicje operacji logicznych:

```
let tt = fun x y -> x
let ff = fun x y -> y
let if_ = fun x y z -> x y z
let and_ b1 b2 = b1 b2 ff
```

- Działanie:

```
# if_ tt 2 3;;
- : int = 2
# and_ tt ff;;
- : '_a -> '_b -> '_b = <fun>
# if_ tt ff tt;;
- : '_a -> '_b -> '_b = <fun>
```

- Nie możemy zdefiniować wprost operatora Y:

```
let y = (fun f -> (fun x -> f (x x)) (fun x -> f (x x)))

# ... f (x x) ...
          ^
This expression has type 'a -> 'b but is here used with type 'a
```

Przykłady (Ocaml), c.d.

- Możemy zdefiniować Y wykorzystując typy rekurencyjne:

```
type 'a t = T of ('a t -> 'a)

let fold x = T x
let unfold = function T f -> f

let y f = (fun x -> f (x (fold x))) (fun x -> f ((unfold x)
```

Wyrażalność w języku Scheme

Język Scheme pozwala na częściowe obejście wspomnianych problemów:

- jest beztypowy, a właściwie:
  - typowany dynamicznie - w momencie definicji zmienna nie ma określonego typu, można pod nią podstawiać dowolne obiekty
  - każdy definiowany obiekt ma typ, którego nie można "dopasować" do innego typu (np. funkcja jest funkcją i nie może być użyta w miejsce liczby)
- pozwala na identyfikację funkcji przez jej nazwę w przypadku, gdy wartością wyrażenia jest obiekt wcześniej zdefiniowany – obiekt rezydualny (wszystkie wartości są w rzeczywistości wskaźnikami)
- nie pozwala na identyfikację funkcji w przypadku konstruowania nowej, anonimowej funkcji

Ultrakrótki kurs języka Scheme

λ – term	składnia Scheme'a
x	x
$\lambda x.t$	`(lambda (x) t)`
$\lambda xy.t$	`(lambda (x y) t)`
$r\ s$	`(r s)`

- Napisy, liczby naturalne - reprezentowane standardowo: 2, "*scheme*"
- Wartości i operacje logiczne: `#t`, `#f`, `if`, `and`, `or`

Ultrakrótki kurs języka Scheme, c.d.

- Mechanizm cytowania pozwala na reprezentowanie kodu programu jako danej: `quote`

```
> (lambda (x) x)
#<procedure>
> (quote (lambda (x) x))
(lambda (x) x)
```

- Pary i listy:

```
> (cons 2 3)
(2 . 3)
> (list 2 #t (lambda (x) x))
(2 #t #<procedure>)
> (car (cons 2 3))
2
> (cdr (cons 2 3))
3
```

Przykłady (Scheme)

```
(define tt (lambda (x y) x))
(define ff (lambda (x y) y))
(define if_ (lambda (x y z) (x y z)))
(define and_ (lambda (x y) (x y ff)))

> (if_ tt 2 ff)
2
> (and_ tt ff)
#<procedure:ff>
```

Operator `eq?` wskazuje na identyczność fizyczną obiektów:

```
> (eq? ff ((lambda (x) x) ff))
#t
> (eq? ff (lambda (x y) y))
#f
```

Kodowanie sygnatur za pomocą par

- Alternatywny schemat reprezentacji polega na kodowaniu struktur danych jako par
- Np. kodowanie konstruktorów list:

$$\lceil \texttt{nil} \rceil = \lambda x.x$$

$$\lceil \texttt{cons}\ a\ b \rceil = \lceil (a,\ b) \rceil$$

- Kodowanie funkcji *head* i *tail* ($head(nil) = nil$ i $tail(nil) = nil$):

$$\lceil head \rceil = \lceil \pi_1 \rceil$$

$$\lceil tail \rceil = \lceil \pi_2 \rceil$$

Reprezentacja struktur danych w Scheme'ie

- Strukturę danych reprezentujemy jako (zagnieżdżoną) parę lub listę
- Pary i listy są obserwowalne
- Do obsługi danych służą funkcje – konstruktory i akcesory

```
(define (make-tree label lson rson) (list label lson rson))
(define (label t) (car t))
(define (lson t) (cadr t))
(define (rson t) (caddr t))
(define (height t)
        (if (null? t) 0
        (if (pair? t) (+ 1 (max (height (lson t))
                                (height (rson t)))) 1)))
```

Definiowalność funkcji numerycznych

- Mówimy, że funkcja całkowita $f : \mathbb{N}^m \to \mathbb{N}$ jest definiowalna w rachunku lambda, jeśli istnieje term F taki, że

$$F \lceil n_1 \rceil \lceil n_2 \rceil \ldots \lceil n_m \rceil \to^* \lceil f(n_1, n_2, \ldots, n_m) \rceil,$$

dla dowolnego systemu numerycznego kodującego liczby naturalne.

Definiowalność funkcji rekurencyjnych

Twierdzenie

Wszystkie funkcje rekurencyjne są definiowalne w lambda rachunku.

- Klasę funkcji rekurencyjnych definiujemy jako najmniejszy zbiór zawierający funkcje początkowe: zerowania, następnika i rzutowania, i zamknięty ze względu na złożenie, rekursję prostą i operację minimum.
- Np. każda z funkcji początkowych jest definiowalna:

$$\Pi_i^n = \lambda x_1 \ldots x_n . x_i$$

$$S = \lambda x . suc\ x$$

$$Z = \lambda x . zero$$

Definiowalność funkcji rekurencyjnych, c.d.

Twierdzenie (Kleene)

Funkcja numeryczna jest definiowalna w lambda rachunku wtedy i tylko wtedy, gdy jest rekurencyjna.

- Jeśli rozszerzymy definicję funkcji definiowalnej na funkcje częściowe: Funkcja częściowa $f : \mathbb{N}^m \to \mathbb{N}$ jest λ-definiowalna, jeśli istnieje term F taki, że:

$F\ \lceil n_1 \rceil \lceil n_2 \rceil \dots \lceil n_m \rceil \to^* \lceil f(n_1 n_2 \dots n_m) \rceil$, gdy $f(n_1, \dots, n_m)$ określone,

$F\ \lceil n_1 \rceil \lceil n_2 \rceil \dots \lceil n_m \rceil$ nie ma hnf, gdy $f(n_1, \dots, n_m)$ nieokreślone,

wówczas mamy równość zbiorów funkcji λ-definiowalnych i częściowo rekurencyjnych.

- Z tezy Churcha-Turinga (f jest częściowo rekurencyjna $\Leftrightarrow$ f jest obliczalna w sensie Turinga) wynika więc, że wszystkie "efektywnie obliczalne" funkcje dadzą się zakodować w rachunku lambda.

Problemy rozstrzygalności

Twierdzenie (Scott)

Niech A będzie nietrywialnym podzbiorem Λ, zamkniętym ze względu na równość. Wówczas A nie jest zbiorem rekurencyjnym, tzn. nie istnieje maszyna Turinga, która dla danego elementu zatrzymuje się z informacją, czy ten element jest w zbiorze A czy nie.

- Problem istnienia postaci normalnej termu jest nierozstrzygalny (tj. zbiór $\{t \in \Lambda : t$ ma postać normalną$\}$ nie jest rekurencyjny), ale jest semi-rozstrzygalny
- Problem: dla danych r, s, czy $r = s$, jest nierozstrzygalny.

Autoewaluacja

- W rachunku lambda możemy zakodować termy rachunku lambda:

$$\lceil x \rceil = \lambda abc.a\, x$$

$$\lceil r\, s \rceil = \lambda abc.b\, \lceil r \rceil\, \lceil s \rceil$$

$$\lceil \lambda x.r \rceil = \lambda abc.c\, (\lambda x.\lceil r \rceil)$$

- Zmienne a, b, c są "świeże"(nie występują po lewej stronie równań)
- W przypadku λ-abstrakcji do związania zmiennej używamy HOAS (higher-order abstract syntax)
- Wówczas związanie zmiennej w abstrakcji-danej osiągamy przez związanie zmiennej w abstrakcji-kodzie
- Taka reprezentacja odpowiada np.

```
type term = Var of var | App of term * term
          | Abs of (term -> term)
```

Autoewaluacja, c.d.

- Ewaluator – term E taki, że dla każdego termu r:

$$E\lceil r\rceil \rightarrow_{\beta}^{*} r$$

- Reduktor (reducer) – term R taki, że dla każdego termu r:

$$R\lceil r\rceil \rightarrow_{\beta}^{*} \lceil \mathtt{nf}(r)\rceil$$

- Autoewaluator:

$$E = Y\,(\lambda em.m\,(\lambda x.x)\,(\lambda mn.(e\,m)\,(e\,n))\,(\lambda m.\lambda v.e\,(m\,v)))$$

- Lub inaczej:

$$E(Var\,x) = x$$

$$E(App\,r\,s) = E(r)\,E(s)$$

$$E(Abs(r)) = \lambda x.E(r\,x)$$

- Redukcje w języku reprezentowanym są wykonywane na metapoziomie; strategia zależy od strategii w metajęzyku
- Źródło: T. Mogensen "Efficient Self-Interpretation in Lambda Calculus"

Reduktor (program normalizujący)

Nieformalnie, reduktor możemy zapisać jako $R(r) = snd(R_0(r))$, gdzie:

$$R_0(Var\ x) = x$$

$$R_0(App\ r\ s) = (fst(R_0\ r))\ (R_0\ s)$$

$$R_0(Abs\ r) = (\lambda v.R_0(r\ v),\ Abs(\lambda w.snd((\lambda v.R_0(r\ v))\ (P(Var\ w)))))$$

$$P(r) = (\lambda v.P(App\ r\ (snd(v))), r)$$

Plan

Rachunek kombinatorów (logika kombinatoryczna)

- Formalizm oparty na kombinatorach (Curry, Schönfinkel)
- Alternatywny dla rachunku lambda – brak wiązania zmiennych
- Rachunek lambda - model dla języków programowania, rachunek kombinatorów - model dla implementacji

Składnia

- Zauważmy, że dla dowolnego termu otwartego r ($FV(r) = \vec{x}$), istnieje term zamknięty t taki, że
$$t\,\vec{x} = r$$
- Wystarczy wziąć $t = \lambda\vec{x}.r$
- Równoważnie, term t można skonstruować tylko za pomocą dwóch kombinatorów i aplikacji

Składnia, c.d.

- Termy rachunku kombinatorów:

$$r := x \mid r\ r \mid \mathbf{S} \mid \mathbf{K}$$

- Aplikacja wiąże do lewej
- Wszystkie wystąpienia zmiennych w termach są wolne (nie ma operatora wiązania)

Wnioskowanie w CL

- System wnioskowania definiują reguły:

$$\frac{}{s = s} \qquad \frac{s = u}{u = s} \qquad \frac{r = s \quad s = u}{r = u}$$

$$\frac{}{\mathbf{K}\, rs = r} \qquad \frac{}{\mathbf{S}\, rst = (r\ t)\ (s\ t)}$$

$$\frac{s = r}{s\ t = r\ t} \qquad \frac{s = t}{r\ s = r\ t}$$

- Przykład: Możemy zdefiniować $\mathbf{I} := \mathbf{SKK}$, ponieważ $\mathbf{I}x = \mathbf{SKK}x = (\mathbf{K}x)(\mathbf{K}x) = x$.
- W CL definiujemy również relację redukcji przez skierowanie równań (podobnie jak w $\lambda\beta$), np. $\mathbf{SKKK} \to (\mathbf{KK})(\mathbf{KK}) \to \mathbf{K}$

Translacje

- Termy rachunku lambda można zdefiniować w rachunku kombinatorów:

$$\overline{x} := x \qquad \overline{\lambda x.r} := \lambda^* x.\overline{r} \qquad \overline{r\,s} := \overline{r}\,\overline{s}$$

$$\lambda^* x.x := \mathbf{SKK}$$

$$\lambda^* x.r := \mathbf{K}r \qquad \text{jeśli } x \notin FV(r)$$

$$\lambda^* x.r\,s := \mathbf{S}(\lambda^* x.r)(\lambda^* x.s)$$

- W drugą stronę:

$$\underline{x} := x$$

$$\underline{r\,s} := \underline{r}\,\underline{s}$$

$$\underline{\mathbf{K}} := \lambda xy.x$$

$$\underline{\mathbf{S}} := \lambda xyz.(x\,z)\,(y\,z)$$

- Translacja $\underline{\cdot}$ stanowi prototyp kompilatora (D. Turner)

Przykład

- $\overline{\lambda xy.x} = \mathbf{S}\,(\mathbf{K}\,\mathbf{K})\mathbf{I}$
- $\underline{\mathbf{S}\,(\mathbf{K}\,\mathbf{K})\mathbf{I}} =_\beta \underline{\mathbf{K}}$, ale $CL \nvdash \mathbf{S}\,(\mathbf{K}\,\mathbf{K})\mathbf{I} = \mathbf{K}$.
- system CL jest silniejszy niż $\lambda\beta$

Porównanie CL i $\lambda\beta$

- Mamy następującą własność:

$$CL \vdash r = s \quad \Rightarrow \quad \lambda\beta \vdash \underline{r} = \underline{s}$$

- Aby otrzymać implikację w drugą stronę, należy rozszerzyć teorię CL o aksjomat słabej ekstensjonalności:

$$(\mathrm{ext}) \quad \frac{r = s}{\lambda^* x.r = \lambda^* x.s}$$

- Wtedy

$$CL + \mathrm{ext} \vdash r = s \quad \Leftrightarrow \quad \lambda\beta \vdash \underline{r} = \underline{s}$$

Plan

Własności języka programowania

- Skupimy się na następujących wymaganiach w stosunku do języka programowania:
  - składnia do zapisu programów $\Rightarrow$ lambda termy
  - deterministyczna semantyka operacyjna (proces "wykonania programu") $\Rightarrow$ β-redukcja i wybrana deterministyczna strategia redukcji
  - wynik działania programu $\Rightarrow$ wartość obserwowalna przez użytkownika

Składnia

- minimalna gramatyka lambda termów jest wystarczająca, ale niezbyt praktyczna (m.in. możemy "zakodować" w lambda rachunku dowolną strukturę danych i napisać program kodujący dowolny algorytm)
- programem jest dowolny term zamknięty (bez zmiennych wolnych)
- zwykle rozszerzamy składnię o dodatkowe, popularne konstrukcje (pewne skróty syntaktyczne, ang. "syntactic sugar") oraz rozszerzamy semantykę operacyjną o odpowiednie reguły redukcji, aby nadać im pożądaną funkcjonalność

Deterministyczne strategie redukcji

- Zwykle ustalamy konkretną strategię redukcji, która w danym kroku wybiera dokładnie jeden z możliwych redeksów.
- Typowe strategie to:
  - normalna (lewostronna zewnętrzna) - zawsze wybieraj redeks położony w termie najbardziej na lewo (t. który zaczyna się najbardziej na lewo)
  - aplikatywna (lewostronna wewnętrzna) - zawsze wybieraj redeks położony najbardziej na lewo, ale niezawierający w sobie innego redeksu
- W językach programowania dodatkowo przyjmuje się założenie, że ciało funkcji nie musi być redukowane; wobec tego nigdy nie wybiera się redeksu położonego wewnątrz lambda abstrakcji.
- Postać normalna w takich strategiach to dowolna lambda abstrakcja:

$$t_{nf} := \lambda x.t.$$

Jest to (zamknięta) słaba czołowa postać normalna.

Wynik działania programu

- Efekt wykonania programu powinien być obserwowalny przez użytkownika.
- W języku opartym na czystym rachunku lambda, postacią normalną (przy standardowych strategiach) jest dowolna lambda abstrakcja (funkcja).
- Nie mamy dostępu do wewnętrznej postaci takiej funkcji, zatem jedynym efektem działania programu w takim języku jest informacja, że obliczenie zostało zakończone i jego wartością jest funkcja (por. Ocaml, Scheme, etc).
- Zwykle w językach beztypowych rozszerza się składnię o stałe bazowe (numerały, wartości logiczne, znaki); wówczas będą one również możliwymi wynikami obliczeń.

Minimalny język programowania Λ_n

- Definiujemy język Λ_n następująco:
  - program to dowolny zamknięty lambda term
  - strategia redukcji to strategia normalna, generująca słabe czołowe postaci normalne
  - wartością programu jest lambda abstrakcja

$$v := \lambda x.r$$

Minimalny język programowania Λ_n

- Strategia normalna ("wołanie-przez-nazwę", 'call-by-name")
- Semantykę operacyjną wielokrokową realizującą tę strategię możemy przedstawić następująco:

$$\frac{}{\lambda x.r \Downarrow_n \lambda x.r}$$

$$\frac{r \Downarrow_n \lambda x.u \quad u\,[s/x] \Downarrow_n v}{r\,s \Downarrow_n v}$$

- Oznaczenia:

$$r \Downarrow_n := \exists v.r \Downarrow_n v$$

$$r \Uparrow := \neg(r \Downarrow_n)$$

- $\Downarrow_n$ jest funkcją częściową (strategia jest deterministyczna)

Równoważność programów

- Jak porównać wyniki działania dwóch programów, jeśli wiemy o nich tylko tyle, że są to funkcje?
  - aplikatywna symulacja
  - obserwacyjna równoważność

Aplikatywna symulacja

- Dwa lambda termy są "równoważne", jeśli zastosowanie ich do tego samego argumentu daje ten sam efekt, czyli albo oba się zatrzymują, albo oba się nie zatrzymują. Możemy powtarzać ten eksperyment, aż zachowanie obu funkcji będzie się różnić (jedna się zatrzyma, druga nie), albo w nieskończoność...
- Np. termy $\lambda x.x$ i $\lambda xy.y$ nie są równoważne – można je rozróżnić za pomocą Ω:

$$(\lambda x.x)\Omega \to \Omega \to \ldots$$

$$(\lambda xy.y)\Omega \to \lambda y.y$$

Aplikatywna symulacja

- Formalnie, definiujemy rodzinę relacji $\sqsubseteq_k \subseteq \Lambda_0^2$:
  - dla dowolnych s, s', $s \sqsubseteq_0 s'$
  - dla dowolnych s, s', $s \sqsubseteq_{k+1} s'$ jeśli

$$\forall (v := \lambda x.p).s \Downarrow_n v \rightarrow (\exists (v' := \lambda x.p').s' \Downarrow_n v' \wedge \forall q.p\ [q/x] \sqsubseteq_k p'\ [q/x])$$

- $s \sqsubseteq s'$ wtedy i tylko wtedy, gdy dla każdego k zachodzi $s \sqsubseteq_k s'$ (s' zatrzymuje się co najmniej wtedy, kiedy s)
- definicję można rozszerzyć na cały zbiór termów: $s \sqsubseteq s'$ wtedy i tylko wtedy, gdy dla każdego $\sigma : V \rightarrow \Lambda$ zachodzi $s_\sigma \sqsubseteq s'_\sigma$
- termy s i s' są wzajemnie podobne ($s \sim s'$) wtw, gdy $s \sqsubseteq s'$ oraz $s' \sqsubseteq s$ (bisymulacja)
- $\sqsubseteq$ jest relacją zwrotną i przechodnią

Aplikatywna symulacja, c.d.

- W sposób trywialny $\Omega \sqsubseteq r$ dla dowolnego r
- $\lambda x.x$ i $\lambda xy.x$ nie są w relacji $\sqsubseteq$
- Jeśli $s \Uparrow$ i $t \Downarrow$, to $\lambda x_1 \ldots x_n.s \sqsubseteq \lambda x_1 \ldots x_n.t$

Aplikatywna symulacja, c.d.

Twierdzenie (Charakteryzacja $\sqsubseteq$)

Dla dowolnych $s, s' \in \Lambda_0$, $s \sqsubseteq s'$ wtedy i tylko wtedy, gdy dla dowolnego ciągu termów zamkniętych $\vec{t}$, jeśli $s\,\vec{t} \Downarrow_n$, to $s'\,\vec{t} \Downarrow_n$.

Równoważność obserwacyjna (kontekstualna)

- Kontekst programu C – dowolny kontekst niezawierający zmiennych wolnych.
- Równoważność termów możemy scharakteryzować za pomocą kontekstów:

  $s \sqsubseteq_{ctx} s'$ wtw, gdy dla każdego kontekstu C, jeśli $C[s] \Downarrow_n$, to $C[s'] \Downarrow_n$.

- s i s' są obserwacyjnie równoważne ($s \sim_{ctx} s'$), jeśli $s \sqsubseteq_{ctx} s'$ oraz $s' \sqsubseteq_{ctx} s$.

Twierdzenie

Dla dowolnych $s, s' \in \Lambda_0$, $s \sim s'$ wtw, gdy $s \sim_{ctx} s'$.

Wniosek: Jeśli dwa termy są rozróżnialne przez jakiś kontekst, to istnieje ciąg termów (kontekst aplikatywny), który je rozróżnia.