SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Lovro Rožić

FUNKCIJSKO PROGRAMIRANJE

Diplomski rad

Voditelji rada:
prof. dr. sc.
Mladen Vuković
doc. dr. sc.
Jan Šnajder

Zagreb, veljača 2014.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Teorija izračunljivosti 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 nerješenih problema. U relativno kratkom razdoblju, matematičari Alonso Church i Alan Turing predstavili su svoje modele izračunljivosti, λ-račun i Turingove strojeve, te nezavisno 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 reprezentiraju istu klasu funkcija. Na temelju rezultata ekvivalentnosti, Church je iskazao tezu koja tvrdi da su svi zamislivi modeli izračunljivosti ekvivalentni, koju zovemo *Curchova teza*. Ta je teza i danas aktualna jer se za svaki naknadno predstavljen model izračunljivosti pokazalo da je ekvivalentan Turingovim strojevima, odnosno λ-računu.

U ovom radu bavimo se teorijskom podlogom funkcijskog programiranja. Za razliku od imperativnog programiranja, koje kao osnovu koristi Turingov stroj, funkcijsko programiranje kao osnovu koristi λ-račun. Dok Turingov stroj koristi promjenu stanja kao proces izračunavanja, λ-račun sastoji se isključivo od aplikacija funkcija te korištenja njihovih povratnih vrijednosti. Poznati primjeri funkcijskih programskih jezika su LISP i ML, a u novije doba Haskell, OCaml (nadogradnja ML-a), F# i drugi.

Rad je podijeljen u tri dijela: netipizirani λ-račun, tipizirani λ-račun i Haskell. Prva dva poglavlja zamišljena su kao uvod u teorijsku podlogu funkcijskih jezika. S iznimkom točke o kombinatorima, prvo poglavlje sadržajem prati knjigu [1], dok je drugo poglavlje bazirano na radu [2].

Prvo poglavlje bavi se idejom redukcije, odnosno načinom izračunavanja izraza. Daju se osnovna svojstva klasične β-redukcije te se opisuje kako je λ-račun moguće koristiti kao rudimentaran programski jezik. Na kraju poglavlja se dokazuje Turing-potpunost λ-računa čime se taj sustav po ekspresivnosti poistovjećuje s ostalim standardnim modelima izračunljivosti.

Drugo poglavlje nastoji objasniti pojam tipa i tipiziranja. Tipiziranjem se uvodi odnos funkcija-argument među λ-termima, kojeg netipizirana varijanta ne posjeduje. Također se formalizira ideja pridruživanja tipa izrazima, koja je ključna kako u teoriji (zbog Curry-Howardovog izomorfizma) tako i u praktičnom programiranju. Glavni rezultat je Hindley-

Milnerov algoritam koji određuje tip proizvoljnog jednostavno tipiziranog λ-terma.

Ideje prva dva poglavlja opisane su iz teorijske perspektive. Treće poglavlje zamišljeno je da na primjeru programskog jezika Haskell objedini prva dva poglavlja, te demonstrira kako se opisane ideje realiziraju u praksi. Poglavlje nije zamišljeno kao uvod u funkcijsko programiranje, već kao demonstracija nekih ključnih pojmova. Naglasak je na *lijenoj* evaluaciji izraza i njenom korištenju prilikom rješavanja problema, te na proširenom Hindley-Milnerovom sustavu tipova koji omogućuje ne samo automatsko određivanje tipa svih izraza, već i definiranje vlastitih tipova i *klasa* tipova.

Ovim putem želio bih se zahvaliti voditeljima rada, prof. dr. sc. Mladenu Vukoviću i doc. dr. sc. Janu Šnajderu na iznimnom strpljenju i brojnim primjedbama za poboljšnje ovog rada. Također, posebno zahvaljujem prof. dr. sc. Mladenu Vukoviću na inspirativnim predavanjima iz teorije računarstva, te na prijedlogu teme ovog rada, zbog kojih sam otkrio zanimanje za ovo područje znanosti.

Poglavlje 1

Netipizirani λ-račun

1.1 Osnovne definicije λ-računa

Osnovna ideja netipiziranog λ-računa kojeg ovdje predstavljamo je na apstraktan način opisati promjene nekog podatka tokom izvođenja nekog algoritma, s time da je naglasak isključivo na samom procesu izračunavanja, a standardna programerska oruđa poput uvođenja tipova zasada potpuno zanemarujemo.

Kako bismo lakše objasnili metode koje kasnije formalno definiramo, za početak dajemo intuitivno objašnjenje nekih pojmova. Promotrimo 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 predstavlja apstraktan proces kojim dolazimo do konačnog rezultata. Samo ime funkcije/procesa nije relevantno, pa ga možemo izostaviti (primijetimo da smo izostavili i podatke o domeni i kodomeni, oni nas trenutno ne zanimaju). U λ-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 „+" ali nije bitno za prezentaciju ideje). Zatim, želimo da ovakve izraze možemo primijeniti na nekakav podatak. U standardnom načinu zapisivanja pišemo $f(3)$ želimo li primijeniti funkciju na konkretan podatak. U λ-računu aplikacija se naznačava samo dopisivanjem argumenta zdesna tijelu, i odvajajući tijelo od argumenta zagradama, primjerice $(\lambda x.x + 2)3$. Ovakvim zapisom samo smo naznačili da želimo izračunati izlazni podatak, međutim sama manipulacija još nije izvedena. Osnovna metoda kojom iz jednog izraza dobivamo novi u λ-računu naziva se β-redukcija. Primjenom β-redukcije simuliramo sam proces izračunavanja:

$$(\lambda x.x + 2)3 = 3 + 2 = 5$$

Važno je još naglasiti razliku standardnog zapisa $f(3)$ i aplikacije: aplikacija samo nago-

viješta operacije koje je potrebno izvršiti, dok u standardnom zapisu smatramo da je $f(x)$ element kodomene funkcije f, tj. da su transformacije već izvedene.

Funkcijski jezici sintaktički su vrlo srodni λ-računu. Razlike se uglavnom svode na značajke koje olakšavaju čitljivost i omogućavaju veću ekspresivnost (primjerice, ne zahtijeva se od programera da definira operacije nad brojevima već je to ugrađeno u sam jezik). U ovom poglavlju dajemo formalnu definiciju netipiziranog λ-računa te opisujemo njegova osnovna svojstva. Zatim prezentiramo primjere izračunavanja pojedinih izraza u λ-računu, odnosno kako se pojedini operatori mogu realizirati koristeći λ-račun. Također, formalno dokazujemo Turing-potpunost netipiziranog λ-računa, čime opravdavamo njegovo korištenje kao oruđe za opisivanje procesa izračunavanja

Slijedi definicija sintakse λ-računa. Za početak, smatramo da imamo prebrojiv skup varijabli iz kojih tvorimo λ-izraze. Sljedeća definicija definira alfabet kojim tvorimo izraze λ-računa.

Definicija 1.1.1. *Alfabet λ-računa tvore sljedeći simboli:*

$v_0, v_1, \ldots$ - *oznake varijabli*

λ - *apstraktor*

$(,),.$ - *zagrade i točka*

Sada definiramo pravila koja definiraju kakve izraze smatramo λ-termima.

Definicija 1.1.2. *Skup svih λ-terma $\mathbf{\Lambda}$ definiramo rekurzivno na sljedeći način:*

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

Term oblika (MN) nazivamo *aplikacija*, a term oblika $(\lambda x.M)$ *apstrakcija*.

Nadalje malim slovima $x, y, \ldots$ označavamo varijable, a velikim slovima $M, N, \ldots$ λ-terme. Kako do sada nismo definirali što bi značilo da su termi jednaki, ne koristimo znak „=", a želimo li naglasiti da su dva terma grafički jednaka pisat ćemo znak „≡".

Uobičajeno je terme oblika $\lambda x.(\lambda y.(\ldots \lambda z.M)\ldots)$ pisati kao $\lambda xy\ldots z.M$. Niz varijabli oblika „$x_1 x_2 \ldots x_n$" označavamo i 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 aplikacija lijevo asocijativna: term MNP interpretiramo kao $(MN)P$, i u tom slučaju ne pišemo zagrade.

Podterm nekog λ-terma je podniz znakova nekog terma koji je i sam λ-term.

Definicija 1.1.3. *Term N je podterm terma M ukoliko vrijedi $N \in Sub(M)$, pri čemu je skup $Sub(M)$ definiran rekurzivno na sljedeći način:*

- $Sub(x) = \{x\}$
- $Sub(\lambda x.N) = Sub(N) \cup \{\lambda x.N\}$
- $Sub(N_1 N_2) = Sub(N_1) \cup Sub(N_2) \cup \{N_1 N_2\}$

Sada želimo definirati što λ-termi predstavljaju te kako se njima manipulira. Manipulacije terma vrše se sintaktičkim pravilima kojima se varijable u nekom λ-termu supstituiraju drugim termima. Kako bi mogli definirati supstituciju varijabli, prvo definiramo pojam slobodne varijable.

Definicija 1.1.4. *Za λ-term M definiramo skup slobodnih varijabli $FV(M)$ na sljedeći način:*

- *za $M \equiv x$, vrijedi $FV(M) = \{x\}$*
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $FV(MN) = FV(M) \cup FV(N)$

Varijabla x je slobodna *u termu M ako $x \in FV(M)$, u suprotnom kažemo da je* vezana *u termu M.*

Vidimo da je ovako definiran pojam slobodne varijable analogan pojmu slobodne varijable prilikom korištenja integrala. U λ-računu apstraktorom deklariramo varijablu, dok u integralnom računu tu funkciju ima oznaka d. Varijable koje nismo posebno deklarirali a javljaju se u izrazu, smatramo konstantama, odnosno implicira se da je njihova vrijednost određena u kontekstu u kojem evaluiramo dani izraz.

Može se promatrati i kakva je pojedina „pojava" varijable u termu, odnosno je li varijabla na određenoj poziciji u termu vezana nekim apstraktorom ili ne. Primjerice, u termu $x(\lambda x.xy)$ varijabla x javlja se i vezano i slobodno.

Sada definiramo supstituciju slobodne varijable λ-termom u nekom λ-termu.

Definicija 1.1.5. *λ-term nastao zamjenom svake slobodne pojave neke varijable x u termu $M \in \Lambda$ nekim termom N označavamo sa $M[x := N]$ i definiramo sljedećim pravilima:*

- $x[x := N] \equiv N$
- $y[x := N] \equiv y$, *ako* $x \not\equiv y$
- $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$, *ako* $x \not\equiv y$
- $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

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: λ-izraze možemo promatrati kao algoritme, odnosno kao programe koji definiraju na koji se način neka radnja odvija.

Pravilima redukcije želimo definirati kriterije prema kojima terme smatramo „sličnima", stoga želimo da pravila budu konzistentna s pravilima kreiranja λ-terma iz definicije 1.1.1. Drugim riječima, ukoliko dva terma M i N smatramo „sličnima", tada je razumno da i termi $\lambda x.M$ i $\lambda x.N$ te MZ i NZ za $Z \in \mathbf{\Lambda}$ budu „slični". Sljedeća definicija formalizira tu ideju.

Definicija 1.1.6. *Za binarnu relaciju* $\mathbf{R}$ *na* $\mathbf{\Lambda}$ *kažemo da je* kompatibilna *(sa sintaksom λ-računa) 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* $Z \in \mathbf{\Lambda}$
- $(\lambda x.M, \lambda x.N) \in \mathbf{R}$

Svaka kompatibilna, refleksivna i tranzitivna relacija na $\mathbf{\Lambda}$ naziva se *relacija redukcije* na $\mathbf{\Lambda}$. Postoje brojne relacije redukcije koje se koriste za definiranje različitih λ-teorija. Relacije redukcije uvelike određuju svojstva teorije koja se definira. U ovom ćemo poglavlju uvesti pojam β-redukcije te promotriti neka svojstva takve redukcije. U poglavlju 3 demonstrirat ćemo posebnu vrstu redukcije koju koristi Haskell.

Definicija 1.1.7. *Neka je R binarna relacija na* $\mathbf{\Lambda}$*. Relacija R na* $\mathbf{\Lambda}$ *inducira sljedeće relacije: Relacija* $\rightarrow_R$ *je kompatibilno zatvorenje relacije R, a naziva se „jedan korak R-redukcije". Sa* $\twoheadrightarrow_R$ *označavamo tranzitivno i refleksivno zatvorenje relacije* $\rightarrow_R$*. Ta relacija naziva se R-redukcija. Sa* $=_R$ *označavamo simetrično zatvorenje relacije* $\twoheadrightarrow_R$*. Ovu relaciju nazivamo R-ekvivalentnost ili R-konvertibilnost.*

Za jedan korak R-redukcije umjesto zapisa

$$(M, N) \in \rightarrow_R$$

pišemo

$$M \rightarrow_R N$$

Takvim zapisom sugerira se da se prvi term transformira u drugi, odnosno naglašava se njihova sintaktička veza. Analogno, koristimo zapise $M \twoheadrightarrow_R N$ i $M =_R N$ za odgovarajuće relacije redukcije i ekvivalencije.

Prijašnje definicije bile su nam potrebne kako bismo definirali pojam β-redukcije.

Definicija 1.1.8. *Neka je*

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

relacija na $\mathbf{\Lambda}$*. Za term M kažemo da* β*-reducira u term N ukoliko vrijedi*

$$M \twoheadrightarrow_\beta N$$

Primjerice, vrijede sljedeće relacije:

$$(\lambda x.xx)(\lambda y.y) \to_\beta (\lambda y.y)(\lambda y.y)$$
$$(\lambda xy.x)(\lambda z.z) \to_\beta \lambda y.(\lambda z.z) \equiv \lambda yz.z$$

jer se term s desne strane može generirati vršenjem supstitucije na termu s lijeve strane.

Vratimo se sada na slobodne varijable i supstituciju. U termu $(\lambda xy.xy)$, radimo li naivnu supstituciju varijable x varijablom y, mogli bismo zaključiti sljedeće:

$$(\lambda xy.xy)yx \to_\beta (\lambda y.yy)x \to_\beta xx$$

Međutim, intuitivno je jasno da je poželjan rezultat term yx. Imajući na umu da naziv vezane varijable ne igra ulogu u načinu na koji term reducira (primjerice, term $(\lambda x.x)M$ reducira u M, potpuno identično kao i term $(\lambda y.y)M$), ovakvo se ponašanje može izbjeći tako da smatramo da su imena vezanih varijabli uvijek različita od imena slobodnih varijabli, odnosno da prije redukcije varijable preimenujemo. U našem primjeru, redukcija sada izgleda ovako:

$$(\lambda xy.xy)yx \equiv (\lambda uv.uv)yx \to_\beta (\lambda v.yv)x \to_\beta yx$$

Time garantiramo da se sve varijable javljaju ili vezano ili slobodno, te da nema konflikata.

Činjenica da se termi razlikuju samo po imenu vezanih varijabli povijesno se naziva α-kongruencija (simbol $\equiv_\alpha$), a promjena imena vezanih varijabli α-konverzija. Po uzoru na [1] ove metode nećemo formalizirati već koristiti kako je navedeno u prethodnom primjeru.

Kako je jedan od ciljeva λ-računa formaliziranje algoritma, pojmovi „beskonačne petlje" te „završetka algoritma" također su od velike važnosti. Sada ćemo definirati *normalnu formu* terma, koja reprezentira finalni rezultat izvršavanja nekog algoritma. Analogno, nepostojanje normalne forme nekog terma možemo interpretirati kao beskonačnu petlju.

Definicija 1.1.9. *Neka je R binarna relacija na* $\mathbf{\Lambda}$*. Term M je R-*redeks *ukoliko postoji* $N \in \mathbf{\Lambda}$ *takav da je* $(M, N) \in R$*. U tom slučaju term N nazivamo R-*kontraktum *terma M.*

Jednostavno je vidjeti da prema gornjoj definiciji termi koji su β-redeksi imaju oblik tipa $(\lambda x.M)N$, odnosno nužno su aplikacija sačinjena od apstrakcije i proizvoljnog terma.

Definicija 1.1.10. *Neka je R binarna relacija na* Λ*. Za term kažemo da je u R-*normalnoj formi *ukoliko ne sadrži podterm koji je R-redeks. Za term kažemo da* ima *R-*normalnu formu *ukoliko je R-ekvivalentan nekom termu koji je u R-normalnoj formi.*

Kao primjer, promotrimo ponovno β-redukciju. Sljedeći termi su u β-normalnoj formi:

$$\lambda x.x$$
$$\lambda xyz.xz(yz)$$

jer ne postoji term u kojeg reduciraju.
Sada dajemo dva primjera terma i pripadnih β-normalnih formi. Term $(\lambda x.x)(\lambda x.x)$ nije u β-normalnoj formi jer sadrži β-redeks. No, očito vrijedi $(\lambda x.x)(\lambda x.x) \to_\beta (\lambda x.x)$, pa je term $\lambda x.x$ β-normalna forma danog terma.

Slično, term $(\lambda xyz.zyx)(\lambda xy.yx)(\lambda x.x)(\lambda x.x)$ nije u β-normalnoj formi, no primjenom β-redukcije dobivamo redom:

$$\begin{aligned}(\lambda xyz.zyx)(\lambda xy.yx)(\lambda x.x)(\lambda x.x) &\to_\beta (\lambda x.x)(\lambda x.x)(\lambda xy.yx)\\ &\to_\beta (\lambda x.x)(\lambda xy.yx) \to_\beta \lambda xy.yx\end{aligned}$$

Posljednji term je β-normalna forma početnog terma.

Promotrimo li definiciju relacije redukcije, vidimo da terme možemo promatrati i kao funkcije koje za dani ulazni term vraćaju neki izlazni term, međutim prilikom takve interpretacije treba biti oprezan jer netipizirani λ-račun ne radi semantičku podjelu između funkcije i njenog argumenta, pa ta analogija nije u potpunosti točna. Ipak, u tekstu će se koristiti funkcijska terminologija poput „djeluje na" i „izlaz" kada je jasno na što se izrazi odnose.

U poglavlju 1.4 vidjet ćemo svojstva β-redukcije te demonstrirati zašto ona nije pogodna za direktnu implementaciju u programskim jezicima. Također, umjesto $\to_\beta$, $\twoheadrightarrow_\beta$ i $=_\beta$ pisat ćemo samo $\to$, $\twoheadrightarrow$ i $=$, te podrazumijevati da se radi o β-redukciji osim ako naznačimo suprotno.

Teoriju netipiziranog λ-računa sa standardnom β-ekvivalencijom moguće je definirati i aksiomatski, gdje su aksiomi upravo zahtjevi konzistentnosti i relacije redukcije, te definicija β-redukcije iz gornjih definicija. Prema tome, aksiomi λ-teorije s β-ekvivalencijom

kako je ranije definirana bili bi sljedeći:

$$
\begin{aligned}
&(\lambda x.M)N = M[x := N] \\
&M = M \\
&M = N \implies N = M \\
&M = N,\ N = L \implies M = N \\
&M = N \implies MZ = NZ \\
&M = N \implies ZM = ZN \\
&M = N \implies \lambda x.M = \lambda x.N
\end{aligned}
$$

Pritom se može dokazati (primjerice u [1]) da definiranje ekvivalencije terma pomoću β-ekvivalencije i aksiomatsko definiranje ekvivalencije definiraju istu teoriju. Želimo li naglasiti da je neku ekvivalenciju $M = N$ moguće dokazati u tako definiranoj teoriji, koristimo zapis

$$\boldsymbol{\lambda} \vdash M = N$$

Kao primjer alternativne λ-teorije, gornje aksiome možemo proširiti aksiomom ekstenzionalnosti:

$$\lambda x.Mx = M$$

Time terme izjednačavamo po njihovom djelovanju. Može se pokazati da je ovaj aksiom nezavisan od ostalih. Funkcijski jezici implicitno koriste ovaj aksiom prilikom definiranja funkcija (više u poglavlju 3).

1.2 Kombinatori

U ovoj točki 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 λ-računu moguće definirati neke matematičke teorije poput logike sudova.

Definicija 1.2.1. *Za $M \in \boldsymbol{\Lambda}$ kažemo da je* kombinator *ako vrijedi $FV(M) = \emptyset$. Skup svih kombinatora označavamo sa $\boldsymbol{\Lambda}^0$.*

Kombinatori su važni jer na njih ne utječe okolina, tj. njihovo je djelovanje uvijek jednako, neovisno o kontekstu u kojem se nalaze, zbog čega su vrlo korisni u funkcijskom programiranju jer programski prevodioc o njima može puno zaključiti već prilikom statičkog prevođenja programa. Zbog neovisnosti o kontekstu, prilikom evaluacije 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 $I \equiv \lambda x.x$ na sve λ-terme djeluje potpuno jednako, tako da vrati njih same kao izlaz. Stoga se taj kombinator može definirati sljedećim izrazom:

$$\forall M \in \mathbf{\Lambda} \;\; IM = M$$

Kombinator pritom shvaćamo kao klasu ekvivalencije na skupu svih terma gdje je relacija ekvivalencije funkcijska jednakost kombinatora. Uočimo da ti termi ne moraju biti niti α-kongruentni, niti β-ekvivalentni (primer su kombinatori fiksne točke u poglavlju 1.3).

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.

Definicija 1.2.2. *Za proizvoljan term $M \in \mathbf{\Lambda}$ definiramo* zatvorenje *kao term*

$$\lambda x_1 \ldots x_n.M$$

takav da vrijedi $FV(M) = \{x_1, \ldots, x_n\}$.

Uočimo da zatvorenja ima koliko i mogućih permutacija skupa slobodnih varijabli.

Parcijalna aplikacija i funkcije višeg reda

Promotrimo sljedeći pseudo λ-izraz:

$$plus \equiv \lambda xy.x + y$$

Tim izrazom smo definirali jedan kombinator. Prisjetimo se da je $\lambda xy.x + y$ samo pokrata za izraz $\lambda x.(\lambda y.x + y)$. S time na umu, promotrimo redukciju prilikom primjene izraza plus na jedan argument.

$$(plus)2 \equiv (\lambda xy.x + y)2 \equiv (\lambda x.(\lambda y.x + y))2 \rightarrow \lambda y.2 + y$$

Primijetimo da smo time ponovno dobili funkciju, koja je identična početnoj osim što je jedna varijabla zamijenjena konkretnim izrazom. Takva operacija naziva se *parcijalna aplikacija*. Parcijalna aplikacija daje nam zanimljivu interpretaciju λ-izraza s više varijabli. Umjesto da izraz „plus" shvatimo kao funkciju koja prima dvije varijable, koristeći rezoniranje kao u gornjem primjeru, „plus" možemo shvatiti kao izraz koji prima jednu varijablu, a kao izlaz vraća ponovno funkciju, koja također prima jedan argument. Takva interpretacija λ-izraze promatra kao *funkcije višeg reda*, odnosno funkcije čija kodomena sadrži i funkcije.

Kombinatori također imaju sljedeće važno svojstvo (dokaz u [1]): sve zatvorene λ-izraze moguće je generirati parcijalnom aplikacijom samo jednog kombinatora samog na

sebe. Međutim, najčešće se koristi sustav standardnih kombinatora kojeg tvore dva kombinatora **S** i **K**.

$$\mathbf{K} \equiv \lambda xy.x$$
$$\mathbf{S} \equiv \lambda xyz.(xz)(yz)$$

Primjerice, kombinator identiteta može se dobiti na sljedeći način:

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

Sada ćemo definirati još neke kombinatore koji će nam trebati u kasnijim poglavljima. Osim oruđa za dokazivanje važnih rezultata, ovi kombinatori demonstriraju mogućnost korištenja λ-izraza za kodiranje različitih matematičkih teorija, primjerice aritmetike prirodnih brojeva.

Logika sudova

U ovoj točki definiramo terme koji predstavljaju analogone booleovskih vrijednosti. Osim što su potrebni da dokažemo Turing-potpunost netipiziranog λ-računa, ovi termi omogućuju simuliranje logike sudova u λ-računu. Neka je sada

$$\mathbf{T} \equiv \lambda xy.x$$

$$\mathbf{F} \equiv \lambda xy.y$$

Ovi izrazi su jednostavne projekcije prve odnosno druge varijable. Sljedeći korak je definiranje kondicionala. Za proizvoljne λ-terme P, M, N, promatramo term PMN. Iako tipovi nisu eksplicino definirani te na mjesto terma P može doći bilo koji λ-izraz, ovaj term zamišljen je tako da na mjesto terma P dolazi jedna od booleovskih vrijednosti (odnosno term koji reducira u booleovski term). Tada vrijedi sljedeće:

$$PMN \equiv \mathbf{T}MN \twoheadrightarrow M$$
$$PMN \equiv \mathbf{F}MN \twoheadrightarrow N$$

što je upravo ponašanje kakvo bismo željeli od kondicionala. 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 booleovski operatori:

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

U gornjim izrazima, zamišljeno je da se na mjesta varijabli u i v supstituiraju termi **T** ili **F**. Kako su oni sami projekcije, ovisno o vrijednosti oni „odabiru" odgovarajuću daljnju akciju. Primjerice, logička disjunkcija je definirana na sljedeći način: $A \vee B \iff A \equiv T$ ili $B \equiv T$. Redukcija terma **or** slijedi tu logiku. Primjerice, redukcija gdje je prvi argument **F** a drugi **T** izgledala bi ovako:

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

Liste

Sada ćemo demonstrirati konstrukciju liste u λ-računu. Liste su ključne strukture podataka za funkcijske jezike jer omogućuju iteriranje po elementima. Liste koje ćemo ovdje definirati su vrlo slične onima u Haskellu pa tako možemo formalno rezonirati o njihovim svojstvima.

Prvo ćemo definirati uređeni par elemenata, a tada iskoristiti tu definiciju kako bismo definirali listu. Uređen par $[M, N]$ definiramo sa

$$[M, N] \equiv \lambda z.zMN$$

Proširenje na proizvoljnu n-torku je logično:

$$[M_0, \ldots, M_n] \equiv [M_0, [M_1, \ldots, M_n]]$$

Važno je uočiti da se novi element uvijek dodaje *slijeva*. To omogućuje jednostavniju 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 dohvaćaju svi elementi osim prvog. Operator konkatenacije možemo realizirati na sljedeći način:

$$\mathbf{cons} \equiv \lambda xyz.zxy$$

Promotrimo djelovanje operatora **cons** na proizvoljne terme:

$$(\mathbf{cons})MN \equiv (\lambda xyz.zxy)MN \twoheadrightarrow \lambda z.zMN \equiv [M, N]$$

Uočimo ponovno da nema restrikcija na tip pojedinih terma u uređenom paru. Ipak, ukoliko je term N i sam lista, ponašanje je upravo kakvo bi očekivali od operatora konkatenacije. Primjerice:

$$(\mathbf{cons})M_0\ [M_1, \ldots, M_n] \equiv (\lambda xyz.zxy)M_0\ [M_1, \ldots, M_n] \twoheadrightarrow \lambda z.zM_0[M_1, \ldots, M_n]$$

Nadalje, vrijedi:

$$\lambda z.zM_0[M_1, \ldots, M_n] \equiv [M_0, [M_1, \ldots, M_n]] \equiv [M_0, M_1, \ldots, M_n]$$

Ovakva definicija list ima zanimljive posljedice. Primjerice, dohvaćanje n-tog elementa liste ne može se izvesti u konstantnom vremenu. Ipak, vidjet ćemo da se ovaj problem često može zanemariti koristimo li specijalne vrste redukcije.

Operatore ***head*** i ***tail*** definiramo sljedećim izrazima:

$$\mathbf{head} \equiv \lambda x.x\mathbf{T}$$
$$\mathbf{tail} \equiv \lambda x.x\mathbf{F}$$

Jednostavno se vidi da ti operatori reduciraju na željeni način.

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

Liste su elementarna struktura podataka u funkcijskim jezicima, zbog čega postoje mnoge funkcije koje automatiziraju česte operacije nad njima. U točki 1.3 ćemo definirati funkcije **map**, **foldl** i **foldr**, nakon što demonstriramo rekurziju u λ-računu.

Prirodni brojevi

Sada definiramo analogone prirodnih brojeva, *numerale*. Ovi termi služe kako bismo u λ-računu imali terme koji reprezentiraju prirodne brojeve, s ciljem da omogućimo usporedbu klasa parcijalno rekurzivnih funkcija i klase λ-definabilnih funkcija u poglavlju 1.5.

Definicija 1.2.3. *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 1.2.4. *Funkcije sljedbenika i prethodnika definiramo sa:*

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

*Term **Zero** kojim ispitujemo je li dani numeral jednak $\ulcorner 0 \urcorner$ definiramo sa:*

$$\textbf{\textit{Zero}} \equiv \lambda x.x\mathbf{T}$$

Jednostavno se vidi 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 λ-računu, te osnovnu aritmetiku na njima.

1.3 Fiksne točke i rekurzija

Rekurzija je ključno oruđe u funkcijskim jezicima, pa ćemo se u ovom poglavlju posebno posvetiti rekurziji i njenoj realizaciji. Prvo dajemo važan rezultat koji uz svoju praktičnu vrijednost pokazuje i neka neintuitivna svojstva netipiziranog λ-računa.

Teorem 1.3.1 (Teorem o fiksnoj točki)**.** *Za svaki $F \in \mathbf{\Lambda}$ postoji $X \in \mathbf{\Lambda}$ tako da vrijedi*

$$FX = X$$

Dokaz. Definiramo $W \equiv \lambda x.F(xx)$ i $X \equiv WW$. Sada vrijedi redom:

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$$

□

Ovaj nam teorem govori da svaki λ-term, promatran kao funkcija, ima fiksnu točku. Kao posljedica ovog teorema javlja se činjenica da svaka jednadžba oblika $FX = X$ u λ-računu ima rješenje. Sada ćemo definirati jedan term koji nam omogućava algoritamski trivijalno dobivanje rješenja ovakvih jednadžbi, odnosno način da se postigne rekurzija u funkcijskim jezicima.

Definicija 1.3.2. *Za kombinator $Y \in \mathbf{\Lambda}$ kažemo da je* kombinator fiksne točke *ako vrijedi sljedeće:*

$$\forall X \in \mathbf{\Lambda} \quad YX = X(YX)$$

Važno svojstvo ovakvih kombinatora je da aplikacijom kombinatora fiksne točke na bilo koji λ-term dobivamo fiksnu točku tog terma.

Propozicija 1.3.3. *Sljedeći termi su kombinatori fiksne točke:*

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$\mathbf{\Theta} \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

Dokaz. Uzmimo proizvoljan λ-term F. Neka je $W \equiv \lambda x.F(xx)$. Sada vrijedi:

$$\mathbf{Y}F = WW = F(WW) = F(\mathbf{Y}F)$$

Pritom smo koristili činjenicu da je $\mathbf{Y}F = WW$. Primijetimo da ne vrijedi $WW \twoheadrightarrow \mathbf{Y}F$, pa prema tome ne vrijedi niti $\mathbf{Y}F \twoheadrightarrow F(\mathbf{Y}F)$. Kombinator $\mathbf{\Theta}$ ima i dodatno svojstvo da vrijedi $\mathbf{\Theta}F \twoheadrightarrow F(\mathbf{\Theta}F)$. Označimo s $V \equiv \lambda xy.y(xxy)$. Tada vrijedi $\mathbf{\Theta} \equiv VV$ iz čega slijedi:

$$\mathbf{\Theta}F \equiv VVF \to F(VVF) \to F(\mathbf{\Theta}F)$$

□

Gornji kombinatori zbog svoje su važnosti imenovani prema svojim autorima: $\mathbf{Y}$ je imenovan *Churchov (paradoksalni) kombinator*, a $\mathbf{\Theta}$ *Turingov kombinator fiksne točke*.

Kao primjer korištenja kombinatora fiksne točke u praksi prezentirat ćemo definiranje funkcije koja za dani prirodni broj n vraća n-ti Fibonaccijev broj. Dakle, želimo funkciju F koja ima sljedeće djelovanje:

$$Fn = F(n-2) + F(n-1)$$

Sada F možemo zapisati sa

$$F = \lambda n.F(n-2) + F(n-1)$$

U λ-računu ne postoji alat koji bi omogućavao definiranje terma koji sam sebe referencira na sintaktičkoj razini. Stoga desnu stranu prepišemo u obliku aplikacije, gdje je argument funkcija F.

$$F = (\lambda fn.f(n-2) + f(n-1))F$$

Iz toga zaključujemo da je F jedna fiksna točka funkcije $E \equiv \lambda fn.f(n-2) + f(n-1)$. Koristeći kombinator fiksne točke jednostavno je vidjeti da je YE fiksna točka terma E, vrijedi $YE = E(YE)$, direktno iz definicije.

Kombinatori fiksne točke također se koriste za iteriranje željene funkcije. Promotrimo kao primjer par koraka redukcije gore definirane funkcije YE primijenjene na broj 3:

$$YE3 = E(YE)3 \equiv (\lambda fn.f(n-2) + f(n-1))(YE)3 \twoheadrightarrow (YE)1 + (YE)2$$

Ponekad nije dovoljno koristiti funkciju koja rekurzivno ovisi samo o vlastitim vrijednostima, već se može dogoditi da nam je potrebno definirati dvije funkcije koje su međusobno rekurzivno ovisne. Promotrimo primjer određivanja parnosti prirodnog broja. Problem je moguće sročiti na sljedeći način (kako je predstavljeno u [16]):

$$(\textbf{even})n = \text{„ako } n = 0 \text{ onda } \mathbf{T} \text{ inače } (\textbf{odd})(n-1)\text{”}$$
$$(\textbf{odd})n = \text{„ako } n > 0 \text{ onda } (\textbf{even})(n-1) \text{ inače } \mathbf{F}\text{”}$$

Kao u primjeru Fibonaccijevih brojeva, prvo pomoću λ-izraza napišemo nerekurzivne verzije danih funkcija.

$$(\textbf{even}) = (\lambda fn.\text{„ako } n = 0 \text{ onda } \mathbf{T} \text{ inače f}(n-1)\text{”})(\textbf{odd})$$
$$(\textbf{odd}) = (\lambda fn.\text{„ako } n > 0 \text{ onda f}(n-1) \text{ inače } \mathbf{F}\text{”})(\textbf{even})$$

Definirajmo sada sljedeće terme:

$$E_1 \equiv (\lambda fn.\text{„ako } n = 0 \text{ onda } \mathbf{T} \text{ inače f}(n-1)\text{”})$$
$$E_2 \equiv (\lambda fn.\text{„ako } n > 0 \text{ onda f}(n-1) \text{ inače } \mathbf{F}\text{”})$$

Vidimo da ulančavanjem terma E_1 i E_2 možemo dobiti funkcije koje rekurzivno ovise samo o jednoj funkciji. Tada možemo koristiti standardni kombinator fiksne točke. Primjerice:

$$(\textbf{even}) \equiv E_1(E_2(\textbf{even})),$$

te konačno:

$$(\textbf{even}) \equiv \mathbf{Y}(\lambda f.E_1(E_2 f))$$

Primjenom kombinatora **Y** sada dobivamo traženu funkciju. Zamjenom redoslijeda terma E_1 i E_2 dobivamo funkciju **odd**.

Na ovaj način, programski jezici mogu definirati bilo kakvu funkciju realiziranu pomoću rekurzije. U poglavlju 1.5 vidjet ćemo da je za slučaj netipiziranog λ-računa rekurzija (odnosno Turingov kombinator) ključna u dokazu Turing-potpunosti λ-računa.

Iteriranje po rekurzivnim strukturama

Sada definiramo funkcije **map**, **foldl** i **foldr** koje smo najavili ranije. Kao u prošlim primjerima, prvo definiramo nerekurzivni term na koji zatim primjenjujemo kombinator fiksne točke da bismo dobili željeno djelovanje. Počnimo s funkcijom **map**. Želimo sljedeće djelovanje:

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

Funkcijom **head** prvo izdvajamo prvi element liste te ga zatim transformiramo 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** korektno definirana izrazom:

$$\textbf{map} \equiv \mathbf{Y}(\lambda c f l.((\textbf{cons})\ \ (f\ ((\textbf{head})\ l))\ \ c\ f\ ((\textbf{tail})\ l)))$$

Vrijedi:

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

gdje je **L** proizvoljna lista (booleovskih) terma. Pritom smo koristili pravila redukcije kombinatora definiranih u 1.2.

Promotrimo sada kako bismo, primjerice, rekurzivno sumirali listu brojeva. U svakom koraku rekurzije potrebno je izdvojiti element liste i nadodati ga na postojeću sumu. Općenito kada želimo vršiti analognu operaciju, potrebno je specificirati funkciju kojom akumuliramo rezultat, početnu vrijednost te listu elemenata. Ovisno o tome kojim redom apliciramo funkciju, razlikujemo lijevu (**foldl**) i desnu (**foldr**) varijantu.

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

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

U poglavlju 3.3 promotrit ćemo svojstva ovih funkcija uz *lijenu* evaluaciju izraza.

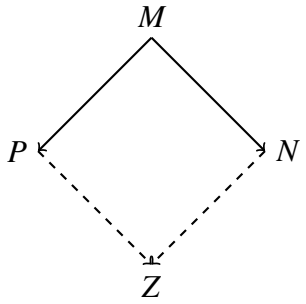
1.4 Svojstva β-redukcije

U programskim jezicima, β-redukcija se ne koristi direktno jer ne definira strogi redoslijed reduciranja, zbog čega se javlja svojevrsni nedeterminizam, što ćemo demonstrirati. Međutim, redukcije koje se koriste u suštini su β-redukcije s restringiranim uvjetima korištenja pa ima smisla promatrati općenita svojstva ovakvih transformacija.

Pitanje postojanja normalne forme nekog terma često se pojavljuje u meta-teoriji λ-računa. U programskim jezicima to je pitanje od velike praktične vrijednosti jer nepostojanje normalne forme uzrokuje beskonačnu petlju.

Uzmimo kao primjer term $\omega \equiv \lambda x.xx$. Pokazat ćemo da ovaj term nema normalnu formu. Promotrimo moguće redukcije terma $\Omega \equiv \omega\omega$:

$$\Omega \equiv \omega\omega \equiv (\lambda x.xx)\omega \rightarrow \omega\omega \equiv \Omega$$



Slika 1.1: Grafički prikaz prvog Church-Rosserovog teorema

Vidimo da term Ω reducira uvijek sam u sebe. Kako je uvijek moguće reducirati, Ω nema normalnu formu. Međutim, nepostojanje normalne forme ne mora biti toliko očito. Promotrimo term $\mathbf{KI}\Omega$. Vrijedi

$$\mathbf{KI}\Omega \equiv (\lambda xy.x)\mathbf{I}\Omega \twoheadrightarrow \mathbf{I}$$

Međutim reduciramo li prvo term $\Omega \equiv (\omega\omega)$, slijedi

$$\mathbf{KI}\Omega \twoheadrightarrow \mathbf{KI}\Omega \twoheadrightarrow \ldots$$

prema prijašnjem primjeru. Ipak, β-redukcija slijedi neke pravilnosti koje su karakterizirane Church-Rosserovim teoremima koje ćemo sada navesti. Ovdje ih nećemo dokazivati jer nas zanimaju samo posljedice, a dokazi se mogu naći u [1]. U suštini, ovi teoremi garantiraju 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 normalne forme, ukoliko ona postoji.

Teorem 1.4.1. *Neka su $M, N, P \in \mathbf{\Lambda}$ 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. Primjerice, za term $\mathbf{KI}\Omega$ nije bitno koliko puta reduciramo term Ω, uvijek možemo doseći term $\mathbf{I}$.

Zbog grafičkog prikaza gornjeg teorema, ovakvo svojstvo naziva se još i *diamond property*.

Jednostavan korolar ovog teorema je da, ukoliko su termi N i P u normalnoj formi, tada vrijedi $N \equiv_\alpha P$ (gdje $\equiv_\alpha$ označava α-kongruenciju opisanu na stranici 7). To je stoga što u tom slučaju niti N niti P ne mogu reducirati, a teorem garantira da mora postojati zajednički term u kojeg oba mogu reducirati. Prema tome mora se raditi o jednakim termima.

Sljedeći teorem je također posljedica teorema 1.4.1. Iskaz teorema je sličan, međutim nešto je općenitiji. Teorem garantira da dva β-ekvivalentna terma imaju term u koji oba mogu reducirati.

Teorem 1.4.2. *Neka su $M, N \in \mathbf{\Lambda}$ 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 Haskell i srodne programske jezike.

Teorem 1.4.3. *Ukoliko term ima normalnu formu, tada redukcija krajnje lijevog redeksa uvijek dovodi do normalne forme.*

Ponovno kao primjer promatramo term $\mathbf{KI}\Omega \equiv \mathbf{KI}(\omega\omega)$. Pod krajnje lijevim redeksom smatramo onaj redeks čiji znak apstraktora je prvi slijeva. U našem primjeru to je redeks **KI**, koji reducira u funkciju $\lambda y.\mathbf{I}$ koja tada zanemaruje argument Ω. U ranijim računima vidjeli smo da reduciranje terma Ω vodi u beskonačnu petlju dok redukcija cijelog terma $\mathbf{KI}\Omega$ reducira u normalnu formu u dva koraka.

Redukcija krajnje lijevog redeksa naziva se *redukcija normalnim redom* (*normal order reduction*), jer uvijek dovodi do normalne forme ukoliko ona postoji. Redukcija krajnje desnog redeksa naziva se redukcija aplikativnim redom.

Intuitivno, aplikativni redoslijed redukcije uvijek reducira prvo argument funkcije, a tek tada samu funkciju, dok je kod redukcije normalnim redom redoslijed obratan. To dovodi do toga da pri aplikativnom reduciranju možemo beskonačno reducirati argument koji, gledano iz perspektive cijelog izraza, možda nije uopće relevantan (primjerice kod konstantnih funkcija). Normalan redoslijed prvo evaluira funkciju, pa će potencijalno problematičan argument možda biti zanemaren.

1.5 Parcijalno rekurzivne funkcije

U prijašnjim poglavljima vidjeli smo kako netipizirani λ-račun možemo promatrati i kao jednostavan programski jezik. U ovom poglavlju posvetit ćemo se apstraktnijem problemu *izračunljivosti* i pokazat ćemo da je λ-račun ekvivalentan ostalim modelima izračunljivosti (u Churchovom smislu).

Točnije, u ovom ćemo poglavlju pokazati da se parcijalno rekurzivne funkcije mogu simulirati klasom λ-definabilnih funkcija i obratno, čime ćemo dokazati da su ti modeli izračunljivosti ekvivalentni. Za početak je potrebno navesti definicije i dokazati neke pomoćne rezultate.

Želimo li kodirati parcijalne funkcije u λ-računu, potrebno je kodirati i slučaj kada funkcija nije definirana za neki vektor prirodnih brojeva. Kao prvi pokušaj čini se da bi

bilo razumno poistovjetiti nepostojanje 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. U lemi 1.5.11 vidjet ćemo da su pojmovi rješivosti i postojanja normalne forme ipak vrlo bliski, iako nisu u potpunosti ekvivalentni.

Sada ćemo definirati klasu parcijalno rekurzivnih funkcija. Za to je potrebno prvo definirati operacije primitivne rekurzije, kompozicije i μ-operator. Parcijalno rekurzivne funkcije kao domenu imaju neki neprazan skup $S \subseteq \mathbb{N}^k$, za $k \in \mathbb{N}$, što znači da nisu nužno definirane za svaki ulazni podatak $\vec{n} \in \mathbb{N}^k$. Ukoliko funkcija f nije definirana za ulazni podatak $\vec{n} \in \mathbb{N}^k$ (odnosno $\vec{n}$ ne pripada domeni S), pišemo $f(\vec{n}) \uparrow$. U suprotnom pišemo $f(\vec{n}) \downarrow$.

Sa $f(\vec{x}) \simeq g(\vec{x})$ označavamo činjenicu da funkcije f i g za dani vektor brojeva $\vec{x}$ ili obje nisu definirane, ili vrijedi $f(\vec{x}) = g(\vec{x})$.

Definicija 1.5.1. *Neka su $\chi, \psi_1, \ldots, \psi_n$ funkcije, te neka je funkcija φ definirana sa $\varphi(\vec{x}) \simeq \chi(\psi_1(\vec{x}), \ldots, \psi_n(\vec{x}))$. Kažemo da je φ definirana pomoću* kompozicije *funkcija.*

Definicija 1.5.2. *Neka je $k \in \mathbb{N}$ i χ totalna k-mjesna funkcija. Neka je ψ totalna $(k+2)$ mjesna funkcija. Za $(k+1)$-mjesnu funkciju φ definiranu s*

$$\varphi(0, \vec{x}) = \chi(\vec{x})$$
$$\varphi(y+1, \vec{x}) = \psi(\varphi(y, \vec{x}), y, \vec{x})$$

kažemo da je definirana pomoću primitivne rekurzije.

Definicija 1.5.3. *Neka je $\varphi : \mathbb{N}^{k+1} \to \mathbb{N}$ funkcija i $\vec{x} \in \mathbb{N}^k$. Tada definiramo funkciju $\mu y(\varphi(\vec{x}, y))$ na sljedeći način:*

$$\mu y(\varphi(\vec{x}, y) \simeq 0) \simeq \begin{cases} \text{najmanji } z \text{ takav da je } \varphi(\vec{x}, y) \downarrow \text{ za sve } y < z, \\ \quad \text{te je } \varphi(\vec{x}, z) = 0, \text{ ako takav } z \text{ postoji} \\ \uparrow, \text{ inače} \end{cases}$$

Definicija 1.5.4. *Klasa parcijalno rekurzivnih funkcija je najmanja klasa funkcija koja sadrži sljedeće funkcije:*

- *nul-funkcija* - $Z(n) = 0, \ \forall n \in \mathbf{N}$
- *projekcija* - $I_n^m(x_1, \ldots, x_n) = x_m$
- *funkcija sljedbenika* - $S^+(n) = n + 1$

a zatvorena je na primitivnu rekurziju, kompoziciju i μ-operator. Nul-funkciju, projekcije i funkciju sljedbenika nazivamo inicijalne funkcije.

Sada definiramo klasu λ-definabilnih funkcija.

Definicija 1.5.5. *Za $M \in \mathbf{\Lambda}^0$ kažemo da je rješiv ako $\exists n \exists N_1, \ldots, N_n \in \mathbf{\Lambda}$ takvi da vrijedi:*

$$MN_1 \ldots N_n = \mathbf{I}$$

gdje je $\mathbf{I}$ kombinator identiteta. Za proizvoljan $N \in \mathbf{\Lambda}$ kažemo da je rješiv ukoliko postoji zatvorenje od N koje je rješivo.

Definicija 1.5.6. *Neka je $S \subseteq \mathbb{N}^k$. Za funkciju $\varphi : S \to \mathbb{N}$ kažemo da je λ-definabilna ukoliko postoji $F \in \mathbf{\Lambda}$ takav da vrijedi*

$$\begin{aligned} \forall \vec{n} \in \mathbb{N}^k \quad & F^{\ulcorner}\vec{n}^{\urcorner} = {\ulcorner}\varphi(\vec{n}){\urcorner}, \text{ ako } \varphi(\vec{n}) \downarrow \\ & F^{\ulcorner}\vec{n}^{\urcorner} \text{ je nerješiv, inače} \end{aligned}$$

Kako bismo dokazali da su klase λ-definabilnih i parcijalno rekurzivnih funkcija jednake, za početak dokazujemo da su inicijalne funkcije λ-definabilne. Jednostavnosti radi vektore numerala koji reprezentiraju redom brojeve u vektoru $\vec{n}$ označavamo s ${\ulcorner}\vec{n}{\urcorner}$.

Lema 1.5.7. *Inicijalne funkcije su λ-definabilne.*

Dokaz. Definiramo sljedeće terme:

- $\mathbf{Z} \equiv \lambda x.{\ulcorner}0{\urcorner}$
- $\mathbf{I}^k_j \equiv \lambda x_1 \ldots x_k.x_j$
- $\mathbf{S}^+ \equiv \lambda x.[\mathbf{F}, x]$

Pokažimo da one redom definiraju nul-funkciju, projekciju te funkciju sljedbenika. Iz definicije terma $\mathbf{Z}$ očito je da vrijedi $\mathbf{Z}{\ulcorner}n{\urcorner} \to {\ulcorner}0{\urcorner}$, $\forall n \in \mathbb{N}$, a kako je $Z(n) = 0$, $\forall n \in \mathbb{N}$, iz definicije λ-definabilnosti slijedi da term $\mathbf{Z}$ λ-definira funkciju Z. Neka je $\vec{x} \in \mathbb{N}$. Tada očito vrijedi $\mathbf{I}^k_j{\ulcorner}\vec{x}{\urcorner} \equiv (\lambda y_1 \ldots y_k.y_j){\ulcorner}\vec{x}{\urcorner} \to {\ulcorner}x_j{\urcorner}$. Konačno, term $\mathbf{S}^+$ je upravo term definiran u točki 1.2, te vrijedi

$$\mathbf{S}^+{\ulcorner}n{\urcorner} \twoheadrightarrow {\ulcorner}n+1{\urcorner}$$

□

Sljedeći rezultati bit će nam potrebni u kasnijim dokazima.

Definicija 1.5.8. *1. Neka je $M \in \mathbf{\Lambda}$ proizvoljan λ-term. Za term M^* kažemo da je supstitucijska instanca terma M ukoliko je*

$$M^* \equiv M[x_1 := N_1] \cdots [x_m := N_m]$$

za neke $N_1, \ldots, N_m \in \mathbf{\Lambda}$.

2. *M^* je zatvorena supstitucijska instanca terma M ukoliko je zatvoren term, odnosno $FV(M^*) = \emptyset$.*

Lema 1.5.9. *Neka je $M \in \mathbf{\Lambda}$. Tada je M rješiv ako i samo ako postoji zatvorena supstitucijska instanca M^* od M i konačni niz zatvorenih termi $\vec{N}$, tako da je $M^*\vec{N} = I$.*

Dokaz. Ukoliko je M rješiv, tada iz definicije rješivosti za proizvoljan term slijedi da postoji $\vec{N}$ takav da

$$(\lambda x_1 \cdots x_n.M)\vec{N} = \mathbf{I}$$

gdje je $\lambda x_1 \cdots x_n.M$ zatvorenje od M i $\vec{N} \equiv N_1 \cdots N_m$. Primijetimo da u niz varijabli $x_1, \ldots, x_n$ uvijek možemo dodati eventualne slobodne varijable terma $N_1, \ldots, N_m$. Nadalje, uvijek možemo garantirati da je $m > n$ dodavanjem kombinatora identiteta zdesna na obje strane jednakosti.

Nadalje pretpostavljamo da su termi $N_1, \ldots, N_m$ zatvoreni. Prema tome, vrijedi

$$M[x_1 := N_1] \cdots [x_n := N_n]N_{n+1} \cdots N_m = \mathbf{I}$$

pa postoji zatvorena supstitucijska instanca $M^* \equiv M[x_1 := N_1] \cdots [x_n := N_n]$ koja je rješiva. Obrat slijedi jednostavno iz činjenice da ukoliko je $M^* \equiv M[x_1 := N_1] \cdots [x_n := N_n]$ zatvoren term, tada je term $\lambda x_1 \cdots x_n.M$ zatvorenje terma M. Iz toga slijedi da je zatvorenje terma M rješivo, pa je M rješiv prema definiciji. □

Napomena 1.5.10. *Ukoliko je M nerješiv kombinator, tada je očito da je za proizvoljan $N \in \mathbf{\Lambda}$ term MN također nerješiv: kad bi MN bio rješiv, tada bi postojao $\vec{K}$ takav da $MN\vec{K} = \mathbf{I}$, pa bi prema tome i M bio rješiv. Rezultat gornje leme omogućuje nam da istu argumentaciju primijenimo i na term M koji ima slobodne varijable, tako da term M zamijenimo s M^*. Nadalje, ukoliko je M nerješiv, tada je i $\lambda x.M$ nerješiv: pretpostavimo li da je $\lambda x.M$ rješiv, tada postoji zatvorenje $\hat{M} \equiv \lambda y_1 \cdots y_n x.M$ i $\vec{N}$ takav da $\mathbf{I} = \hat{M}\vec{N} = M[y_1 := N_1] \cdots [y_n := N_n][x := N_{n+1}]N_{n+2} \cdots N_m$, pa po gornjoj lemi slijedi da postoji zatvorena supstitucijska instanca terma M koja je rješiva, što je kontradikcija. Navedene činjenice bit će nam potrebne prilikom dokazivanja zatvorenosti λ-definabilnih funkcija na kompoziciju.*

Sljedeća lema daje nam vezu među pojmovima normalne forme i rješivosti. Dokaz se može pronaći u [1].

Lema 1.5.11. *M je nerješiv ako i samo ako za svaku supstitucijsku instancu M^* i niz $\vec{N}$ term $M^*\vec{N}$ nema normalnu formu.*

Sada dokazujemo zatvorenost λ-definabilnih funkcija na primitivnu rekurziju.

Lema 1.5.12. *Klasa λ-definabilnih funkcija zatvorene je na primitivnu rekurziju.*

Dokaz. Kako je primitivna rekurzija definirana samo za slučaj totalnih funkcija, ovdje ne moramo koristiti pojam rješivosti zbog čega je dokaz jednostavniji. Označimo s $x^- \equiv P^- x$ (term P^- definiran je na stranici 13). Neka je φ funkcija definirana sa

$$\varphi(0, \vec{n}) = \chi(\vec{n})$$

$$\varphi(k+1, \vec{n}) = \psi(\varphi(k, \vec{n}), k, \vec{n}),$$

gdje su χ i ψ λ-definirane s G i H. Želimo term koji će, ovisno o vrijednosti k, vraćati ili $\chi(\vec{n})$ ili $\psi(\varphi(k, \vec{n}), k, \vec{n})$. Definiramo term $F = \lambda x\vec{y}$. „ako **Zero** x, onda $G\vec{y}$, inače $H(Fx^-\vec{y})x^-\vec{y}$". Pogledajmo sada djelovanje F na numerale. Iz definicije kondicionala slijedi:

$$F\ulcorner 0\urcorner\vec{y} = (\mathbf{Zero}\ulcorner 0\urcorner)(G\vec{y})(H(F\ulcorner 0\urcorner^-\vec{y})\ulcorner 0\urcorner^-\vec{y})$$

Kako vrijedi $\mathbf{Zero}\ulcorner 0\urcorner = \mathbf{T}$, slijedi da je gornji izraz jednak $G\vec{y}$ (sjetimo se da je $\mathbf{T}$ projekcija prvog argumenta). Ukoliko je $\vec{y}$ neki vektor numerala $\ulcorner\vec{n}\urcorner$, iz činjenice da je χ λ-definirana s G slijedi $G\vec{y} = \ulcorner\chi(\vec{n})\urcorner$.

Promotrimo sada izraz $F\ulcorner n\urcorner\vec{y}$, za proizvoljan n. Analognim računanjem kao u prethodnom dijelu dobivamo da je

$$F\ulcorner n+1\urcorner\vec{y} = \mathbf{F}(G\vec{y})(H(F\ulcorner n\urcorner\vec{y})\ulcorner n\urcorner\vec{y}) = H(F\ulcorner n\urcorner\vec{y})\ulcorner n\urcorner\vec{y},$$

a to je upravo tražena tvrdnja. □

Sljedeći jednostavan rezultat nam omogućava λ-definiranje kompozicije funkcija i μ-operatora.

Lema 1.5.13. *$\forall n \in \mathbb{N}$, vrijedi $\ulcorner n\urcorner\mathbf{KII} = \mathbf{I}$, odnosno svi numerali su rješivi.*

Dokaz. Lemu dokazujemo direktno, promatrajući strukturu numerala. Posebno izdvajamo $\ulcorner 0\urcorner \equiv \mathbf{I}$ jer je ima posebnu strukturu.

$$\ulcorner 0\urcorner\mathbf{KII} = \mathbf{IKII} = \mathbf{I}$$

Za $n \neq 0$ vrijedi:

$$\ulcorner n\urcorner\mathbf{KII} = [\mathbf{F}, \ulcorner n-1\urcorner]\,\mathbf{KII} \equiv (\lambda x.x\mathbf{F}\ulcorner n-1\urcorner)\mathbf{KII} =$$
$$\mathbf{KF}\ulcorner n-1\urcorner\mathbf{II} = (\lambda xy.x)\mathbf{F}\ulcorner n-1\urcorner\mathbf{II} = \mathbf{FII}$$

□

Preostaje nam pokazati zatvorenost klase λ-definabilnih funkcija na kompoziciju i μ-operator.

Lema 1.5.14. *Klasa λ-definabilnih funkcije zatvorena je na kompoziciju.*

Dokaz. Neka je funkcija φ definirana na sljedeći način:

$$\varphi(\vec{n}) \simeq \chi(\psi_1(\vec{n}), \ldots, \psi_k(\vec{n}))$$

gdje su $\chi, \psi_1, \ldots, \psi_k$ funkcije λ-definirane redom termima $G, H_1, \ldots, H_k$. Definiramo term F na sljedeći način:

$$F \equiv \lambda\vec{x}.(H_1\vec{x}\mathbf{KII})\cdots(H_k\vec{x}\mathbf{KII})(G(H_1\vec{x})\cdots(H_k\vec{x}))$$

Tvrdimo da F λ-definira funkciju φ. Prvo želimo pokazati da ukoliko je $\psi(\vec{n})$ nedefiniran, da je tada i term $H_i\vec{x}\mathbf{KII}$ nerješiv. Taj je rezultat jednostavna posljedici leme 1.5.9: ukoliko vrijedi $\psi_i(\vec{n})\uparrow$, onda $H_i\ulcorner\vec{n}\urcorner$ nije rješiv prema definiciji λ-definabilnosti. Iz leme 1.5.9 slijedi da tada niti term $H_i\ulcorner\vec{n}\urcorner\mathbf{KII}$ nije rješiv. Lema 1.5.9 također garantira da u tom slučaju niti F nije rješiv, što smo i željeli pokazati.

S druge strane, neka je $\psi_i(\vec{n})\downarrow$, tj. $\psi_i(\vec{n}) = m_i$ za svaki i. Tada je i $H_i\ulcorner\vec{n}\urcorner$ rješiv prema definiciji λ-definabilnosti. Kako vrijedi $H_i\ulcorner\vec{n}\urcorner = \ulcorner m_i\urcorner$, slijedi:

$$H_i\ulcorner\vec{n}\urcorner\mathbf{KII} \to \ulcorner m_i\urcorner\mathbf{KII} \to \mathbf{I}$$

prema lemi 1.5.13. Sada je očito:

$$F\ulcorner\vec{n}\urcorner \twoheadrightarrow G(H_1\ulcorner\vec{n}\urcorner)\cdots(H_k\ulcorner\vec{n}\urcorner) \twoheadrightarrow G\ulcorner m_1\urcorner\cdots\ulcorner m_k\urcorner \twoheadrightarrow \ulcorner\chi(m_1,\ldots m_k)\urcorner$$

jer je χ λ-definiran s G. Time je rezultat dokazan. □

Sada ćemo definirati analogon μ-operatora za λ-račun, za što će nam biti potrebni neki kombinatori definirani u ranijim poglavljima.

Definicija 1.5.15. *Neka je P kombinator. Definiramo kombinator H_P sa:*

$$H_P \equiv \Theta(\lambda hx.\text{„}ako\ Px\ onda\ x\ inače\ h(S^+x)\text{”})$$

gdje je izraz „ako Px onda x inače h(S⁺x)” definiran na stranici 11. Sada definiramo analogon μ-operatora na kombinatorima kao

$$\mu P \equiv H_P\ulcorner 0\urcorner$$

Uočimo da je gornji izraz definiran i za kombinatore P i terme M takve da PM ne reducira u booleovske terme $\mathbf{F}$ i $\mathbf{T}$. U gornjoj definiciji cilj je simulacija djelovanja μ-operatora na način da se redom ispituje je li zadani predikat istinit za pojedini numeral, počevši od $\ulcorner 0\urcorner$. Primjerice, ako za sve $m < n$ vrijedi $P\ulcorner m\urcorner \twoheadrightarrow \mathbf{F}$, tada:

$$\begin{aligned}\mu P \equiv H_P\ulcorner 0\urcorner &\twoheadrightarrow \text{„Ako } P\ulcorner 0\urcorner \text{ onda } \ulcorner 0\urcorner \text{ inače } H_P\ulcorner 1\urcorner\text{”}\\ &\twoheadrightarrow \ldots \twoheadrightarrow \text{„Ako } P\ulcorner n\urcorner \text{ onda } \ulcorner n\urcorner \text{ inače } H_P\ulcorner n+1\urcorner\text{”}\end{aligned}$$

Pritom kao predikat koristimo λ-definiciju neke rekurzivne relacije. Neka P λ-definira neku relaciju R. Ukoliko je predikat P istinit za neki numeral $\ulcorner n \urcorner$ i ukoliko je taj numeral najmanji za koji je P istinit, tada vidimo da operator reducira u taj numeral. Ukoliko predikat nije definiran za neki n, tada $P\ulcorner n\urcorner$ nije rješiv, a kako je on podterm terma „Ako $P\ulcorner n\urcorner$ onda $\ulcorner n\urcorner$ inače $H_P\ulcorner n+1\urcorner$" koji je β-ekvivalentan početnom termu μP, slijedi da μP nije rješiv. Time smo pokazali da ovako definiran μ-operator dobro opisuje μ-operator parcijalno rekurzivnih funkcija. Sada ćemo dokazati zatvorenost λ-definabilnih funkcija na μ-operator. Prvo trebamo sljedeći pomoćni rezultat.

Lema 1.5.16. *Neka je term P takav da za svaki $n \in \mathbb{N}$ vrijedi $P\ulcorner n\urcorner = \mathbf{F}$. Tada je μP nerješiv.*

Intuitivno je jasno da ukoliko predikat kondicionala uvijek reducira u **F**, kondicional uvijek odabire term koji je redeks, pa početni term nema normalnu formu. Jednostavno se pokaže da u tom slučaju ne postoji zatvorena supstitucijska instanca μP^* i vektor terma $\vec{Z}$ takav da term $\mu P^*\vec{Z}$ ima normalnu formu. Rezultat tada slijedi iz leme 1.5.11. Formalni dokaz se može pronaći u [1].

Sada konačno dokazujemo zatvorenost klase λ-definabilnih funkcija na μ-operator.

Lema 1.5.17. *Klasa λ-definabilnih funkcija je zatvorena na μ-operator.*

Dokaz. Neka je $\varphi(\vec{n}) = \mu m[\chi(\vec{n}, m) = 0]$, gdje je χ λ-definirana termom G. Tvrdimo da

$$F \equiv \lambda\vec{x}.(G\vec{x}\mathbf{KII})\mu\left[\lambda y.\mathbf{Zero}(G\vec{x}y)\right]$$

λ-definira funkciju φ. Pogledajmo prvo slučaj kada $\varphi(\vec{n})\downarrow$. Tada postoji m takav da je $\chi(\vec{n}, m) = 0$ i takav da je za sve $k < m$ $\chi(\vec{n}, k)$ definirano. Kako G λ-definira χ, vrijedi da je za svaki $k < m$ term $G\ulcorner\vec{n}\urcorner\ulcorner m\urcorner$ rješiv, te da izraz $(G\ulcorner\vec{n}\urcorner\mathbf{KII})$ reducira u kombinator identiteta. Iz definicije μ-operatora, točnije iz definicije terma H_P, vrijedi da redukcija terma $G\ulcorner\vec{n}\urcorner\ulcorner m\urcorner$ „staje" onda kada je $\mathbf{Zero}(G\ulcorner\vec{n}\urcorner\ulcorner m'\urcorner) = \mathbf{T}$, a to se postiže upravo za $m' = m$. Stoga vrijedi $F\ulcorner m\urcorner = \varphi(m)$. Neka je sada $\varphi(\vec{n})\uparrow$. Pogledajmo slučajeve kada se to može dogoditi:

- ukoliko ne postoji m takav da $\chi(\vec{n}, m) = 0$
- ukoliko $\chi(\vec{n}, m)\uparrow$, za neki m takav da ne postoji $k < m$ za koji vrijedi $\chi(\vec{n}, k) = 0$

U prvom slučaju možemo zanemariti slučaj kada je $\chi(\vec{n}, k)\uparrow$ za neki $k \in \mathbb{N}$, jer ćemo to riješiti u drugoj točki. Pokažimo sada da u tom slučaju $F\ulcorner\vec{n}\urcorner$ nije rješiv. Vrijedi:

$$\mathbf{Zero}(G\ulcorner\vec{n}\urcorner\ulcorner m\urcorner) \twoheadrightarrow \mathbf{F}$$

Iz leme 1.5.16 slijedi da tada term $\mu\left[\lambda y.\mathbf{Zero}(G\ulcorner\vec{n}\urcorner y)\right]$ nije rješiv, pa iz leme 1.5.9 slijedi da niti term $F\ulcorner\vec{n}\urcorner \equiv (G\vec{x}\mathbf{KII})\mu\left[\lambda y.\mathbf{Zero}(G\ulcorner\vec{n}\urcorner y)\right]$ nije rješiv što je i trebalo pokazati.

Preostaje slučaj kada $\chi(\vec{n}, m) \uparrow$, za neki m takav da ne postoji $k < m$ za koji vrijedi $\chi(\vec{n}, k) = 0$. Tada slijedi da niti izraz $(G\ulcorner\vec{n}\urcorner\mathbf{KII})$ nije rješiv, pa niti term $F\ulcorner\vec{n}\urcorner$ nije rješiv, koristeći jednaku argumentaciju kao i u prvom slučaju. Time je rezultat dokazan. □

Sljedeća lema potrebna nam je da dokažemo da su λ-definabilne funkcije nužno parcijalno rekurzivne.

Lema 1.5.18. *Neka je φ λ-definirana s F. Tada vrijedi:*

$$\varphi(\vec{n}) = m \iff F\ulcorner\vec{n}\urcorner = \ulcorner m\urcorner$$

Dokaz. ($\Rightarrow$) Slijedi direktno iz definicije λ-definabilnosti.
($\Leftarrow$) Pretpostavimo da vrijedi $F\ulcorner\vec{n}\urcorner = \ulcorner m\urcorner$, i te da je φ λ-definirana s F.

Iz leme 1.5.13 slijedi da je $\ulcorner m\urcorner$ rješiv, $\forall m \in \mathbb{N}$, iz čega slijedi da je i $F\ulcorner\vec{n}\urcorner$ rješiv jer su termi β-ekvivalentni. Iz definicije λ-definabilnosti slijedi $\varphi(\vec{n}) \downarrow$, pa vrijedi $\varphi(\vec{n}) = k$, za neki $k \in \mathbb{N}$. Kako F λ-definira φ, slijedi da je $\ulcorner k\urcorner = \ulcorner m\urcorner$, iz čega slijedi $k = m$ zbog načina na koji se numerali definiraju (nužno su α kongruentni, tj. jedinstveni su do na nazive varijabli). Stoga vrijedi $\varphi(\vec{n}) = m$. Time je tvrdnja dokazana. □

Sada dokazujemo konačni rezultat, koristeći činjenice dokazane u prethodnim lemama.

Teorem 1.5.19. *Funkcija je parcijalno rekurzivna ako i samo ako je λ-definabilna.*

Dokaz. ($\Rightarrow$) Prema lemi 1.5.18 vrijedi, ukoliko je φ λ-definirana s F, tada:

$$\varphi(\vec{n}) = m \iff F\ulcorner\vec{n}\urcorner = \ulcorner m\urcorner$$

Očito je skup svih aksioma (vidi stranu 9) λ-računa rekurzivan. Iz toga slijedi da je skup svih teorema, odnosno izraza oblika $M = N$ rekurzivno prebrojiv. Tada iz prethodno navedene ekvivalencije slijedi da je skup $\{(\varphi(\vec{n}, m) : \varphi(\vec{n}) = m\}$ rekuzivno prebrojiv.

Kako je taj skup ujedno i graf funkcije φ, koristeći teorem o grafu parcijalno rekurzivne funkcije, slijedi da je funkcija φ također parcijalno rekurzivna. Time je pokazano da klasa parcijalno rekurzivnih funkcija sadrži λ-definabilne funkcije.

($\Leftarrow$) Leme 1.5.12, 1.5.14 i 1.5.17 pokazale su da su inicijalne funkcije λ-definabilne, te da je skup λ-definabilnih funkcija zatvoren na primitivnu rekurziju, kompoziciju te μ-operator. Time je pokazano da klasa λ-definabilnih funkcija sadrži parcijalno rekurzivne funkcije, i time je dokaz završen. □

Na kraju dajemo još jednu karakterizaciju nerješivosti terma, ovog puta pomoću pojma *head normal form* (koristimo skraćenicu *HNF*).

Definicija 1.5.20. *Za term $M \in \mathbf{\Lambda}$ kažemo da je u HNF ako je oblika*

$$M \equiv \lambda x_1 \ldots x_n.xM_1 \ldots M_m,$$

za $m, n \geq 0$ i $M_i \in \mathbf{\Lambda}$, za $i = 1, \ldots m$. Term ima HNF ukoliko je ekvivalentan termu koji je u HNF.

Sljedeći teorem daje karakterizaciju rješivosti koristeći HNF.

Teorem 1.5.21. *Term je rješiv ako i samo ako ima HNF.*

U poglavlju 3 ovaj rezultat omogućit će nam karakterizaciju izraza koje nije moguće u potpunosti evaluirati u funkcijskim jezicima.

Poglavlje 2

Tipizirani λ-račun i Hindley-Milnerov algoritam

2.1 Definicija i osnove

U prošlom poglavlju vidjeli smo kako pomoću netipiziranog λ-računa možemo opisati proces izračunavanja izraza u jezicima srodnim λ-računu. U ovom poglavlju bavit ćemo se tipiziranim λ-računom kojim ćemo opisati sustav tipova u Haskellu, te principe provjeravanja i automatskog određivanja tipova.

Tipizirani račun srodniji je programskim jezicima jer nudi podjelu podataka na funkcije i argumente, te restringira skup λ-izraza koje smatramo valjanima. Time se olakšava pronalaženje grešaka u kodu jer se već prilikom prevođenja može otkriti mnogo grešaka. Haskellov sustav tipova je strog, u smislu da su tipovi vrlo specifični te se ne dopušta automatska transformacija. Na taj način može se otkriti i greška u logici programa, što se u fleksibilnijim sustavima rijetko može. 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 λ-računa [5].

Uz sintaktičke pogodnosti koje uvodi u pisanje programa, tipizirani λ-račun uvodi i neke probleme koji ne postoje u netipiziranim varijantama. Primjerice, kako je aplikacija restringirana na terme koji su međusobno u odnosu funkcija-argument, prirodno se postavlja pitanje je li neki term valjan u danom računu te može li mu se algoritamski pridružiti neki tip. Srodan, ali različit problem, je i samo određivanje tipa nekog terma, za što se pokazuje da je neodlučiv problem u nekim sustavima tipova. Treći specifičan problem nije direktno 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 λ-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 λ-račun. Ovaj sustav je najjednostavniji primjer tipiziranog λ-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 skup. Za generiranje kompleksnijih tipova koriste se *konstruktori tipova*. U jednostavno tipiziranom λ-računu postoji samo jedan konstruktor tipova, u oznaci „$\rightarrow$". Sada definiramo skup tipova jednostavno tipiziranog računa.

Definicija 2.1.1. *Skup tipova* ***Typ*** *definiramo na sljedeći način:*

- *proizvoljan konačan ili prebrojiv skup* baznih tipova *je podskup skupa* ***Typ***
- *za* $\sigma, \tau \in$ ***Typ****, vrijedi* $(\sigma \rightarrow \tau) \in$ ***Typ***

Kao primjer, neka je skup baznih tipova jednak $\{\sigma, \tau\}$. Tada su primjerice mogući sljedeći tipovi: $(\sigma \rightarrow \tau) \rightarrow \tau$, $(\sigma \rightarrow \sigma) \rightarrow (\tau \rightarrow \tau)$, $\tau \rightarrow (\tau \rightarrow \sigma)$.

Kao bazni tip može se uzeti npr. *nul-tip* (u oznaci $\mathbf{0}$), 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.

Drugo pravilo u gornjoj definiciji sugerira zapis tipova koji je desno asocijativan, odnosno smatramo da tip $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n$ predstavlja tip $\sigma_1 \rightarrow (\sigma_2 \rightarrow (\ldots \rightarrow \sigma_n)\ldots)$.

Jednakost dvaju tipova označavamo kao i u slučaju λ-terma. Dakle, ukoliko su dva tipa σ i τ grafički jednaki, pišemo $\sigma \equiv \tau$.

Alfabet kojim tvorimo terme tipiziranog računa ekvivalentan je netipiziranom slučaju. Za proizvoljan tip σ smatramo da postoji prebrojiv skup *varijabli tipa* σ. Sada definiramo skup terma tipa σ.

Definicija 2.1.2. *Neka je* σ *proizvoljan tip. Skup terma tipa* σ*, u oznaci* $\mathbf{\Lambda}_\sigma$*, definiramo rekurzivno na sljedeći način:*

- *sve varijable tipa* σ *su elementi skupa* $\mathbf{\Lambda}_\sigma$*,*
- *za* $M \in \mathbf{\Lambda}_{\tau \rightarrow \sigma}$ *i* $N \in \mathbf{\Lambda}_\tau$*, vrijedi* $(MN) \in \mathbf{\Lambda}_\sigma$*,*
- *za* σ *definiran kao* $\tau_1 \rightarrow \tau_2$*, te za* $M \in \mathbf{\Lambda}_{\tau_1}$ *i* $x \in \mathbf{\Lambda}_{\tau_2}$*, vrijedi* $(\lambda x.M) \in \mathbf{\Lambda}_{\tau_1 \rightarrow \tau_2}$*.*

Konačno, definiramo skup terma jednostavno tipiziranog λ-računa.

Definicija 2.1.3. *Skup svih jednostavno tipiziranih λ-terma* $\mathbf{\Lambda}^\tau$ *definiramo kao uniju svih skupova terma nekog tipa.*

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

Drugim riječima, time smo opisali kakve terme smatramo *valjano tipiziranima*, odnosno termima kojima se korektno može pridružiti tip. Uzmemo li neki term, iz poznavanja tipova njegovih podterma može se deducirati koji tip bi sam term trebao imati. Uočimo da su nam za to potrebne pretpostavke barem o varijablama koje se u termu javljaju. U daljnjem tekstu formalizirat ćemo dedukciju tipa uz pretpostavke, ali prvo dajemo dva načina promatranja veze tipova i terma u jednostavno tipiziranom računu.

Ovisno o tome pišemo li eksplicitno tip svake varijable u termu, možemo promatrati račun *a la Church* i *a la Curry*. Churchev sustav terme promatra kao *tipizirane terme* te tako svaki term ima jedinstven tip i svaka varijabla eksplicitno naveden svoj tip. Primjerice, term $\mathbf{I}_\sigma$ (kombinator identiteta tipa σ) u Churchevom sustavu zapisujemo:

$$\mathbf{I}_\sigma \equiv (\lambda x : \sigma . x) : \sigma \to \sigma$$

pri čemu s $x : \sigma$ naglašavamo tip varijable x, dok sa $\sigma \to \sigma$ označavamo tip cijelog izraza. Pritom zagrade možemo izostaviti. Nasuprot tome, u Curryjevoj varijanti terme smatramo *termima s pridodanim tipovima* tj. izraze promatramo kao izraze netipiziranog λ-računa kojima se naknadno pridružuje tip. Kombinator identiteta sada bi zapisali kao

$$\mathbf{I}_\sigma \equiv (\lambda x . x) : \sigma \to \sigma$$

bez da eksplicitno navodimo tipove varijabli. U jednostavno tipiziranom računu ova distinkcija nema opsežnije posljedice, međutim u nekim polimorfnim računima Churchev sustav omogućuje određivanje (jedinstvenog) tipa nekog izraza dok traženje tipa u Curryjevoj varijanti postaje neodlučiv problem (vidi [11]). Ipak, iz perspektive funkcijskih jezika, Curryjeva varijanta računa je daleko poželjnija jer ne zahtijeva eksplicitno navođenje tipova svih varijabli te time smanjuje količinu potrebnog koda. Kako su operacije određivanja i provjere tipa trivijalne za Churchev sustav, te kako je u slučaju Haskella korišten Curryjev sustav, nadalje promatramo samo Curryjev sustav. Sada opisujemo što formalno znači kontekst u kojem termu pridružujemo tip, a zatim dajemo formalni sustav za deduciranje tipa iz danog konteksta.

Definicija 2.1.4. *Neka je M λ-term. Ukoliko vrijedi* $M \in \mathbf{\Lambda}_\sigma$*, pišemo* $M : \sigma$ *te čitamo „M* ima tip σ*". Niz deklaracija* $x_1 : \sigma_1, \ldots, x_n : \sigma_n$*, gdje su* $x_1, \ldots, x_n$ *međusobno različite varijable, nazivamo* kontekst *i označavamo s* Γ*. Sa dom*(Γ) *označavamo skup varijabli koje se javljaju u* Γ*.*

Sada definiramo pravila dedukcije tipa zadanog terma.

Definicija 2.1.5. *Neka je $M \in \mathbf{\Lambda}$. Ukoliko pomoću pravila dedukcije možemo zaključiti da vrijedi $M : \sigma$, uz neki kontekst Γ, pišemo $\Gamma \vdash M : \sigma$. U slučaju praznog skupa Γ, pišemo samo $\vdash M : \sigma$. Pravila dedukcije su sljedeća:*

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

Promotrimo sada dedukciju tipa za term $\mathbf{S} \equiv \lambda xyz.xz(yz)$. Označimo s $\Gamma = x : \sigma \to \tau \to \theta, y : \sigma \to \tau, z : \sigma$. Ove pretpostavke potrebne su da bismo odredili tip podterma terma **S**. U točki 2.3 bavit ćemo se određivanjem tipova i konteksta takvih da možemo odrediti tip nekog zadanog terma.

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

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

Pojam β-redukcije i ekvivalencije u potpunosti je jednak onom iz netipiziranog računa. Međutim, skup $\mathbf{\Lambda}^\tau$ nije zatvoren na ovakvu β-ekvivalenciju, što je jednostavna posljedica činjenice da je $\mathbf{\Lambda}^\tau$ pravi podskup skupa svih λ-terma $\mathbf{\Lambda}$, što ćemo vidjeti u sljedećoj točki.

2.2 Svojstva jednostavno tipiziranog λ-računa

U ovoj točki navodimo neka svojstva tipiziranog računa. Dokazi se mogu pronaći u [2].

Jednostavno se vidi da ukoliko vrijedi $\Gamma \vdash M : \sigma$, te ako vrijedi $\Gamma \subseteq \Gamma'$, tada $\Gamma' \vdash M : \sigma$. Sljedeći rezultat daje jednostavnu vezu među slobodnim varijablama terma kojem određujemo tip te konteksta koji je potreban za dedukciju.

Propozicija 2.2.1. *Neka vrijedi $\Gamma \vdash M : \sigma$. Tada je nužno $FV(M) \subseteq dom(\Gamma)$. Nadalje, neka je $\Gamma' \subseteq \Gamma$ kontekst u kojem se javljaju samo slobodne varijable terma M. Tada vrijedi $\Gamma' \vdash M : \sigma$.*

Uočimo jednostavnu posljedicu gornjeg rezultata: ukoliko je term zatvoren, tada nisu potrebne nikakve pretpostavke kako bismo odredili tip tog terma.

Sljedeći rezultat bit će nam potreban prilikom dokazivanja odlučivosti određivanja tipa, a govori da se iz strukture terma može zaključiti o tipu njegovih podterma.

Propozicija 2.2.2. *Vrijedi:*

- *Ako je* $\Gamma \vdash x : \sigma$*, tada je nužno* $(x : \sigma) \in \Gamma$*,*
- *ako* $\Gamma \vdash MN : \sigma$*, tada postoji tip* τ *takav da vrijedi* $\Gamma \vdash M : \tau \to \sigma$ *te* $\Gamma \vdash N : \tau$*, te*
- *ukoliko* $\Gamma \vdash \lambda x.M : \sigma$*, tada postoje tipovi* τ *i* θ *takvi da je* $\sigma \equiv \tau \to \theta$ *i* $\Gamma, x : \tau \vdash M : \theta$*.*

Važan rezultat koji vrijedi općenito za tipizirane račune je činjenica da redukcijom ne mijenjamo tip terma.

Teorem 2.2.3. *Neka su M i N termi takvi da* $M \twoheadrightarrow N$*. Ukoliko je* $M : \sigma$*, tada je i* $N : \sigma$*.*

U kontekstu funkcijskih jezika, to znači da ukoliko odredimo tip izraza prije evaluacije, tada znamo i njegov povratni tip. To 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. Ta činjenica je relativno jednostavna uzmemo li u obzir da, osim smanjenja broja mogućih terma, nismo mijenjali sintaktička svojstva β-redukcije. Međutim, sljedeći rezultat nam govori da je restrikcija koju radimo uvođenjem tipa ipak netrivijalna.

Teorem 2.2.4. *Neka je* $M \in \Lambda^\tau$*. Svaki redukcijski put terma M vodi do normalne forme.*

Gornje svojstvo naziva se svojstvo *stroge normalizacije*, te ima netrivijalne posljedice. Uočimo da svaki Turingov stroj čiji alfabet sadrži alfabet λ-računa i koji uvijek staje u stanju q_{DA} odlučuje Halting problem za jednostavno tipizirani λ-račun. Iz toga jednostavno slijedi da ovaj račun nije Turing-potpun. Nadalje, termi bez normalne forme nisu mogući u ovakvom računu pa vrijedi $\Lambda^\tau \subset \Lambda$.

Ovakav račun može se proširiti do Turing-potpunog računa dodavanjem kombinatora fiksne točke za svaki mogući tip računa. Kombinator fiksne točke tada ima tip $\mathbf{Y} : (\sigma \to \sigma) \to \sigma$.

2.3 Hindley-Milnerov algoritam

U ovoj točki definiramo Hindley-Milnerov algoritam za određivanje tipova u Curryjevom sustavu tipiziranja. Sustav koji demonstriramo preuzet je iz [2] te se razlikuje od originalnog sustava definiranog u [8] jer ne koristi polimorfne tipove niti dodatnu sintaksu. Ipak, ovaj sustav se jednostavno proširuje do originalnog a uvelike pojednostavljuje formalni

dokaz korektnosti. U poglavlju 3 opisat ćemo Haskellov sustav koji je gotovo identičan originalnom Hindley-Milnerovom sustavu.

Gruba ideja algoritma je sljedeća. Pravila derivacije tipa definirana u prošloj točki ovise o sintaksi λ-računa, odnosno načinu na koji je dani term izgrađen iz jednostavnijih terma. Promatrajući građu terma, možemo zaključiti u kojim odnosima moraju biti pojedini podtermi pomoću leme 2.2.2 Za početak se svakom podtermu pripisuje neodređeni tip, a zatim se gradi sustav jednadžbi čijim rješavanjem se dobiva najopćenitiji oblik tipa koji term može imati. Ukoliko term nije zatvoren, također je potrebno pronaći najmanji kontekst u kojem se može odrediti neki tip tog terma.

Napomena 2.3.1. *Ovdje je važno naglasiti distinkciju pristupa neodređenim tipovima koji opisujemo u odnosu na pristup koji koristi varijable tipova (poput originalnog Hindley-Milnerovog sustava). Kada pričamo o nekom općenitom tipu a označavamo ga primjerice sa σ, implicitno govorimo o cijeloj klasi tipova. S time na umu, svaku oznaku tipa koja ne označava eksplicitan tip možemo smatrati varijablom, iako tu ideju ne formaliziramo. Definicija supstitutora koja slijedi definira funkcije koje imaju poželjno svojstvo u smislu da su aditivne na skupu tipova. Na taj način, ukoliko neki tip σ preslikavamo u tip τ, u kompleksnijem tipu koji sadrži σ možemo σ interpretirati kao varijablu. Kako su tipovi nužno konačne duljine, slijedi da uvijek postoje bazni tipovi koji se ne javljaju u nekom tipu. Te tipove možemo koristiti kao varijable.*

Sada formaliziramo ideju rješavanja jednadžbi tipova.

Definicija 2.3.2. Supstitutor *je funkcija* $* : \textbf{\textit{Typ}} \to \textbf{\textit{Typ}}$ *takva da vrijedi*

$$*(\sigma \to \tau) \equiv *(\sigma) \to *(\tau)$$

U daljnjem tekstu umjesto $*(\sigma)$ pišemo σ^*. Nadalje, definiramo *unifikator* tipova.

Definicija 2.3.3. *Neka su $\sigma, \tau \in \textbf{\textit{Typ}}$. Za supstitutor $*$ kažemo da je* unifikator *za σ i τ ako vrijedi $\sigma^* \equiv \tau^*$. Za supstitutor kažemo da je najopćenitiji unifikator za σ i τ ako*

- $\sigma^* \equiv \tau^*$
- *Ukoliko $\sigma^{*_1} \equiv \tau^{*_1}$ za neki supstitutor $*_1$, tada postoji supstitutor $*_2$ takav da vrijedi $*_1 = *_2 \circ *$.*

Dakle unifikator dvaju tipova je supstitutor koji ih preslikava u isti tip. Sada je jasno da riješiti jednadžbu tipova znači pronaći unifikator tipova. Drugi dio definicije govori kako bi unifikator trebao biti „najizravniji" način da riješimo jednadžbu.

Kako smo znak „≡" koristili da naglasimo grafičku jednakost dvaju tipova, želimo li naglasiti da se radi o jednadžbi tipova koristimo znak jednakosti „=".

Definicija 2.3.4. *Neka je* $E = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$ *skup jednadžbi tipova. Kao* unifikator skupa E *definiramo svaki supstitutor koji je unifikator za sve jednadžbe u* E*, te u tom slučaju pišemo* $* \models E$.

Napomena 2.3.5. *Uočimo da iz definicije tipova vrijedi* $\sigma_1 \to \tau_1 \equiv \sigma_2 \to \tau_2$ *ako i samo ako* $\sigma_1 \equiv \sigma_2$ *i* $\tau_1 \equiv \tau_2$*. Iz toga možemo zaključiti da je proizvoljan supstitutor* $*$ *unifikator za skup* $E = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$ *ako i samo ako je unifikator tipova* $\sigma_1 \to \ldots \to \sigma_n$ *i* $\tau_1 \to \ldots \to \tau_n$.

U daljnjem tekstu izrazom $[\sigma := \tau]$ označavamo supstitutor koji je ekvivalentan funkciji identiteta u svim točkama osim točki σ koju preslikava u τ. Ukoliko je θ tip, supstituciju označavamo s $\theta\,[\sigma := \tau]$. Iste oznake koristimo za supstituciju unutar konteksta Γ.

Sljedeća lema govori da su pravila dedukcije tipa u skladu s pravilima supstitucije. Dokaz se može pronaći u [2].

Lema 2.3.6. *Neka vrijedi* $\Gamma \vdash M : \sigma$*. Tada* $\Gamma\,[\alpha := \tau] \vdash \sigma\,[\alpha := \tau]$.

Prirodno se postavlja pitanje je li moguće napisati algoritam koji pronalazi unifikator za dane tipove. Sljedeći nam teorem garantira postojanje takvog algoritma.

Teorem 2.3.7. *Postoji rekurzivna funkcija* U *koja prima uređeni par tipova a vraća supstitutor ili* `fail` *tako da vrijedi:*

- *ako postoji unifikator za* σ *i* τ*, tada je* $U(\sigma, \tau)$ *najopćenitiji unifikator za* σ *i* τ*, te*
- *ukoliko unifikator za* σ *i* τ *ne postoji, tada vrijedi* $U(\sigma, \tau) =$ `fail`.

Dokaz. Funkciju definiramo rekurzivno, po slučajevima. Sa α ćemo označiti proizvoljan bazni tip.

$$U(\tau, \alpha) = U(\alpha, \tau) = \begin{cases} [\alpha := \tau]\text{, za } \tau \text{ koji ne sadrži } \alpha, \\ id_{\mathbf{Typ}}\text{, ako } \tau = \alpha, \\ \texttt{fail}\text{, inače,} \end{cases}$$

$$U(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) = U(\sigma_1^{U(\sigma_2,\tau_2)}, \tau_1^{U(\sigma_2,\tau_2)}) \circ U(\sigma_2, \tau_2),$$

gdje je $id_{\mathbf{Typ}}$ funkcija identiteta na skupu **Typ**. Smatramo da je vrijednost funkcije U jednaka `fail` ukoliko se bilo koja komponenta evaluira u `fail`.

Indukcijom se može pokazati da funkcija U zadovoljava tražene uvjete (vidi [2]). □

Koristeći napomenu 2.3.5 ova se funkcija jednostavno proširuje na skupove. Promotrimo primjer unifikacije za skup $\{\alpha \to \beta \to \gamma = (\gamma \to \gamma) \to \delta \to \alpha\}$. Vrijedi:

$$U(\alpha \to \beta \to \gamma, (\gamma \to \gamma) \to \delta \to \alpha) =$$
$$U(\alpha^{U(\beta\to\gamma,\delta\to\alpha)}, (\gamma \to \gamma)^{U(\beta\to\gamma,\delta\to\alpha)}) \circ U(\beta \to \gamma, \delta \to \alpha)$$

Nadalje:

$$\begin{aligned} &U(\beta \to \gamma, \delta \to \alpha) = U(\beta^{U(\gamma,\alpha)}, \delta^{U(\gamma,\alpha)}) \circ U(\gamma, \alpha) \\ &= U(\beta\,[\gamma := \alpha]\,, \delta\,[\gamma := \alpha]) \circ [\gamma := \alpha] \\ &= U(\beta, \delta) \circ [\gamma := \alpha] = [\beta := \delta, \gamma := \alpha] \end{aligned}$$

Sada dobiveni izraz uvrštavamo u početni:

$$\begin{aligned} &U(\alpha^{U(\beta\to\gamma,\delta\to\alpha)}, (\gamma \to \gamma)^{U(\beta\to\gamma,\delta\to\alpha)}) \circ U(\beta \to \gamma, \delta \to \alpha) \\ &= U(\alpha\,[\beta := \delta, \gamma := \alpha]\,, (\gamma \to \gamma)\,[\beta := \delta, \gamma := \alpha]) \circ [\beta := \delta, \gamma := \alpha] \\ &= U(\alpha, \alpha \to \alpha) \circ [\beta := \delta, \gamma := \alpha] \end{aligned}$$

Iz definicije funkcije U slijedi $U(\alpha, \alpha \to \alpha) = \texttt{fail}$, pa ne postoji unifikator za gornje tipove.

Gornji teorem daje nam konstruktivan dokaz za nalaženje unifikatora nekog skupa jednadžbi tipova. Kako bismo za dani λ-izraz pronašli tip (ili pokazali da ne posjeduje tip), potrebno je još generirati skup jednadžbi koji opisuje odnose među tipovima podterma danog terma, na koji bismo mogli primijeniti funkciju U.

Teorem 2.3.8. *Za svaki kontekst* Γ*, term* $M \in \mathbf{\Lambda}$ *i* $A \in$ **Typ** *takav da* $FV(M) \subseteq dom(\Gamma)$ *postoji konačan skup jednadžbi* $E = E(\Gamma, M, A)$ *takav da za sve supstitutore* $*$ *vrijedi:*

- *ukoliko vrijedi* $* \models E(\Gamma, M, A)$*, tada* $\Gamma^* \vdash M : A^*$*, i obratno,*
- *ukoliko* $\Gamma^* \vdash M : A^*$*, tada* $*_1 \models E(\Gamma, M, A)$*, za neki* $*_1$ *takav da* $*$ *i* $*_1$ *imaju jednako djelovanje na tipove varijabli u* Γ *i* A*.*

Promotrimo prvo pažljivo iskaz teorema. Prva točka teorema garantira postojanje skupa jednadžbi tipova E takvog da, ukoliko postoji unifikator $*$ za taj skup, tada možemo pronaći tip za dani term M. Dodatno, taj tip je upravo tip σ^*, a može se deducirati iz Γ^*. Također nam je dan i obrat, odnosno ukoliko iz modificiranog konteksta Γ^* možemo izvesti tip σ^* za term M, tada $*$ nužno unificira skup E.

Dokaz. Skup $E(\Gamma, M, \sigma)$ definiramo rekurzivno po strukturi terma M na sljedeći način:

$$\begin{aligned} E(\Gamma, x, \sigma) &= \{\sigma = \Gamma(x)\} \\ E(\Gamma, MN, \sigma) &= E(\Gamma, M, \alpha \to \sigma) \cup E(\Gamma, N, \alpha) \\ E(\Gamma, \lambda x.M, \sigma) &= E(\Gamma \cup \{x : \alpha\}\,, M, \beta) \cup \{\alpha \to \beta = \sigma\}\,, \end{aligned}$$

gdje su α i β varijable (u smislu opisanom u 2.3.1) koje se ne pojavljuju u ostalim tipovima u tom retku. Dokaz se provodi indukcijom po strukturi M. Uzmimo da je $M \equiv x$ gdje

je x proizvoljna varijabla. Skup jednadžbi tada je jednak $E(\Gamma, x, \sigma) = \{\sigma = \tau\}$, gdje je τ tip pridružen x u Γ. Uzmimo sada proizvoljni unifikator $*$ tog skupa. Uočimo da ukoliko vrijedi $\Gamma \vdash x : \tau$, tada vrijedi $(x : \tau) \in \Gamma$ po definiciji. Nadalje, tada vrijedi $(x : \tau^* \in \Gamma)$, te konačno $(\Gamma^* \vdash x : \tau^*)$ čime je prva tvrdnja dokazana. Neka je sada $\Gamma^* \vdash x : \tau^*$. Skup E sada je jednak $E(\Gamma, x, \tau) = \{\tau = \tau\}$, iz čega slijedi tvrdnja.

Promotrimo sada slučaj kada je zadani term aplikacija. Neka vrijedi $* \models E(\Gamma, MN, \sigma)$ gdje su $M, N \in \mathbf{\Lambda}$ i pretpostavimo da tražene tvrdnje vrijede za M i N zasebno. Kako iz pretpostavke $*$ unificira skup $E(\Gamma, MN, \sigma)$, posebno vrijedi da $*$ unificira svaki od skupova $E(\Gamma, M, \alpha \to \sigma)$ i $E(\Gamma, N, \alpha)$. Kako smo pretpostavili da tvrdnje vrijede za M i N, slijedi da vrijedi $\Gamma^* \vdash M : (\alpha \to \sigma)^*$ i $\Gamma^* \vdash N : \alpha^*$. Korištenjem aditivnosti supstitutora te pravila dedukcije tipova zaključujemo $\Gamma^* \vdash MN : \sigma^*$.

Neka je sada $\Gamma^* \vdash MN : \sigma^*$. Vrijedi $E(\Gamma, MN, \sigma) = E(\Gamma, M, \alpha \to \sigma) \cup E(\Gamma, N, \alpha)$. Po pretpostavci, svaki $*_1$ koji djeluje na σ te tipove varijabli u Γ jednako kao $*$, vrijedi da $*_1$ unificira svaki od skupova pa unificira i njihovu uniju. Time je tvrdnja dokazana. Sličnim postupkom tvrdnju dokazujemo za apstrakciju. □

Dokazana tvrdnja ključna je za odlučivost problema određivanja tipa zadanog λ-terma, jer upravo unifikatorom skupa E za dani term M pronalazimo odgovarajući kontekst Γ i tip σ takve da $\Gamma \vdash M : \sigma$.

Promotrimo generiranje skupa $E(\Gamma, \lambda xy.xy, \sigma)$. Kako je term $\lambda xy.xy$ zatvoren, za kontekst uzimamo $\Gamma = \emptyset$.

$$\begin{aligned} E(\emptyset, \lambda xy.xy, \sigma) &= E(\{x : \alpha_1\}, \lambda y.xy, \beta_1) \cup \{\alpha_1 \to \beta_1 = \sigma\} \\ &= E(\{x : \alpha_1, y : \alpha_2\}, xy, \beta_2) \cup \{\beta_1 = \alpha_2 \to \beta_2\} \cup \{\alpha_1 \to \beta_1 = \sigma\} \\ &= E(\{x : \alpha_1, y : \alpha_2\}, x, \gamma \to \beta_2) \cup E(\{x : \alpha_1, y : \alpha_2\}, y, \gamma) \cup \\ &\cup \{\beta_1 = \alpha_2 \to \beta_2\} \cup \{\alpha_1 \to \beta_1 = \sigma\} \\ &= \{\alpha_1 = \gamma \to \beta_2, \alpha_2 = \gamma, \beta_1 = \alpha_2 \to \beta_2, \alpha_1 \to \beta_1 = \sigma\} \end{aligned}$$

Definicija 2.3.9. *Neka je $M \in \mathbf{\Lambda}$. Tada za uređeni par (Γ, σ) kažemo da je* glavni par *(principal pair) i pišemo $pp(M)$ ako vrijedi*

- $\Gamma \vdash M : \sigma$
- $\Gamma' \vdash M : \sigma' \Rightarrow$ *postoji supstitutor* $*$ *takav da* $\Gamma^* \subseteq \Gamma'$ *i* $\sigma^* \equiv \sigma'$.

Drugim riječima, glavni par (Γ, σ) nekog terma je uređeni par najmanjeg konteksta koji sadrži tip kojeg term može poprimiti, te samog tipa. Ukoliko je iz drugog konteksta Γ' izvediv neki drugi tip σ' za dani term, tada je kontekst opširniji a tip σ' je moguće prevesti u tip σ.

Uzmimo kao primjer term $\mathbf{K} \equiv \lambda xy.x$. Ovaj term nema slobodnih varijabli pa je ograničenje konteksta iz gornje definicije trivijalno ispunjeno. Vrijedi $\vdash \mathbf{K} : \sigma \to \tau \to \sigma$

(valjanost ovog tipa može se provjeriti direktno iz definicije pravila dedukcije). Uzmimo da su σ i τ proizvoljni bazni tipovi. Tvrdimo da je tada $pp(\mathbf{K}) = (\emptyset, \sigma \to \tau \to \sigma)$. Uzmimo da vrijedi $\vdash \mathbf{K} : \sigma' \to \tau' \to \sigma'$. Kako su σ i τ bazni tipovi, možemo ih preslikati u proizvoljan tip bez da narušimo aditivnost. Stoga očito postoji supstitutor $*$ takav da je $\sigma^* \equiv \sigma'$ i $\tau^* \equiv \tau'$, nakon čega tvrdnja slijedi zbog aditivnosti supstitutora. Intuitivno, supstitutori mogu samo povećati broj strelica, a tip $\sigma \to \tau \to \sigma$, gdje su σ i τ bazni tipovi, ima najmanji mogući broj strelica.

Vidjeli smo da zatvoreni termi ne trebaju kontekst kako bi se odredili dozvoljeni tipovi jer ne postoje slobodne varijable čiji tip ne možemo odrediti iz same strukture terma. U tom slučaju govorimo samo o *glavnom tipu* (engl. principal type), u oznaci $pt(M)$. Slijedi konstruktivni dokaz Hindley-Milnerovog algoritma za određivanje tipa zadanog terma u jednostavno tipiziranom računu.

Teorem 2.3.10. *Postoji izračunljiva funkcija* PP *takva da:*

- *ukoliko M ima tip, tada $PP(M) = (\Gamma, \sigma)$, gdje je $(\Gamma, \sigma) = pp(M)$,*
- *ukoliko M nema tip, tada $PP(M) =$* `fail`.

Dokaz. Neka je $M \in \mathbf{\Lambda}$, $FV(M) = \{x_1, \ldots, x_n\}$ i neka je $\Gamma_0 = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ i neka je $\sigma_0 = \beta$. Pritom pretpostavljamo da su $\alpha_1, \ldots, \alpha_n, \beta$ neodređeni tipovi, u smislu opisanom u napomeni 2.3.1.

Sada definiramo $pp(M)$:

$$PP(M) = \begin{cases} (\Gamma_0^*, A_0^*), & \text{ako } U(E(\Gamma_0, M, \sigma_0)) = *, \\ \texttt{fail}, & \text{ako } U(E(\Gamma_0, M, \sigma_0)) = \texttt{fail} \end{cases}$$

Pokažimo prvo da ukoliko M ima tip, da je tada $PP(M)$ glavni par od M. Uočimo da, ukoliko M ima tip, vrijedi da postoje kontekst Γ i tip σ takvi da $\Gamma \vdash M : \sigma$. Zbog 2.2.1 možemo pretpostaviti $dom(\Gamma) = \{x_1, \ldots, x_n\}$. Iz definicije Γ_0 i σ_0. slijedi da možemo pronaći supstitutor $*$ takav da $\Gamma_0^* = \Gamma$ i $\sigma_0^* \equiv \sigma$. Iz toga slijedi $\Gamma_0^* \vdash M : \sigma_0^*$, pa iz 2.3.8 slijedi da $* \models E(\Gamma_0, M, \sigma_0)$. Iz toga slijedi da je $U(E(\Gamma_0, M, \sigma_0)) \neq$ `fail`.

Pokažimo sada da ukoliko M nema tip, vrijedi $PP(M) =$ `fail`. Pretpostavimo da M nema tip i da vrijedi $PP(M) \neq$ `fail`. Označimo s $*$ supstitutor $U(E(\Gamma_0, M, \sigma_0))$. Tada iz definicije $PP(M)$ slijedi $* \models E(\Gamma_0, M, \sigma_0)$. Tada iz teorema 2.3.8 vrijedi $\Gamma^* \vdash M : \sigma_0^*$, što je kontradikcija s pretpostavkom da M nema tip.

Još je preostalo pokazati da ukoliko M ima tip, vrijedi da je $PP(M) = pp(M)$. Vidjeli smo da ukoliko M ima tip, izraz $U(E(\Gamma_0, M, \sigma_0))$ je definiran i prema teoremu 2.3.7 je najopćenitiji unifikator skupa $E(\Gamma_0, M, \sigma_0)$. Označimo $U(E(\Gamma_0, M, \sigma_0))$ s $*$. Nadalje, vrijedi $\Gamma_0^* \vdash M : \sigma_0^*$ prema 2.3.8. Pretpostavimo da je $\Gamma' \vdash M : \sigma'$ (ponovo pretpostavljamo

da je $dom(\Gamma) = FV(M)$). Neka je $*_1$ takav da je $\Gamma' = \Gamma_0^{*_1}$ i $\sigma' \equiv \sigma^{*_1}$. Tada istom argumentacijom kao ranije slijedi $*_1 \models E(\Gamma_0, M, \sigma_0)$. Iz teorema 2.3.7 slijedi da je $*$ najopćenitiji unifikator skupa $E(\Gamma_0, M, \sigma_0)$ pa slijedi da postoji $*_2$ takav da $*_1 = *_2 \circ *$. Slijedi

$$(\Gamma_0^*)^{*_2} = \Gamma_0^{*_1} \subseteq \Gamma'$$

te

$$(\sigma_0^*)^{*_2} \equiv \sigma_0^{*_1} \equiv \sigma'$$

Time smo demonstrirali da je (Γ_0^*, σ_0^*) glavni par za M. □

Primijenimo li algoritam na zatvoren term, očito je da je kontekst prazan a kao $pt(M)$ je dovoljno uzeti drugu komponentu $PP(M)$.

Konačno, kao korolar imamo sljedeću tvrdnju.

Teorem 2.3.11. *Određivanje te provjera tipa u jednostavno tipiziranom λ-računu* a la Curry *je odlučivo.*

Dokaz. Neka je dan term M. Očito vrijedi da M ima tip ako i samo ako vrijedi $PP(M) \neq$ `fail`. Promotrimo problem provjere tipa. Neka su dani term M i tip σ. Jednostavno se vidi da vrijedi

$$\vdash M : \sigma \iff \text{postoji supstitutor } * \text{ takav da vrijedi } \sigma = pt(M)^*$$

Iz prethodnog teorema znamo da možemo naći $pt(M)$. Još preostaje unifikacija tipova $pt(M)$ i σ, koju možemo provesti koristeći funkciju U iz teorema 2.3.7. □

U ovom smo poglavlju opisali način na koji možemo utvrditi tip zadanog λ-terma. Kao korolar smo dobili i mogućnost provjere za slučaj da nam je tip dan.

Završetkom ovog poglavlja obradili smo dvije glavne komponente funkcijskog programiranja: izračunavanje funkcija i pronalaženje tipova terma. U sljedećem poglavlju na primjeru Haskella demonstrirat ćemo modificirane varijante ovih ideja koje se koriste u praksi.

Poglavlje 3

Programski jezik Haskell

Kako je već spomenuto u poglavlju 1.2, osnovu Haskella čine funkcije, točnije kombinatori. Programiranje u Haskellu svodi se na definiranje potrebnih tipova koristeći skup baznih tipova te navođenje niza jednadžbi kojima definiramo kombinatore.

Želimo li stvoriti izvršni program, potrebno je definirati funkciju `main`. U tom slučaju izvršavanje programa počinje evaluacijom funkcije `main` i to je jedini obavezan dio programa.

Haskell koristi nepromjenjive podatke, što znači da vrijednosti nije moguće „pamtiti" u proizvoljnom trenutku već su sve vrijednosti koje koristimo dobivene kao rezultat evaluacije funkcija. Iako nepromjenjivost podataka nije nužna kako bi se jezik smatrao „funkcijskim" (npr. funkcijski jezik LISP koristi promjenjive podatke), ipak je paradigma specifična za funkcijske jezike.

Haskell također koristi strogi sustav tipova, što znači da nema implicitnih promjena tipova podataka. Ovo svojstvo nije specifično za funkcijsko programiranje (iako slijedi iz načina tipiziranja u λ-računu), pa se time nećemo posebno baviti.

Na početku dajemo osnovnu sintaksu funkcija u Haskellu, kako bismo olakšali praćenje primjera koji se javljaju kasnije. Zatim objašnjavamo razlike jednostavno tipiziranog λ-računa i originalnog Hindley-Milnerovog sustava koji koristi Haskell. Ukratko ćemo objasniti definiranje novih tipova, te pojam *klasa* tipova.

Na kraju promatramo kako se vrši redukcija Haskellovih izraza te koja je teorijska podloga takve redukcije.

Ovo poglavlje nije zamišljeno kao uvod u funkcijsko programiranje, već kao vrlo kratak uvod u formalnu teoriju funkcijskih jezika. Zato je naglasak na povezivanju pojmova definiranih u ranijim poglavljima, a ne na standardnim metodama funkcijskog programiranja ili programiranja u Haskellu.

Za jednostavan uvod u programiranje u Haskellu, te funkcijsko programiranje općenito, vidi [6].

3.1 Funkcije

U ovom poglavlju uvodimo osnovnu sintaksu koja će nam biti potrebna kasnije. Uz sintaksu, pojasnit ćemo neke paradigme, poput rekurzije i kompozicije. Također ćemo u Haskellu definirati kombinatore **map**, **foldl** i **foldr**, definirane u poglavlju 1.2.

Sintaksa

Krenimo s osnovnom sintaksom. Najjednostavniji oblik funkcija koje možemo definirati su funkcije s praznom domenom. Takve funkcije imaju neki bazni tip, odnosno tip koji nije funkcijski, a reprezentiraju sinonim za neku konkretnu vrijednost. Definiranje funkcije sastoji se od dva dijela: deklaracija tipa funkcije te definicija same funkcije. Zbog automatskog određivanja tipa, deklaracije tipa u sljedećim primjerima mogu se izostaviti.

```
const5 :: Int
const5 = 5
```

Ovime se definira funkcija `const5`, koja je cjelobrojnog tipa `Int` te je sinonim za brojku 5. Dvostrukom dvotočkom deklarira se tip, a jednadžbom kombinator čije djelovanje se navodi s desne strane znaka jednakosti.

Imena tipova uvijek započinju velikim slovom. Neki od baznih tipova koji su uvijek dostupni su cjelobrojni tip `Int`, cjelobrojni tip proizvoljne veličine `Integer`, realni tipovi `Float` i `Double`, znakovni tip `Char`, niz znakova `String`, booleovski tip `Bool` itd. Liste vrijednosti nekog tipa označavaju se uglatim zagradama, primjerice, `[Int]` označava listu cijelih brojeva. Liste mogu biti proizvoljne duljine po uzoru na liste definirane u 1.2.

Aplikacija se označava jednostavnim dopisivanjem argumenata zdesna funkciji. Ukoliko su argumenti i sami aplikacije, mora ih se odvojiti zagradom. Moguće je definirati funkcije koje koriste infiksnu notaciju. Primjeri takvih funkcija su standardni aritmetički operatori, operatori usporedbe itd. Želimo li funkciju definiranu u infiksnom obliku koristiti u standardnoj notaciji, potrebno ju je okružiti zagradama: `(+) 2 3` je ekvivalentno `2 + 3`. Proizvoljnu binarnu funkciju možemo koristiti u infiksnoj notaciji ukoliko je okružimo obrnutim apostrofima. Na primjer, izraz `div x y` ekvivalentan je izrazu `x ‘div‘ y` (gdje `div` označava cjelobrojno dijeljenje).

Uzmimo sada za primjer funkciju koja računa euklidsku normu danog vektora (liste) i pokažimo kako je moguće definirati je na različite načine. Za početak, demonstriramo definiciju pomoću rekurzije i pomoćne funkcije.

Pogledajmo sljedeći način definiranja:

```
sqnorm :: [Double] -> Double
sqnorm [] = 0
sqnorm l = (head l)^2 + sqnorm (tail l)
```

```
norm :: [Double] -> Double
norm = sqrt . sqnorm
```

Funkcija `sqnorm` služi nam da izračunamo kvadriranu normu liste, koju zatim prosljeđujemo funkciji `norm` kako bismo izračunali korijen. Funkciju `sqnorm` definiramo po točkama. Haskell dozvoljava definiranje funkcije prema njenom djelovanju na pojedinim elementima domene. U gornjem slučaju, `sqnorm` provjerava je li ulazna lista prazna (u oznaci „`[]`"). Ukoliko nije, prelazi se na redak ispod te se rekurzivno računa norma. Promotrimo korak evaluacije:

```
sqnorm [1,2,3,4]
       = (head [1,2,3,4])^2 + sqnorm (tail [1,2,3,4])
       = 1 + sqnorm [2,3,4]
```

i tako dalje. Funkciju `norm` definiramo pomoću kombinatora kompozicije dviju funkcija (označenog točkom). Prvo se primijenjuje desna funkcija `sqnorm`, a zatim se rezultat prosljeđuje funkciji koja računa korijen realnog broja. Uočimo da nismo eksplicitno koristili varijable, već smo za definiranje koristili funkcijsku jednakost, koja je analogna η-ekvivalenciji terma generiranoj η-redukcijom definiranom na stranici 9. Ovakvo definiranje naziva se definiranje bez točaka (engl. *point-free*) jer se ne koriste oznake varijabli, koje u matematičkom smislu možemo interpretirati kao *točke* domene funkcije koju koristimo.

Operator kompozicije ima ista svojstva kao u klasičnom funkcijskom računu.

Još jedna mogućnost za definiranje funkcija je korištenje lambda-funkcija. To su funkcije definirane sintaksom sličnom λ-računu, čije ime se ne javlja u globalnom kontekstu. Sintaksa je sljedeća:

```
plus2 = \x -> x + 2
```

gdje znak „\" označava znak „λ", a strelica je ekvivalent točke u λ-računu. Uočimo da smo u ovom slučaju pomoću funkcijske jednakosti vezali anonimnu funkciju uz ime `plus2`.

Kako lambda-funkcije nisu vezane uz ime, da bismo postigli rekurziju moramo koristiti kombinator fiksne točke *fix*

```
norm = sqrt .
    (fix
        (\f l -> if null l
                 then 0
                 else (head l)^2 + f (tail l)))
```

gdje je `null` funkcija koja provjerava je li dana lista prazna, a izraz `if A then B else C` kondicional. Kondicional ima svojstva opisana u 1.2, uz dodatak da je izraz A nužno tipa `Bool`, a B i C imaju isti tip.

U točki 1.2 objasnili smo pojam parcijalne aplikacije i funkcije višeg reda. U Haskellu su funkcije osnovni tip podataka, te se kao takve mogu prenositi kao argumenti drugim funkcijama. Prisjetimo se kombinatora **foldl** i **map**. Njihovi tipovi u Haskellu su oblika

```
map :: (a -> b) -> [a] -> [b]
foldl :: (a -> b -> a) -> a -> [b] -> a
```

gdje su `a` i `b` proizvoljni tipovi. Ove dvije funkcije su polimorfne u tipovima `a` i `b`, više o tome u točki 3.2.

Normu sada možemo definirati na sljedeći način:

```
norm = sqrt . (foldl (+) 0) . map (^2)
```

Uočimo da smo funkcije `foldl` i `map` parcijalno aplicirali na funkcije zbrajanja i potenciranja. U točki 3.3 vidjet ćemo da ovako definirana funkcija prolazi kroz listu samo jednom, iako na prvi pogled to nije očito.

Lokalne funkcije

U ovoj točki predstavljamo dvije ključne riječi koje omogućuju definiranje lokalnih funkcija (funkcija koje nisu vezane uz globalno ime već se mogu koristiti samo unutar tijela funkcije u kojoj su definirane).

Prvo predstavljamo ključnu riječ *where*. Promotrimo sintaksu na primjeru iz prošle točke:

```
norm = sqrt . sqnorm
    where
        sqnorm [] = 0
        sqnorm l = (head l)^2 + sqnorm (tail l)
```

Sada funkcija `sqnorm` više nije dostupna van tijela funkcije `norm`. Uočimo da su svi načini definiranja funkcija mogući i na ovaj način. Ukoliko definiramo više lokalnih funkcija, redoslijed definiranja nije bitan.

Alternativno možemo koristiti konstrukt *let*, koji uz mogućnost vezanja vrijednosti uz lokalne varijable ima dodatna svojstva koja ćemo proučiti u poglavljima 3.3 i 3.2. Izraz je uveden u ranim programskim jezicima nakon čega se počinje koristiti i kao sintaktičko proširenje λ-računa.

```
norm = let sqnorm [] = 0
           sqnorm l = (head l)^2 + sqnorm (tail l)
       in sqrt . sqnorm
```

Osnovna razlika ovih dvaju načina je da izraz *let* uvijek ima povratnu vrijednost, dok je *where* samo sintaktički dodatak.

3.2 Sustav tipova

Prema standardu definiranom u [13], Haskell koristi Hindley-Milnerov sustav tipova proširen s *klasama tipova*. U sljedećim točkama pojasnit ćemo neke značajke ovog sustava.

Polimorfizam

U poglavlju 2.3 vidjeli smo kako se λ-račun može tipizirati te kako koristiti tipove da bismo restringirali domene funkcija. Također smo opisali algoritam kojim se u jednostavno tipiziranom λ-računu može odrediti tip proizvoljnog terma. Pritom tip koji smo određivali nije bio jedinstven, odnosno umjesto jednog tipa odredili smo cijelu klasu mogućih tipova. U ovom poglavlju demonstrirat ćemo kako se opisani algoritam može proširiti na *polimorfne* tipove koji uz bazne tipove sadrže i *varijable tipova*.

Promotrimo sljedeći term i pokušajmo mu odrediti tip u jednostavno tipiziranom računu:

$$(\lambda y.yy)(\lambda x.x)$$

Prema teoremu 2.2.3, znamo da tip tog terma mora biti jednak tipu terma $(\lambda x.x)(\lambda x.x)$ jer u njega reducira. Međutim, pridružimo li neki tip σ termu $(\lambda x.x)$ u originalnom izrazu, vidimo da bi nakon redukcije trebao također imati tip $\sigma \to \sigma$. U primjeru na stranici 34, vidjeli smo da skup $\{\sigma = \sigma \to \sigma\}$ nema unifikator, pa slijedi da $(\lambda y.yy)(\lambda x.x)$ nema tip.

Vrijedi $\lambda x.x : \sigma \to \sigma$. Pritom σ nije nužno bazni tip, već proizvoljan tip iz cijele klase tipova. Međutim, u nekom izrazu, dani term može imati samo jedan tip. Polimorfni tipovi omogućuju kvantificiranje po tipovima, čime se omogućuje da jedan te isti term u nekom izrazu ima više tipova, ovisno o tome na koji se term aplicira. Zapis tipa polimorfne funkcije identiteta u takvom računu je sljedeći:

$$\lambda x.x : \forall\alpha.\alpha \to \alpha$$

Sada nakon redukcije oba podterma imaju tip $\lambda x.x : \forall\alpha.\alpha \to \alpha$. Taj se polimorfni tip zatim može *instancirati* u tip koji je *manje općenit*. Instanciranje znači potencijalna zamjena polimorfnih varijabli konkretnim tipovima, ili dodavanje konstruktora tipova da bi se oblikom terma ograničio skup potencijalnih tipova. Primjerice, tip $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ svojim oblikom obuhvaća manji broj mogućih tipova, pa je manje općenit od tipa $\forall\alpha.\alpha \to \alpha$. Instanciranjem se prvom termu $\lambda x.x$ pridružuje tip $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$, a drugom $\forall\alpha.\alpha \to \alpha$.

Više o polimorfnim tipovima i instanciranju može se pronaći u [5].
Promotrimo sada funkciju identiteta `id` kako je definirana u Haskellu:

```
id :: a -> a
id x = x
```

Funkcije koje smo do sada definirali imale su tipove sastavljene isključivo od baznih tipova i konstruktora tipova `->`. Tipovi označeni malim slovima označavaju varijable koje su implicitno univerzalno kvantificirane. Izraz `id id` valjan je u Haskellu, te vrijedi

```
id id ::  a -> a.
```

Međutim, sljedeći izraz nije valjan u Haskellu:

```
id' = (\x -> x x) id
```

Problem je u tome što se tip terma određuje statički, u trenutku prevođenja programa. Prisjetimo se Hindley-Milnerovog algoritma, točnije, funkcije E. Skup koji se njome generira je skup jednadžbi koji opisuje u kakvim su odnosima podtermi danog terma. Kako je već navedeno, iako određujemo skup mogućih tipova, dani term u nekom izrazu može imati samo jedan, konkretan tip.

Hindley-Milnerov algoritam izvorno je opisan u članku [8], gdje je definiran pomoću konstrukta *let* i polimorfnih tipova kakve smo opisali ranije. Tip oblika `a -> a` shvaćamo kao tip `forall a.a -> a`. Da bismo naglasili da želimo instancirati različite tipove istog terma, koristimo izraz *let*:

```
id' = let x = id in x x
```

U izrazu `x x` prvi se `x` instancira u term `id :: (a -> a) -> (a -> a)`, a drugi u `id :: a -> a`. Uočimo da su rezultantni tipovi i dalje polimorfni. Dakle, kao i u originalnom sustavu [8], Izraz *let* omogućuje polimorfno instanciranje izraza s polimorfnim tipovima.

Kako bi se tip nekog terma mogao odrediti, algoritam opisan u 2.3.7 potrebno je proširiti slučajem kada je term definiran izrazom *let*. Tada je pronalazak tipa u ovakvom sustavu također odlučiv (vidi [8]).

Ograničenje Hindley-Milnerovog polimorfnog sustava tipova je da su kvantifikatori tipova uvijek s krajnje lijeve strane, a nikad unutar pojedinih podtipova. Primjerice, dozvoljeni su sljedeći oblici tipova:

$$\forall\alpha.\alpha \to \alpha$$
$$\forall\alpha.\forall\beta.\alpha \to \beta,$$

no ne i izrazi oblika

$$(\forall\alpha.\alpha) \to (\forall\beta.\beta)$$

jer sadrže kvantifikator unutar podtipa.

Polimorfni tipizirani λ-račun (λ-račun drugog reda) naziva se *System F*, a omogućuje kvantifikator na bilo kojem mjestu u tipu. Neki programski prevodioci (poput prevodioca

Glasgow Haskell Compiler [14]) implementiraju ekstenzije koje omogućuju eksplicitno navođenje kvantifikatora te mogućnost proizvoljnog umetanja kvantifikatora. Važno je naglasiti da se time gubi mogućnost automatskog određivanja tipa jer je pronalazak tipa neodlučiv problem u sistemu F (vidi [11]).

Algebarski tipovi

Haskellova sintaksa dozvoljava definiranje vlastitih tipova. Definicija tipa sastoji se od dvije komponente: naziv tipa te naziv konstruktora objekata tog tipa. Promotrimo jednostavan primjer definicije booleovskog tipa.

```
data Bool = True | False
```

Ovakvom definicijom definiraju se dva objekta: *podatkovni* konstruktor (*data constructor*) (ovdje True ili False) i konstruktor *tipa* (*type constructor*) (ovdje Bool).

Podatkovni konstruktor je funkcija koja kao izlaz ima objekt definiranog tipa:

```
True :: Bool
False :: Bool
```

Navedeni podatkovni konstruktori su funkcije prazne domene, poput `const5` definirane na stranici 40. Moguća je definicija i konstruktora podataka koji primaju parametre. Sljedećim primjerom definiramo tip koji reprezentira uređeni par cijelog broja i niza znakova:

```
data MyPair = Empty | Pair Int [Char]
```

Kako su i sami funkcije, podatkovni konstruktori imaju sva svojstva kao i obične funkcije, pa je tako moguća kompozicija, parcijalna aplikacija itd.

Kada smo definirali tipove jednostavno tipiziranog računa u 2.1.2, spomenuli smo da takav račun ima samo jedan konstruktor tipova, kojim iz jednostavnijih tipova definiramo kompleksnije. Vidjeli smo da Haskell također sadrži taj konstruktor, koji možemo shvatiti kao binarnu funkciju.

Definiranjem tipa `Bool` u Haskellu stvaramo funkciju `Bool` prazne domene (nularna funkcija), kojom definiramo novi tip. Dakle izraz `Bool` je i naziv tipa i konstruktor tipa bez parametara.

Da bismo omogućili polimorfne parametre u podatkovnim konstruktorima, moramo koristiti jednomjesne ili višemjesne konstruktore tipova. Primjerice, trodimenzionalni vektor koji može sadržavati podatke proizvoljnih tipova definiramo na sljedeći način:

```
data Vector3 a = Vector a a a
```

Vidimo da je podatkovni konstruktor sada polimorfna funkcija. Također se sada jasno vidi razlika između tipa i konstruktora tipa: `Vector3` je naziv konstruktora tipa, a ne samog tipa. Da bismo pomoću konstruktora tipa dobili konkretan tip, potrebno ga je primijeniti na neki konkretan tip.

```
v1 :: Vector3 Bool
v1 = Vector True False False

v2 :: Vector3 Char
v2 = Vector 'a' 'b' 'c'

v3 :: Vector3 [Char]
v3 = Vector "asd" "qw" "a"

v :: Vector3 (Vector3 Bool)
v = Vector v1 v1 v1
```

Tip `Vector3 Bool` primjer je konkretnog tipa pa ga se može koristiti kao parametar u drugim konstruktorima tipova. Konstruktori tipova također mogu koristiti proizvoljan broj polimorfnih parametara.

Sada ćemo na primjeru liste demonstrirati definiranje rekurzivnih tipova podataka. Uočimo da je sintaksa vrlo slična Backus-Naurovoj formi za opisivanje gramatika.

Term **cons** bio je ključan za konstrukciju liste u točki 1.2. Koristeći **cons**, dodavanjem novih elemenata u listu, počevši od liste sa samo dva elementa, mogli smo proizvesti listu proizvoljne duljine. Nadopunimo li tako definiranu listu s objektom koji reprezentira praznu listu, dobivamo listu kakvu definira Haskell:

```
data List a = Null | Cons a (List a)
```

Operator `Cons` koristimo kao podatkovni konstruktor. Kako bi se pojednostavila sintaksa, umjesto izraza `Cons` koristi se infiksni operator označen s „:”, dok je oznaka prazne liste „[]”. Navođenje elemenata unutar uglatih zagrada samo je pokrata za uzastopno korištenje konstruktora:

```
[a1, a2, a3] = a1 : [a2, a3] = ... = a1 : a2 : a3 : []
```

Kako smo već napomenuli, indeksiranje elemenata ovakve liste nije moguće izvesti u konstantnom vremenu.

Uočimo važan detalj: konstruktor prazne liste nema polimorfni parametar. Ovdje je važno prisjetiti se da, unatoč tome, prazna lista ima konkretan tip. Primjerice, sljedeća usporedba nije dozvoljena iako obje strane reduciraju u praznu listu, jer im se tipovi ne podudaraju.

```
tail ['c'] == tail [True]
```

Vrijedi:

```
tail ['c'] :: [Char]
tail [True] :: [Bool]
```

Uočimo da u slučaju liste i konstruktor tipa i podatkovni konstruktor označavamo uglatim zagradama. Općenito, konstruktori tipova i podatkovni konstruktori „žive" u različitim kontekstima, te mogu imati iste nazive iako je koncept značajno različit.

Haskell ne omogućuje prosljeđivanje tipa kao parametra funkcijama (prema standardu opisanom u [13]), međutim postoje i računi u kojima nema semantičke podjele na tipove i terme, poput *Calculus of Inductive Constructions (CiC)* (vidi [4]). Ovakvi sustavi koriste se primjerice u automatskim verifikatorima teorema (poput funkcijskog jezika *Coq* [15]).

Podudaranje uzoraka

Uz konstruktore tipova veže se pojam podudaranje uzoraka (engl. *pattern matching*). U prošloj točki definirali smo listu te pojasnili da je oznaka liste samo pokrata za uzastopno korištenje operatora konkatenacije.

Uočimo da je za bilo koju listu objekata redoslijed korištenja konstruktora jedinstveno određen. Primjerice, lista `[1,2,3]` sastavljena je korištenjem konstruktora `Null`, te trima uzastopnim aplikacijama konstruktora `Cons` redom s parametrima 3, 2 i 1. Ovo pravilo vrijedi i općenitije: za svaki podatak možemo odrediti kojim je podatkovnim konstruktorom generiran.

To znači da za svaki algebarski tip možemo uvijek odrediti koji je konstruktor zadnji korišten da bi se dobio konačni podatak. Promotrimo sada ponovno definiciju funkcije `sqnorm` sa strane 40:

```
sqnorm [] = 0
sqnorm l = (head l)^2 + sqnorm (tail l)
```

Pojasnili smo da prvim retkom provjeravamo je li dana lista prazna. Uočimo da je izraz `[]` jedan od konstruktora liste. Koristeći činjenicu da je uvijek moguće odrediti konstruktor koji je korišten, moguća je sljedeća sintaksa:

```
sqnorm [] = 0
sqnorm (x:xs) = x^2 + sqnorm xs
```

U drugom se retku dana lista „rastavlja" na sastavne dijelove, a svaki od dijelova veže se uz neko ime. U ovom slučaju uz ime `x` vezali smo prvi parametar konstruktora `Cons`, a s `xs` drugi.

Sada možemo jednostavno definirati operatore `map` i `foldr`/`foldl`.

```
map [] = []
map f (x:xs) = f x : map f xs

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Case izraz

Definiranje funkcije po slučajevima na navedeni način samo je pokrata za izraz *case*. Ekvivalentan način definiranja, s eksplicitnim korištenjem izraza *case*, izgledao bi ovako:

```
sqnorm l = case l of
                 [] -> 0
             (x:xs) -> x^2 + sqnorm xs
```

Podudaranje uzoraka moguće je koristiti kod bilo kojeg definiranja vrijednosti.

Zapisi

Želimo li primjerice usporediti prve dvije komponente vektora definiranog na stranici 45, koristimo podudaranje uzoraka na konstruktoru vektora. Pritom znakom „_” označavamo da danu vrijednost ne vežemo uz neko ime već je ignoriramo.

```
data Vector3 a = Vector a a a
usporedi (Vector x y _) = x == y
```

Želimo li samo saznati vrijednost neke komponente možemo, koristeći podudaranje uzoraka, definirati funkcije za svaku komponentu. Mana ovakvog pristupa je da broj funkcija raste s brojem argumenata, pogotovo kada postoji više konstruktora. Stoga je stvorena sintaksa kojom se automatizira ekstrakcija vrijednosti iz danog objekta, na način da se pojedini parametri imenuju. Sintaksa je sljedeća:

```
data Vector3 a = Vector { prva :: a
                        , druga :: a
                        , treca :: a }
```

Sada vrijedi `prva (Vector 1 2 3) = 1`. Ovakva definicija naziva se definicija *zapisom* (engl. *record*).

Klase tipova

Osim mogućnosti definiranja tipova, Haskell pruža mogućnost definiranja *klasa tipova* (engl. *type classes*). Klase tipova omogućuju *preopterećivanje* (engl. *overloading*) funkcija, odnosno definiranje funkcija čije djelovanje ovisi o tipu podatka na koji se primjenjuje. Po načinu korištenja, klase tipova srodne su pojmu sučelja u objektno orijentiranim jezicima.

Ponekad se prirodno javljaju ograničenja na tip podataka nad kojim je moguće izvršiti neku operaciju. Primjerice, želimo li vidjeti javlja li se neka vrijednost u danoj listi, potrebno je iterirati po listi i svaki element usporediti s danom vrijednošću. To znači da na tipu vrijednosti koju tražimo mora biti definirana operacija usporedbe, kojom možemo ustvrditi jesu li dvije vrijednosti jednake.

Pogledajmo kao primjer tip funkcije `elem` kojom se utvrđuje je li dana vrijednost element liste:

```
elem :: Eq a => a -> [a] -> Bool
```

Deklaracija tipa počinje oznakom `Eq a`, kojom se označava da tip koji je dozvoljeno supstituirati na mjesto varijable `a` mora pripadati klasi `Eq`. Ta klasa objedinjuje sve tipove koje je moguće uspoređivati. Definicija klase je sljedeća:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Želimo li da neki tip bude element klase, potrebno je za dani tip definirati sve funkcije koje ta klasa zahtijeva. U slučaju klase `Eq` to su funkcije == i /= kojima se ispituje jesu li dani objekti jednaki. Uočimo da tijela funkcija nisu definirana jer konkretna definicija ovisi o tipu `a` koji nije poznat.

Tipovi koji pripadaju određenoj klasi nazivaju se *instance* te klase. Za tip `Vector3`, instanciranje izgleda ovako:

```
instance (Eq a) => Eq (Vector3 a) where
        (Vector x1 y1 z1) == (Vector x2 y2 z2) =
                x1 == x2 && y1 == y2 && z1 == z2
        x /= y = not (x == y)
```

Tip `a` u gornjoj definiciji i sam mora pripadati klasi `Eq` jer u definiciji jednakosti vektora smatramo da možemo usporediti elemente tipa `a`. Za formalno proširenje Hindley-Milnerovog sustava tipova klasama tipova i odlučivost tipiziranja u takvom sustavu, vidi [9].

Sada ćemo ukratko opisati klasu `Monad`, koja služi za nizanje transformacija nad nekim podacima. Ova klasa ima brojne primjene u funkcijskom programiranju (vidi primjerice [6]) koje daleko premašuju sadržaj ovog rada. Mi ćemo ovdje demonstrirati samo jednostavnu signalizaciju greške. Za početak, definiramo tip `Maybe a` na sljedeći način:

```
data Maybe a = Nothing | Just a
```

Ovaj tip zamišljen je kao omotač za operaciju čiji rezultat ne mora biti definiran. Kao motivaciju, promotrimo funkciju `sqrt` koja nije definirana za negativne brojeve. Kako bismo signalizirali grešku, povratni tip ove funkcije možemo modificirati tako da se vraća

`Nothing` u slučaju greške, ili `Just v` (gdje je `v` traženi korijen), ukoliko je broj nenegativan.

```
sqrtM :: Floating a => a -> Maybe a
sqrtM x = if x < 0
              then Nothing
              else Just (sqrt x)
```

Problem ovakve definicije jest da više nije moguće nizanje:

```
sqrtM (sqrtM 10)
```

nije valjan izraz jer se povratni tip i ulazni tip operacije ne poklapaju.

Postavlja se pitanje kako bismo mogli nizati gornju operaciju. Želimo funkciju koja kao ulazni podatak prima `Just x` ili `Nothing`, i funkciju `sqrtM`, a kao rezultat vraća rezultat evaluacije izraza `sqrtM x`, ili `Nothing` u slučaju greške. Definiramo:

```
apply :: Maybe a -> (a -> Maybe a) -> Maybe a
apply Nothing f = Nothing
apply (Just x) f = f x
```

Sada je nizanje moguće:

```
((Just 9) `apply` sqrtM) `apply` sqrtM
```

Uočimo da ukoliko je u bilo kojem trenu međurezultat `Nothing`, tada je i konačni rezultat `Nothing`. Na taj način možemo signalizirati da se u nizu operacija dogodila greška.

Po uzoru na gornji primjer, klasa `Monad` definirana je sljedećim funkcijama:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Prva razlika u odnosu na viđene definicije jest da varijablom `m` označavamo *konstruktor* tipova koji prima jedan parametar. Ta činjenica je vidljiva u oznaci tipa `m a`, gdje primjenjujemo konstruktor tipa `m` na polimorfni tip `a`. Funkcija `(>>=)` analogna je funkciji `apply` koju smo ranije definirali. Funkcija `return` stvara „omotač" oko dane vrijednosti, kako bi se funkcija `(>>=)` mogla primijeniti (analogno korištenju konstruktora `Just`). Opisan tip `Maybe` instanca je ove klase, te vrijedi:

```
return = Just
(>>=) = apply
```

3.3 Evaluacija po potrebi

Evaluacija po potrebi (engl. *call-by-need*) ili lijena (*lazy*) evaluacija naziv je posebne vrste redukcije kojom se nastoji izbjeći negativna svojstva β-redukcije, te stvoriti redukciju pogodnu za implementaciju. Ovakva redukcija spoj je redukcije s *dijeljenim redeksima* i evaluacije *call-by-name*, koje ćemo objasniti u daljnjem tekstu.

Dijeljenje redeksa

U poglavlju 1.4 spomenuli smo da redukcija normalnim redosljedom uvijek vodi do normalne forme ukoliko ona postoji. Iz perspektive programskih jezika, to svojstvo je poželjno jer daje kanonski način reduciranja terma koji je najbolji mogući u smislu da evaluacija programa staje ako je to moguće.

Međutim, redukcija normalnim redosljedom nije nužno *najbrži* put do normalne forme ukoliko brojimo korake redukcije. Promotrimo pobliže supstituciju varijable nekim termom. Primjerice, u izrazu

$$(\lambda x.xx)M \to MM$$

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 izraz reducira u normalnu formu. Uočimo da bi aplikativnim redom term M reducirali samo jednom.

Prisjetimo se, znakom „$\equiv$" naznačavali smo grafičku jednakost, ali sam znak se ne pojavljuje u λ-računu i služi samo za pojednostavljivanje zapisa. Primjerice, term M u izrazu $(\lambda x.xx)M$ samo označava da se na odgovarajućem mjestu nalazi term označen s M. Ukoliko je $M \equiv (\lambda x.x)N$, vrijedi

$$MM \equiv ((\lambda x.x)\, N)\ ((\lambda x.x)\, N)$$

te ne postoji način poistovjećivanja prve i druge pojave terma $(\lambda x.x)N$.

Ovdje ćemo dati samo kratku intuitivnu predodžbu metode korištene za premošćivanje tih problema. Za formalan opis, upućujemo na [1] za formalni koncept indeksiranih redeksa i redukcije, te [7] i [10] za dva pristupa semantici računa s evaluacijom po potrebi, kojim se nastoji dati opis sustava korištenog u Haskellu.

Kako bismo dobili označene redekse, u svakom se redeksu prvom znaku λ slijeva pridružuje neki indeks. Gornji primjer sada transformiramo u sljedeći izraz:

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

Označimo korak redukcije u kojoj kontraktiramo redeks indeksa i s $\to_i$. Sada vrijedi:

$$(\lambda_1 x.xx)\,((\lambda_2 y.y)N) \to_1 ((\lambda_2 y.y)N)\ ((\lambda_2 y.y)N) \to_2 NN$$

Na taj smo način u drugom koraku kontraktirali oba redeksa istovremeno. Ova metoda naziva se dijeljenje redeksa (engl. *sharing*), ili alternativno redukcija grafa (engl. *graph reduction*). Ime dolazi iz činjenice da u reprezentaciji terma grafom (kako je opisano u [10]), čvorove grafa koji reprezentiraju isti term M stapamo u jedan čvor, a terme koji sadrže term M spajamo s novostvorenim čvorom, čime smo smanjili broj čvorova u grafu.

Ovakva redukcija nikad neće imati više koraka od redukcije aplikacijskim redosljedom (vidi [3]).

Za naglašavanje ovakvog dijeljenja redeksa koristi se konstrukt *let* te smatramo da je sljedeće ekvivalentno:

$$(\lambda x.M)N \equiv \text{let } x = N \text{ in } M$$

Naravno, izrazi su ekvivalentni samo iz perspektive redukcije. U točki 3.2 demonstrirali smo razliku u interpretaciji polimorfnih izraza u λ-funkcijama i izrazima *let*.

Važno je naglasiti da ovakva redukcija nije propisana standardom definiranim u [13], već je samo implementacijski detalj, korišten primjerice u prevodiocu *GHC*, opisanom u [14].

Redukcija *call-by-name*

U ovoj točki dajemo pojašnjenje redukcije u Haskellovim izrazima. Haskell koristi lijenu (engl. *lazy*) evaluaciju izraza, koja je ime dobila prema svojstvu da se niti jedan izraz ne evaluira više nego je „potrebno". Kako smo spomenuli, lijena evaluacija spoj je evaluacije *call-by-name* i redukcije s dijeljenim redeksima. U ovom poglavlju dat ćemo primjere evaluacija izraza te pojasniti što znači da je neki izraz potrebno evaluirati.

Dijeljenje redeksa opisali smo u prošloj točki. Ovdje predstavljamo redukciju *call-by-name*. To je varijanta redukcije normalnim redosljedom u kojoj se ne reduciraju izrazi koji su apstrakcije.

Prisjetimo se terma Ω za koji vrijedi $\Omega =_\beta \Omega\Omega$. Promotrimo sada term $\lambda x.\Omega$. Ovaj term nema normalnu formu u klasičnoj β-redukciji jer i normalan i aplikativni redoslijed reduciraju term Ω. Međutim, ukoliko uvedemo ograničenje redukcije na terme koji nisu apstrakcija, ovaj term je u normalnoj formi za takvu redukciju.

U poglavlju 1.5 naglasili smo da poistovjećivanje svih terma bez normalne forme vodi do inkonzistentne teorije, te smo stoga koristili pojam rješivosti. Nadalje, u definiciji 1.5.20 definirali smo HNF, a u teoremu 1.5.21 pokazali smo da je term rješiv ako i samo ako ima HNF. U kontekstu programskih jezika, ova se definicija oslabljuje te se koristi pojam slabe (engl. *weak*) HNF, u oznaci WHNF, koji obuhvaća sve izraze koji su apstrakcije (vidi [12]). U tom kontekstu izraz $\lambda x.\Omega$ je u WHNF.

Nadalje, ukoliko term nema WHNF, tada nema niti HNF pa nije rješiv. To nam daje opravdanje da terme bez WHNF poistovjetimo, te označimo s $\bot$.

Redukcija *call-by-name* primjer je nestriktne redukcije. 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. Očito je redukcija *call-by-name* nestriktna, jer vrijedi $(\lambda xy.y)\Omega \to \lambda y.y$. Međutim, vrijedi i više: $(\lambda xy.x)\Omega \to \lambda y.\Omega$, te je rezultantni term u normalnoj formi ovakve redukcije. To nam govori da je moguće da term sadrži nedefiniran izraz, a da evaluacija cijelog izraza ipak završi. Sada ćemo vidjeti kakve to posljedice ima na funkcijske jezike.

U kontekstu programa, izraz je 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* (engl. *impure*), 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.

Potpuno aplicirani konstruktori kao povratnu vrijednost imaju apstraktan objekt nekog tipa. Kako su ti objekti konstante, interpretiramo ih kao terme u normalnoj formi pa su nužno i u WHNF. Shvatimo li parcijalno aplicirane funkcije u Haskellu kao analogone apstrakcije u λ-računu (u biti su nešto općenitije jer funkcija ne mora biti apstrakcija da bi bila parcijalno aplicirana), smatramo da je izraz u WHNF ukoliko je parcijalno aplicirana funkcija. Uočimo da tu pripadaju i parcijalno aplicirani 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, kako ćemo vidjeti u primjerima koji slijede.

Primjeri

Sada ćemo dati dva primjera koja koriste evaluaciju po potrebi. Kako smo napomenuli, pojedini izrazi mogu biti u različitim fazama evaluacije, ali uvijek u WHNF. U daljnjem tekstu s # označavat ćemo izraz koji u potpunosti nije evaluiran.

Kao prvi primjer vraćamo se na funkciju `sqnorm` definiranu ranije. Na kraju točke 3.1 naglasili smo da tako definirana funkcija prolazi po listi samo jednom. Sada ćemo proći kroz evaluaciju korak po korak. Funkciju `(+)` koristimo u infiksnom obliku kako bismo naglasili redoslijed evaluiranja. Vrijedi:

```
sqnorm [1, 2, 3, 4] = ((foldr (+) 0) . map (^2)) [1, 2, 3, 4]
                    = foldr (+) 0 (map (^2) [1, 2, 3, 4])
                    = (+) #1 (foldr (+) 0 #2)
```

gdje vrijedi

```
#1 : #2 = map (^2) [1, 2, 3, 4]
```

Uočimo da u ovom trenu evaluacija funkcije `sqnorm` ne može stati, jer je rezultantni izraz aplikacija. Stoga se evaluacija nastavlja evaluiranjem izraza `#1` i `#2`. Slijedi:

```
map (^2) [1, 2, 3, 4] = 1^2 : map (^2) [2, 3, 4]
```

Pomoću podudaranja uzoraka, funkcija `foldr` sada može deducirati sljedeće: `#1 = 1^2` i `#2 = map (^2) [2, 3, 4]`. Sada se evaluacija funkcije `foldr` nastavlja:

```
#1 + foldr (+) 0 #2
             = (+) 1^2 (foldr (+) 0 (map (^2) [2, 3, 4]))
             = (+) 1^2 (#3 + foldr (+) 0 #4)
```

Daljnja redukcija odvija se već opisanim redom. Konačni je rezultat

```
(+) 1^2 ((+) 2^2 ((+) 3^2 4^2)) = 1^2 + 2^2 + 3^2 + 4^2
                                 = 1 + 4 + 9 + 16
                                 = 30
```

Kako su opisani izrazi svi bili aplikacije (pa nisu u WHNF), evaluacija nije mogla stati u niti jednom trenutku dok nije dobiven konačni rezultat. Također, uočimo da smo kroz danu listu iterirali samo jednom, što je posljedica ranije opisanih svojstava normalne redukcije: prvo se evaluira krajnje lijevi redeks, što u ovom kontekstu znači da se efekti primijenjenih funkcija akumuliraju te istovremeno primjenjuju na argumente.

Sada dajemo primjer u kojem ćemo, osim reda evaluacije, demonstrirati rad s beskonačnim strukturama podataka. Promotrimo definiciju funkcije kojom računamo Fibonaccijeve brojeve.

```
fibs = 0 : 1 : recSum fibs (tail fibs)
    where
        recSum (x:xs) (y:ys) = (x + y) : recSum xs ys
```

Pomoćna funkcija `recSum` uzima dvije liste a vraća treću čiji elementi su redom sume elemenata ulaznih listi. Uočimo da `recSum` ne mora imati definirane rubne uvjete jer unaprijed pretpostavljamo da će rezultantna lista biti beskonačna. Funkcija `fibs` je rekurzivno definirana lista. Označimo li nedefinirani dio liste s `#1`, vrijedi sljedeće:

```
fibs = 0 : 1 : recSum (0 : 1 : #1) (tail (0 : 1 : #1))
```

Uočimo da je ovaj izraz u WHNF jer se s krajnje lijeve strane (u infiksnoj notaciji) nalazi konstruktor liste. Prema tome, izraz u ovom obliku ne mora se dalje evaluirati ukoliko se to ne zatraži. Vrijedi:

```
head fibs = head (0 : 1 : #1) = 0
```

Pretpostavimo sada da u nekom vanjskom izrazu tražimo n-ti element liste i pokažimo da će evaluacija u tom slučaju uvijek stati. Za $n > 2$ potrebno je generirati sljedeći element liste.

```
fibs = 0 : 1 : recSum (0 : 1 : #1) (tail (0 : 1 : #1))
     = 0 : 1 : recSum (0 : 1 : #1) (1 : #1)
     = 0 : 1 : (0 + 1) : recSum (1 : #1) #1
```

Uočimo da sada vrijedi

```
#1 = 1 : #2
```

gdje je `#2` novi neevaluirani dio liste. Sada slijedi:

```
fibs = 0 : 1 : 1 : recSum (1 : 1 : #2) (1 : #2)
```

Sada je definiran i treći element liste `fibs`. Nastavkom ovakve evaluacije možemo generirati bilo koji element. Uočimo da će uvijek preostati dio liste #*n* koji nije definiran. Također, u svakom koraku evaluacije ovaj je izraz u WHNF, zbog čega evaluacija može stati.

Time smo demonstrirali redukciju *call-by-name*.

Bibliografija

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

[2] H. P. Barendregt, W. Dekkers, R. Statman, *Lambda Calculus with Types*, `www.cs.ru.nl/~henk/book.pdf`, elektronička knjiga, preuzeto 20. 11. 2013.

[3] R. Bird, P. Wadler, *Introduction to functional programming*, Prentice Hall International Series in Computing Science, 1988.

[4] T. Coquand, G. Huet, *The Calculus of Constructions*, Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[5] H. Geuvers, *Introduction to Type Theory*, Language Engineering and Rigourous Software Development, LNCS 5520, Springer, 2009.

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

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

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

[9] P. Wadler, S. Blott, *How to make* ad-hoc *polymorphism less* ad-hoc, Glasgow University, 1988.

[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

[12] *FOLDOC: Free On-Line Dictionary Of Computing*,
`http://foldoc.org`,
pristupljeno 22.1.2014.

[13] `http://www.haskell.org/onlinereport/haskell2010`,
službena dokumentacija, pristupljeno 29.1.2014.

[14] `http://www.haskell.org/ghc/docs/7.6.3/html/`,
službena dokumentacija, pristupljeno 20.1.2014.

[15] `http://coq.inria.fr/distrib/current/refman/`,
službena dokumentacija, pristupljeno 2.2.2014.

[16] `http://okmij.org/ftp/Computation/fixed-point-combinators.html`,
pristupljeno 21.1.2014.

Sažetak

U ovom radu, demonstrirali smo dva osnovna koncepta formalne teorije funkcijskog programiranja: *redukciju* i *tipiziranje*. U prvom smo poglavlju, na primjeru netipiziranog λ-računa, proučili svojstva β-redukcije kao osnovnog načina transformacije λ-terma, te dokazali da je takav račun dovoljno moćan da izrazi bilo koju izračunljivu funkciju.

U drugom poglavlju cilj je bio, uz definiciju tipova i tipiziranih termi, formalno predstaviti Hindley-Milnerov algoritam. Definirali smo što znači rješiti jednadžbu tipova, a kao korolar Hindley-Milnerovog algoritma demonstrirali smo da je određivanje tipa nekog terma, u jednostavno tipiziranom λ-računu *a la Curry*, izračunljivo.

U zadnjem smo poglavlju na manje formalan način pojasnili neke praktične varijante redukcije i tipova, kakve se implementiraju u programske jezike, koristeći pritom jezik *Haskell* kao primjer. Fokus je bio na polimorfnim tipovima i klasama tipova kao sustavu tipova te na *lijenoj* redukciji kao načinu reduciranja izraza.

Summary

In this work, we have demonstrated two basic concepts of the formal theory of functional languages: *reduction* and *typing*. In the first chapter, using the untyped calculus as a basis, we have studied properties of β-reduction as a basic method of transformation of λ-terms, and proved that the untyped λ-calculus is a theory powerful enough to express any computable function.

In the second chapter, the goal was to present the Hindley-Milner algorithm in a formal manner, after defining types and typed terms. We defined what it means to solve type equations and, as a corollary of the Hindley-Milner algorithm, proved typing is computable in the case of the simply typed λ-calculus *a la Curry*.

The final chapter was a less formal presentation of some more practical variants of reduction and typing used in programming languages, using the language *Haskell* as an example. The main focus was on polymorphic types and type-classes, and the *call-by-need* reduction, concerning type systems and reduction respectively.

Životopis

Rođen sam 20. travnja 1989. godine u Čakovcu. U Varaždinu, 1996. godine upisujem Drugu osnovnu školu Varaždin, koju završavam 2004. godine, kada upisujem matematički smjer Prve gimnazije Varaždin. U Varaždinu također pohađam Školu stranih jezika Kezele gdje tokom cijelog osnovnoškolskog i srednješkolskog obrazovanja učim engleski jezik, a od 2000. do 2003. godine i njemački jezik. Srednju školu završio sam 2008. godine, te iste godine upisao preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Preddiplomski studij završavam 2011. godine. Studij nastavljam na smjeru „Računarstvo i matematika", gdje sam na prvoj godini studija bio studentski demonstrator iz kolegija „Matematička logika". Tokom diplomskog studija samoinicijativno proučavam lambda račun i funkcijsko programiranje, koji na poticaj prof. dr. sc. Mladena Vukovića postaju tema mog diplomskog rada. Krajem diplomskog studija nagrađen sam Priznanjem za izniman uspjeh na studiju.