**math.e**

***Hrvatski matematički elektronički časopis***

# Lambda račun kao osnova funkcijskog programiranja

**funkcijsko programiranje** **Haskel**

Lovro Rožić Jan Šnajder Mladen Vuković

lorozic33@gmail.com jan.snajder@fer.hr vukovic@math.hr

**Sažetak.** Funkcijsko programiranje stil je programiranja koji se zasniva na izračunavanju funkcija. U ovome članku izlažemo teorijske osnove funkcijskog programiranja. Za razliku od imperativnog programiranja, koje kao teorijski model izračunavanja koristi Turingov stroj, funkcijsko programiranje kao osnovu koristi $\lambda$-račun. Dok Turingov stroj koristi promjenu stanja kao postupak izračunavanja, $\lambda$-račun sastoji se isključivo od primjena funkcija te korištenja njihovih povratnih vrijednosti. Kao primjer konkretnog funkcijskog programskog jezika, u članku opisujemo Haskell, moderan funkcijski programski jezik koji se temelji na tipiziranom $\lambda$-računu.

## 1 Uvod

Na samome početku razvoja računala i programskih jezika, 40-ih godina prošlog stoljeća, bilo je prirodno da programski jezik bude u skladu s arhitekturom računala. Pošto su osnovni dijelovi računala procesor i memorija, jasno je da se program sastojao od instrukcija koji su mijenjale sadržaj memorije. Iako se u konačnici svi programi prevode u strojni kôd te manipuliraju memorijom i registrima procesora, s vremenom su se razvile brojne apstrakcije i različiti pristupi strukturiranju samog koda te načinu razmišljanja o procesu izračunavanja. Takvi apstraktni modeli omogućili su predviđanje rezultata izvršavanja nekog programa bez znanja o konkretnom načinu na koji je rezultat dobiven u hardveru. *Imperativni stil* programiranja jedan je od prvih pristupa, te je kao takav i danas konceptualno blizak hardveru. Tako se programski kôd imperativnih programskih jezika, poput Pascala

ili C-a, sastoji od niza instrukcija, a rješavanje problema odnosno *izračunavanje* svodi se na slijedno izvođenje instrukcija koje mijenjaju sadržaj memorije ili mijenjaju tijek izvođenja programa.

Međutim, postupci za rješavanje raznih matematičkih i analitičkih problema postojali su i mnogo prije pojave prvog računala (isto vrijedi i za same koncepte izračunavanja i izračunljivosti). Vrlo važnu ulogu u matematici imaju funkcije. Možemo reći da funkcije izražavaju vezu između dvaju skupova vrijednosti, tj. da svakoj vrijednosti nekog *ulaznog* skupa pridružuju vrijednost u *izlaznom* skupu. Funkcije su vrlo dobar način za razmatranje pojma izračunavanja, te su one osnova *funkcijskog programiranja*. U funkcijskom programiranju programski kôd sastoji se od definicije jedne ili više funkcija, a izvođenje programa svodi se na izračunavanje funkcijskih izraza. Funkcijski programski jezici pokušavaju biti usklađeni s matematičkom tradicijom, udaljavajući se time od konkretne računalne arhitekture na kojima se programi izvode. Udaljavanje od arhitekture povijesno je kao posljedicu imalo smanjene performanse funkcijskih jezika (u odnosu na imperativne) te su funkcijski jezici dugo igrali marginalnu ulogu u svijetu računarstva van akademske zajednice.

S druge strane, prednost funkcijskih jezika jest u tome što su svojom strukturom bliži ljudskom poimanju nego računalima. Kako računala postaju sve jeftinija a rad programera sve skuplji, praktična vrijednost funkcijskog programiranja raste te je vjerojatno da će u budućnosti funkcijsko programiranje biti sve zastupljenije. Iz tih razloga, teorija funkcijskih jezika danas je vrlo aktivno područje istraživanja.

Kada pričamo o teoriji funkcijskih jezika, podrazumijevaju se jezici sa sljedećim svojstvima:

- *nepromjenjivost* (engl. *immutability*) memorijskih lokacija, što znači da se ne može mijenjati vrijednost pojedinih varijabli jednom kada su inicijalizirane. Točnije, smatra se da su sve vrijednosti konstante, a jedini način za generiranje novih vrijednosti jest izračunavanje povratnih vrijednosti funkcija.
- *čistoća* (engl. *purity*) funkcija, koja garantira da funkcije neće utjecati na vrijednosti koje joj nisu direktno proslijeđene Ovo svojstvo garantira da će pokretanje neke funkcije više puta s istim argumentima generirati istu povratnu vrijednost svaki puta. Ovo svojstvo bitno je za optimizaciju programa jer se prijašnje povratne vrijednosti mogu pamtiti te ih nije potrebno ponovo računati.
- korištenje *funkcija višeg reda* (engl. *higher-order functions*), tj. funkcija koje primaju druge funkcije kao argumente te vraćaju funkcije kao povratne vrijednosti. Ovo svojstvo kao posljedicu često povlači funkcije kao "građane prvog reda", odnosno kao osnovni objekt kojim jezik manipulira.

Međutim, nerijetko se događa da neki jezik zadovoljava samo neko od tih svojstava a da ga ipak smatramo "funkcijskim", poput primjerice jezika LISP koji ima promjenjivu memoriju. Također, mnogi jezici koje ne smatramo funkcijskima implementiraju jedno ili više navedenih svojstava. U ovom ćemo se tekstu baviti *čistim* funkcijskim jezicima, tj. jezicima koji zadovoljavaju sva tri navedena svojstva, s

programskim jezikom Haskell kao primjerom čistog funkcijskog jezika.

Govoriti o funkcijskom programiranju gotovo je nemoguće bez $\lambda$-računa. Međutim, osim "programerske" strane, $\lambda$-račun ima veliku važnost u teoriji izračunljivosti.

Teorija izračunljivosti (engl. *computability theory*) nastala je u prvoj polovici 20. stoljeća inspirirana radovima Kurta Gödela o nepotpunosti matematičkih teorija, koji su dali opravdan razlog da se sumnja u rješivost nekih do tada neriješenih problema. U relativno kratkom razdoblju matematičari Alonso Church i Alan Turing predstavili su svoje modele izračunljivosti, $\{\lambda$-račun$\}$ odnosno *Turingov stroj*, te su nezavisno jedan od drugoga dali negativan odgovor na problem odlučivosti logike prvog reda (*Entscheidungsproblem*). Ubrzo je pokazano da su ti modeli, zajedno s *teorijom rekurzivnih funkcija*, ekvivalentni u smislu da definiraju istu klasu funkcija (za dokaze vidi [4] i [9]). Na temelju rezultata ekvivalentnosti, Church je iskazao tezu koja tvrdi da su svi zamislivi modeli izračunljivosti ekvivalentni, koju zovemo *Churchova teza*. Ta je teza i danas aktualna jer se za svaki naknadno predstavljen model izračunljivosti pokazalo da je ekvivalentan Turingovom stroju, odnosno $\lambda$-računu.

Članak je u nastavku podijeljen u četiri dijela. U drugome dijelu dajemo osnove netipiziranog $\lambda$-računa te razmatramo ideju redukcije, odnosno načine izračunavanja izraza. Daju se osnovna svojstva klasične $\beta$-redukcije te se opisuje kako je $\lambda$-račun moguće koristiti kao rudimentaran programski jezik. Ilustrirat ćemo kako u $\lambda$-računu možemo realizirati logiku sudova i prirodne brojeve. Na kraju točke razmatra se potpunost $\lambda$-računa (u Turingovom smislu), čime se taj sustav po ekspresivnosti poistovjećuje s ostalim standardnim modelima izračunljivosti.

U trećem dijelu nastojimo objasniti pojmove *tipa* i *tipiziranja*. Tipiziranjem se uvodi odnos funkcija–argument među $\lambda$-termima, koji netipizirana varijanta ne posjeduje. Također se formalizira ideja pridruživanja tipa izrazima, koja je ključna kako u teoriji (zbog Curry-Howardovog izomorfizma) tako i u praktičnome programiranju (zbog provjere semantičke ispravnosti programa).

U četvrtome dijelu na primjeru programskog jezika Haskell demonstriramo kako se ideje iz prethodne dvije točke realiziraju u praksi. Važno je, međutim, naglasiti da taj dio nije zamišljen kao uvod u funkcijsko programiranje, već kao demonstracija nekih njegovih ključnih pojmova.

Ovaj je članak sažetak diplomskog rada [8]. Zbog ograničenog prostora, ovdje nisu uključeni dokazi te je dano razmjerno malo primjera. Više detalja, uključivo dokaze, čitatelj može pronaći u navedenome diplomskom radu.

Kako bismo lakše objasnili metode koje kasnije formalno definiramo, za početak dajemo intuitivno

# 2 Netipizirani $\lambda$-račun

objašnjenje nekih osnovnih pojmova. Promotrimo kao primjer funkciju $f(x) = x + 2$. Važno je uočiti koje su glavne komponente ovakvog zapisa: prvo, s lijeve strane zapisujemo ime (u ovom slučaju $f$) koje dajemo funkciji; drugo, u zagradama naglašavamo koji podatak variramo, odnosno koji je podatak predmet manipuliranja o kojem ovisi krajnji rezultat; treće, s desne strane jednakosti dajemo samo tijelo funkcije, koje opisuje apstraktan postupak kojim dolazimo do konačnog rezultata. Samo ime funkcije/postupka nije od značaja, pa ga možemo izostaviti (primijetimo da smo također izostavili i podatke o domeni i kodomeni jer nas oni trenutno ne zanimaju). U $\lambda$-računu označavamo samo varijable te tijelo funkcije, a ime zanemarujemo. Zapis funkcije je tada $\lambda x.\, x + 2$ (pritom bi naravno trebalo prvo definirati što znači "2" a što "+", no za ilustraciju ideje to trenutno nije bitno). Zatim, želimo da ovakve izraze možemo primijeniti na neku vrijednost. U standardnom načinu zapisivanja, da bismo funkciju primijenili na npr. vrijednost 3, pišemo $f(3)$. U $\lambda$-računu primjena (aplikacija) se naznačava samo dopisivanjem argumenta zdesna tijelu funkcije te odvajanjem tijela od argumenta zagradama, npr. $(\lambda x.\, x + 2)3$. Ovakvim zapisom samo smo naznačili da želimo izračunati izlazni podatak, međutim sama manipulacija još nije izvedena.

Osnovna transformacija u $\lambda$-računu kojom se iz jednog izraza izvodi novi naziva se $\{\beta$-redukcija$\}$. Primjenom $\beta$-redukcije simuliramo sam postupak izračunavanja. Primjerice, izračunavanje $(\lambda x.\, x + 2)3 = 3 + 2 = 5$ svodi se na uzastopnu primjenu $\beta$-redukcije.

## 2.1 $\lambda$–termi i $\beta$-redukcija

Funkcijski programski jezici sintaktički su u osnovi vrlo slični $\lambda$-računu. Razlike se uglavnom svode na neke sintaktičke dodatke koji olakšavaju čitljivost i programiranje (primjerice, ne zahtijeva se od programera da definira operacije nad brojevima već je to ugrađeno u sam jezik). Slijedi definicija sintakse $\lambda$-računa.

**Definicija 1.** Alfabet $\lambda$-računa je skup $\{v_0, v_1, \ldots\} \cup \{\lambda, (, )\}$, pri čemu je $\{v_0, v_1, \ldots\}$ prebrojiv skup čije elemente nazivamo *varijable*. Simbol $\lambda$ nazivamo *apstraktor*. Skup svih $\lambda$-terma $\Lambda$ definiramo rekurzivno na sljedeći način:

- svaka varijabla je element skupa $\Lambda$,
- za $M, N \in \Lambda$, vrijedi $(MN) \in \Lambda$,
- za $M \in \Lambda$ i za proizvoljnu varijablu $x$, vrijedi $(\lambda x.\, M) \in \Lambda$.

Ukoliko je $\lambda$-term oblika $(MN)$ nazivamo ga *primjena* (*aplikacija*), a ukoliko je oblika $(\lambda x.\, M)$ *apstrakcija*.

Nadalje, malim slovima $x, y, \ldots$ označavamo varijable, a velikim slovima $M, N, \ldots$ {$\lambda$-terme.} Ukoliko smatramo da dva simbola $M$ i $N$ reprezentiraju isti $\lambda$-term, pišemo $M \equiv N$. U daljnjem tekstu, ukoliko nije drugačije naznačeno, izraz *term* označavat će $\lambda$-term.

Uobičajeno je terme oblika $\lambda x.\, (\lambda y.\, (\ldots \lambda z.\, M) \ldots)$ pisati kao $\lambda xy \ldots z.\, M$. Niz varijabli $x_1\, x_2\, \ldots\, x_n$ kraće označavamo s $\vec{x}$ te u tom slučaju term oblika $\lambda x_1\, .\, (\lambda x_2\, .\, (\ldots \lambda x_n\, .\, M) \ldots)$ možemo zapisati i kao $\lambda \vec{x}.\, M$. Smatramo da je primjena lijevo asocijativna: term $MNP$ interpretiramo kao $(MN)P$, i u tom slučaju ne pišemo zagrade.

Podterm nekog $\lambda$-terma je podriječ znakova nekog terma koja je i sama $\lambda$-term. Za varijablu $x$ kažemo da je *slobodna* u nekom termu $N$ ukoliko se ne pojavljuje uz apstraktor (znak $\lambda$). Varijabla je *vezana* ukoliko nije slobodna, tj. ukoliko se nalazi uz apstraktor. Primjerice, u termu $\lambda y.\, xy$ varijabla $x$ je slobodna, a varijabla $y$ vezana, dok su u termu $\lambda x.\, (\lambda y.\, xy)$ obje varijable vezane. Za term koji nema slobodnih varijabli kažemo da je *zatvoren*.

Zamjenu slobodne varijable $x$ u nekom termu $M$ nekim termom $N$ nazivamo *supstitucija*, a označavamo s $M[x := N]$. Primjerice, vrijedi $xy[x := (\lambda z.\, z)] \equiv (\lambda z.\, z)y$.

Pravilo kojim mijenjamo terme odnosno kojim terme transformiramo u neke druge terme nazivamo *redukcija*. Operacije koje ovdje definiramo omogućavaju manipulaciju termima na sintaktičkoj razini, ali nam omogućuju i intuitivniju interpretaciju: $\lambda$-izraze možemo promatrati kao algoritme, odnosno kao programe koji definiraju na koji se način neki postupak odvija. Pravilima redukcije definirat ćemo zapravo kriterije prema kojima terme smatramo "sličnima", stoga želimo da pravila budu konzistentna s pravilima izgradnje $\lambda$-terma iz definicije 1. Sljedeća definicija formalizira tu ideju.

**Definicija 2.** Za binarnu relaciju $\mathbf{R}$ na $\Lambda$ kažemo da je *kompatibilna* ako za svaka dva terma $M$ i $N$ takva da je $(M, N) \in \mathbf{R}$ vrijedi:

- $(ZM, ZN) \in \mathbf{R}, (MZ, NZ) \in \mathbf{R}$, za $Z \in \Lambda$
- $(\lambda x.\, M, \lambda x.\, N) \in \mathbf{R}$

Najvažnija redukcija koju ćemo razmatrati je $\beta$-redukcija koju sada definiramo.

**Definicija 3.** Neka je $\beta \subseteq \Lambda \times \Lambda$ binarna relacija koja je definirana sa:

$$\beta = \{((\lambda x.\, M)N, M[x := N]) \mid M, N \in \Lambda, x \text{ varijabla}\}$$

Označimo sa $\twoheadrightarrow$ kompatibilno, refleksivno i tranzitivno zatvorenje relacije $\beta$. Za term $M$ kažemo da $\beta$-reducira, ili samo kratko da reducira, u term $N$ ukoliko vrijedi $M \twoheadrightarrow N$.

Intuitivno, navedena zatvorenja relacije $\beta$ kao posljedicu imaju mogućnost reduciranja podterma pojedinih termi, tranzitivnost (ukoliko $M \twoheadrightarrow N$ i $N \twoheadrightarrow P$, tada $N \twoheadrightarrow P$), te svojstvo da za svaki term $M$ vrijedi $M \twoheadrightarrow M$.

Na primjer, vrijede sljedeće relacije:

$$(\lambda x.\, xx)(\lambda y.\, y) \twoheadrightarrow (\lambda y.\, y)(\lambda y.\, y) \qquad \text{i} \qquad (\lambda xy.\, x)(\lambda z.\, z) \twoheadrightarrow \lambda y.\, (\lambda z.\, z) \equiv \lambda yz.\, z,$$

jer se term s desne strane može generirati primjenom supstitucije na termu s lijeve strane.
Kako je jedan od ciljeva $\lambda$-računa formaliziranje algoritama, pojmovi "beskonačne petlje" te "završetka algoritma" također su od velike važnosti. Sada ćemo definirati *normalnu formu* terma, koja reprezentira završni rezultat izvršavanja nekog algoritma. Analogno, nepostojanje normalne forme nekog terma možemo interpretirati kao beskonačnu petlju.

**Definicija 4.** Term $M$ nazivamo *redeks* ako postoji term $N$ tako da vrijedi $(M, N) \in \beta$. Za term $P$ kažemo da je u $\beta$-normalnoj formi ukoliko ne sadrži podterm koji je redeks.

Kažemo da neki term $M$ ima $\beta$-normalnu formu ako postoji term $N$ u $\beta$-normalnoj formi $N$ tako da vrijedi $M \twoheadrightarrow N$.

Razmotrimo nekoliko primjera terma i njihovih $\beta$-normalnih formi. Termi $\lambda x.\, x$ i $\lambda xyz.\, xz(yz)$ su u $\beta$-normalnoj formi jer ne postoji termi u koje se reduciraju. Term $(\lambda x.\, x)(\lambda x.\, x)$ nije u $\beta$-normalnoj formi jer možemo primijeniti $\beta$-redukciju. Očito vrijedi $(\lambda x.\, x)(\lambda x.\, x) \twoheadrightarrow (\lambda x.\, x)$, pa je term $\lambda x.\, x\beta$-normalna forma danog terma. Slično, term $(\lambda xyz.\, zyx)(\lambda xy.\, yx)(\lambda x.\, x)(\lambda x.\, x)$ nije u $\beta$-normalnoj formi, no primjenom $\beta$-redukcije dobivamo redom:

$$(\lambda xyz.\, zyx)(\lambda xy.\, yx)(\lambda x.\, x)(\lambda x.\, x) \twoheadrightarrow (\lambda x.\, x)(\lambda x.\, x)(\lambda xy.\, yx) \twoheadrightarrow (\lambda x.\, x)(\lambda xy.\, yx) \twoheadrightarrow \lambda xy.\, yx$$

Posljednji term je $\beta$-normalna forma početnog terma $(\lambda xyz.\, zyx)(\lambda xy.\, yx)(\lambda x.\, x)(\lambda x.\, x)$.

Navedimo i primjer jednog terma koji nema $\beta$-normalnu formu. Neka je $\omega \equiv \lambda x.\, xx$ , te $\Omega \equiv \omega\omega$ . Budući da vrijedi $\Omega \equiv \omega\omega \equiv (\lambda x.\, xx)\omega \twoheadrightarrow \omega\omega \equiv \Omega$ , vidimo da se term $\Omega$ reducira uvijek sam u sebe, pa taj term nema $\beta$-normalnu formu.

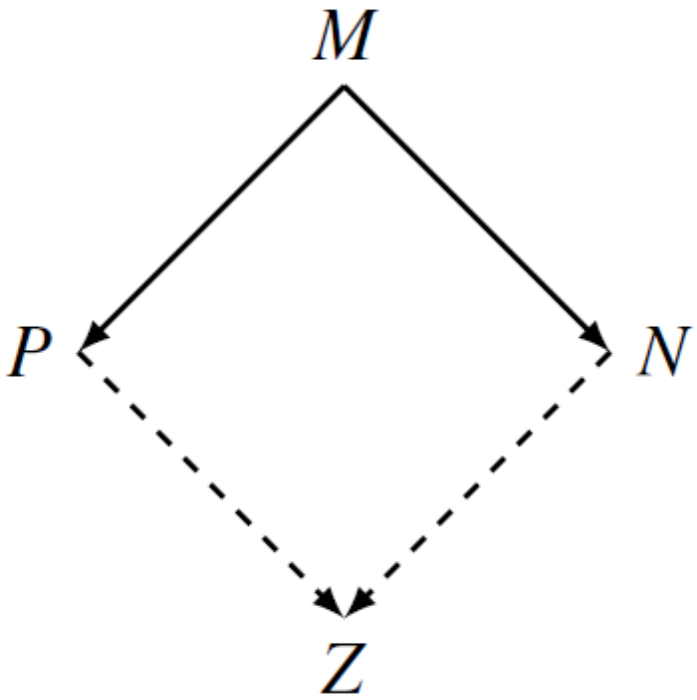
U programskim jezicima $\beta$-redukcija se ne koristi direktno jer ne definira strogi redoslijed reduciranja, zbog čega se javlja svojevrsni nedeterminizam. Međutim, redukcije koje se koriste u suštini su $\beta$-redukcije s ograničenim uvjetima korištenja, stoga ima smisla promatrati općenita svojstva ovakvih transformacija.

Pitanje postojanja $\beta$-normalne forme nekog terma često se pojavljuje u metateoriji {$\lambda$-računa.} Kao što smo već naveli, u programskim jezicima to je pitanje od velike praktične vrijednosti jer nepostojanje normalne forme uzrokuje beskonačnu petlju.

Pravilo $\beta$-redukcija slijedi neke pravilnosti koje su karakterizirane Church-Rosserovim teoremima koje ćemo sada navesti. Ovi teoremi jamče da različitim načinima redukcije nećemo doći do suviše različitih terma, odnosno uvijek ćemo moći doći do terma koji je zajednički svim smjerovima redukcije. Za funkcijske programske jezike to znači da se nikad nećemo morati vraćati, te da je svaki redukcijski put valjan u smislu da iz njega možemo doći do $\beta$-normalne forme, ukoliko ona postoji.

**Teorem 5.** *Neka su $M, N, P \in \Lambda$ takvi da vrijedi $M \twoheadrightarrow N$ i $M \twoheadrightarrow P$. Tada postoji term $Z$ takav da $N \twoheadrightarrow Z$ i $P \twoheadrightarrow Z$.*

Dakle, neovisno o tome reduciramo li term $M$ u term $N$ ili term $P$, uvijek možemo doseći neki term $Z$. Zbog grafičkog prikaza prethodnog teorema, ovakvo svojstvo naziva se još i svojstvo romba (engl. *diamond property*).




Slika 1: Grafički prikaz teorema 5

Jednostavan korolar ovog teorema jest da, ukoliko su termi $N$ i $P$ dvije $\beta$-normalne forme jednog te istog terma, tada se oni mogu razlikovati samo do na imena vezanih varijabli. To je stoga što u tom slučaju niti $N$ niti kojeg se oba mogu reducirati. Prema tome, mora se raditi o jednakim termima.

U daljnjem tekstu sa $=$ označavamo simetrično zatvorenje relacije $\twoheadrightarrow$. Intuitivno, $M = N$ vrijedi ukoliko vrijedi $M \twoheadrightarrow N$ ili $N \twoheadrightarrow M$.

Sljedeći teorem je također posljedica teorema 5. Teorem jamči da dva $\beta$-ekvivalentna terma imaju term u koji se oba mogu reducirati.

**Teorem 6.** *Neka su $M, N \in \Lambda$ te neka vrijedi $M = N$. Tada postoji term $Z$ takav da $M \twoheadrightarrow Z$ i $N \twoheadrightarrow Z$.*

Na kraju navodimo rezultat čije posljedice su posebno bitne za funkcijske programske jezike, budući da upućuje na praktičnu strategiju redukcije koja nas dovodi do završetka izvođenja algoritma.

**Teorem 7.** *Ako neki term ima $\beta$-normalnu formu, tada redukcija krajnje lijevog redeksa uvijek dovodi do normalne forme.*

Pritom, pod krajnjim lijevim redeksom nekog terma mislimo na podterm koji je redeks te čiji apstraktor je prvi slijeva u termu. Primjerice, u termu $\lambda x.\,(\lambda z.\,z)(\lambda y.\,y)(\lambda u.\,u)$, term $(\lambda z.\,z)(\lambda y.\,y)$ je krajnji lijevi redeks.

## 2.2 Kombinatori

U ovom dijelu definiramo pojam kombinatora te objašnjavamo važnost i primjenu kombinatora u funkcijskom programiranju. Pokazat ćemo na koji je način moguće definirati neke standardne kombinatore koji se koriste u funkcijskom programiranju, te kako je u $\lambda$-računu moguće definirati neke matematičke teorije poput logike sudova.

**Definicija 8.** Za $M \in \Lambda$ kažemo da je *kombinator* ako ne sadrži slobodne varijable. Skup svih kombinatora označavamo sa $\Lambda^0$.

Kombinatori su važni jer na njih ne utječe okolina, tj. njihovo je djelovanje uvijek jednako, neovisno o kontekstu u kojem se nalaze. Kombinatori su zbog toga korisni u funkcijskom programiranju jer programski prevodioc o njima može puno zaključiti već prilikom statičkog prevođenja programa. Zbog neovisnosti o kontekstu, prilikom vrednovanja izraza s kombinatorima može se zanemariti konkretan način na koji pojedini kombinator reducira te promatrati samo krajnji rezultat te redukcije. Stoga kombinatore možemo definirati tako da odredimo samo njihovo generalno djelovanje, a zanemarimo njihovu pravu strukturu. Primjerice, kombinator identiteta $\mathbf{I} \equiv \lambda x.\,x$ na sve $\lambda$-terme djeluje potpuno jednako, tako da vrati njih same kao izlaz. Stoga se taj kombinator može definirati sljedećim izrazom:

$(\forall M \in \Lambda)\ \ \mathbf{I}M = M$. Kombinator pritom shvaćamo kao klasu ekvivalencije na skupu svih terma gdje je relacija ekvivalencije funkcijska jednakost kombinatora. Funkcijsko programiranje koristi sličnu konvenciju. Kombinatori čine osnovne jedinice programa, a u kontekstu funkcijskih jezika nazivaju se funkcije.

Uz kombinatore se prirodno veže i pojam zatvorenja terma. Za proizvoljan term $M \in \Lambda$ definiramo *zatvorenje* kao term $\lambda x_1 \ldots x_n . M$, gdje je $\{x_1, \ldots, x_n\}$ skup koji sadrži sve varijable koje imaju slobodan nastup u termu $M$.

Kombinatori također imaju sljedeće važno svojstvo (dokaz u [1]): sve zatvorene $\lambda$-terme moguće je generirati djelomičnom primjenom samo jednog kombinatora samog na sebe. Međutim, najčešće se koristi sustav standardnih kombinatora koji tvore dva kombinatora $\mathbf{K}$ i $\mathbf{S}$, gdje je $\mathbf{K} \equiv \lambda xy.\, x$ i $\mathbf{S} \equiv \lambda xyz.\, (xz)(yz)$. Primjerice, kombinator identiteta $\mathbf{I}$ može se dobiti na sljedeći način:

$$\mathbf{SKK} \equiv (\lambda xyz.\, (xz)(yz))\mathbf{KK} \twoheadrightarrow \lambda z.\, (\mathbf{K}z)(\mathbf{K}z) \twoheadrightarrow \lambda z.\, (\lambda y.\, z)(\lambda y.\, z) \twoheadrightarrow \lambda z.\, z \equiv \mathbf{I}$$

## 2.3 Logika sudova

U ovom dijelu želimo ilustrirati kako u $\lambda$-računu možemo simulirati logiku sudova. Osim u tu svrhu, neke ćemo kombinatore iz ovog dijela koristiti i u daljnjim razmatranjima.

Neka je $\mathbf{T} \equiv \lambda xy.\, x$ i $\mathbf{F} \equiv \lambda xy.\, y$. Ovi su izrazi jednostavne projekcije prve odnosno druge varijable, a u $\lambda$-računu predstavljaju istinitosne vrijednosti *true* odnosno *false*. Sljedeći korak je definiranje kondicionala. Za proizvoljne $\lambda$-terme $P$, $M$ i $N$ promatramo term $PMN$. Iako tipovi nisu eksplicitno definirani te na mjesto terma $P$ može doći bilo koji $\lambda$-term, ovaj je term zamišljen tako da na mjesto terma $P$ dolazi jedna od istinitosnih vrijednosti. Jednostavno je provjeriti da vrijedi $PMN \equiv \mathbf{T}MN \twoheadrightarrow M$ i $PMN \equiv \mathbf{F}MN \twoheadrightarrow N$, što je upravo ponašanje kakvo bismo željeli od kondicionala oblika: "ako $P$ onda $M$ inače $N$". Stoga, želimo li naglasiti da term $PMN$ promatramo kao kondicional, pišemo "ako $P$ onda $M$ inače $N$" te smatramo da je taj izraz ekvivalentan termu $PMN$. Jednostavno se definiraju i logički veznici:

$$\mathbf{and} \equiv \lambda uv.\, u(v\mathbf{TF})\mathbf{F}, \qquad \mathbf{or} \equiv \lambda uv.\, u\mathbf{T}(v\mathbf{TF}) \qquad \text{i} \qquad \mathbf{not} \equiv \lambda u.\, u\mathbf{FT}$$

U gornjim izrazima zamišljeno je da se na mjesta varijabli $u$ i $v$ supstituiraju termi $\mathbf{T}$ ili $\mathbf{F}$. Kako su oni sami projekcije, ovisno o vrijednosti oni "odabiru" odgovarajuću daljnju akciju. Primjerice, logička je

disjunkcija definirana na sljedeći način: $A \vee B \iff A \equiv T$ ili $B \equiv T$. Redukcija terma **or** slijedi isti princip. Primjerice, redukcija gdje je prvi argument $\mathbf{F}$ a drugi $\mathbf{T}$ izgledala bi ovako:

$$(\mathbf{or})\mathbf{FT} \equiv (\lambda uv.\, u\mathbf{T}(v\mathbf{TF}))\mathbf{FT} \twoheadrightarrow \mathbf{FT}(\mathbf{TTF}) \twoheadrightarrow \mathbf{TTF} \twoheadrightarrow \mathbf{T}$$

## 2.4 Liste

Sada ćemo demonstrirati kako se u $\lambda$-računu mogu simulirati liste te kako se njima može manipulirati. Lista kakvu ćemo ovdje definirati vrlo je slična onoj kakva se koristi u funkcijskim jezicima, što nam omogućava da u funkcijskim jezicima formalno rasuđujemo o svojstvima listi.

Prvo ćemo definirati uređeni par elemenata, a tada iskoristiti tu definiciju kako bismo definirali listu. Uređeni par terma $M$ i $N$, u oznaci $[M, N]$, definiramo sa $[M, N] \equiv \lambda z.\, zMN$ . Sada listu od $n$ proizvoljnih terma $M_1, \ldots, M_n$ definiramo ovako:

$$[M_1, \ldots, M_n] \equiv [M_1, [M_2, \ldots, M_n]] \ .$$

Kako u funkcijskom programiranju nije moguća promjena stanja, ključno je imati strukture koje čuvaju veći broj podataka a omogućuju brzo i jednostavno manipuliranje elementima. Ovako definirane liste omogućuju dohvaćanje i ubacivanje krajnjeg lijevog elementa u konstantnom vremenu (tj. u vremenu koje ne ovisi o duljini liste). Ukoliko se ubacivanje i izbacivanje elemenata vrši uvijek s iste strane, definira se veza prethodnik-sljedbenik među elementima liste, što omogućuje jednostavnu iteraciju po listi te trivijalno korištenje nekih operacija karakterističnih za funkcijsko programiranje. Uz liste se prirodno vežu i sljedeći kombinatori: operator konkatenacije ***cons***, operator ***head***, koji vraća prvi element liste, te operator ***tail*** kojim se mogu izdvojiti svi elementi osim prvog. Operator konkatenacije možemo ostvariti na sljedeći način: $\mathbf{cons} \equiv \lambda xyz.\, zxy$ . Lako je vidjeti da vrijedi $(\mathbf{cons})MN = [M, N]$ i $(\mathbf{cons})M_1\ [M_2, \ldots, M_n] = [M_1, M_2, \ldots, M_n]$ . Operatore ***head*** i ***tail*** definiramo sljedećim izrazima: $\mathbf{head} \equiv \lambda x.\, x\mathbf{T}$ i $\mathbf{tail} \equiv \lambda x.\, x\mathbf{F}$ . Lako je vidjeti da ti operatori reduciraju na željeni način:

$$(\mathbf{head})[M, N] \equiv (\lambda x.\, x\mathbf{T})(\lambda z.\, zMN) \twoheadrightarrow (\lambda z.\, zMN)\mathbf{T} \twoheadrightarrow \mathbf{T}MN \twoheadrightarrow M$$
$$(\mathbf{tail})[M, N] \equiv (\lambda x.\, x\mathbf{F})(\lambda z.\, zMN) \twoheadrightarrow (\lambda z.\, zMN)\mathbf{F} \twoheadrightarrow \mathbf{F}MN \twoheadrightarrow N$$

Liste su elementarna struktura podataka u funkcijskim jezicima, zbog čega postoje mnoge funkcije koje

automatiziraju česte operacije nad njima. U dijelu 2.6 razmotrit ćemo neke funkcije za rad s listama.

## 2.5 Prirodni brojevi

Sada definiramo terme koji su analogoni prirodnih brojeva u $\lambda$-računu. Nazivamo ih *numerali*. Ovi termi služe kako bismo u $\{\lambda$-računu$\}$ imali izraze koji predstavljaju prirodne brojeve, a kako bismo omogućili usporedbu klasa parcijalno rekurzivnih funkcija i klase $\lambda$-definabilnih funkcija.

**Definicija 9.** Za svaki $n \in \mathbb{N}$, term $\ulcorner n \urcorner$ definiramo rekurzivno:

$$\ulcorner 0 \urcorner \equiv \mathbf{I}$$
$$\ulcorner n+1 \urcorner \equiv [\mathbf{F}, \ulcorner n \urcorner]$$

i nazivamo $n$–ti numeral.

Nakon što smo definirali analogone prirodnih brojeva, možemo definirati aritmetiku na njima.

**Definicija 10.** Funkcije sljedbenika i prethodnika definiramo sa:

$$S^+ \equiv \lambda x.\, [\mathbf{F}, x] \qquad \text{i} \qquad P^- \equiv \lambda x.\, x\mathbf{F}$$

Term **Zero**, kojim ispitujemo je li dani numeral jednak $\ulcorner 0 \urcorner$, definiramo sa: $\mathbf{Zero} \equiv \lambda x.\, x\mathbf{T}$.

Lako je vidjeti da termi reduciraju na očekivani način:

$$\begin{aligned}
S^+\ulcorner n \urcorner &\equiv (\lambda x.\, [\mathbf{F}, x])\ulcorner n \urcorner \twoheadrightarrow [\mathbf{F}, \ulcorner n \urcorner] \equiv \ulcorner n+1 \urcorner \\
P^-\ulcorner n \urcorner &\equiv (\lambda x.\, x\mathbf{F})\ulcorner n \urcorner \twoheadrightarrow \ulcorner n \urcorner \mathbf{F} \equiv (\lambda x.\, x\mathbf{F}\ulcorner n-1 \urcorner)\mathbf{F} \twoheadrightarrow \ulcorner n-1 \urcorner \\
\mathbf{Zero}\ulcorner 0 \urcorner &\equiv (\lambda x.\, x\mathbf{T})\mathbf{I} \twoheadrightarrow \mathbf{IT} \twoheadrightarrow \mathbf{T} \\
\mathbf{Zero}\ulcorner n \urcorner &\equiv (\lambda x.\, x\mathbf{T})(\lambda x.\, \mathbf{F}\ulcorner n-1 \urcorner) \twoheadrightarrow (\lambda x.\, \mathbf{F}\ulcorner n-1 \urcorner)\mathbf{T} \\
&\twoheadrightarrow \mathbf{TF}\ulcorner n-1 \urcorner \twoheadrightarrow \mathbf{F}, \text{ za } n \neq 0
\end{aligned}$$

Time smo predstavili način definiranja analogona brojeva u $\lambda$-računu te osnovnu aritmetiku na njima. Primjenom kombinatora fiksne točke, koje ćemo definirati u nastavku, mogu se razmatrati analogoni funkcija zbrajanja i množenja u $\lambda$-računu (detalje možete naći u [1]).

## 2.6 Fiksne točke i rekurzija

Rekurzija je ključan alat u funkcijskim programskim jezicima, pa ćemo se u ovome dijelu posebno posvetiti rekurziji i tome kako se ona ostvaruje. Prvo dajemo važan rezultat koji, uz svoju praktičnu vrijednost, pokazuje i neka neintuitivna svojstva netipiziranog $\lambda$-računa.

**Teorem 11.** *[Teorem o fiksnoj točki] Za svaki $F \in \Lambda$ postoji $X \in \Lambda$ tako da vrijedi $FX = X$.*

Ovaj nam teorem govori da svaki $\lambda$-term, promatran kao funkcija, ima fiksnu točku. Kao posljedica ovog teorema javlja se činjenica da svaka jednadžba oblika $FX = X$ u {$\lambda$-računu} ima rješenje. Sada ćemo definirati posebnu vrstu terma koji nam omogućava algoritamski trivijalno dobivanje rješenja ovakvih jednadžbi, a koji ustvari odgovara rekurziji u funkcijskim programskim jezicima.

**Definicija 12.** Za kombinator $Y \in \Lambda$ kažemo da je *kombinator fiksne točke* ako za svaki $\lambda$-term $X$ vrijedi $YX = X(YX)$ .

Važno svojstvo kombinatora fiksne točke jest da njihovom primjenom na bilo koji dani term dobivamo fiksnu točku tog danog terma. Sljedeći termi su kombinatori fiksne točke:
$\mathbf{Y} \equiv \lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx))$ i $\Theta \equiv (\lambda xy.\,y(xxy))(\lambda xy.\,y(xxy))$ . Ovi kombinatori zbog svoje su važnosti imenovani prema svojim autorima: $\mathbf{Y}$ je imenovan *Churchov (paradoksalni) kombinator*, a $\Theta$ *Turingov kombinator fiksne točke*.

Kao primjer korištenja kombinatora fiksne točke prezentirat ćemo definiranje funkcije koja za dani prirodni broj $n$ vraća $n$-ti Fibonaccijev broj. Dakle, želimo funkciju $F$ koja ima sljedeće svojstvo: $F(n) = F(n-2) + F(n-1)$ . To svojstvo možemo zapisati i ovako:
$F = \lambda n.\,F(n-2) + F(n-1)$ . U $\lambda$-računu ne postoji način koji bi omogućavao definiranje terma koji sam sebe referencira na sintaktičkoj razini. Stoga desnu stranu prepisujemo u obliku primjene (aplikacije), čiji je argument funkcija $F$. Time dobivamo: $F = (\lambda fn.\,f(n-2) + f(n-1))F$ . Iz toga

zaključujemo da je $F$ jedna fiksna točka sljedećeg izraza: $E \equiv \lambda fn.\, f(n-2) + f(n-1)$. Koristeći kombinator fiksne točke jednostavno je vidjeti da je $\mathbf{Y}E$ fiksna točka terma $E$, tj. vrijedi $\mathbf{Y}E = E(\mathbf{Y}E)$ .

Kao još jednu ilustraciju primjene kombinatora fiksne točke, dat ćemo opis definicija funkcija **map**, **foldl** i **foldr** za rad s listama. Kao i u prošlom primjeru, prvo definiramo "nerekurzivni" term, na koji zatim primjenjujemo kombinator fiksne točke da bismo dobili željeno djelovanje.

Počnimo s funkcijom **map**. Želimo da vrijedi sljedeće:

$$(\mathbf{map})\ l\ f \equiv (\mathbf{cons})\ \ (f\ ((\mathbf{head})\ l))\ \ ((\mathbf{map})\ f\ ((\mathbf{tail})\ l))\ ,$$

odnosno želimo da funkcija **head** izdvoji prvi element liste te ga zatim transformira funkcijom $f$. Rezultantna lista generira se nadovezivanjem izdvojenog transformiranog elementa na ostatak liste, modificiran na isti način.

Koristeći ranije opisanu metodu, vidimo da je funkcija **map** ispravno definirana sljedećim izrazom: $\mathbf{map} \equiv \mathbf{Y}(\lambda cfl.\, ((\mathbf{cons})\ \ (f\ ((\mathbf{head})\ l))\ \ c\ f\ ((\mathbf{tail})\ l))$. Vrijedi:

$$\begin{aligned}(\mathbf{map})((\mathbf{and})\mathbf{T})\,[\mathbf{T},\mathbf{L}] &\twoheadrightarrow (\mathbf{cons})\ ((\mathbf{and})\ \mathbf{T}\ ((\mathbf{head})\,[\mathbf{T},\mathbf{L}])((\mathbf{map})\ ((\mathbf{and})\mathbf{T})\ ((\mathbf{tail})\ \ [\mathbf{T},\mathbf{L}]))\\ &\twoheadrightarrow (\mathbf{cons})\ ((\mathbf{and})\ \mathbf{T}\ \mathbf{T})\ ((\mathbf{map})\ ((\mathbf{and})\mathbf{T})\ \mathbf{L})\\ &\twoheadrightarrow [\mathbf{T},(\mathbf{map})\ ((\mathbf{and})\mathbf{T})\ \mathbf{L}]\end{aligned}$$

gdje je L proizvoljna lista terma. Pritom smo koristili pravila redukcije kombinatora definirana u 2.2. Promotrimo sada kako bismo, primjerice, rekurzivno zbrojili listu brojeva. U svakom koraku rekurzije potrebno je izdvojiti element liste i pridodati ga postojećem zbroju. Zbrajanje liste je samo jedan primjer općenite operacije agregacije vrijednosti liste, kojoj odgovaraju funkcije **foldl** odnosno **foldr**. Kod takve je operacije potrebno specificirati kombinirajuću funkciju kojom akumuliramo rezultat (funkciju koja element liste kombinira s akumulatorom), početnu vrijednost akumulatora te listu elemenata. Kod zbrajanja elemenata liste, kombinirajuća funkcija je funkcija zbrajanja, početna vrijednost akumulatora je nula, a lista elemenata je lista čije elemente želimo zbrojiti. Ovisno o tome kojim redom apliciramo funkciju, razlikujemo lijevu (**foldl**) i desnu (**foldr**) varijantu:

$$\begin{aligned}(\mathbf{foldl})\ f\ e\ l &\equiv (\mathbf{foldl})\ f\ (f\ e\ ((\mathbf{head})\ l))\ ((\mathbf{tail})\ l)\\ (\mathbf{foldr})\ f\ e\ l &\equiv f\ ((\mathbf{head})\ l)\ ((\mathbf{foldr})\ f\ e\ ((\mathbf{tail})\ l)\end{aligned}$$

Konkretan term za obje funkcije može se dobiti kao u slučaju funkcije **map**.

## 2.7 $\lambda$-definabilne funkcije

U prethodim točkama vidjeli smo kako netipizirani $\lambda$-račun možemo promatrati i kao jednostavan programski jezik. U ovoj točki želimo posebno naglasiti da je $\lambda$-račun apstraktan model izračunavanja, te još više da je $\lambda$-račun ekvivalentan ostalim modelima izračunavanja (u Turingovom smislu).

Ovdje ćemo iskazati vezu između parcijalno rekurzivnih funkcija i $\lambda$-računa. Nećemo formalno definirati parcijalno rekurzivne funkcije, jer smatramo da je čitatelju poznat taj pojam (definiciju možete vidjeti u [4] i [9]). Želimo li kodirati parcijalne funkcije u {$\lambda$-računu,} potrebno je kodirati i slučaj kada funkcija nije definirana. U prvi tren čini se da bi bilo razumno poistovjetiti nepostojanje $\beta$-normalne forme s nedefiniranošću, međutim uspostavilo se da takvo definiranje dovodi do inkonzistentne teorije (vidi [1]). Stoga je kao alternativa odabran pojam rješivosti terma koji je analogan pojmu rješivosti jednadžbe, odnosno traženju inverzne funkcije.

**Definicija 13.** Za term $M \in \Lambda^0$ kažemo da je rješiv ako postoje termi $N_1, \ldots, N_n$ takvi da vrijedi: $MN_1 \ldots N_n = \mathbf{I}$, gdje je $\mathbf{I}$ kombinator identiteta.

Za term $N \in \Lambda$ kažemo da je rješiv ukoliko postoji zatvorenje od $N$ koje je rješivo.

Neka je $S \subseteq \mathbb{N}^k$. Za funkciju $f : S \to \mathbb{N}$ kažemo da je $\lambda$-definabilna ukoliko postoji term $F \in \Lambda$ takav da sve $\vec{n} \in \mathbb{N}^k$ vrijedi: $F\ulcorner\vec{n}\urcorner = \ulcorner f(\vec{n})\urcorner$ ako $\vec{n} \in S$, te je term $F\ulcorner\vec{n}\urcorner$ nerješiv inače.

Dokaz sljedećeg teorema možete vidjeti u [1], odnosno [8].

**Teorem 14.** *Funkcija je parcijalno rekurzivna ako i samo ako je $\lambda$-definabilna.*

# 3 Tipizirani $\lambda$-račun

Uočimo da sintaksa netipiziranog $\lambda$-računa za svaka dva terma $M$ i $N$ dozvoljava tvorenje terma $MN$ i $NM$, tj. ne postoji mogućnost razlučivanja funkcije od argumenta. Tipizirani račun nastoji se približiti programskim jezicima tako da omogućava podjelu podataka na funkcije i argumente te reducira skup $\lambda$-izraza koje smatramo dobro definiranim. Time se olakšava pronalaženje grešaka u kodu budući da se

već prilikom prevođenja može otkriti mnogo grešaka. Primjerice, programski jezik Haskell koristi tzv. strogi tipski sustav. To znači da su unaprijed definirani tipovi svih izraza te da se ne dopušta automatska ili implicitna promjena tipova. Na taj se način nerijetko može otkriti i greška u logici programa, što je u jezicima s fleksibilnijim tipskim sustavima teško ostvarivo. Naravno, važnost tipiziranog računa ne leži samo u praktičnosti pri implementaciji, već i u teoriji, pa se tako mogu vući netrivijalne paralele nekih teorija u logici te tipiziranih varijanti $\lambda$-računa.

Uz sintaktičke pogodnosti koje uvodi u pisanje programa, tipizirani $\lambda$-račun uzrokuje i neke probleme koji ne postoje u netipiziranim varijantama. Primjerice, kako je primjena ograničena na terme koji su međusobno u odnosu funkcija-argument, prirodno se postavlja pitanje je li neki term dobro definiran u danom računu te može li mu se algoritamski pridružiti neki tip. Sličan, ali ipak različit problem, jest i samo određivanje tipa nekog terma, što se pokazuje kao nerješiv problem u nekim sustavima tipova. Treći, specifičan problem nije izravno vezan uz funkcijsko programiranje, ali je od velike teorijske važnosti: za proizvoljan tip postavlja se pitanje postoji li term koji je tog tipa (tj. *nastanjuje* li neki term taj tip). Važnost ovog pitanja leži u Curry–Howardovom izomorfizmu, kojim $\lambda$-tipove možemo interpretirati kao teoreme, a terme koji su tog tipa kao dokaze tih teorema. U tom pogledu pronalaženje terma danog tipa ekvivalentno je dokazivanju teorema (ili čak pronalaženju algoritma koji ga rješava), dok je nepostojanje terma primjer nedokazivog teorema.

U nastavku definiramo *jednostavno tipizirani*$\lambda$-račun}. Ovaj sustav je najjednostavniji primjer tipiziranog $\lambda$-računa, pa je posebno pogodan za demonstriranje ključnih pojmova i koncepata općenitih tipiziranih sustava. Uz terme, potrebno je definirati i tipove. Za početak smatramo da imamo skup *baznih tipova*, koji može biti proizvoljan neprazan konačan ili prebrojiv skup. Uz bazne tipove, također postoji operator (u oznaci $\to$) kojim iz dvaju postojećih tipova $\sigma$ i $\tau$ tvorimo novi tip.

**Definicija 15.** Skup tipova **Typ** definiramo kao najmanji skup koji sadrži skup *baznih tipova*, te za sve $\sigma, \tau \in$ **Typ** vrijedi $(\sigma \to \tau) \in$ **Typ**.

Kao bazni tip može se uzeti npr. *nul-tip* (u oznaci **0**), ali u svrhe opisivanja programskih jezika i standardni tipovi podataka poput `Int`, `Float` itd. Uzimanje više od jednog tipa olakšava zapise, ali ne povećava ekspresivnost računa. Mi ćemo pretpostaviti da imamo prebrojiv skup baznih tipova. Kao i kod $\lambda$-terma, slučaj kada dva simbola $\sigma$ i $\tau$ reprezentiraju isti tip označavat ćemo s $\sigma \equiv \tau$.

Sustav koji definiramo sadrži iste terme kao i netipizirani $\lambda$-račun, uz dodatan zahtjev da je termu moguće pridružiti neki tip prema pravilima opisanim u sljedećoj definiciji. Označimo s $Var$ prebrojiv skup varijabli.

**Definicija 16.** Za proizvoljan tip $\sigma$ definiramo skup terma tipa $\sigma$, u oznaci $\Lambda_\sigma$. Ako je $\sigma$ neki bazni tip, tada definiramo da je $\Lambda_\sigma = Var$. Ako je $\sigma$ neki tip koji nije bazni tada je $\Lambda_\sigma$ najmanji skup koji sadrži skup $Var$ kao podskup, te zadovoljava sljedeća svojstva:

- za $M \in \Lambda_{(\tau\to\sigma)}$ i $N \in \Lambda_\tau$ vrijedi $(MN) \in \Lambda_\sigma$;
- za tip $\sigma \equiv (\tau_1 \to \tau_2)$, te za $M \in \Lambda_{\tau_1}$ i $x \in \Lambda_{\tau_2}$, vrijedi $(\lambda x.\, M) \in \Lambda_\sigma$.

Drugo pravilo u gornjoj definiciji sugerira zapis tipova koji je desno asocijativan, odnosno smatramo da tip $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$ predstavlja tip $\sigma_1 \to (\sigma_2 \to (\ldots \to \sigma_n)\ldots)$. Uočimo da jedan te isti term može pripadati u više skupova tipiziranih termi.

Konačno, definiramo skup terma jednostavno tipiziranog $\lambda$-računa.

**Definicija 17.** Skup svih jednostavno tipiziranih $\lambda$-terma $\Lambda^\tau$ definiramo kao uniju svih skupova terma nekog tipa, tj.

$$\Lambda^\tau = \bigcup_{\sigma\in\mathbf{Typ}} \Lambda_\sigma$$

Smatramo da je uz svaki *term zadan i pripadni tip.*[1] Kombinator identiteta sada bi zapisali kao $\mathbf{I}_\sigma \equiv (\lambda x.\, x) : \sigma \to \sigma$, bez eksplicitnog navođenja tipova varijabli.

Sada opisujemo što formalno znači kontekst u kojem termu pridružujemo tip, a zatim dajemo formalan sustav za deduciranje tipa iz danog konteksta.

**Definicija 18.** Neka je $M$ neki $\lambda$-term. Ukoliko vrijedi $M \in \Lambda_\sigma$ pišemo $M : \sigma$, te čitamo "*term* $M$ ima tip $\sigma$}". Niz $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, gdje su $x_1, \ldots, x_n$ međusobno različite varijable, nazivamo *kontekst* i označavamo s $\Gamma$.

Primjer dedukcije tipa nekog terma prikazan je u dodatku 6. Pojmovi $\beta$-redukcije i ekvivalencije u

potpunosti su analogni onima iz netipiziranog računa, uz neka ograničenja opisana u sljedećem poglavlju.

## 3.1 Svojstva jednostavno tipiziranog $\lambda$-računa

U ovoj točki navodimo neka svojstva tipiziranog računa. Dokazi se mogu pronaći u [3]. Lako je vidjeti da, ako vrijedi $\Gamma \vdash M : \sigma$, te ako vrijedi $\Gamma \subseteq \Gamma'$, tada $\Gamma' \vdash M : \sigma$. Važan rezultat koji vrijedi općenito za tipizirane račune jest činjenica da redukcijom ne mijenjamo tip terma.

**Teorem 19.** *Neka su $M$ i $N$ termi takvi da $M \twoheadrightarrow N$. Ako $M : \sigma$, tada je i $N : \sigma$.*

U funkcijskim programskim jezicima to znači da, ukoliko odredimo tip izraza prije evaluacije, tada znamo i njegov povratni tip. To pak znači da, jednom kada je utvrđena korektnost izraza, tipove više nije potrebno provjeravati tokom evaluacije.

Sličnost s netipiziranim računom očituje se u Church-Rosserovu teoremu, koji vrijedi i u ovom računu. Jednostavno je utvrditi da je tome tako, uzmemo li u obzir da, osim smanjenja broja mogućih terma, nismo mijenjali sintaktička svojstva $\beta$-redukcije. Međutim, sljedeći nam rezultat govori da je restrikcija koju radimo uvođenjem tipova ipak netrivijalna.

**Teorem 20.** *Neka je $M \in \Lambda^\tau$. Svaki redoslijed reduciranja podtermi terma $M$ vodi do normalne forme.*

Prethodni teorem naziva se teorem o *strogoj normalizaciji* te ima netrivijalne posljedice. Kako redukcijom jednostavno tipiziranih termi uvijek dolazimo do normalne forme, očito je da termi bez normalne forme nisu mogući u ovakvom računu, pa vrijedi $\Lambda^\tau \subset \Lambda$. Nadalje, ovakav račun nije potpun u Turingovom smislu. No, može se proširiti do računa koji je potpun u Turingovom smislu tako da dodamo kombinatore fiksne točke za svaki mogući tip računa. Kombinator fiksne točke $\mathbf{Y}$ tada ima tip $(\sigma \to \sigma) \to \sigma$. Može se pokazati da je određivanje, te provjera tipa, u jednostavno tipiziranom $\lambda$-računu odlučivo (vidi [8]).

# 4 Primjena u programskom jeziku Haskell

U ovome ćemo dijelu prethodna teorijska razmatranja povezati s jednim konkretnim programskim jezikom, koji se naziva *Haskell*.[2] Haskell je moderan funkcijski programski jezik koji, osim što se temelji na tipiziranom $\lambda$-računu, također implementira i niz drugih zanimljivih koncepata funkcijskog programiranja, poput lijene evaluacije, tipskih razreda, monada, itd. Ovdje ćemo predstaviti neke osnovne koncepte u Haskellu, dok se u dodatku 7 prezentira primjer *lijene* redukcije kakvu koristi Haskell.

Programiranje u Haskellu svodi se na definiranje potrebnih tipova koristeći skup baznih tipova te navođenje niza jednadžbi kojima definiramo kombinatore. Pritom neki izraz možemo vezati uz najviše jedno ime te jednom vezani izraz više nije moguće mijenjati. Takav sustav naziva se sustav s nepromjenjivim (engl. *immutable*) podacima. U ovoj točki predstavit ćemo sustav tipova koji koristi Haskell. Taj je sustav identičan Hindley-Milnerovom sustavu, s dodatkom *tipskih razreda*.

## 4.1 Funkcije

Funkcije su osnovne jedinice funkcijskih programskih jezika. Svi izrazi koje je u Haskellu moguće definirati jesu funkcije. Moguće je definirati funkcije prazne domene koje reprezentiraju konstante. Funkcije definiramo jednadžbama gdje slijeva navodimo ime funkcije, a zdesna tijelo kojim definiramo djelovanje. Tijelo definiramo kombinirajući postojeće funkcije pomoću kompozicije i rekurzije, te korištenjem *lambda-izraza*, koji su sintaktički srodni $\lambda$-termima, a predstavljaju anonimne funkcije (funkcije kojima nije pridjeljen identifikator). Primjerice, funkcija koja izračunava sljedbenika zadanog prirodnog broja definirana je kao:

```
let succ x = x + 1
```

## 4.2 Tipovi

Haskell koristi Hindley-Milnerov sustav tipova proširen tipskim razredima. Sustav je izvorno predstavljen u članku [7]. Hindley-Milnerov sustav tipova je *polimorfan* sustav, što znači da postoje

varijable tipova koje se mogu univerzalno kvantificirati. Haskellov je sustav proširen tipskim *razredima* te omogućuje definiranje vlastitih tipskih *konstruktora*.

### 4.2.1 Polimorfizam i tipski razredi

Postoje termi kojima se u jednostavno tipiziranom računu ne može pridružiti tip. Primjerice, u jednostavno tipiziranom $\lambda$-računu term $(\lambda y.\, yy)(\lambda x.\, x)$ nema tip. Kako bismo se to uvjerili, označimo sa $\mathbf{I}_\sigma \equiv \lambda x.\, x : \sigma \to \sigma$. Budući da vrijedi $(\lambda y.\, yy)\mathbf{I}_\sigma \to \mathbf{I}_\sigma\mathbf{I}_\sigma$, lako je vidjeti da bi trebalo vrijediti $\sigma \equiv \sigma \to \sigma$, što nije moguće.

Polimorfizam omogućuje da jedan te isti term u nekom izrazu poprimi više različitih tipova. U polimorfnom računu smatramo da $\sigma$ ne označava nužno bazni tip, već neki proizvoljni element skupa $\mathbf{Typ}$, odnosno $\sigma$ smatramo *tipskom varijablom*. Kako bismo naglasili da se radi o varijablama a ne baznim tipovima, koristimo početna slova grčkog alfabeta ($\alpha, \beta, \ldots$). Primjerice, zapis tipa polimorfnog kombinatora identiteta je $\forall\alpha.\, \alpha \to \alpha$, gdje kvantifikatorom naznačavamo da s $\alpha$ označavamo proizvoljan element skupa svih tipova.

Zamjena varijable tipa konkretnim tipom naziva se *instanciranje* tipa. U izrazu $\mathbf{I}_\alpha\mathbf{I}_\alpha$, za proizvoljan tip $\sigma$, tip desnog terma $\mathbf{I}_\alpha$ možemo instancirati u tip $\sigma \to \sigma$. Lijevi term $\mathbf{I}_\alpha$ tada je tipa $(\sigma \to \sigma) \to (\sigma \to \sigma)$.

Koristeći samo lambda-izraze, nije moguće postići da jedan te isti term istovremeno poprimi više različitih tipova u nekom izrazu. Kako bismo omogućili instanciranje polimorfnog tipa u više tipova istovremeno, koristimo izraz *let*. Sintaksa je preuzeta iz originalnog Hindley-Milnerovog računa:

```
let x = M in N
```

Primjerice, term $(\lambda y.\, yy)(\lambda x.\, x)$ tada možemo zapisati kao:

```
let y = (\x -> x) in y y
```

Definiramo li polimorfnu funkciju, u deklaraciji tipa funkcije za oznaku polimorfnih tipova koristimo mala latinična slova $a, b, \ldots$.

*Tipski razredi* omogućuju polimorfizam kod kojeg je kvantificiranje ograničeno na manji podskup

tipova. Pojam tipskog razreda analogan je pojmu sučelja u objektno orijentiranim jezicima, a pruža način grupiranja tipova prema operacijama koje su dostupne na podacima tih tipova.

Primjerice, razred `Eq` sadrži svaki tip takav da je za izraze tog tipa moguće provjeriti jednakost. Kako bi neki tip bio element razreda `Eq`, potrebno je definirati operatore uspoređivanja `==` i `/=`. Drugi često korišteni razredi su `Num`, `Ord` i `Bounded`, koji redom definiraju brojevne, uređene te omeđene tipove podataka.

Važan je i razred `Monad`, koji se u funkcijskom programiranju koristi kako bi se, primjerice, simuliralo stanje ili od *čistog* (engl. *pure*) k\^oda odvojio *nečist* (engl. *impure*) k\^od. Nečist k\^od je svaki onaj koji dovodi do *popratnih učinaka (engl. side effects) koji mijenjaju memoriju računala*. Više o monadama i njihovom korištenju može se pronaći u [5].

### 4.2.2 Tipski konstruktori i podudaranje uzoraka

U točki 3 spomenuli smo pojam *baznih* tipova te naveli da složenije tipove tvorimo korištenjem tipskih konstruktora $\rightarrow$. Haskellov sustav tipova dozvoljava definiranje vlastitih tipskih konstruktora koji nisu nužno binarne funkcije.

Standardni tipovi poput `Int`, `Bool` ili `Float` u biti su tipski konstruktori bez parametara (tj. nul-mjesne funkcije). Da bismo definirali novi tip, potrebno je također definirati i *podatkovne konstruktore*, kojima tvorimo objekte novodefiniranog tipa. Primjerice, podatkovni konstruktor `True` je nul-mjesna funkcija koja kao povratnu vrijednost ima apstraktan podatak tipa `Bool`.

Kao primjer, tip koji reprezentira listu elemenata nekog tipa ima jedan tipski konstruktor te dva podatkovna konstruktora. Konstruktor tipa ima jedan polimorfni parametar, čijom supstitucijom dobivamo neki konkretan tip, primjerice: `[Int]` (lista cijelih brojeva) ili `[[Int]]` (lista listi cijelih brojeva).

Konstrukcija same liste pomoću podatkovnih konstruktora analogna je onoj opisanoj u dijelu 2.4, uz razliku da se, uz podatkovni konstruktor `Cons`, ovdje koristi i konstruktor `Nil`, koji predstavlja praznu listu. Navedeni konstruktori imaju sljedeće pokrate: konkatenaciju označavamo dvotočkom (`:`), a praznu listu praznim uglatim zagradama (`[]`).

Promatramo li listu `[1,2,3]`, znamo da je ona samo pokrata za uzastopno korištenje konstruktora liste:

```
[1,2,3] = 1 : [2,3] = 1 : 2 : [3] = 1 : 2 : 3 : []
```

Za danu listu uvijek je moguće odrediti koji je konstruktor posljednji korišten, te s kojim parametrima (u gornjem primjeru, za konstrukciju liste posljednji je korišten operator konkatenacije s brojkom 1 te listom `[2,3]` kao parametrima).

Lako se vidi da je za svaki podatak moguće odrediti kojim konstruktorima i parametrima je konstruiran. Ta činjenica omogućuje definiranje funkcija čije djelovanje ovisi o tipu konstruktora koji je korišten za konstrukciju argumenata. Takav način definiranja naziva se *podudaranje uzoraka* (engl. *pattern matching*) jer se, prilikom definiranja funkcije, argument nastoji izjednačiti s nekim od podatkovnih konstruktora.

Podudaranje uzoraka omogućuje jednostavno definiranje rekurzivnih funkcija tako da se djelovanje funkcije podijeli na rekurzivni dio i rubne slučajeve. Primjerice, funkciju `length` koja računa broj elemenata u listi možemo definirati na sljedeći način:

```
length [] = 0 length (x:xs) = 1 + length xs
```

U prvom retku obrađuje se slučaj kada je dana lista prazna. U drugom retku rekurzivno definiramo duljinu liste građene operatorom konkatenacije, gdje $x$ predstavlja element koji se dodaje u listu a $xs$ ostatak liste. Vrijedi:

```
length [1,2] = length (1:2:[]) = 1 + length (2:[]) = 1 + 1 + length [] = 1 + 1 + 0 = 2
```

Podudaranje uzoraka funkcionira i na konstruktorima bez parametara (nul-mjesnim konstruktorima). Primjerice, funkcija negacije može se napisati kao:

```
not True = False not False = True
```

U ovom primjeru provjerava se je li argument izgrađen podatkovnim konstruktorom `True` ili `False` te, ovisno o tome, odabire odgovarajuća povratna vrijednost.

## 5 Zaključak

U ovome smo članku dali pregled funkcijskog programiranja kroz prizmu $\lambda$-računa i programskoga

jezika Haskell. Funkcijsko je programiranje stil programiranja u kojemu se izvođenje programa svodi na primjenu matematičkih funkcija. Za razliku od imperativnog stila programiranja, kod kojega se problem rješava slijednim izvođenjem instrukcija koje mijenjaju stanje računala, kod funkcijskog se programiranja program izvodi izračunavanjem funkcijskih izraza bez promjena stanja računala, što je bliže matematičkom načinu rješavanja problema.

U teorijskom smislu, funkcijsko se programiranje zasniva na $\lambda$-računu, modelu koji izračunavanje opisuje kao izračunavanje funkcija. Formalan jezik $\lambda$-računa sastoji se od terma, primjena terma i apstrakcija. Izračunavanje se u $\lambda$-računu ostvaruje sintaktičkom transformacijom terma pomoću pravila $\beta$-redukcije. Tipizirani $\lambda$-račun dodatno uvodi tipove terma, čime se doduše ograničava skup dopuštenih terma, ali se također omogućava lakše nalaženje pogrešaka i učinkovitije izvođenje programa. Haskell je primjer modernoga funkcijskog programskog jezika koji se temelji na tipiziranome $\lambda$-računu, a koji uključuje i mnoge druge napredne koncepte funkcijskog programiranja, kao što je lijena evaluacija.

Funkcijsko programiranje i $\lambda$-račun temelj su računarske znanosti, no njihov značaj nikako nije isključivo teorijske prirode. Haskell i drugi funkcijski programski jezici sve su prisutniji u inženjerskoj praksi te svoju primjenu nalaze u mnogim područjima u kojima su ranije dominirali imperativni programski jezici. Istovremeno, imperativni programski jezici preuzimaju neke koncepte funkcijskih programskih jezika, kao što su, primjerice, funkcije višeg reda, lijena evaluacija ili rad s listama. Uvažavajući značaj koji $\lambda$-račun i funkcijsko programiranje imaju u obrazovanju studenata računarstva, na Sveučilištu u Zagrebu održavaju se već niz godina kolegiji posvećeni toj tematici: *Matematička logika u računarstvu*[3] na Matematičkome odsjeku Prirodoslovno-matematičkoga fakulteta te *Programiranje u Haskellu*[4] na Fakultetu elektrotehnike i računarstva. Više od 200 studenata je odslušalo te kolegije, od kojih su mnogi nastavili samostalno razvijati svoje vještine funkcijskog programiranja.

## Bibliografija

[1] H. P. Barendregt, *The Lambda Calculus : Its Syntax and Semantics*, North Holland, Amsterdam, 1984.

[2] H. P. Barendregt, *The Impact of the Lambda Calculus in Logic and Computer Science*, working material, International Summer School, Marktoberdorf, 1997.

[3] H. P. Barendregt, W. Dekkers, R. Statman, *Lambda Calculus with Types*, http://www.cs.ru.nl/ henk/book.pdf

[4] M. Doko, V. Novaković, *Izračunljivost i apstraktni strojevi*, math.e, br. 9, 2006. http://e.math.hr/old/indexno9.html

[5] M. Lipovača, *Learn You a Haskell for Great Good: A Beginner's Guide*, No Starch Press, San Francisco, 2011.

[6] J. Launchbury, *A Natural Semantics for Lazy Evaluation*, Glasgow University, 1993.

[7] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17 (1978), 348-375

[8] L. Rožić, *Funkcijsko programiranje*, diplomski rad, PMF–MO, Zagreb, 2014. https://www.math.pmf.unizg.hr/sites/default/files/pictures/rozic-funkcijsko-programiranje.pdf

[9] M. Vuković, Izračunljiovst, skripta, PMF-MO, 2009. https://www.math.pmf.unizg.hr/sites/default/files/pictures/izn-skripta-2009.pdf

[10] P. Wadler, J. Maraist, M. Odersky, *The Call–by–Need Lambda Calculus*, Journal of Functional Programming 8 (3) (1998), 275-317

[11] J. B. Wells, *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, In Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (1994), 176–185

# 6 Dodatak A

Ovdje definiramo pravila dedukcije tipa zadanog terma te dajemo primjer dedukcije.

**Definicija 21.** Neka je $M \in \Lambda$. Ukoliko pomoću pravila dedukcije možemo zaključiti da vrijedi $M : \sigma$, uz neki kontekst $\Gamma$, pišemo $\Gamma \vdash M : \sigma$. U slučaju praznog skupa $\Gamma$ pišemo samo $\vdash M : \sigma$. Pravila dedukcije su sljedeća:

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \qquad \frac{\Gamma \vdash M{:}\sigma \to \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P{:}\tau}{\Gamma \vdash \lambda x.P{:}\sigma \to \tau}$$

Za ilustraciju promotrimo sada dedukciju tipa za term $\mathbf{S} \equiv \lambda xyz.\, xz(yz)$. Označimo s $\Gamma$ sljedeći kontekst: $x : \sigma \to \tau \to \theta, y : \sigma \to \tau, z : \sigma$. Ove pretpostavke potrebne su da bismo odredili tip

podterma terma $\mathbf{S}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\vdash x:\sigma\to\tau\to\theta \quad \Gamma\vdash z:\sigma}{\Gamma\vdash xz:\tau\to\theta} \quad \dfrac{\Gamma\vdash y:\sigma\to\tau \quad \Gamma\vdash z:\sigma}{\Gamma\vdash yz:\tau}}{\Gamma\vdash xz(yz):\theta}}{x:\sigma\to\tau\to\theta,y:\sigma\to\tau\vdash\lambda z.xz(yz):\sigma\to\theta}}{x:\sigma\to\tau\to\theta\vdash\lambda yz.xz(yz):(\sigma\to\tau)\to\sigma\to\theta}}{\vdash \lambda xyz.\, xz(yz) : (\sigma \to \tau \to \theta) \to (\sigma \to \tau) \to \sigma \to \theta}$$

Iako su nam potrebne pretpostavke za dedukciju tipova pojedinih podterma terma $\mathbf{S}$, vidimo da u konačnici tip samog terma $\mathbf{S}$ možemo deducirati iz praznog skupa.

# 7 Dodatak B

Haskell implementira tzv. *lijenu evaluaciju* (engl. *lazy evaluation*) ili *poziv po potrebi* (engl. *call-by-need*). Kod te se evaluacijske strategije izrazi izračunavaju tek onda kada su doista potrebni, što ima niz prednosti (izbjegavanje izračunavanja nepotrebnih izraza, izračunavanje s beskonačnim podatkovnim strukturama). Poziv po potrebi ostvaruje se kombinacijom tehnika *poziva po imenu* (engl. *call by name*) i dijeljenjem redeksa.

## 7.1 Poziv po imenu

Prisjetimo se da prilikom definiranja $\beta$-redukcije (odjeljak 2.1) nismo definirali strogi redoslijed reduciranja. Dva jednostavna načina uvođenja strogog redoslijeda jesu reduciranje krajnje desnog redeksa (*aplikativan redoslijed*) te reduciranje krajnje lijevog redeksa (*normalan redoslijed*). Teorem 7 jamči da, ukoliko normalna forma postoji, tada normalna redukcija vodi do normalne forme.

Poziv po imenu odgovara redukciji normalnim redoslijedom. U kontekstu izvođenja programa, to znači da će se prvo evaluirati funkcijski dio, a tek onda argument. Međutim, kod lijene evaluacije redukcija se ne provodi skroz do normalne forme, već do *slabe početno normalne forme* (engl. *weak head normal form*, WHNF). Neformalno, term je u slaboj početno normalnoj formi ako njegov najljeviji podterm (glava) nije redeks ili ako je oblika apstrakcije.

Redukcija prijevodom po imenu primjer je nestriktne redukcije. Ukoliko s $\perp$ označimo izraz koji je nedefiniran (što uključuje i beskonačne podatkovne strukture), neformalno možemo definirati striktnu redukciju kao redukciju u kojoj je rezultat evaluacije $\perp$ kao argumenta također $\perp$. Točnije, redukcija je *striktna* ako za svaku funkciju $f$ vrijedi: $f\perp = \perp$. U suprotnom kažemo da je redukcija *nestriktna*. Poziv po imenu očito je nestriktna redukcija, budući da vrijedi $(\lambda xy.\, y)\Omega \twoheadrightarrow \lambda y.\, y$. Međutim, vrijedi i više: $(\lambda xy.\, x)\Omega \twoheadrightarrow \lambda y.\, \Omega$. Rezultantni je term u WHNF. To nam govori da je moguće da term sadrži nedefiniran izraz, a da evaluacija cijelog izraza ipak završi. Promotrimo sada kakve posljedice to svojstvo ima na funkcijske programske jezike.

U kontekstu programa, smatramo da je izraz potrebno evaluirati ukoliko se vrijednost tog izraza koristi u ulazno-izlaznoj operaciji ili podudaranju uzoraka u podatkovnom konstruktoru. Za funkcije koje se bave ulazom i izlazom kažemo da su *nečiste*, u smislu da mogu mijenjati stanje memorije računala. Prema tome svaka funkcija čija povratna vrijednost se primjerice ispisuje, mora se evaluirati do normalne forme. Vrijednosti koje se ne zatraže ne evaluiraju se.

Podatkovne konstruktore i konstante interpretiramo kao terme u normalnoj formi, pa su nužno i u WHNF. Shvatimo li djelomično primijenjene funkcije u Haskellu kao analogone apstrakcije u $\lambda$-računu, smatramo da je izraz u WHNF ukoliko je djelomično primijenjena funkcija. Uočimo da tu pripadaju i djelomično primijenjeni podatkovni konstruktori.

Evaluacija izraza u Haskellu može stati u mnogo različitih međustanja evaluacije terma, između potpuno neodređenog izraza i normalne forme. Razina evaluacije izraza ovisi o količini podataka koji se zatraže. Ono što sva međustanja dijele jest da su izrazi uvijek u WHNF. Kao primjer promotrimo izraz kojim se definira beskonačna lista prirodnih brojeva:

```
let nats x = x : nats (x+1) in nats 1
```

Promotrimo prvih nekoliko koraka evaluacije tog izraza:

```
nats 1 = 1 : nats (1+1) = 1 : nats 2 = 1 : 2 : nats (2+1)
```

Uočimo da je u svakom koraku krajnji lijevi izraz aplikacija konstruktora liste. Prema definiciji, to znači da je izraz u svakom koraku u WHNF. Zahvaljujući evaluaciji normalnim redoslijedom te nestriktnosti evaluacije, ukoliko u nekom trenu zatražimo prvih $n$ elemenata liste, evaluacija izraza će završiti iako dio izraza sadrži beskonačnu listu.

## 7.2 Dijeljenje redeksa

Vidjeli smo da poziv po imenu odgovara redukciji normalnim redoslijedom. Međutim, negativno svojstvo normalne redukcije jest da broj koraka redukcije nije nužno optimalan. Primjerice, u izrazu $(\lambda x.\, xx)M \twoheadrightarrow MM$ term $M$ se inicijalno pojavljuje samo jednom, dok se nakon redukcije normalnim redom pojavljuje dva puta. Ukoliko je $M$ i sam redeks, dupliciranjem smo sigurno povećali broj koraka potrebnih da se izraz reducira u normalnu formu. Uočimo da bi aplikativnim redom term $M$ reducirali samo jednom.

Dijeljenje redeksa metoda je kojom se smanjuje broj koraka potrebnih da izraz reducira. Kako bismo dobili označene redekse, u svakom se redeksu prvom znaku $\lambda$ slijeva pridružuje indeks. Gornji primjer sada transformiramo u sljedeći izraz:

$$(\lambda x.\, xx)\,((\lambda y.\, y)M) \equiv (\lambda_1 x.\, xx)\,((\lambda_2 y.\, y)M)$$

Označimo korak redukcije u kojem kontraktiramo redeks indeksa $i$ sa $\twoheadrightarrow_i$. Sada vrijedi:

$$(\lambda_1 x.\, xx)\,((\lambda_2 y.\, y)M) \twoheadrightarrow_1 ((\lambda_2 y.\, y)M)\;((\lambda_2 y.\, y)M) \twoheadrightarrow_2 MM$$

Na taj smo način u drugom koraku reducirali oba redeksa istovremeno. Intuitivna predodžba metode jest da umjesto redeksa indeksa $2$ pamtimo pokazivač na taj redeks. Nakon reduciranja indeksiranog terma, pokazivač pokazuje na konačan oblik terma. Zbog reprezentacije označenih redeksa kao čvorova grafa te reprezentacije pokazivača kao bridova, ova metoda naziva se i *redukcija grafa* (engl. *graph reduction*). Može se pokazati da ovakva redukcija nikad neće trebati više koraka od redukcije aplikativnim redoslijedom.

Term iz prethodnog primjera pomoću izraza *let* zapisali bismo kao: $\mathrm{let}\ x = N\ \mathrm{in}\ xx$ . Izrazi $\mathrm{let}\ x = N\ \mathrm{in}\ M$ i $(\lambda x.\, M)N$ su ekvivalentni, u smislu da reduciraju u isti term.

---

[1]To je tzv. *Curryeva varijanta* pridjeljivanja tipova $\lambda$-termima.
[2]Instalacijski paket za Haskell možete preuzeti sa http://www.haskell.org/platform/ .
[3] https://www.math.pmf.unizg.hr/hr/matematicka-logika-u-racunarstvu
[4] http://www.fer.unizg.hr/predmet/puh

ISSN 1334-6083