Instytut Informatyki

Programowanie funkcyjne

Wykład 11. Siła wyrazu rachunku lambda

Zdzisław Spławski

- Wstęp
- Reguły delta
- Wartości logiczne
- Pary
- Liczby naturalne jako liczebniki Churcha
- Kombinatory punktu stałego
- Kombinator punktu stałego w Haskellu i OCamlu
- Funkcje rekurencyjne — definicje
- Przykłady funkcji pierwotnie rekurencyjnych
- Lambda-definiowalność funkcji rekurencyjnych
- Semantyka prostego języka imperatywnego PJI
- Zadania kontrolne

Język funkcyjny można potraktować jak rachunek lambda (beztypowy lub z typami) z wybraną semantyką redukcyjną, dodanymi stałymi (z odpowiednimi regułami redukcji) i dużą ilością „lukru syntaktyczego”. Te rozszerzenia rachunku lambda stosuje się w celu ułatwienia pisania programów, polepszenia ich czytelności i zwiększenia efektywności. Logicznie nie są one konieczne.

Ekstensjonalność. Praktycznie, z punktu widzenia programowania funkcyjnego, rachunek $\lambda\beta\eta$ nie jest tak ważny jak rachunek $\lambda\beta$, ponieważ reguła (η) zwykle nie jest implementowana. Term $\lambda x.Mx$ jest w słabej czołowej postaci normalnej (WHNF) i jest odróżniany od termu M. Pierwszy jest traktowany jako wartość w konwencjonalnych językach fukcyjnych, drugi może prowadzić do obliczeń nieskończonych.
W tym wykładzie będziemy rozważać rachunek $\lambda\beta$, chyba że wyraźnie zostanie wprowadzony inny wariant.

Reguły delta

W czystym, beztypowym rachunku λ wszystkie funkcje obliczalne są reprezentowalne jako λ-termy. Jednak ze względu na efektywność i wygodę można rozszerzyć ten język. Można wprowadzić pewne stałe (literały), reprezentujące wartości pierwotne (np. liczebniki) oraz operacje (np. dodawanie). Należy wówczas wprowadzić nowe reguły redukcji (δ-redukcje), definiujace semantykę operacyjną tych operacji. Jest to bardzo użyteczne w przypadku stosowanego rachunku λ.

Definicja.

(i) Niech δ będzie pewną stałą. Wówczas $\Lambda\delta$ jest zbiorem λ-termów zbudowanych ze zmiennych i stałej δ za pomocą aplikacji i abstrakcji w zwykły sposób. Analogicznie definiuje się $\Lambda\vec{\delta}$, gdzie $\vec{\delta}$ oznacza ciąg stałych.

(ii) Niech $\vec{M}$ oznacza ciąg zamkniętych λ-termów w postaci normalnej. Do rachunku λ należy dodać reguły δ-redukcji w postaci $\delta\vec{M} \to_\delta f(\vec{M})$.

Reguły delta

Dla zadanej funkcji f δ-redukcja nie jest pojedynczą regułą, lecz schematem reguł. Tak wzbogacony system nazywamy rachunkiem $\lambda\delta$. Relacje kontrakcji i redukcji są oznaczane odpowiednio przez $\rightarrow_{\beta\delta}$ i $\twoheadrightarrow_{\beta\delta}$.

Twierdzenie. Niech f w regule redukcji $\delta\vec{M} \rightarrow_{\delta} f(\vec{M})$ będzie funkcją na zamkniętych λ-termach w postaci normalnej. Wówczas relacja redukcji $\twoheadrightarrow_{\beta\delta}$ spełnia twierdzenie Churcha-Rossera.

Pojęcie redeksu, postaci normalnej i strategii redukcji w sposób naturalny uogólnia się na rachunek $\lambda\delta$. Prawdziwe jest też twierdzenie o standardyzacji.

W poniższych przykładach zbudujemy *stosowany* rachunek λ, definiując nowe stałe i wprowadzając nowe reguły redukcji. Jednocześnie będziemy jednak definiowali reprezentacje tych stałych w postaci λ termów i wyprowadzali reguły redukcji dla nich w celu zilustrowania siły wyrazu *czystego* rachunku λ.

Dodajemy następujące stałe: **false**, **true**, **if** oraz dwie reguły δ:

$$\textbf{if}\ \textbf{true}\ M\ N \to M \qquad\qquad \textbf{if}\ \textbf{false}\ M\ N \to N$$

Odpowiednie termy można zdefiniować na przykład tak (Church):

$$\textbf{true} \equiv \lambda xy.x \qquad \textbf{false} \equiv \lambda xy.y \qquad \textbf{if} \equiv \lambda buv.buv \qquad (\twoheadrightarrow_\eta \lambda b.b)$$

Wyprowadzimy pierwszą regułę δ.

$$\begin{array}{rcll} \textbf{if}\ \textbf{true}\ M\ N & \equiv & (\lambda buv.b\,u\,v)\ \textbf{true}\ M\ N & \to_\beta\ (\lambda uv.\textbf{true}\ u\ v)M\ N \\ & \to_\beta & (\lambda v.\textbf{true}\ M\ v)\ N & \to_\beta\ \textbf{true}\ M\ N \\ & \equiv & (\lambda xy.x)\ M\ N & \to_\beta\ (\lambda y.M)N \\ & \to_\beta & M & \end{array}$$

Analogicznie można wyprowadzić drugą regułę.
W celu zwiększenia czytelności czasem zamiast **if** $B\ M\ N$ będziemy pisali $if\, B\, then\, M\, else\, N$.

Pary

Dodajemy następujące stałe: **pair**, **fst**, **snd** oraz dwie reguły δ:

$$\mathbf{fst}\,(\mathbf{pair}\,M\,N) \to M \qquad\qquad \mathbf{snd}\,(\mathbf{pair}\,M\,N) \to N$$

Odpowiednie termy mogą być zdefiniowane np. tak:

$$\mathsf{fst} \equiv \lambda p.p\lambda xy.x \qquad \mathsf{snd} \equiv \lambda p.p\lambda xy.y \qquad \mathsf{pair} \equiv \lambda uvs.s\,u\,v$$

Wyprowadzimy pierwszą regułę δ.

$$\begin{array}{lclcl}
\mathsf{fst}\,(\mathsf{pair} M N) & \equiv & (\lambda p.p\lambda xy.x)(\mathsf{pair} M N) & \to_\beta & \mathsf{pair} M N \lambda xy.x \\
& \equiv & (\lambda uvs.s\,u\,v) M N \lambda xy.x & \twoheadrightarrow_\beta & (\lambda s.sMN)(\lambda xy.x) \\
& \to_\beta & (\lambda xy.x) M N & \to_\beta & (\lambda y.M)N \\
& \to_\beta & M & &
\end{array}$$

Analogicznie można wyprowadzić drugą regułę.

Liczebniki Churcha

Dla każdego $n \in \mathbb{N}$ wybierana jest stała o nazwie c_n (lub po prostu $\mathbf{n}$). Wybieramy jeszcze dwie stałe: **Iter**, **suc** i dodajemy dwie reguły δ:

$$\begin{cases} \mathbf{Iter}\ \mathbf{0}\ M\ N & \to \quad N \\ \mathbf{Iter}\ (\mathbf{suc}\ \mathbf{n})\ M\ N & \to \quad M(\mathbf{Iter}\ \mathbf{n}\ M\ N) \end{cases}$$

Liczebniki Churcha są definiowane następująco: $\mathsf{c}_n \equiv \mathbf{n} \equiv \lambda fx.f^n x$, gdzie $F^n M$ jest zdefiniowane indukcyjnie ($n \in \mathbb{N}$, a $F, M \in \Lambda$):

$$F^0 M \equiv M \qquad\qquad F^{n+1} M \equiv F(F^n M)$$

Widać, że liczebniki Churcha są iteratorami:

$$\mathbf{0} \equiv \lambda fx.x \qquad \mathbf{1} \equiv \lambda fx.fx \qquad \mathbf{2} \equiv \lambda fx.f(fx) \qquad \text{itd.}$$

Wobec tego definiujemy:

$\mathbf{suc} \equiv \lambda nfx.f(nfx)$ $\qquad$ $\mathbf{Iter} \equiv \lambda nfa.nfa \quad (\twoheadrightarrow_\eta \lambda n.n)$

(lub $\mathbf{suc} \equiv \lambda nfx.nf(fx)$)

Dodawanie i mnożenie

Dodajemy dwie stałe **add**, **mult** oraz dwie reguły δ:

$$\begin{array}{rrcl} (i) & \mathbf{add}\,\mathbf{c}_m\,\mathbf{c}_n & \to & \mathbf{c}_{m+n} \\ (ii) & \mathbf{mult}\,\mathbf{c}_m\,\mathbf{c}_n & \to & \mathbf{c}_{m*n} \end{array}$$

Odpowiednie λ-termy można zdefiniować następująco:

$$\begin{array}{rcl} \mathbf{add} & \equiv & \lambda mnfx.mf(nfx) \\ \mathbf{mult} & \equiv & \lambda mnf.m(nf) \end{array}$$

Wyprowadzenie reguł δ jest łatwe.
Wskazówka. Przez indukcję po m udowodnij $f^m(f^n x) = f^{m+n}x$,
a następnie $(\mathbf{c}_n f)^m x = f^{m*n}x$.

Test na zero

Dodajemy stałą **isZero** oraz dwie reguły δ:

$$\begin{cases} \textsf{isZero}\,\textsf{0} & \rightarrow & \textsf{true} \\ \textsf{isZero}\,(\textsf{suc}\,\textsf{n}) & \rightarrow & \textsf{false} \end{cases}$$

Odpowiedni term można zdefiniować następująco:

$$\textsf{isZero} \equiv \lambda n.\textsf{Iter}\; n\; (\lambda b.\textsf{false})\; \textsf{true}$$

Łatwo teraz wywieść wymagane reguły redukcji.

Poprzednik

$$\begin{cases} \textbf{pred } \mathbf{0} & \rightarrow & \mathbf{0} \\ \textbf{pred } (\textbf{suc } \mathbf{n}) & \rightarrow & \mathbf{n} \end{cases}$$

Ideę obliczania poprzednika wyjaśnia poniższy program.

$$\begin{array}{l} \left.\begin{array}{l} y := 0; \\ z := 0; \end{array}\right\} \langle 0, 0 \rangle \\ \left.\begin{array}{l} \textbf{while } y \neq n \textbf{ do } (*\ z = \textbf{pred } y\ *) \\ \textbf{begin} \\ \quad \left.\begin{array}{l} z := y; \\ y := \textbf{suc } y; \end{array}\right\} q\langle y, z \rangle \rightarrow \langle \textbf{suc } y, y \rangle \\ \textbf{end} \end{array}\right\} n\text{-krotna iteracja operacji } q \\ (*\ z = \textbf{pred } n\ *) \end{array}$$

Teraz ten algorytm możemy zapisać w postaci lambda-termu.

$$\textbf{pred} \equiv \lambda n.\textbf{snd}\ (\textbf{Iter}\ n\ (\lambda p.\textbf{pair}(\textbf{suc}(\textbf{fst}\ p))(\textbf{fst}\ p))\ (\textbf{pair}\ \mathbf{0}\ \mathbf{0}))$$

Równania stałopunktowe w matematyce

Rozważmy równania algebraiczne, np.

$$x = 5/x \qquad \text{czy} \qquad x = 6 - x$$

Problem rozwiązania tych równań można też sformułować jako problem znalezienia punktów stałych funkcji:

$$f_1 \equiv \lambda x.5/x \qquad \text{i} \qquad f_2 \equiv \lambda x.6 - x$$

Punktem stałym funkcji jest wartość należąca do dziedziny funkcji, odwzorowywana przez funkcję na siebie. Poszukujemy więc takich wartości w_1 i w_2, że

$$f_1\, w_1 = w_1 \qquad \text{i} \qquad f_2\, w_2 = w_2$$

Oczywiście $w_1 = \sqrt{5}$ i $w_2 = 3$.
Punkt stały może być jeden, może ich być wiele (nawet nieskończenie wiele, np. dla $\lambda x.x$), może też nie być żadnego (np. dla $\lambda x.x + 1$).
Istnienie i liczba punktów stałych funkcji zależy od jej dziedziny.

Równania stałopunktowe w programowaniu

Podobny problem pojawia się przy definicjach funkcji rekurencyjnych, np.

$$s(n) = \text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * s(n-1)$$

lub inaczej:

$$s = \lambda n.\text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * s(n-1)$$

Z matematycznego punktu widzenia jest to równanie (wyższego rzędu) z jedną niewiadomą s, podobnie jak $x = 6 - x$ jest równaniem z jedną niewiadomą x. Symbol $=$ oznacza tu β (lub $\beta\eta$) konwersję. Problem rozwiązania tego równania, tj. znalezienia lambda termu, spełniającego równanie, sprowadza się do znalezienia punktu stałego funkcjonału:

$$F \equiv \lambda gn.\text{if}\, n = 0\, \text{then}\, 1\, \text{else}\, n * g(n-1)$$

lub bardziej formalnie:

$$F \equiv \lambda gn.\textbf{if}\, (\textbf{isZero}\, n)\, \textbf{1}\, (\textbf{mult}\, n\, (g(\textbf{pred}\, n)))$$

Kombinator punktu stałego $\mathbf{Y}$

Definicja. *Kombinatorem punktu stałego* nazywamy każdy term M taki, że $\forall F. MF = F(MF)$.

Twierdzenie o punkcie stałym.

(i) $\forall F. \exists X. X = FX$

(ii) Istnieje kombinator punktu stałego $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ taki, że $\mathbf{Y}F = F(\mathbf{Y}F)$.

Dowód.

(i) Niech $W \equiv \lambda x.F(xx)$ i $X \equiv WW$. Wówczas $X \equiv WW \equiv (\lambda x.F(xx))W \to F(WW) \equiv FX$

(ii) $\mathbf{Y}F \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F$
$\to_\beta (\lambda x.F(xx))(\lambda x.F(xx))$
$\to_\beta F((\lambda x.F(xx))(\lambda x.F(xx)))$
$=_\beta F((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F) \equiv F(\mathbf{Y}F)$

Kombinator punktu stałego $\mathbf{\Theta}$ (Turing)

Czasem potrzebujemy kombinatora punktu stałego M z nieco mocniejszą własnością:

$$\forall F.MF \twoheadrightarrow_\beta F(MF).$$

Turing zaproponował następujący kombinator:

$$\mathbf{\Theta} \equiv AA \quad \text{gdzie } A \equiv \lambda xy.y(xxy).$$

$$\begin{aligned} \mathbf{\Theta}F &\equiv (\lambda xy.y(xxy))AF \\ &\to_\beta (\lambda y.y(AAy))F \\ &\to_\beta F(AAF) \\ &\equiv F(\mathbf{\Theta}F) \end{aligned}$$

Zauważ, że dla kombinatora $\mathbf{Y}$ zachodzi tylko $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$.
Zarówno $\mathbf{Y}$ jak i $\mathbf{\Theta}$ mają czołową postać normalną (HNF), ale żaden z nich nie ma postaci β-normalnej. To jest własność wszystkich kombinatorów punktu stałego (jest ich nieskończenie wiele). Jak widać zbiór użytecznych λ-termów jest większy niż zbiór termów, posiadających postać β-normalną.

Przykład użycia kombinatora punktu stałego: silnia

Teraz możemy zdefiniować lambda term dla silni jako:

$s \equiv \mathbf{\Theta} F$ (lub $\mathbf{Y} F$) gdzie $F \equiv \lambda gn.\mathrm{if}\, n = 0 \text{ then } 1 \text{ else } n * g(n-1)$

Uwaga. Przy ewaluacji tego termu należy stosować redukcję normalną.

$$
\begin{array}{lcl}
s\, n & \equiv & (\mathbf{\Theta} F)\, n \\
 & \twoheadrightarrow_\beta & F(\mathbf{\Theta} F)\, n \\
 & \equiv & (\lambda gn.\mathrm{if}\, n = 0 \text{ then } 1 \text{ else } n * g(n-1))(\mathbf{\Theta} F)\, n \\
 & \twoheadrightarrow_\beta & \mathrm{if}\, n = 0 \text{ then } 1 \text{ else } n * (\mathbf{\Theta} F)(n-1) \\
 & \equiv & \mathrm{if}\, n = 0 \text{ then } 1 \text{ else } n * s(n-1)
\end{array}
$$

Oczywiście, możliwa jest inna definicja silni za pomocą iteratora, co odpowiadałoby rekursji ogonowej.

Powyższy przykład możemy uogólnić.

Twierdzenie. Niech $C \equiv C[f, \vec{x}]$ będzie termem ze zmiennymi wolnymi f i $\vec{x}$. Wówczas:

(i) $\exists F. \forall \vec{N}. F\vec{N} = C[F, \vec{N}]$

(ii) $\exists F. \forall \vec{N}. F\vec{N} \twoheadrightarrow_\beta C[F, \vec{N}]$

Dowód. W obu przypadkach możemy wziąć $F \equiv \Theta(\lambda f\vec{x}.C[f, \vec{x}])$. Zauważ, że dla przypadku (i) wystarczy wykorzystać $\mathsf{\mathbf{Y}}$.

Przykład.

$$C[f, n] \equiv \mathsf{\mathbf{if}}\,(\mathsf{\mathbf{isZero}}\,n)\,\mathsf{\mathbf{1}}\,(\mathsf{\mathbf{mult}}\,n\,(f(\mathsf{\mathbf{pred}}\,n)))$$

i (i) gwarantuje istnienie termu, nazwijmy go F, który zachowuje się jak funkcja silni, czyli:

$$Fn = \mathsf{\mathbf{if}}\,(\mathsf{\mathbf{isZero}}\,n)\,\mathsf{\mathbf{1}}\,(\mathsf{\mathbf{mult}}\,n\,(F(\mathsf{\mathbf{pred}}\,n)))$$

Definicja kombinatora punktu stałego jest uniwersalnie skwantyfikowana po wszystkich λ-termach. Poniższy lemat charakteryzuje kombinatory punktu stałego poprzez ich interakcję z jednym termem:

Lemat. Niech $G \equiv \lambda yf.f(yf)$. Wówczas $M \in \Lambda$ jest kombinatorem punktu stałego wtw, gdy $M = GM$, tzn. wtw, gdy M jest kombinatorem punktu stałego dla G.
Dowód. Zadania kontrolne.

Zdefiniujmy następujący ciąg kombinatorów:

$$\begin{cases} \mathsf{Y}^0 \equiv \mathsf{Y} \\ \mathsf{Y}^{n+1} \equiv \mathsf{Y}^n G \end{cases}$$

Lemat. Wszystkie termy ciągu $\mathsf{Y}^0, \mathsf{Y}^1, \ldots$ są kombinatorami punktu stałego.
Dowód. Przez indukcję po n. Zadania kontrolne.
Są też kombinatory punku stałego nie należące do ciągu Y^n.

Kombinator punktu stałego w Haskellu.

Bezpośrednia definicja kombinatora $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ w języku Haskell spowoduje błąd typu. Możemy jednak zdefiniować typ danych:

```
newtype Self a = Fold {unfold :: Self a -> a}
-- unfold :: Self a -> Self a -> a
```

a następnie funkcjonał `fixl :: (a -> a) -> a`

```
fixl f = let w = λ x -> f ( unfold x x) in w (Fold w)
```

Ten funkcjonał można wykorzystać do zdefiniowania np. nieskończonej listy jedynek:

```
ones = fixl (λ x -> 1:x)
```

lub funkcji obliczającej wartość silni bez użycia rekursji:

```
fact = fixl (λ f n -> if n==0 then 1 else n*f(n-1))
```

Przekonaj się, że jest to rzeczywiście funkcja silni.

Kombinator punktu stałego w OCamlu. I

Bezpośrednia definicja kombinatora $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ w języku OCaml spowoduje błąd typu. Możemy jednak (podobnie jak w Haskellu) zdefiniować typ danych:

```
type 'a self = Fold of ('a self -> 'a);;
let unfold (Fold t) = t;;
```

a następnie funkcjonał `fixl : ('a -> 'a) -> 'a`

```
let fixl f =
    let w = fun x -> f ( unfold x x) in w (Fold w);;
```

Niestety działa on poprawnie tylko w językach z wartościowaniem leniwym. W OCamlu, który stosuje wartościowanie gorliwie, wartościowanie `fixl` nie kończy się i nie uda się nam zdefiniować w ten sposób np. nieskończonej listy jedynek (sprawdź to!):

```
let ones = fixl (function x -> 1::x);;
```

(ale można to zrobić inaczej (wykład 6): `let rec ones = 1::ones;;`)

Kombinator punktu stałego w OCamlu. II

Można jednak nieco zmodyfikować `w` w funkcjonale `fixl`

```
let fixl f =
    let w = fun x -> f ( unfold x x) in w (Fold w);;
```

wykorzystując η-ekspansję i otrzymać funkcjonał:

```
let fix f = let w = fun x y-> f (unfold x x) y
            in w (Fold w);;
```

lub:

```
let fix f = let w = fun x -> f (fun y -> (unfold x x) y)
            in w (Fold w);;
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

który znajduje punkty stałe funkcji (ale już nie list!). Przekonaj się o tym, definiując funkcję obliczającą silnię zadanej liczby bez jawnego użycia rekursji.

Kombinator punktu stałego w OCamlu. III

Z tego powodu w OCamlu w konstrukcjach `let rec` $f = M$ i `let rec` $f = M$ `in` N term M powinien być funkcją. Term np. `let rec` $f = M$ `in` N jest zasadniczo rozumiany jako `let` $f = \text{fix } \lambda f.M$ `in` N, a kombinator punktu stałego znajduje punkty stałe funkcji. Użycie jawnej rekursji w definicjach wartości, które nie są funkcjami, pozwala definiować struktury cykliczne (jeśli to jest możliwe; patrz wykład 6, str. 34–35).

Funkcjonał *`fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`* można więc zdefiniować bez użycia typu *`'a self`*, np.

```
let fix f = let rec fixf x = f fixf x in fixf;;
```

Można go teraz użyć do zdefiniowania silni bez użycia rekursji:

```
let fact = fix (fun f n -> if n=0 then 1 else n*f(n-1));;
```

Przekonaj się, że jest to rzeczywiście funkcja silni.

Operacja minimum

1. *Funkcją numeryczną* jest dowolna funkcja $f : \mathbb{N}^n \to \mathbb{N}$ dla pewnego n.
2. *Operacja minimum*. Niech $f : \mathbb{N}^{n+1} \to \mathbb{N}$. $(\mu m. f(x_1, \ldots, x_n, m) = 0) : \mathbb{N}^n \to \mathbb{N}$ oznacza najmniejszą liczbę m, dla której zachodzi $f(x_1, \ldots, x_n, m) = 0$ i $f(x_1, \ldots, x_n, k)$ jest zdefiniowana dla wszystkich $k \leq m$; w przeciwnym razie wynik jest niezdefiniowany.
3. *Operacja minimum ograniczonego*. Niech $f : \mathbb{N}^{n+1} \to \mathbb{N}$. $(\mu m < z. f(x_1, \ldots, x_n, m) = 0) : \mathbb{N}^n \to \mathbb{N}$ oznacza najmniejszą liczbę $m < z$, dla której zachodzi $f(x_1, \ldots, x_n, m) = 0$ i $f(x_1, \ldots, x_n, k)$ jest zdefiniowana dla wszystkich $k \leq m$; w przeciwnym razie wynik jest równy z.
4. $\vec{x} \stackrel{\text{def}}{=} \langle x_1, \ldots, x_n \rangle$ dla $n \in \mathbb{N}$.

Funkcje bazowe

Definicja. *Funkcje bazowe* (*początkowe*):

1. $Z(x) = 0$, dla każdego x — funkcja zerująca
2. $S(x) = x + 1$, dla każdego x — funkcja następnika (ang. successor)
3. $U_n^i(x_1, \ldots, x_n) = x_i$, dla $1 \leq i \leq n$ — funkcje projekcji

Schematy tworzenia nowych funkcji I

Definicja. Niech A będzie zbiorem funkcji numerycznych.

1. A jest zamknięty ze względu na operację *złożenia* (*superpozycji funkcji*), jeśli dla wszystkich f zdefiniowanych następująco:

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_r(\vec{x}))$$

   gdzie $g, h_1, \ldots, h_r \in A$, zachodzi $f \in A$.

2. A jest zamknięty ze względu na operację *rekursji prostej*, jeśli dla wszystkich f zdefiniowanych następująco (druga kolumna odnosi się do przypadku $\vec{x} = \langle\rangle$):

$$\begin{cases} f(0, \vec{x}) = g(\vec{x}), \\ f(S(n), \vec{x}) = h(n, f(n, \vec{x}), \vec{x}), \end{cases} \qquad \begin{cases} f(0) = a, \quad \text{dla } a \in \mathbb{N} \\ f(S(n)) = h(n, f(n)), \end{cases}$$

   gdzie $g, h \in A$, zachodzi $f \in A$.

Schematy tworzenia nowych funkcji II

3. A jest zamknięty ze względu na operację *minimum efektywnego*, jeśli dla wszystkich f zdefiniowanych następująco:

   $$f(\vec{x}) = (\mu m.g(\vec{x}, m) = 0),$$

   gdzie $g \in A$ i $\forall\vec{x}.\exists m.g(\vec{x}, m) = 0$, zachodzi $f \in A$.

4. A jest zamknięty ze względu na operację *minimum*, jeśli dla wszystkich f zdefiniowanych następująco:

   $$f(\vec{x}) = (\mu m.g(\vec{x}, m) = 0),$$

   gdzie $g \in A$, zachodzi $f \in A$.

Klasa $\mathcal{PR}$ funkcji *pierwotnie rekurencyjnych*

Definicja. Klasa $\mathcal{PR}$ funkcji *pierwotnie rekurencyjnych* jest najmniejszą klasą funkcji numerycznych zawierających wszystkie funkcje bazowe i zamkniętą ze względu na operacje

(i) złożenia (superpozycji),

(ii) rekursji prostej.

Klasy $\mathcal{R}_0$ i $\mathcal{R}$ funkcji ogólnie i częściowo rekurencyjnych

Definicja. (Herbrand, Gödel) Klasa $\mathcal{R}_0$ funkcji (ogólnie) rekurencyjnych jest najmniejszą klasą funkcji numerycznych zawierających wszystkie funkcje bazowe i zamkniętą ze względu na operacje

(i) złożenia (superpozycji),

(ii) rekursji prostej,

(iii) minimum efektywnego.

Definicja. (Kleene) Klasa $\mathcal{R}$ funkcji częściowo rekurencyjnych jest najmniejszą klasą funkcji numerycznych zawierających wszystkie funkcje bazowe i zamkniętą ze względu na operacje

(i) złożenia (superpozycji),

(ii) rekursji prostej,

(iii) minimum.

Dodawanie

Nieformalnie:

$$\begin{cases} add(0,n) = n \\ add(m+1,n) = add(m,n)+1 \end{cases}$$

Formalnie:

$$\begin{cases} add(0,n) = U_1^1(n) \\ add(S(m),n) = h(m, add(m,n), n) \end{cases}$$

gdzie:

$$h(x_1,x_2,x_3) = S(U_3^2(x_1,x_2,x_3))$$

Poprzednik

Specyfikacja:

$$\delta(n) = \begin{cases} n-1, & \text{dla } n > 0 \\ 0, & \text{dla } n = 0 \end{cases}$$

Definicja (nieformalna):

$$\begin{cases} \delta(0) = 0 \\ \delta(S(n)) = n \end{cases}$$

Formalnie:

$$\begin{cases} \delta(0) = 0 \\ \delta(S(n)) = U_2^1(n, \delta(n)) \end{cases}$$

Został tu zastosowany schemat rekursji prostej, w którym $\vec{x} = \langle\rangle$.

Odejmowanie

(ang. cut-off subtraction, positive subtraction, monus operation)
Specyfikacja:

$$m \dot{-} n = \begin{cases} m - n, & \text{dla } m \geq n \\ 0, & \text{dla } m < n \end{cases}$$

Definicja (nieformalna):

$$\begin{cases} m \dot{-} 0 = m \\ m \dot{-} S(n) = \delta(m \dot{-} n) \end{cases}$$

W tym i następnych przykładach definicje funkcji będą podawane nieformalnie, ale należy zdawać sobie sprawę, że jest to jedynie pewien skrót notacyjny, który w razie potrzeby należy umieć zastąpić definicją w pełni formalną, zgodną z wprowdzonymi wyżej schematami definiowania funkcji rekurencyjnych.

Rozpoznawanie zera

Specyfikacje:

$$sg(n) = \begin{cases} 0, & \text{dla } n = 0 \\ 1, & \text{dla } n > 0 \end{cases} \qquad \overline{sg}(n) = \begin{cases} 1, & \text{dla } n = 0 \\ 0, & \text{dla } n > 0 \end{cases}$$

Definicje:

$$\begin{cases} sg(0) = 0 \\ sg(S(n)) = 1 \end{cases} \qquad \overline{sg}(n) = 1 \dot{-} sg(n)$$

Twierdzenie. Każda funkcja pierwotnie rekurencyjna jest całkowita (ang. total).
Dowód. Widzieliśmy, że funkcje bazowe są całkowite, złożenie funkcji całkowitych daje funkcję całkowitą oraz funkcje otrzymane z funkcji całkowitych za pomocą schematu rekursji prostej są całkowite. $\square$

Dlaczego funkcje pierwotnie rekurencyjne są ważne?

1. Złożenie funkcji i schemat rekursji prostej są łatwe do zrozumienia i proste w użyciu.
2. Większość całkowitych (ang. total) funkcji obliczalnych to funkcje pierwotnie rekurencyjne.
3. Każdą funkcję obliczalną można zdefiniować za pomocą dwóch funkcji pierwotnie rekurencyjnych, wykorzystując operację minimum i składania funkcji (postać normalna Kleene'a).

Twierdzenie (Kleene'a o postaci normalnej). Każdą funkcję częściowo rekurencyjną f można zapisać w postaci

$$f(\vec{x}) = g(\mu m.h(\vec{x}, m) = 0)$$

gdzie g, h są pierwotnie rekurencyjne.

Funkcja Ackermanna

Funkcja Ackermanna-Peter $A(m, n)$ jest uproszczeniem oryginalnej, trzyargumentowej funkcji Ackermanna.

$$
\begin{aligned}
A(0, n) &= n + 1 \\
A(m + 1, 0) &= A(m, 1) \\
A(m + 1, n + 1) &= A(m, A(m + 1, n))
\end{aligned}
$$

Powyższa definicja zawiera podwójną rekursję, która jest nieco silniejsza, niż schemat rekursji prostej. Zauważmy jednak, że przy każdym wywołaniu rekurencyjnym jeden z argumentów maleje, więc po *skończonej* liczbie wywołań obliczenia muszą się zakończyć.

Twierdzenie. Jeśli $f : \mathbb{N} \to \mathbb{N}$ jest pierwotnie rekurencyjna, to istnieje takie m, że dla wszystkich n, $f(n) < A(m, n)$, tzn. funkcja Ackermanna asymptotycznie rośnie szybciej, niż jakakolwiek funkcja pierwotnie rekurencyjna.

Mamy więc następujące

Twierdzenie. $\mathcal{PR} \subsetneq \mathcal{R}_0 \subsetneq \mathcal{R}$.

Definicja. Funkcja częściowo rekurencyjna $f : \mathbb{N}^k \to \mathbb{N}$ jest *definiowalna* w beztypowym rachunku lambda, jeśli istnieje zamknięty term F, spełniający dla dowolnych $n_1 \ldots n_k \in \mathbb{N}$ następujące warunki ($\mathbf{n} \equiv \lambda f x. f^n x$ jest liczebnikiem Churcha, reprezentującym $n \in \mathbb{N}$):

(i) Jeśli $f(n_1 \ldots n_k) = m$, to $F \mathbf{n}_1 \ldots \mathbf{n}_k =_\beta \mathbf{m}$;

(ii) Jeśli wartość $f(n_1 \ldots n_k)$ jest nieokreślona, to $F \mathbf{n}_1 \ldots \mathbf{n}_k$ nie ma postaci normalnej.

Lemat. Funkcje bazowe (początkowe) Z, S, U_n^i są λ-definiowalne.

Dowód. Weźmy następujące kombinatory jako termy definiujące.

$$\mathsf{Z} \equiv \lambda n.\mathbf{0}$$

$$\mathsf{suc} \equiv \lambda nfx.f(nfx)$$

$$\mathsf{U}_n^i \equiv \lambda x_1 \ldots x_n.x_i$$

Lemat. Funkcje λ-definiowalne są zamknięte ze względu na operację składania funkcji.

Dowód. Niech funkcje $g, h_1, \ldots, h_r$ będą λ-definiowalne odpowiednio za pomocą termów $G, H_1, \ldots, H_r$. Wówczas funkcja

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_r(\vec{x}))$$

jest λ-definiowalna za pomocą termu

$$F \equiv \lambda\vec{x}.G(H_1\,\vec{x}) \ldots (H_r\,\vec{x})$$

Lemat. Funkcje λ-definiowalne są zamknięte ze względu na rekursję prostą.

$$\begin{cases} \mathsf{Rec}\, g\, h\, \mathbf{0}\, \vec{x} = g\, \vec{x} \\ \mathsf{Rec}\, g\, h\, (\mathsf{suc}\, n)\, \vec{x} = h\, n\, (\mathsf{Rec}\, g\, h\, n\, \vec{x})\, \vec{x} \end{cases} \qquad \begin{cases} f(0, \vec{x}) = g(\vec{x}), \\ f(S(n), \vec{x}) = h(n, f(n, \vec{x}), \vec{x}), \end{cases}$$

Ideę obliczania $f(k, \vec{x})$ wyjaśnia poniższy program (porównaj z programem dla poprzednika).

$$\left.\begin{array}{l} \left.\begin{array}{l} y := 0; \\ z := g(\vec{x}); \end{array}\right\} \langle 0, g(\vec{x})\rangle \\ \textbf{while}\ y \neq n\ \textbf{do}\ (*\ z = f(y, \vec{x})\ *) \\ \textbf{begin} \\ \quad \left.\begin{array}{l} z := h(y, z, \vec{x}); \\ y := \mathsf{suc}\, y; \end{array}\right\} q\langle y, z\rangle \to \langle \mathsf{suc}\, y, h(y, z, \vec{x})\rangle \\ \textbf{end} \end{array}\right\} \begin{array}{l} n\text{-krotna iteracja} \\ \text{operacji } q \end{array}$$

$$(*\ z = f(n, \vec{x})\ *)$$

Teraz ten algorytm możemy zapisać w postaci λ-termu.

$$\mathsf{Rec} \equiv$$
$$\lambda g h n \vec{x}.\mathsf{snd}\, (\mathsf{Iter}\, n\, (\lambda p.\mathsf{pair}\, (\mathsf{suc}\, (\mathsf{fst}\, p))\, (h\, (\mathsf{fst}\, p)\, (\mathsf{snd}\, p)\, \vec{x}))\, (\mathsf{pair}\, \mathbf{0}\, (g\vec{x})))$$

Lemat. Funkcje λ-definiowalne są zamknięte ze względu na operację minimum.
Dowód. Niech funkcja g będzie λ-definiowalna za pomocą termu G. Wówczas funkcja $f(\vec{x}) = (\mu m.g(\vec{x}, m) = 0)$ jest λ-definiowalna za pomocą termu

$$\mathbf{mi} \equiv \lambda \vec{x}.\mathbf{szukaj}\,\mathbf{0}\,\vec{x} \qquad (\twoheadrightarrow_\eta \mathbf{szukaj}\,\mathbf{0})$$

gdzie
$\mathbf{szukaj} \equiv \mathbf{\Theta}\,\lambda f m \vec{x}.\mathit{if}\ \mathbf{isZero}\,(G\,\vec{x}\,m)\ \mathit{then}\ m\ \mathit{else}\ f\,(\mathbf{suc}\,m)\,\vec{x}$.
Ponieważ $\mathbf{\Theta}\,F \twoheadrightarrow F(\mathbf{\Theta}\,F)$, to

$$\mathbf{szukaj}\,m\,\vec{x} \twoheadrightarrow \mathit{if}\ \mathbf{isZero}\,(G\,\vec{x}\,m)\ \mathit{then}\ m\ \mathit{else}\ \mathbf{szukaj}\,(\mathbf{suc}\,m)\,\vec{x}$$

Ituicyjnie, $\mathbf{szukaj}\,\mathbf{0}\,\vec{x}$ zaczynając od zera poszukuje najmniejszej liczby m takiej, że $(G\,\vec{x}\,m) = 0$, co jest zgodne z definicją operacji minimum. Zasadniczo jest to reprezentacja pętli while w rachunku lambda.

Twierdzenie. Wszystkie funkcje częściowo rekurencyjne są λ-definiowalne.

Teza Churcha [Churcha-Turinga]. Każda funkcja obliczalna w nieformalnym sensie jest λ-definiowalna (rekurencyjna).

Składnia i semantyka prostego języka imperatywnego. I

Składnia abstrakcyjna
$V \in$ Zmienna
$N \in$ Liczba
$B ::= \textbf{true} \mid \textbf{false} \mid B \&\& B \mid B || B \mid \textbf{not}\ B \mid E < E \mid E = E$
$E ::= N \mid V \mid E + E \mid E * E \mid E - E \mid -E$
$C ::= \textbf{skip} \mid C; C \mid V := E \mid \textbf{if}\ B\ \textbf{then}\ C\ \textbf{else}\ C\ \textbf{fi} \mid \textbf{while}\ B\ \textbf{do}\ C\ \textbf{od}$

Semantykę denotacyjną języka zadaje się definiując funkcje semantyczne, które opisują, jak semantyka wyrażenia może być otrzymana z semantyki składowych tego wyrażenia (semantyka kompozycyjna).

Składnia i semantyka prostego języka imperatywnego. II

Funkcja semantyczna dla instrukcji C jest funkcją częściową $S_C[\![C]\!]$, zmieniającą wektor stanu programu $\mathbb{S}$ (pamięć operacyjną). Dziedzinę $\mathbb{S}$ reprezentujemy za pomocą zbioru funkcji $\sigma : \mathrm{Zmienna} \to \mathbb{Z}$.
Załóżmy dla uproszczenia, że funkcje semantyczne dla wyrażeń logicznych B i arytmetycznych E są znane. Tutaj zdefiniujemy w rachunku lambda funkcję semantyczną $S_C[\![C]\!] : \mathbb{S} \rightsquigarrow \mathbb{S}$ dla instrukcji.

Semantyka denotacyjna

$S_B[\![B]\!] : \mathbb{S} \to \mathbb{B},$ gdzie $\mathbb{B} = \{\mathtt{F}, \mathtt{T}\}$

$S_E[\![E]\!] : \mathbb{S} \to \mathbb{Z}$

$S_C[\![\textsf{skip}]\!]\sigma = \sigma$

$S_C[\![V := E]\!]\sigma \simeq \sigma[V \mapsto S_E[\![E]\!]\sigma]$

$S_C[\![C_1; C_2]\!]\sigma \simeq S_C[\![C_2]\!](S_C[\![C_1]\!]\sigma)$

$$S_C[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]\sigma \simeq \begin{cases} S_C[\![C_1]\!]\sigma & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{T} \\ S_C[\![C_2]\!]\sigma & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{F} \end{cases}$$

Składnia i semantyka prostego języka imperatywnego. III

Wykonując jednokrotnie pętlę **while** otrzymujemy poniższe równanie dla funkcji semantycznej $S_C[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!]$:

$$S_C[\![\mathbf{while}\,B\,\mathbf{do}\,C\,\mathbf{od}]\!]\sigma \simeq \begin{cases} S_C[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!](S_C[\![C]\!]\sigma) & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{T} \\ \sigma & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{F} \end{cases}$$

Z analogicznym problemem mieliśmy do czynienia w przypadku funkcji silnia. Należy znaleźć najmniejszy punkt stały funkcjonału:

$$F \equiv \lambda f.\lambda\sigma. \begin{cases} f(S_C[\![C]\!]\sigma) & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{T} \\ \sigma & \text{gdy } S_B[\![B]\!]\sigma = \mathtt{F} \end{cases}$$

czyli, używając kombinatora punktu stałego $\mathbf{\Theta}$ (lub $\mathbf{Y}$):
$S_C[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!] \simeq \mathbf{\Theta}F$
Istnienie najmniejszego punktu stałego gwarantuje teoria dziedzin.

1. Wykorzystując term **if** zdefiniuj pozostałe spójniki logiczne.
2. Udowodnij, że termy **0**, **suc**, **Iter** spełniają specyfikację algebraiczną ze strony 8.
3. Niech $G \equiv \lambda yf.f(yf)$. Udowodnij, że $M \in \Lambda$ jest kombinatorem punktu stałego wtw, gdy $M = GM$, tzn. wtw, gdy M jest kombinatorem punktu stałego dla G.
4. Niech $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, $G \equiv \lambda yf.f(yf)$. Udowodnij, że wszystkie kombinatory zdefiniowane następująco:

$$\begin{cases} \mathbf{Y}^0 \equiv \mathbf{Y} \\ \mathbf{Y}^{n+1} \equiv \mathbf{Y}^n G \end{cases}$$

   są kombinatorami punktu stałego.
5. Pokaż, że $\mathbf{Y}^1 \twoheadrightarrow_\beta \Theta$.
6. Zdefiniuj kombinator $\boldsymbol{\Theta}$ w Haskellu i OCamlu.

7. Pokaż, że poniższe funkcje są pierwotnie rekurencyjne.
   (a) $m \cdot n$ = iloczyn argumentów
   (b) m^n = m do potęgi n, gdzie $0^0 = 1$
   (c) $\min(m, n)$ = mniejszy z argumentów
   (d) $\max(m, n)$ = większy z argumentów
   (e) $\min_k(n_1, \ldots, n_k)$
   (f) $\max_k(n_1, \ldots, n_k)$
   (g) $|m - n|$ = wartość bezwzględna różnicy
   (h) $n!$
   (i) $m \,\%\, n$ = reszta z dzielenia m przez n (gdzie $m\%0 = m$)
   (j) $\lfloor m/n \rfloor$ = część całkowita ilorazu (gdzie $\lfloor m/0 \rfloor = 0$)

   Dla wszystkich wartości argumentów zachodzi zwykły związek
   $m = \lfloor m/n \rfloor * n + m\%n$.

8. Udowodnij, że funkcja

$$\sum_{n<k} f(n,\vec{x}) = \begin{cases} 0, & \text{gdy } k = 0 \\ f(0,\vec{x}) + \cdots + f(k-1,\vec{x}), & \text{gdy } k > 0 \end{cases}$$

jest pierwotnie rekurencyjna (przy założeniu, że f jest pierwotnie rekurencyjna).

9. Udowodnij, że klasa funkcji pierwotnie rekurencyjnych jest zamknięta ze względu na definicje warunkowe

$$f(\vec{x}) \quad = \quad \begin{cases} h_1(\vec{x}), & \text{gdy } R_1(\vec{x}) \\ h_2(\vec{x}), & \text{gdy } R_2(\vec{x}) \\ \dots\dots\dots\dots\dots & \\ h_m(\vec{x}), & \text{gdy } R_m(\vec{x}) \end{cases}$$

gdzie h_i i funkcje charakterystyczne relacji R_i $(1 \leq i \leq m)$ są pierwotnie rekurencyjne, a relacje $R_i(n,\vec{x})$ wykluczają się wzajemnie oraz wyczerpują wszelkie możliwości.