Modularni interpreteri u Haskell-u
Verzija 1.0

Viktor Kunčak, IV_C

Diplomski rad

Mentor: Prof. Dr. Mirjana Ivanović

Prirodno-matematički fakultet, Novi Sad

8. Juni 2000

Apstrakt

Cilj ovog rada je bilo ispitivanje prednosti koje u razvoju složenih programa pružaju mogućnosti savremenih funkcionalnih programskih jezika. U realizaciji je korišćena implementacija lenjog čisto funkcionalnog programskog jezika Haskell koja podržava i niz proširenja u odnosu na standard Haskell98, među kojima su i multiparametarske klase. Korišćenjem ovih mogućnosti jezika razvijene su komponente za modularni interpreter zasnovan na denotacionoj semantici. Komponente podržavaju različite semantičke osobine jezika i mogu se međusobno kombinovati. Time je kreirano okruženje koje omogućava jednostavnu realizaciju prototipova interpretera i eksperimentisanje sa različitim dizajnima jezika. Postignut je viši stepen modularnosti u odnosu na neke ranije pokušaje jer je i specifikacija leksike i sintakse jezika modularna. Leksika komponenti interpretera se zadaje regularnim izrazima, a leksički analizator je zasnovan na lenjoj konstrukciji prelaza i stanja deterministčkog konačnog automata. Modularnost specifikacije sintakse je ostvarena zadavanjem sintakse pomoću infiksnih, prefiksnih i postfiksnih operatora sa prioritetima. Uz ranije rezultate koji realizuju modularnost apstraktne sintakse i denotacione semantike, ovo omogućava modularnu specifikaciju kompletnog interpretera, a pristup se može proširiti i na pisanje kompajlera.

Sadržaj

1 Predgovor

U početku je za cilj ovog rada postavljena realizacija nekih od savremenih tehnika za implementaciju funkcionalnih programskih jezika. Kako su ove tehnike prilično složene, bilo je potrebno opravdati potrebu za programiranjem u funkcionalnom stilu. To je navelo autora da i samu realizaciju implementacije jezika započne u jednom funkcionalnom programskom jeziku i tako ispita pogodnost funkcionalnih jezika kao jezika za pisanje prototipova složenih programa. Značajno mesto u trenutnim pristupima pisanju jezičkih procesora u funkcionalnim programskim jezicima pripada upravo modularnim intepreterima. Pri tome ovde modularnost ne znači samo podelu programa na module, već mogućnost da se nezavisno realizuju komponente interpretera za različite osobine jezika (npr. aritmetika, naredba dodele, petlje, lambda izrazi, obrada grešaka, nedeterminizam). Kombinovanjem ovih komponenti se dobija interpreter koji podržava sve te osobine. Ovaj pristup se činio posebno privlačan zbog toga što živimo u trenutku kada modularni razvoj programa zaista ima priliku da promeni način na koji funkcioniše softverska industrija. Pisanje interpretera na modularan način stoga je predstavljalo težnju da se istovremeno dotakne implementacija programskih jezika, problemi modularnosti u razvoju programa i pogodnost Haskell-a za razvoj složenih programa.

1.1 Programski jezici

Ovaj rad je motivisan problemom dizajna i implementacije programskih jezika i to na dva načina. S jedne strane se ispituju mogućnosti programiranja u Haskell-u kao jeziku koji podržava mnoge savremene koncepte programskih jezika. Sa druge strane, ove mogućnosti se ispituju na primeru pisanja modularnog interpretera, što je pristup koji sam po sebi predstavlja eksperiment kako u oblasti dizajna programskih jezika, tako i u primeni modularnosti kao generalnog pristupa u softverskom inženjerstvu.

Opravdanje za ovakvu tematiku leži u značaju koji programski jezici imaju za rešavanje problema na računaru. Kao što su lingvistička istraživanja pokazala da postoji veza između načina na koji pripadnici određene zajednice razmišljaju i prirodnog jezika kojim se služe, tako je i iskustvo pokazalo da i programski jezik utiče na način rešavanja problema koji se u njemu opisuju. Zato nije iznenađujuće što danas postoji na hiljade programskih jezika. Svedoci smo kako evolucije programskih jezika opšte namene, tako i pojave mnoštva jezika za rešavanje problema iz specifičnih domena. Razvoj hardvera daje sve veću slobodu u dizajnu programskih jezika. Večiti kompromis između efikasnosti izvršavanja programa i kvaliteta procesa programiranja se pomera u pravcu sve sofisticaranijih programskih jezika. To se manifestuje u toleranciji sve većeg stepena apstrakcije u zadavanju procesa računanja.

Kao krajnji cilj razvoja viših programskih jezika može se postaviti komunikacija sa računarom na prirodnom jeziku. Jedna od razlika između prirodnog i programskog jezika je što je programski jezik u osnovi formalan matematički jezik, dok prirodni jezik sadrži različite izvore nepreciznosti, a komunikacija pomoću njega se intenzivno oslanja na upotrebu konteksta. Današnja praktična istraživanja u oblasti prirodnog jezika stoga su usmerena na akviziciju podataka iz standardizovanih dokumenata, a dublje razumevanje teksta se čini još vrlo dalekim. Zbog svega toga, razumno je za cilj kome programski jezici treba da teže postaviti formalnu specifikaciju problema. Većina dovoljno izražajnih formalnih specifikacija

baziranih na matematičkim formalizmima je neodlučiva, pa se ne mogu neposredno koristiti kao programski jezici. Zato se teži konstrukciji izvršnih specifikacija, koje se mogu prevesti u dovoljno efikasne programe sa zadovoljavajućom efikasnošću prevođenja, dozvoljavaju formalnu manipulaciju i dokazivanje matematičke korektnosti, a sam proces pisanja specifikacije je prirodan sa stanovišta programera.

1.2 Funkcionalno programiranje

Problematika programskih jezika se u ovom radu posmatra kroz prizmu funkcionalnog programiranja. Ovaj pristup tretira programe kao funkcije koje se zadaju korišćenjem kompozicije funkcija i rekurzije. Ovi mehanizmi su podržani i u savremenim proceduralnim jezicima, ali ono što karakteriše funkcionalni stil programiranja je da oni predstavljaju osnovni mehanizam za zadavanje izračunljivih funkcija. Zbog toga se u funkcionalnim jezicima funkcije mogu koristiti mnogo fleskibilnije, a implementacije ovih jezika posvećuju veliku pažnju optimizaciji rada sa funkcijama da bi programi napisani u ovom stilu bili što efikasniji.

Funkcionalni jezici spadaju u deklarativne programske jezike. Za razliku od imperativnih jezika, deklarativni programski jezici apstrahuju ne samo od konkretnog skupa mašinskih instrukcija računara, veći i od samog modela računara kao mašine sa stanjima i memorijskim lokacijama koje se menjaju izvršavanjem naredbi. U funkcionalnom stilu programiranja program predstavlja matematički izraz, a izvršavanje programa se realizuje kao računanje vrednosti tog izraza.

Značajno je uporediti osnovni koncept funkcionalnog programiranja sa pristupom logičkog programiranja. U logičkom programiranju, program se zadaje kao niz predikatskih formula prvog reda. Ovaj pristup je još bliži ideji o programiranju kao formalnoj specifikaciji problema. Osnovni problem ovog pristupa je, čini se, upravo to što je apstrahovan jedan korak previše, apstrahovana je razlika između izračunljivog i neizračunljivog. Naime, problem provere da li dati skup formula predikatskog računa prvog reda ima model je neodlučiv, pa čak i ako je specifikacija korektna, nije jasno da li će program dati rezultat u konačnom vremenu. Pored toga, primena kompletnih procedura za nalaženje modela skupa formula rezultuje u eksponencijalnom vremenu izvršavanja čak i jednostavnih formula. To je verovatno bio i glavni razlog što je logički programski jezik Prolog usvojio nekompletnu strategiju za izvršavanje programa i dozvolio upotrebu predikata odsecanja. Ove dve osobine Prologa se čine nužnim za praktičnu primenu, ali upravo one grubo narušavaju deklarativnu prirodu programa. Za razumevanja programa koji koriste ove mogućnosti nije više dovoljno poznavanje predikatskog računa, već je neophodno poznavanje mehanizma izvođenja formula, koji je još složeniji od imperativnog modela sa naredbama i lokacijama. Time programi u Prologu gube svojstva specifikacija.

Nasuprot tome, u funkcionalnom programiranju se izračunljivost posmatra kao suštinsko svojstvo semantike programa. Zapisi programa se mogu posmatrati kao zapisi rekurzivnih funkcija nad struktuiranim domenima. Ako je rekurzija dobro zasnovana, i program će dati rezultat. Apstrakcija mehanizma izvršavanje i ovde otežava rezonovanje o efikasnosti programa, ali je moguće dobiti približnu sliku o efikasnosti funkcionalnih programa ako se za jedan korak izrvršavanja programa uzme izračunavanje vrednosti atomične funkcije ili zamena imena korisničke funkcije njenom definicijom.

Istorija funkcionalnih programskih jezika započinje sa jezikom LISP (LISt Processing), kojeg je definisao McCarthy krajem 50-ih godina. Jednostavnost LISP-a je doprinela njegovoj popularnosti koja i danas traje, iako bi se ovom jeziku sa današnjeg stanovišta mogle uputiti mnoge zamerke. LISP je netipiziran jezik pojednostavljene sintakse bazirane na s-izrazima. Pošto je LISP često bio implementiran kao interpreter, mnoge njegove verzije su zasnovane na dinamičkom vezivanju promenljivih i po pravilu imaju striktnu semantiku. Zbog ovoga se ponekad LISP i ne smatra funkcionalnim jezikom u savremenom smislu te reči. Savremeni pristup funkcionalnom programiranju započinje sa Landinom koji je 1966. godine definisao jezik ISWIM (If you See What I Mean), koji je bio baziran na lambda računu, te je podržavao statičko vezivanje i nestriktnu semantiku. Pored toga, on je doneo i savremeni stil sintakse funkcionalnih programa sa infiksnim matematičkim operatorima i korišćenjem uvlačenja umesto separatora za pregledniji zapis programa. Sledeći značajan korak je bio jezik ML (Meta Language). Nastao je inicijalno kao jezik za zadavanje taktike u dokazivaču teorema LCF (Logic for Computable Functions) razvijen u Edinburgu. Njegov najznačajniji doprinos je uvođenje sistema tipova u funkcionalni programski jezik. Zahvaljujući Milner-Hindley algoritmu, kompajler sam otkriva tipove funkcija i ukazuje na greške u primeni funkcija na argumente. Od tada sofisticirani sistemi tipova postaju značajna komponenta savremenih funkcionalnih jezika i jedan od aspekata po kome će funkcionalni jezici biti znatno iznad ostalih. ML je imao striktnu semantiku koja omogućava efikasnu implementaciju, ali su se kasnije pojavile i verzije sa lenjom semantikom (Lazy ML). Bočne efekte ML dozvoljava, ali ne ohrabruje stil programiranja koji ih intenzivno koristi. ML je danas dosta u upotrebi, a razvijaju se i nova proširenja jezika.

Konačno, najmoderniji pravac u oblasti razvoja funkcionalnih jezika je započeo serijom jezika D. Turner-a. Funkcije definisane u ovim programima liče na jednačine ([116]), što pogoduje dokazivanju njihovih svojstava indukcijom po strukturi programa ([13]). Ovi jezici su nestriktne semantike, podržavaju algebarske tipove, pattern matching i ZF izraze. 1990. godine definisan je programski jezik Haskell sa ciljem da se uspostavi standard u oblasti nestriktnih, čisto funkcionalnih programskih jezika koji bi povećao mogućnosti funkcionalnih jezika da izađu iz laboratorija i dožive širu upotrebu. Koliko se u tome uspelo, i posle 10 godina ostaje otvoreno pitanje. Stopama Haskell-a je krenuo i čisto funkcionalni jezik Clean. On koristi sistem tipova sa (konstruktorskih) klasa kao i u Haskell-u, ali dodaje u sistem tipova elemente linearne logike što omogućava efikasniju implementaciju i interakciju sa spoljašnjim okruženjem. Takođe dozvoljava i egzistencijalne tipove. U toku je razvoj nove verzije jezika Clean koja bi podržala i multiparametarske klase i druge eksperimentalne osobine prisutne u nekim implementacijama Haskell-a.

Funkcionalni programski jezici se i dalje znatno manje koriste u praksi nego imperativni. Ovo se delom može pripisati tome što na popularnost nekog jezika utiče čitav niz okolnosti koje nisu direktno vezani za kvalitet programskog jezika, već su istorijskog ili ekonomskog karaktera. Treba, međutim, priznati da funkcionalni jezici još nisu dostigli performanse koje pružaju imperativni jezici. Osim toga, sofisticirani sistemi tipova funkcionalnih programskih jezika su teško razumljivi prosečnom programeru, iako je njihov potencijal u poboljšanju softvera veliki. Dobra matematička zasnovanost, po svemu sudeći, nije odlučujući faktor za efikasnost razvoja softvera. Formalna verifikacija softvera je dugotrajan posao koji je isplativ samo za sisteme od kojih se očekuje visok stepen pouzdanosti, a takvi sistemi se

često izvršavaju u realnom vremenu, što predstavlja dodatne, iako ne i nerešive, probleme za implementaciju funkcionalnih programskih jezika. Ovakva situacija dovodi do problema sa finansiranjem projekata razvoja funkcionalnih programskih jezika. Mala upotreba ovih jezika u praksi otežava i nalaženje kriterijuma za ocenu praktičnog značaja pojedinih teorijski privlačnih mogućnosti funkcionalnih programskih jezika. Ne može se, međutim, poreći da komercijalno primenljivi i popularni programski jezici povremeno usvajaju ideje primenjene u funkcionalnim jezicima. Zato se oblast funkcionalnih programskih jezika može smatrati za izvor teorijski dobro zasnovanih ideja koje će vremenom naći svoju primenu ukoliko suštinski doprinose procesu razvoja softvera.

1.3 Cilj rada

Cilj rada je ispitivanje mogućnosti za realizaciju modularnih interpretera u savremenim funkcionalnim programskim jezicima za čijeg predstavnika je izabran Haskell. Ovaj cilj je pretpostavljao kako rešavanje problema vezanih za realizaciju modularnosti u pisanju interpretera, tako i ispitivanje mogućnosti funkcionalnog jezika Haskell i njegovih proširenja za kreiranje prototipova složenih programa. Ove aktivnosti treba da predstavljaju pripremu za dalje eksperimentisanje u pravcu sistematskog razvoja interpretera i kompajlera za različite programske jezike, pri čemu se umesto usko specijalizovanih formalizama kao što su atributivne gramatike, kao okruženje koristi programski jezik Haskell.

Praktični deo rada je ostvaren pisanjem programskih modula u programskom jeziku Haskell. Realizovani su moduli za specifikaciju semantike, apstraktne sintakse, konkretne sintakse i leksičke analize. Korišćenjem ovih modula napisane su komponente za jednostavan interpreter.

1.4 Doprinosi

Dok su raniji radovi ([85], [75], [76], [38], [26]) pokazali kako je moguće modularno zadati denotacionu semantiku programskih jezika, i kako je primenom generalizovanih fold operacija moguće modularnost primeniti i na apstraktnu sintaksu, u ovom radu se daje jedan pristup za modularno zadavanje konkretne sintakse i leksike jezika. Za razliku od ranijih pristupa, sintaksa se zadaje na način koji omogućava efikasnu implementaciju. Leksička analiza se takođe zadaje modularno, a realizuje se preko konačnih automata. Time je ostvarena modularnost svih komponenti interpretera. Sklapanjem različitih komponenti mogu se dobiti interpreteri za različite jezike. Sama specifikacija jezika je jednostavna zahvaljujući realizaciji biblioteka za specifikaciju leksike, sintakse i semantike jezika.

Ovaj rad je istovremeno predstavljao eksperiment u oblasti programiranja u savremenom funkcionalnom programskom jeziku kao što je Haskell i ispitivanju multiparametarskih klasa. Problem sklapanja interpretera od komponenti je rešen isključivo programiranjem unutar jezika, bez generisanja teksta izvornog programa kao što je uobičajeno za većinu alata za generisanje kompajlera. Dalje je pokazano je kako bi primena memo funkcija omogućila elegantniju implementaciju konačnog automata sa lenjom konstrukcijom stanja i prelaza. Razvijen je i algoritam za parsiranje izraza zadatih specifikacijom n-arnih infiksnih, prefiksnih i postfiksnih operacija različitih prioriteta.

1.5 Pregled rada

U drugom delu data su razmatranja vezana za semantiku programskih jezika i funkcionalno programiranje u Haskell-u. To uključuje teoriju katagorija, lambda račun, kratak pregled semantike programskih jezika sa naglaskom na denotacionu semantiku, lenjo izračunavanje, memoizaciju, osnovne elemente jezika Haskell, monade, i parsiranje u funkcionalnim jezicima.

U trećem delu opisana je osnovna problematika modularne denotacione semantike i opisani su prethodni pristupi ovom problemu. Zatim je prikazana struktura modularnog interpretera realizovanog u Haskell-u i opisana realizacija apstraktnih tipova pomoću kojih se interpreter specificira. Opisan je način zadavanja semantike pomoću monad transformera, zadavanje apstraktne sintakse zasnovane na pojmu algebre u teoriji kategorija, specifikacija sintakse korišćenjem operatora različitih prioriteta i specifikacija leksike sa implementacijom leksičkog analizatora zasnovanog na lenjoj konstrukciji prelaza i stanja automata. Na kraju je dat primer specifikacije jednostavnog jezika korišćenjem razvijene strukture za pisanje modularnih interpretera, kao i primer programa u tako specificiranom jeziku.

U četvrtom delu se razmatraju prednosti i nedostaci primenjenog pristupa i mogućnosti za dalji rad u tom pravcu. Razmatra se upotreba generatora koda kao alternativnog načina implementacije modularnih interpretera i objašnjava veza sa atributivnim gramatikama.

Rad je pisan sa namerom da se čita sekvencijalno. Čitaocu koji želi da brzo stekne osnovnu predstavu o suštini rada predlažemo čitanje osnova Haskell-a (2.6), klasa u Haskell-u (2.7), denotacione semantike (2.3.4), monada u funkcionalnom programiranju (2.8.2), i konačno osnovne ideje modularnih interpretera (3.1).

1.6 Izrazi zahvalnosti

Želeo bih da se zahvalim članovima komisije za odbranu diplomskog rada, pre svega mentoru prof. Dr. Mirjani Ivanović na pruženoj slobodi u izboru teme rada kao i ukazanom strpljenju i pomoći u njegovoj realizaciji. Vrlo sam zahvalan prof. Dr. Zoranu Budimcu koji je pobudio moje interesovanje za savremene funkcionalne programske jezike na predavanju u Istraživačkoj Stanici Petnica, a kasnije mi omogućio da dođem do izvorne literature iz ove oblasti. Takođe se zahvaljujem na korisnim sugestijama u vezi teksta ovog diplomskog rada. Zahvaljujem se prof. Silviji Ghilezan na posvećenom vremenu i literaturi, što je znatno doprinelo proširivanju mog znanja vezanog za lambda račun i semantiku programskih jezika. Zahvalan sam Dr. Draganu Mašuloviću koji je bio mentor mojih radova u Istraživačkoj Stanici Petnica i uticao na moju odluku da se opredelim za studije informatike na Univerzitetu u Novom Sadu. Takođe mu se zahvaljujem na pomoći u razumevanju veze između monada u teoriji kategorija i monada u funkcionalnom programiranju. Juhász Lajos-u, mom kolegi i cimeru, se zahvaljujem na obećanju da neće postavljati nezgodna pitanja na odbrani rada. Profesorima Univerziteta u Novom Sadu, a pre svega Instituta za matematiku, dugujem zahvalnost na razumevanju za moja raznolika interesovanja i nesebičnoj pomoći u nalaženju literature. Profesoru Siniši Crvenkoviću se, između ostalog, zahvaljujem što me je uputio u neke probleme savremenog teorijskog računarstva i ukazao na značaj term-rewriting sistema. Zahvaljujem se i profesoru Gradimiru Vojvodiću koji je pomogao mojoj afirmaciji već od prve godine fakulteta. Istraživačkoj Stanici Petnica sam zahvalan na bogatom iskustvu

koje mi je pružila u toku srednje škole. Uz dodatne napore profesora Gimnazije “Veljko Petrović” u Somboru, to je imalo veliku ulogu u razvoju mog interesovanja za nauku.

Iznad svega sam zahvalan roditeljima koji su me uvek podržavali u mojoj radoznalosti i težnji da razumem svet koji me okružuje, čak i u vremenu u kojem su dešavanja oko nas svet nauke činili pomalo dalekim.

2 Uvod

Ovaj deo sadrži odeljke koji obrađuju različite oblasti od značaja za semantiku programskih jezika i funkcionalno programiranje.

2.1 Teorija kategorija

Ovaj odeljak sadrži kratak i neformalan pregled osnovnih pojmova teorije kategorija, sa naglaskom na njenu primenu u semantici programskih jezika. Data je skica interpretacije tipiziranog lambda računa pomoću teorije kategorija koja daje opravdanje za primenu teorije kategorija u funkcionalnim programskim jezicima. Ovaj odeljak ujedno predstavlja i pripremu za uvođenje monada u 2.8.

Teorija kategorija uopštava mnoge pojmove i teoreme iz univerzalne algebre i algebarske topologije. Nastala je uglavnom apstrakcijom od algebarske topologije. Osnovni pojmovi u teoriji kategorija su objekti i morfizmi (strelice, eng. arrows). Primeri kategorija su familija skupova sa totalnim preslikavanjima, familija skupova sa parcijalnim preslikavanjima, familija konačnih skupova sa preslikavanjima konačnih skupova, monoidi sa homomorfizmima monoida, grupe sa homomorfizmima grupa, familija algebarskih struktura fiksirane signature sa homomorfizmima, vektorski prostori i linearne transformacije, parcijalni poreci sa monotonim preslikavanjima, metrički prostori sa kontrakcijama, topološki prostori sa neprekidnim preslikavanjima, i druge strukture. U nastavku dajemo definicije osnovnih pojmova teorije kategorija, posebno Kartezijanski zatvorenih kategorija (CCC, eng. Cartesian Closed Categories).

Kategorija **C** se sastoji od

1. kolekcije *objekata*;
2. kolekcije *morfizama* (strelica);
3. operacija koje svakom morfizmu f pridružuju objekat $\mathrm{dom}\, f$ (domen morfizma) i kodomen $\mathrm{cod}\, f$ (kodomen morfizma). Ako je $\mathrm{dom}\, f = A$ i $\mathrm{cod}\, f = B$, pišemo $f : A \to B$, a skup svih morfizama sa domenom A i kodomenom B označavamo sa $C(A, B)$;
4. operatora kompozicije koji svakom paru morfizama $f : A \to B$ i $g : B \to C$ dodeljuje morfizam $g \circ f : A \to C$, tako da važi zakon asocijativnosti tj. za $f : A \to B$, $g : B \to C$ i $h : C \to D$ važi
$$h \circ (g \circ f) = (h \circ g) \circ f;$$
5. identičkog morfizma id_A za svaki objekat A, tako da da za $f : A \to B$ važi $f \circ \mathrm{id}_A = f$ i $id_B \circ f = f$.

Umesto $g \circ f$ ponekad pišemo $f; g$. U prethodnoj definiciji smo koristili termin *kolekcija* umesto termina *skup* jer se svi skupovi mogu posmatrati kao objekti jedne kategorije. U tom slučaju bi kolekcija skupova predstavljala pravu klasu ([83]). Opštost prethodne definicije je ono što omogućava tako široku primenu teorije kategorija. Osim navedenih primera, kao

kategorije se mogu posmatrati i tipovi u nekom programskom jeziku, a morfizmi i objekti se mogu interpretirati i kao iskazi i implikacije u intuicionističkoj logici.

Dijagrami. Teorija kategorija se prepoznaje po intenzivnom korišćenju dijagrama. Prednost dijagrama je što omogućavaju lako uočavanje niza jednakosti koje važe među morfizmima, sakrivajući implicitnu upotrebu osobina jednakosti i osobina kompozicije morfizama. Dijagram je kolekcija čvorova i usmerenih linija koje predstavljaju objekte i morfizme kategorije. Svaki put između dva objekta koji je sastavljen od usmerenih linija odgovara kompoziciji odgovarajućih morfizama. Kažemo da dijagram *komutira*, ako za svaka dva objekta X i Y na dijagramu, kompozicija morfizama na svakom putu koji vodi od X do Y predstavlja jedan isti morfizam iz X u Y.

Sa formalne strane se teorija kategorija može posmatrati kao jednakosna teorija ([47]). Ovakav pogled je koristan za formalizaciju pojmova teorije kategorija i može se upotrebiti kao paradigma za implementaciju kompajlera za funkcionalne jezike baziranih na okruženjima ([47], [20], [107]).

Vrste morfizama i dualnost. Morfizam f je *monomorfizam* ako $f \circ g = f \circ h$ implicira $g = h$ za sve g i h za koje su odgovarajuće kompozicije definisane. Analogno, morfizam f je *epimorfizam*, ako iz $g \circ f = h \circ f$ sledi $g = h$. Dualnost između epimorfizama i monomorfizama je posledica opšte dualnosti u teoriji kategorija koja je vidljiva i iz same definicije pojma kategorije. Tako za svaku kategoriju $\mathbf{C}$ postoji dualna kategorija $\mathbf{C}^{op}$ u kojoj su zamenjene operacije dom i cod, a kompozicija definisana na odgovarajući način. *Izomorfizam* u kategoriji je morfizam $f : A \to B$ za koji postoji morfizam $f^{-1} : B \to A$ tako da $f \circ f^{-1} = \mathrm{id}_B$ i $f^{-1} \circ f = \mathrm{id}_A$. U opštem slučaju u kategoriji može da postoji morfizam koji je monomorfizam i epimorfizam, a da nije izomorfizam.

Inicijalni i terminalni objekat. Objekat O je *inicijalni objekat* u kategoriji $\mathbf{C}$ ako za svaki objekat A iz $\mathbf{C}$ postoji tačno jedan morfizam $f_A : O \to A$. Dualni pojam inicijalnom objektu je *terminalni objekat.* Terminalni objekat se označava sa 1. U kategoriji skupova sa totalnim preslikavanjima prazan skup je inicijalan objekat, a jednoelementni skupovi su terminalni objekti. Morfizmi iz terminalnih objekata se nazivaju *globalni elementi* i mogu se koristiti kao kategoričko uopštenje elemenata skupa. Inicijalni objekti su takođe od velikog značaja. Tako je, na primer, u kategoriji algebri fiksirane signature inicijalni objekat slobodna algebra generisana konstantama i operacijama. Ovo zapažanje je polazna tačka u algebarskom pristupu semantici programskih jezika ([32]).

Limit u kategoriji. Neka je dat dijagram D u kategoriji $\mathbf{C}$. *Konus* (eng. cone) dijagrama D je familija morfizama $f_i : X \to D_i$ za svaki objekat D_i iz D, takvih da za svaki morfizam $g : D_i \to D_j$ iz D važi $g \circ f_i = f_j$. Konus označavamo sa $\{f_i : X \to D_i\}$. Elemenat X zvaćemo *vrh konusa*.

Limit u kategoriji $\mathbf{C}$ je takav konus $\{f_i : X \to D_i\}$ da za svaki konus $\{f'_i : X' \to D_i\}$ postoji *jedinstveni* morfizam $k : X' \to X$ takav da $f'_i = f_i \circ k$.

Drugim rečima, limit je terminalni objekat u kategoriji koju formiraju konusi elemenata X dok X prolazi objektima kategorije $\mathbf{C}$. Direktno iz uslova jedinstvenosti morfizma $k : X' \to X$ sledi da je vrh limita jedinstven do na izomorfizam (ako postoje dva limita onda su njihovi vrhovi izomorfni). Dualna konstrukcija limitu je *kolimit*. Limit je vrlo opšta konstrukcija. Važan primer limita je proizvod.

Proizvod objekata A i B u kategoriji $\mathbf{C}$ se javlja kao vrh konusa dijagrama D gde D sadrži samo objekte A i B i ne sadrži strelice.

Drugim rečima, objekat $A \times B$ je proizvod objekata A i B, ako postoje morfizmi $\pi_1 : A \times B \to A$ i $\pi_2 : A \times B \to B$ takve da za svaki objekat Y kategorije $\mathbf{C}$ za koji postoje morfizmi $f : Y \to A$ i $g : Y \to B$ postoji jedinstven morfizam (u oznaci $\langle f, g \rangle$), takav da je $f = \pi_1 \circ \langle f, g \rangle$ i $g = \pi_2 \circ \langle f, g \rangle$.

Ako za svaka dva objekta A i B kategorije $\mathbf{C}$ postoji neki proizvod $A \times B$, kažemo da $\mathbf{C}$ *ima proizvode*. Analognu terminologiju koristimo i za limite ostalih dijagrama. U opštem slučaju može postojati više izomorfnih objekata koji predstavljaju proizvod $A \times B$. U takvim situacijama je često pogodno odabrati jedan od tih objekata. Tada $\times$ postaje binarni operator nad objektima, i kažemo da kategorija $\mathbf{C}$ ima *označene proizvode* (eng. distinguished products).

Ako su $A \times C$ i $B \times D$ dva proizvoda, a $f : A \to B$ i $g : C \to D$ morfizmi, tada je proizvod morfizama f i g, u oznaci $f \times g : A \times C \to B \times D$, dat sa $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$.

Dualno se definiše i pojam *sume* dva objekta. Definicija proizvoda i sume se prirodno može generalizovati na proizvoljnu familiju objekata kategorije. Pojmovi proizvoda sume su od velikog značaja za teoriju programskih jezika jer se njima mogu modelirati tipovi podataka. U kategoriji u kojoj su objekti skupovi, a morfizmi totalna preslikavanja, proizvod odgovara Dekartovom proizvodu skupova, a suma odgovara disjunktnoj uniji. U denotacionoj semantici se takođe koriste konstrukcije sume i unije domena, što je indikacija da je poželjno ovakve pojmove razmatrati u opštijem kontekstu. Dodatna prednost ove apstrakcije je što se lakše uočavaju dualne pojave. Tako je primena operacija tipa `fold` u funkcionalnom programiranju poznata duže vreme, dok je značaj dualne operacije `unfold` prepoznat kasnije ([31], [52]).

Zbog visokog nivoa apstrakcije teorija kategorija se koristi u različitim oblastima teorijskog računarstva. Ovakav pristup rezultuje u smanjenju zavisnosti od formalnog jezika koji se koristi za opis teorija, kao što je to slučaj i u katagoričkoj logici. Tako su teorijske osnove pojedinih sistema za verifikaciju programa zasnovane na primeni kategoričke logike ([11], [93], [117]). Ovde je od interese pre svega primena teorije kategorija u semantici programskih jezika, što je moguće raditi na više načina. Dok Goguen zastupa tezu da je dovoljno koristiti teorije prvog reda, teorija kategorija se može koristiti i za interpretaciju teorija višeg reda kao što je lambda računa ([9]). Za ovu konstrukciju je od ključnog značaja postojanje *stepena* u kategoriji.

Stepeni. Neka kategorija $\mathbf{C}$ ima binarne proizvode. Neka su A i B proizvoljna dva objekta. Tada je stepen objekata A i B objekat B^A, (koristi se i oznaka $[A \to B]$ ili $(A \Rightarrow B)$), ako postoji morfizam $\mathrm{eval}_{AB} : B^A \times A \to B$ takav da za svaki objekat X za koji postoji morfizam

$g : X \times A \to B$ postoji jedinstveni morfizam curry(g) takav da

$$g = \text{eval}_{AB} \circ (\text{curry}(g) \times \text{id}_A).$$

Stepen predstavlja terminalni objekat u kategoriji koju formiraju dijagrami $f_X : X \times A \to B$ dok X prolazi objektima kategorije $\mathbf{C}$. Morfizmi ove kategorije su morfizmi $g : X_1 \to X_2$ za koje $f_{X_2} = f_{X_1} \circ (\text{id}_A \times g)$. Postupak kojim se određeni objekat karakteriše kao terminalni objekat u kategoriji određenoj dijagramima datog oblika naziva se *univerzalna konstrukcija*. Limiti i stepeni predstavljaju specijalne slučajeve univerzalne konstrukcije. U opštem slučaju univerzalna konstrukcija u kategoriji $\mathbf{C}$ se zadaje tako što se unutar kategorije $\mathbf{C}$ konstruišu objekti i morfizmi nove kategorije $\mathbf{K}$. Željeni pojam se zatim definiše kao terminalni objekat u kategoriji $\mathbf{K}$. Dualno, postoji *ko-univerzalna konstrukcija* kada se u kategoriji $\mathbf{K}$ posmatra inicijalni objekat.

Ako u kategoriji $\mathbf{C}$ za svaka dva objekta A i B postoji stepen A^B, kažemo da kategorija *ima stepene*.

Kartezijanski zatvorena kategorija, u oznaci CCC (Cartesian Closed Category), je kategorija koja ima terminalne objekte, binarne proizvode i stepene.

CCC je dovoljna za interpretaciju tipiziranog lambda računa. Jednostavnosti radi, ovde dajemo samo suštinu prevođenja Church-ove verzije lambda računa sa prostim tipovima ([10]) u morfizme CCC kategorije sa označenim terminalnim objektom, proizvodima i stepenima. To nam dozvoljava da 1 tretiramo kao objekat u kategoriji, a $\times$ i $\Rightarrow$ kao binarne operacije nad objektima. Interpretacija je preuzeta iz [47] u nešto modifikovanom obliku koji uključuje i prevođenje u deBruijn-ovu formu. Uvodimo operaciju $[M]_{E,v}$ koja daje prevod ispravnog lambda terma M u okruženju E, v. Pri tom je E proizvod objekata CCC kategorije, a v je lista imena promenljivih iz terma. Sa $v; x$ označavamo dodavanje promenljive x na listu v. Prevod zatvorenog lambda terma M se dobija kao $[M]_{1,[]}$ gde je sa [] označena prazna lista. Operaciju $[\_]\_\_$ definišemo na sledeći način:

$$[x]_{E,v} = \texttt{lookup}\, x\, E\, v$$

$$[\lambda x{:}A.M]_{E,v} = \text{curry}([M]_{E\times A, v;x})$$

$$[MN]_{E,v} = \text{eval} \circ \langle [M]_{E,v}, [N]_{E,v} \rangle$$

Pri tome je pomoćna funkcija $\texttt{lookup}\, x\, E\, v$ definisana rekurzijom po strukturi liste promenljivih v, na sledeći način.

$\texttt{lookup}\, x\, (E \times A)\, (v; x) = \pi_2$, gde je $\pi_2 : E \times A \to A$;

$\texttt{lookup}\, x\, (E \times A)\, (v; y) = (\texttt{lookup}\, x\, E\, v) \circ \pi_1$, ako je $x \neq y$, gde je $\pi_1 : E \times A \to E$.

Navedene definicije skrivaju u sebi prevođenje lambda termova u deBruijn-ovu notaciju, samo se umesto prirodnih brojeva koriste odgovarajuće projekcije. eval i curry su potrebni kako bi se premostila razlika koja u kategoriji postoji između skupa morfizama između dva objekta, i objekta koji reprezentuje taj skup.

Primer Prevođenjem lambda izraza $\lambda f : (A \Rightarrow B).\lambda x : A.fx$ dobijamo morfizam

$$\text{curry}(\text{curry}(\mathit{eval} \circ \langle \pi_2 \circ \pi_1, \pi_2 \rangle)) : 1 \to (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$$

koji predstavlja identički morfizam nad objektom koji reprezentuje morfizme iz skupa A u skup B.

U nastavku dajemo neke fundamentalne pojmove teorije kategorija koji su neophodni za definisanje pojma monade.

Funktor. Neka su $\mathbf{C}$ i $\mathbf{D}$ kategorije. *Funktor* $F : \mathbf{C} \to \mathbf{D}$ je preslikavanje koje svakom objektu A iz $\mathbf{C}$ dodeljuje objekat $F(A)$ iz $\mathbf{D}$ i svakom morfizmu $f : A \to B$ iz $\mathbf{C}$ morfizam $F(f) : F(A) \to F(B)$ tako da važi

- $F(\text{id}_a) = \text{id}_{F(a)}$;
- $F(g \circ f) = F(g) \circ F(f)$.

Endofunktor je funktor $f : \mathbf{C} \to \mathbf{D}$ gde je $\mathbf{C} = \mathbf{D}$. Funktori predstavljaju preslikavanja između kategorija. Ako se kategorije posmatraju kao objekti, tada funktori predstavljaju morfizme između kategorija. Tako dobijamo kategoriju kategorija, u oznaci **Cat**. Kompozicija funktora F i G je data prirodno kao $(F \circ G)(A) = F(G(A))$ i $(F \circ G)(f) = F(G(f))$. Identički funktor identički slika i objekte i morfizme.

Za nas značajan primer funktora su konstruktori tipova u programskim jezicima.

Primer Neka je $F(x) = [x]$ konstruktor listi u kategoriji čiji su objekti tipovi, a morfizmi funkcije u funkcionalnom programskom jeziku. Za svaki tip a postoji tip liste elemenata iz a, u oznaci $[a]$, koji predstavlja sliku objekta pod funktorom. Za svaku funkciju $f : a \to b$ postoji funkcija $\texttt{map}f : [a] \to [b]$ koja predstavlja sliku morfizma pod ovim funktorom, i data je sa

$$\texttt{map}f\ [a_1, a_2, \ldots, a_n] = [f(a_1), f(a_2), \ldots, f(a_n)]$$

Da je konstruktor listi funktor, sledi iz poznatih zakona koji važe za `map`:

```
map id_a = id_[a]

map (f ∘ g) = map f ∘ map g
```

Prirodna transformacija. Neka su $\mathbf{C}$ i $\mathbf{D}$ kategorije, a F i G funktori $\mathbf{C} \to \mathbf{D}$. Prirodna transformacija η iz F u G, u oznaci $\eta : F \dot{\to} G$, je preslikavanje koje svakom objektu A iz $\mathbf{C}$ pridružuje morfizam $\eta_A : F(A) \to G(A)$ iz $\mathbf{D}$, tako da za svaki morfizam $f : A \to B$ iz $\mathbf{C}$ važi

$$G(f) \circ \eta_A = \eta_B \circ F(f). \tag{1}$$

Ako je η_A izomorfizam za svako A iz $\mathbf{C}$, tada za η kažemo da je *prirodni izomorfizam*.

Na značaj prirodnih transformacija ukazuje i činjenica da je teorija kategorija inicijalno uvedena upravo da bi bi se na sistemaski način proučile prirodne transformacije. Prirodne transformacije su posebno značajne za funkcionalne programske jezike. Naime, kako funktori

predstavljaju konstruktore tipova, oni su osnova za parametrizovane tipove. Funkcije između takvih tipova su *polimorfne* funkcije, pa se mogu predstaviti prirodnim transformacijama. Uslov (1) govori o činjenici da polimorfne funkcije ne rade sa vrednostima elemenata, već samo sa načinom na koji su oni međusobno povezani u strukturi podataka.

Primer Posmatrajmo funktor liste [ ] iz prethodnog primera. Tada polimorfnu funkciju rev : `[a] -> [b]` možemo posmatrati kao prirodnu transformaciju funktora [ ] u isti taj funktor [ ]. Ova prirodna transformacija svakom tipu `a` pridružuje funkciju

$$\mathrm{rev}_a \ : \ \texttt{[a]} \to \texttt{[a]}$$

koja obrće redosled elemenata u listi. Zakon (1) tada postaje

$$\texttt{map f} \circ \mathrm{rev}_a = \mathrm{rev}_b \circ \texttt{map f}$$

Kako je uz to $\mathrm{rev}_a \circ \mathrm{rev}_a = \mathrm{id}_a$, sledi da je rev prirodni izomorfizam.

Prirodne transformacije predstavljaju preslikavanja između dva funktora. Kompozicija dve prirodne transformacije σ i τ se definiše sa $(\sigma \circ \tau)_A = \sigma_A \circ \tau_A$. Jedinična prirodna transformacija funktora F u funktor F svakom objektu A pridružuje morfizam $\mathrm{id}_{F(A)}$. Tako za kategorije $\mathbf{C}$ i $\mathbf{D}$ dobijamo kategoriju $\mathbf{D}^{\mathbf{C}}$ čiji su objekti funktori iz $\mathbf{C}$ u $\mathbf{D}$, a morfizmi prirodne transformacije. Kompozicija $\circ$ prirodnih transormacija se naziva i *vertikalna kompozicija*. Postoji i *horizontalna kompozicija* prirodnih transformacija, koja je vezana za kompoziciju funktora na sledeći način. Ako su $F_1, F_2 : \mathbf{C} \to \mathbf{D}$ i $G_1, G_2 : \mathbf{D} \to \mathbf{E}$ funktori, a $\eta : F_1 \dot{\to} F_2$ i $\sigma : G_1 \dot{\to} G_2$ prirodne transformacije, tada je $\eta; \sigma : G_1 \circ F_1 \dot{\to} G_2 \circ F_2$ horizontalna kompozicija prirodnih transformacija definisana sa $(\eta; \sigma)_A = G_2\eta_A \circ \sigma_{F_1 A} = \sigma_{F_2 A} \circ G_1\eta_A$, gde poslednja jednakost važi prema uslovu (1) iz definicije prirodne transformacije.

U kategoriji **Cat** svih kategorija, za svake dve kategorije $\mathbf{C}$ i $\mathbf{D}$ imali smo *skup* funktora kao morfizama u **Cat**. Prirodne transformacije obogaćuju ovu strukturu, pa sada između svake dve kategorije postoji *kategorija* morfizama iz $\mathbf{C}$ u $\mathbf{D}$. Ova situacija se opet može generalizovati, a prema [85] za to postoje i praktična opravdanja. Tako dolazimo do pojma 2-kategorije. 2-kategorija $\mathbf{C}$ je kategorija u kojoj morfizmi između svaka dva objekta A i B čine kategoriju $\mathbf{C}(A, B)$. Morfizmi kategorije $\mathbf{C}$ se nazivaju 1-morfizmi, a morfizmi svake od kategorija $\mathbf{C}(A, B)$ se nazivaju 2-morfizmi. Zahtev za egzistencijom operatora kompozicije 1-morfizama u 2-kategoriji se proširuje na egzistenciju funktora kompozicije, koji preslikava dva 1-morfizma u kompoziciju 1-morfizama, a dva 2-morfizma u novi 2-morfizam. Kompozicija 1-morfizama je generalizacija kompozicije funktora, a kompozicija 2-morfizama je generalizacija horizontalne kompozicije prirodnih transformacija.

2.2 Lambda račun

Ovde su date osnove lambda računa koji se često koristi za opis semantike čisto funkcionalnih programskih jezika, a odlikuje se jednostavnošću i ekspresibilnošću. Za njegov nastanak je zaslužan prvenstveno logičar A. Church koji je imao nameru da stvori sistem koji bi poslužio za zasnivanje matematike. Teorija razvijena u tu svrhu se pokazala kao protivrečna, ali je izdvojen njen deo koji se odnosi na definisanje i izračunavanje funkcija. To je ono što nazivamo lambda račun.

Postoji nekoliko desetina varijanti lambda računa. Unificiran tretman različitih verzija tipiziranog računa opisan je u obliku "lambda kocke" (lambda cube) u ([10]). Pored

tamo opisanih varijanti, postoje i mnoge koje su uvedene specijalno za potrebe teorijskog proučavanja nekih osobina programskih jezika ([1]). Veza između lambda računa i implementacije funkcionalnih programskih jezika data je u [63].

2.2.1 Netipizirani lambda račun

Razmotrićemo ovde ukratko prvo netipiziran lambda račun. Zainteresovanog čitaoca upućujemo na [9].

Lambda račun formalizuje intuitivnu ideju o funkcijama kao pravilima za računanje koja se opisuju izrazima, nasuprot poimanju funkcije u teoriji skupova kao specijalnog slučaja relacije. Pravi se jasna razlika između izraza u kojem učestvuje promenljiva, kao što je $x+3$, i funkcije $f(x) = x + 3$. Ova funkcija se u lamba računu označava sa $\lambda x.x + 3$, pa možemo pisati $f = \lambda x.x+3$. Lambda izrazi se formiraju polazeći od promenljivih, pomoću apstrakcije i aplikacije. Aplikaciju (primenu) jednog izraza na drugi označavamo jednostavno pisanjem jednog izraza uz drugi. Preciznije rečeno, apstraktnu sintaksu skupa izraza netipiziranog lambda računa, u oznaci Λ, definišemo na sledeći način.

$$\Lambda = \mathrm{Var} \mid \lambda x.\Lambda \mid \Lambda\Lambda$$

Var je prebrojiv skup promenljivih. Pri tom koristimo zagrade da bismo izbegli dvosmislenost izraza, a višak zagrada uklanjamo pravilom da lambda apstrakcija vezuje slabije od aplikacije, i da je aplikacija levo asocijativna. Ova jednostavna sintaksa izražava suštinu lambda računa kao računa o funkcijama. U različitim primenama lambda račun se proširuje tako što se pojedine promenljive tretiraju na poseban način, kao konstante ili kao predefinisane funkcije.

Lambda apstrakcija od izraza pravi funkciju po jednoj promenljivoj. Funkcije po dve promenljive se dobijaju uzastopnom primenom lambda apstrakcije, jer se funkcija od dva argumenta može posmatrati kao funkcija koja za fiksiranu vrednost prvog argumenta vraća novu funkciju koja prihvata drugi argument. Ovaj postupak se naziva Curry-ing.

Pojava promenljive x u stablu izraza se naziva *vezanom* ako se na putu do lista koji odgovara toj pojavi u apstraktnom stablu nalazi λx, u suprotnom se naziva *slobodna pojava*. Lambda izraz je *zatvoren* ako ne sadrži slobodne pojave promenljivih. Vezane promenljive su lokalnog karaktera, pa je smisao izraza $\lambda x.x + 3$ i $\lambda y.y + 3$ isti. Ovo pravilo se formalizuje u obliku tzv. α *konverzije*, ali se često ([9]) ono izbegava tako što se radi sa klasama lambda izraza moduo reimenovanje vezanih promenljivih. Dodatno opravdanje za ovakav tretman je i to što postoje reprezentacije lambda izraza, kao što je deBruin-ova notacija, u kojoj je izbegnuto korišćenje imena promenljivih. Imena promenljivih se koriste prvenstveno zbog veće čitljivosti i pretpostavlja se da važi konvencija po kojoj su sve vezane promenljive izraza koji se u datom kontekstu posmatraju izabrane tako da su različite od svih slobodnih promenljivih. Osnovno pravilo lambda računa je β-konverzija, data sa

$$(\lambda x.M)N \rightarrow_\beta M[x := N] \tag{2}$$

Notacija $M[x := N]$ označava zamenu svih pojava promenljive x u M izrazom N. Po pomenutoj konvenciji, sve slobodne promenljive terma N su različite od vezanih promenljivih terma M (time se izbegava tzv. name capture problem). Izraz sa leve strane relacije $\rightarrow_\beta$ u (2) se naziva *redeks*.

Za relaciju $\rightarrow_\beta$ dalje treba pretpostaviti da je kongruentna u odnosu na lambda apstrakciju i aplikaciju, što znači da pravilo (2) možemo primenjivati i na podizraze nekog izraza. Uvodi se oznaka $\twoheadrightarrow_\beta$ kao tranzitivno i refleksivno zatvorenje za $\rightarrow_\beta$. Dalje se uvodi relacija jednakosti kao ekvivalencijsko zatvorenje relacije $\rightarrow_\beta$.

Od teorijskog značaja je i pravilo η, dato sa $\lambda x.Mx \rightarrow_\eta M$. Ono je ekvivalentno sa pravilom ekstenzionalnosti po kojem iz $Mx = Nx$ možemo izvesti $M = N$. To nam omogućava jednostavnije dokazivanje jednakosti izraza u mnogim slučajevima.

Za nas je prvenstveno od značaja pravilo β (beta redukcija), jer se njime u principu može opisati proces izračunavanja izraza u funkcionalnom programskom jeziku. Nadalje ćemo posmatrati samo ovu relaciju. Lambda izraz koji nema redekse se naziva *normalna forma*. Ako izraz ima više redeksa tada postoje različiti putevi da se on redukuje, zavisno od redosleda kojim se redeksi redukuju. U opštem slučaju redukcija može da se završi normalnom formom ili može biti beskonačna. Pri tome je od suštinskog značaja činjenica da je ne može desiti da različitim putevima redukcije dobijemo različite normalne forme. Kažemo da je *normalna forma lambda izraza jedinstvena*. Ovo svojstvo se obično podrazumeva za izraze kojima se služimo u matematici. Zasnovano je na činjenici da svakom izrazu možemo interpretacijom pridružiti neku vrednost, a ta vrednost se ne menja ako podizraz tranformišemo primenom neke od operacija unutar izraza. Kako je za lambda izraze problem interpretacije složeniji, od interesa je i čisto sintaksni pristup ovom problemu. Jedinstvenost normalne forme sledi iz teoreme Church-Rosser-a, koja kaže da ako $M \twoheadrightarrow_\beta M_1$ i $M \twoheadrightarrow_\beta M_2$, onda postoji M_3 tako da $M_1 \twoheadrightarrow_\beta M_3$ i $M_2 \twoheadrightarrow_\beta M_3$. Svojstva kao što je Church-Rosser se intenzivno proučavaju u kontekstu term-rewriting sistema ([21]).

Postoje lambda izrazi koji imaju i konačne i beskonačne nizove primene pravila $\rightarrow_\beta$. Postupak kojim se za dati lambda izraz određuje koji redeks se redukuje naziva se *strategija redukcije*. Sledeći ključni rezultat je da postoji strategija, tzv. spoljna redukcija (eng. outermost reduction), kojom se garantuje redukcija izraza do normalne forme ako normalna forma postoji. Ako izraz nema normalnu formu, svaka njegova redukcija je beskonačna. Kako je pokazano u [9], i za neke od tih izraz ima smisla smatrati da označavaju određenu vrednost. To se postiže uvođenjem glavene normalne forme (eng. head normal form). Spoljna redukcija ima čak i svojstvo da dovodi izraz do glavene normalne forme ako ona postoji. Ova strategija je u vezi sa lenjim izračunavanjem (odeljak 2.4) i predstavlja objašnjenje činjenice da jezici sa lenjom semantikom mogu da izračunaju i neke od onih izraza čije se računanje u jezicima sa striktnom semantikom nikad ne završava. Implementacije lenjih funkcionalnih programskih jezika uglavnom implementiraju računanje do slabe glavene normalne forme (eng. weak head normal form, WHNF), koja se od glavene normalne forme razlikuje samo po tome što ne izračunava telo lambda apstrakcija.

U lambda računu se može definisati kombinator rekurzije $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ sa svojstvom $YF = F(YF)$. Kodiranjem prirodnih brojeva kao izraza lambda računa specijalnog oblika moguće je unutar lambda računa zadavati funkcije nad brojevima. Klasa funkcija koje je moguće na taj način definisati su upravo rekurzivne funkcije. To opravdava upotrebu lambda računa kao osnove programskih jezika. Sa druge strane, odatle sledi i neodlučivost nekih pitanja lambda računa, analognih halting problemu ([10]).

Lambda račun je od početka posmatran kao sintaksni račun koji funkcije tretira kao izraze. Postavlja se pitanje da li postoji model lambda računa u kojem bi se funkcije posmatrale

kao preslikavanja jednog skupa u drugi. Problem koji se pri tom javlja je da su dozvoljeni lambda izrazi oblika xx, što znači da x mora biti funkcija čiji domen A sadrži preslikavanja iz A u A. Pokušaj interpretacije funkcija lambda računa kao proizvoljnih preslikavanja nekog skupa A u skup A se zato mora odbaciti, jer po teoriji kardinalnih brojeva skupova ne može biti $A^A \subseteq A$. Konstrukciju modela koji rešava ove probleme izvršio je Scott. On je interpretirao funkcije lambda računa kao neprekidne funkcije u specijalnoj topologiji. Neprekidnost se tako koristi kao apstraktna karakterizacija izračunljivosti funkcija i omogućava definisanje domena D koji sadrži skup svih neprekidnih funkcija $D \to D$. Jedan opis slične konstrukcije dat je u [106].

2.2.2 Kombinatori

Osnovno pravilo lambda računa, beta redukcija, je intuitivno jasno, ali formalno nije jednostavna operacija jer uključuje zamenu $M[x := N]$. Zamena se može rekurzivno definisati indukcijom po strukturi izraza, što nas navodi na to da i nju posmatramo kao složenu operaciju. Jedan od načina da se eliminiše pravilo zamene je upotreba *kombinatora*. Kombinatori se mogu opisati lambda izrazima bez slobodnih promenljivih, ali suština njihove primene je upravo u tome što se mogu zadati preko svog dejstva na druge izraze, bez korišćenja promenljivih. Osnovni sistem kombinatora čine kombinatori S, K, I, dati sledećim pravilima.

$$Sxyz = xz(yz)$$

$$Kxy = x$$

$$Ix = x$$

Pri tome je $I = SKK$, pa kombinator I nije neophodan. Značaj ovih kombinatora je u tome što za svaki zatvoreni lambda izraz M postoji izraz M^* sastavljen isključivo od kombinatora S, K, I, takav da M i M^* imaju isto dejstvo na sve lambda izraze (tj. ekstenzionalno su jednaki). Izraz M^* se definiše rekurzivno po strukturi izraza M.

- $x^* = x$
- $(MN)^* = M^*N^*$
- $(\lambda x.M)^* = [x]M^*$

Pri tome je $[x]M$ definisano nad lambda izrazima koji ne sadrže lambda apstrakciju na sledeći način.

- $[x]x = I$
- $[x]y = Ky$, za $y \neq x$
- $[x]MN = S([x]M)([x]N)$

Ovaj pristup je značajan za implementaciju funkcionalnih programskih jezika jer se primena kombinatora jednostavnije implementira nego zamena terma promenljivom, posebno ako je u pitanju jezik nestriktne semantike. Pri praktičnoj primeni ovog postupka koristi se širi skup kombinatora kako bi izrazi koji se dobijaju prevođenjem bili što manji, a primena kombinatora što efikasnija. Sam postupak prevođenja u proizvoljan fiksirani skup kombinatora se naziva "bracketing".

Nasuprot pristupu sa fiksiranim skupom kombinatora, mogu se za svaki program koji se prevodi generisati posebni kombinatori. U tom slučaju se promenljive iz lambda izraza eliminišu procesom zvanim lambda lifting ([63], [64]), kojim se slobodne promenljive unutar ugnježdenih lambda apstrakcija pretvaraju u dodatne parametre lambda apstrakcije čime ona postaje kombinator. Ovo je analogno procesu kojim se globalne promenljive procedure pretvaraju u njene dodatne parametre.

2.2.3 Tipizirani lambda račun

U tipiziranom lambda računu promenljivim i izrazima se dodeljuju tipovi. Postoje dve varijante tipiziranog lambda računa: Church-ova verzija i Curry-jeva verzija ([10]). U Curry-jevoj verziji lambda izrazi imaju formu kao u netipiziranom računu, dok u Church-ovoj verziji i sami lambda izrazi sadrže oznake tipova. Ovde ćemo opisati samo Curry-jevu verziju najjednostavnijeg lambda računa sa prostim tipovima.

Tipove označavamo grčkim slovima. Skup tipova Type dat je sa

$$\text{Type} = \text{TypeVar} \mid \text{Type} \to \text{Type}$$

Dakle, tipovi se izgrađuju od promenljivih primenom operatora $\to$. Intuitivno, $\sigma \to \tau$ označava tip funkcija iz σ u τ. Podrazumevamo da je operator $\to$ nad tipovima desno asocijativan, pa umesto $\sigma \to (\tau \to \phi)$ možemo pisati $\sigma \to \tau \to \phi$. Notacija $\Gamma \vdash M : \tau$ označava činjenicu da izraz M ima tip τ pod pretpostavkom da slobodne promenljive u M imaju tipove date sa Γ. Pri tom je $\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n\}$ konačan skup pretpostavki koji nazivamo kontekst. Izvođenje tipova dato je sledećim pravilima. Notacija $\frac{A}{B}$ znači da se iz pretpostavki A može izvesti B.

(aksioma) $\quad \Gamma \vdash x : \tau \qquad$ za $(x : \sigma) \in \Gamma$

(eliminacija $\to$) $\quad \dfrac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$

(uvođenje $\to$) $\quad \dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \to \tau)}$

Važno svojstvo tipiziranog lambda računa je njegova veza sa intuicionističkom logikom. To je tzv. Curry-Howard izomorfizam (propositions as types interpretation) koji omogućava da se u $\Gamma \vdash M : \tau$ tip τ posmatra kao iskaz gde $\to$ označava implikaciju, dok je izraz M konstruktivni dokaz za iskaz τ u intuicionističkoj logici. Beta redukcija u lambda računu odgovara normalizaciji dokaza kojom se dokaz svodi na kanonički oblik.

Za svaki tipiziran račun i fiksiran kontekst Γ mogu se postaviti sledeća pitanja:

1. $M : \tau$? (da li važi $\Gamma \vdash M : \tau$)

2. $M : ?$ (ako je dat term M, da li postoji tip τ tako da $\Gamma \vdash M : \tau$)

3. $? : \tau$ (da li postoji term M tako da važi $M : \tau$)

Jezgro sistema tipova u funkcionalnim programskim jezicima se može opisati lambda računom sa prostim tipovima. Programi tada predstavljaju lambda izraze čije su slobodne promenljive ugrađene funkcije čiji su tipovi poznati. Prvo pitanje tada odgovara proveri tipova koja je zastupljena i u klasičnim programskim jezicima kao što je Modula-2. Odgovor na drugo pitanje zahteva da kompajler sam pronađe i tip koji se može dodeliti programu. Ovo pitanje je odlučivo za lambda račun sa prostim tipovima, zahvaljujući Hindley-Milner-ovom postupku zasnovanom na unifikaciji ([100]), koji se neformalno može opisati na sledeći način. Pretpostavimo da želimo da pronađemo tip lambda izraza M ukoliko postoji. Dodelimo svakom podizrazu M_i izraza M promenljivu θ_i koja označava tip koji može imati M_i. Ako je $M_k = M_i M_j$, onda po definiciji relacije $\vdash$ mora biti $\theta_i = \theta_j \to \theta_k$. Ako je $M_k = \lambda x.M_i$, tada sve pojave pojave promenljive x moraju imati isti tip, pa ako su M_m i M_n dve takve pojave, mora biti $\theta_n = \theta_m$, a osim toga mora važiti $\theta_k = \theta_n \to \theta_i$. Opisanim postupkom dobijamo niz jednačina među tipovima koje možemo rešiti postupkom unifikacije. Tačan opis algoritma koji omogućava efikasnu proveru tipova je dat u [63]. Rezultat postupka unifikacije je najopštiji tip koji dati term može imati. Tako se, na primer, za izraz $\lambda x.x$ dobija da može da ima tip $\theta_1 \to \theta_1$ za bilo koje θ_1. To je inicijalno bila osnova parametarskog polimofizma u funkcionalnim programskim jezicima kao što je ML, ali se danas za proveru tipova često koristi i lambda račun drugog reda, $\Lambda 2$, koji eksplicitno izražava zavisnost termova od tipova.

Odgovor na poslednje od tri postavljena pitanja o odnosu izraza i tipova je od značaja prvenstveno sa stanovišta Curry-Howard izomorfizma: nalaženje izraza koji odgovara datom tipu odgovara nalaženju dokaza za dati iskaz u intuicionističkoj logici. Postoje pokušaji primene ove interpretacije i u programskim jezicima ([6]), u kojima se specifikacijom tipa istovremeno dokazuje korektnost programa i njegovih komponenti. Za ovakvu primenu je potrebno koristi sistem *zavisnih tipova* (eng. dependent types) da bi se mogla dokazati netrivijalna svojstva programa.

Treba imati u vidu da je skup lambda izraza kojima se može pridružiti tip u lambda računu sa prostim tipovima znatno uži od skupa svih lambda izraza. Za ovaj skup lambda izraza važi ne samo svojstvo Churh-Rosser-a, već i svojstvo jake normalizacije, što znači da je svaka beta redukcija konačna ([10]). Izrazima kao što je Y kombinator se ne može pridružiti tip u ovoj varijanti lambda računa. Prilikom provere tipova u funkcionalnim programskim jezicima je stoga potrebno rekurzivne definicije tretirati na poseban način, za šta postoji više mogućnosti ([63]). Ograničenja kao što je "monomorphism restriction" ([65]) navode na pomisao da potpuno odgovarajuće rešenje još nije pronađeno.

Tipiziran lambda račun je danas predmet intenzivnog proučavanja. Njegova prednost sa stanovišta programskih jezika je jasna, jer je prihvaćen stav da tipizirani programski jezici znatno smanjuju broj grešaka u programiranju. Za matematičko zasnivanje funkcionalnih programskih jezika kao što je Haskell poseban značaj imaju složenije varijante tipiziranog lambda računa. Tako se za objašnjenje polimorfizma, posebno uz eksplicitnu upotrebu egzistencijalnih tipova, koristi tipizirani lambda račun drugog reda (izrazi koji zavise od tipova), dok upotreba konstruktora tipova zahteva upotrebu lambda računa kod kojeg tipovi zavi-

se od tipova. Noviji trend je da se varijante lambda računa velikih izražajnih mogućnosti koriste se kao osnova međukoda prilikom kompajliranja funkcionalnih programskih jezika (međujezici "Henk" [62] i "FLINT" [104]). I sa teorijskog stanovišta tipiziran lambda račun ima prednosti, pa se može koristi čak i za dokazivanje svojstava netipiziranog lambda računa ([29]).

2.3 Semantika programskih jezika

U ovom odeljku dajemo kratak pregled semantike programskih jezika sa naglaskom na denotacionu semantiku koja predstavlja osnovu i našeg modularnog interpretera. Cilj većine ovih pristupa je da se matematičkim aparatom opišu konstrukcije programskog jezika da bi se mogla dokazivati svojstva programa, da bi se obezbedio standard za implementaciju jezičkih procesora i da bi se stekao dublji uvid u suštinu procesa računanja. Za pregled semantike programskih jezika videti takođe [84], [76], [86], [127].

Postoji veliki broj načina zadavanja semantike. Ovde ćemo se posebno osvrnuti na *aksiomatsku*, *operacionu*, i *denotacionu* semantiku.

2.3.1 Aksiomatska semantika

Za nastanak i razvoj aksiomatske semantike su zaslužni pre svega Floyd ([28]), Hoar ([43]) i Dijkstra ([22]). Ovaj postupak se zasniva na aksiomatizaciji toka izvršavanja programa u predikatskom računu prvog reda. Koriste se invarijante petlji da bi se matematičkom indukcijom pokazalo da određena svojstva važe. Jedna od osnovnih teškoća u ovom pristupu je pronalaženje odgovarajuće indukcijske hipoteze (invarijante) koja međusobno povezuje trenutne vrednosti promenljivih u toku programa.

Ovakav postupak se široko koristi i automatizovan je u mnogim sistemima za verifikaciju imperativnih programa ([30], [33]).

Treba primetiti da aksiomatska semantika može da se posmatra kao apstrakcija denotacione semantike. Denotaciona semantika predstavlja jedan od mogućih modela za aksiomatsku semantiku i prirodno je koristiti aksiomatizaciju i za oblike računanja koji se razlikuju od sekvencijalnog, imperativnog modela. Aksiomatski pristup se čini posebno pogodan za *parcijalnu specifikaciju* kojom se ne određuje potpuno semantika programa, ali se pokazuje neko njegovo bitno svojstvo.

2.3.2 Operaciona semantika

Operaciona semantika formalizuje način izvršavanja programa. Najčešće se zadaje pomoću različitih apstraktnih mašina.

Operaciona semantika je u početku dosta kritikovana kao neadekvatno sredstvo za specifikaciju semantike jezika. Plotkin ([97]) je 1980. godine uveo *strukturnu operacionu semantiku*. Ona je zasnovana na logičkim pravilima koja ne fiksiraju u potpunosti jedan tok izvršavanja programa, već dopuštaju više dozvoljenih prelaza iz istog stanja.

Postoji veliki broj primera upotrebe operacione semantike. P-mašina se koristi za kompajliranje Pascal programa ([36]), SECD mašina za implementaciju lambda računa, STG mašina ([67], [90]) za implementaciju lenjih funkcionalnih jezika i mnoge druge. STG mašina

predstavlja mali funkcionalni programski jezik sa jasno definisanom denotacionom semantikom, a u isto vreme ima i preciznu operacionu implementaciju koja specificira lenjo izračunavanje. Po tome je bliska *prirodnoj semantici* koja se koristi kao osnovna operaciona specifikacija lenjog izračunavanja.

Prednost specifikacije semantike pomoću apstraktnih mašina je relativno jednostavna implementacija. Apstraktna mašina se obično definiše tako da oba procesa, prevođenje iz jezika u apstraknu mašinu, i prevođenje iz apstraktne mašine u mašinski jezik, mogu da se obave efikasno.

Nedostatak ovog pristupa je otežana matematička manipulacija programima koji su izraženi u operacionalnoj semantici. Dokazivanje svojstava programa se u principu svodi zaključivanje o toku procesa njegovog izvršavanja. Ovakav formalizam je manje podložan primeni algebarskih zakona koji bi omogućili računanje sa programima.

2.3.3 Denotaciona semantika

Denotaciona semantika se može posmatrati kao opisivanje semantike programskih jezika pomoću lambda računa. U ovom delu razmatramo razvoj i neke osobine denotacione semantike, a način njene upotrebe je dat u sledećem odeljku (2.3.4).

U razvoju denotacione semantike ključnu ulogu su imali Strechey, Scott i Stoy ([106]). Početne faze ovog razvoja karakteriše oštra suprotstavljenost operacionom pristupu. Cilj je definisanje matematičke suštine funkcija, bez obzira na način njene reprezentacije. Naglašava se značaj ekstenzionalnosti po kojem funkcije koje mogu biti sintaksno različite predstavljaju istu matematičku funkciju. Scott je konstrukcijom semantičkih domena dao matematički model za takve funkcije. I danas su prisutna mišljenja da denotaciona semantika treba da predstavlja matematički aparat koji ne mora nužno da rezultuje u specifikacijama koje su izvršne.

Nasuprot tome, mnogi autori uočavaju da je proces pisanja denotacione semantike sličan programiranju u funkcionalnom jeziku. Uočava se da je u denotacionoj semantici, kao i u operacionoj semantici, prisutan problem preterane specifikacije (overspecification). Drugim rečima, opisuje se potpuno značenje programa a ne samo njegove osobine koje su bitne u datom kontekstu. Eksplicitno uvođenje struktura za predstavljanje stanja i kontinuacija u denotacionoj semantici pokazuje da je i ona na niskom nivou apstrakcije. Ova osobina denotacione semantike omogućava dobijanje izvršnih specifikacija iz denotacionih opisa.

Pristup koji predlaže Stoy ([106]) i koji je primenjen na nekoliko dokaza korektnosti kompajlera zasnovan je na dokazima kongruencije. Izvorni jezik i ciljni jezik se opisuju denotacionom semantikom da bi se pokazala korektnost prevođenja. Ovakva transformacija se može izvršiti u više koraka, postepenim uvođenjem implementacionih koncepata ([106]). Takođe su brojni pristupi sa transformacijom denotacione semantike u druge formalizme koji se zatim kompajliraju. Jedan savremeni pristup transformaciji denotacione semantike u oblik u kome je moguće izvršiti efikasnu implementaciju dat je u [82]. Neki od sistema za implementaciju programskih jezika zasnovani na denotacionoj semantici su SIS (Mosses), PSP (Paulson), PSG (Bahlke, Snelting), MESS (Lee, Pleban). Za pregled ovih sistema videti [84].

Savremene metode implementacije funkcionalnih programskih jezika idu u prilog razvoju

upotrebljivih okruženja baziranih na denotacionoj semantici. U [84] se opisuje funkcionalni programski jezik Navel koji sadrži i proširenja za opis leksike i sintakse jezika. Rezultujuće specifikacije jezika su često kraće, ali performanse izvršavanja programa u jeziku koji se definiše zaostaju za onim u sistemima PSG i MESS. Za mnoge funkcionalne programske jezike postoje alati za generisanje leksičkih i sintaksnih analizatora, ali oni nisu integrisani u sam jezik već se koriste kao posebna faza u generisanju implementacije jezika.

Osnovni nedostatak direktne primene denotacione semantike je nizak nivo apstrakcije. Odluka da se koriste neprekidne funkcije nad domenima ostavlja mnogo prostora za opis semantike. Ono što mnogi autori zameraju denotacionoj semantici jeste potreba da se gotovo sve semantičke definicije zamene prilikom prelaska sa direktne semantike na kontinuacije. Da bi se ovi problemi prevazišli potrebno je uvesti kako apstrakcije konkretnih domena u denotacionoj semantici, tako i apstraktne operacije za međusobno kombinovanje elemenata koji vrše računanje ([87]). Srećom, funkcije višeg reda omogućavaju da se oba ova problema reše primenom apstraktnih tipova podataka.

Rad na modularnim interpreterima predstavlja jedan pristup uvođenja modularnosti u denotacionu semantiku. On se može shvatiti kao metod za izgradnju apstraktne aksiomatske semantike na temelju denotacione semantike, pri čemu se različiti procesi računanja opisuju na sistematski način, specifikacijom njihovih osobina i interakcija sa ostalim vidovima računanja.

Savremeni trend u opisivanju semantike programskih jezika je upotreba teorije kategorija. Dok je denotaciona semantika apstrahovala pojam izračunljive funkcije pojmom neprekidne funkcije u odgovarajućoj topologiji, pristup pomoću teorije kategorija apstrahuje i samu prirodu funkcija koje predstavljaju programe, posmatrajući ih kao morfizme u CCC. Taj postupak se čini opravdan, jer se struktura topološkog prostora nad kojim su definisane neprekidne funkcije retko koristi u primeni denotacione semantike ([106]). Ono što je bitno je samo egzistencija modela lambda računa, dok se za većinu funkcija sa kojima se radi ispostavlja da su neprekidne. Po svom apstraktnom načinu tretiranja semantičkih domena primena teorije kategorija je slična primeni tipiziranog lambda računa. Ono što je potencijalna prednost teorije kategorija u ispitivanju semantike je veća opštost ovog matematičkog aparata. Ta opštost ponekad rezultuje u manjoj pristupačnosti dobijenih rezultata, pa je jedan od mogućih pristupa ([85]) da se razmatranja iz teorije katagorija koja su relevantna za dati jezik ili klasu jezika aksiomatizuju u vidu odgovarajuće varijante lambda računa i time pojednostavi njihova primena.

1990. godine Moggi je predložio korišćenje pojma *monade* iz teorije kategorija za opis različitih oblika procesa računanja koji se javljaju u denotacionoj semantici. Ovaj pristup je značajan jer ima mogućnost da reši probleme koji su otežavali primenu denotacione semantike, a da istovremeno zadrži sve prednosti dobro razrađene teorijske osnove denotacione semantike. Jednostavni sistemi za implementaciju jezika zasnovani na ovom principu su realizovani u [26], [76], [127] i [72], ali se stiče utisak da rad na izgradnji efikasnijih sistema za generisanje kompajlera tek predstoji.

2.3.4 Neke konstrukcije denotacione semantike

Ovde dajemo kratak uvod u postupke denotacione semantike i ilustrujemo modeliranje stanja i toka kontrole kao bitnih osobina programskih jezika. Posebnu pažnju posvetićemo neformalnom uvođenju kontinuacija. Cilj odeljka je predstavljanje osnovnih ideja o opisu procesa računanja pomoću lambda računa, a preciznije i opširnije o ovoj problematici se može naći na primer u [106], [113] i [34].

Pri opisu programskih jezika denotacionom semantikom zadajemo sintaksne domene i apstraktnu sintaksu, semantičke domene i interpretaciju apstraktne sintakse. Semantičke definicije zadaju se rekurzivno po strukturi apstraktne sintakse.

Sintaksni domeni odgovaraju tokenima jezika. Apstraktna sintaksa predstavlja opis sintaksnog stabla (videti 2.9).

Semantički domeni označavaju vrednosti koje mogu da uzimaju različiti fragmenti programa. Konstruišu se polazeći od nekih osnovnih domena primenom operacija proizvoda domena $(A \times B)$, sume domena $(A + B)$ i stepena domena $[A \to B]$. Ove konstrukcije su slične pojmovima Dekartovog proizvoda, disjunktne unije i skupa B^A svih funkcija iz A u B.

Razlika je u tome što je nad domenima definisana struktura kompletnog parcijalnog poretka (eng. complete partial order, CPO). Pri tom najmanji elemenat obeležavamo sa $\perp$ i on označava nedefinisanu vrednost (proces računanja koji se nikad ne završava tj. divergira). Posmatraju se isključivo neprekidne funkcije nad ovim domenima, pri čemu je neprekidnost apstrakcija pojma izračunljivosti. Ova neprekidnost se može definisati direktno u terminima parcijalnog poretka, a odgovara neprekidnosti u Scott-ovoj topologiji formiranoj nad tom strukturom ([103], [9]). Operacije $\times$, $+$, $\to$ preslikavaju domene u domene, definišući strukturu parcijalnog poretka na složenim domenima. Taj parcijalni poredak je od ključnog značaja za rekurzivne definicije domena, jer omogućava primenu teoreme Tarskog o nepokretnoj tački. Za operacije kojima se pristupa elementima složenih domena se takođe pokazuje da su neprekidne. Dalje se definišu neprekidne operacije koje interpretiraju apstrakciju i aplikaciju u lambda računu, što omogućava da se slobodno radi sa tipiziranim funkcijama višeg reda nad domenima bez udubljivanja u detaljnu strukturu konstrukcije domena.

Ovakav matematički aparat omogućava da se direktno interpretiraju funkcionalni jezici koji su zasnovani na lambda računu: njihovo prevođenje se svodi na eliminaciju sintaksnog šećera (eng. syntax sugar) i zamenu sintaksnih konstrukcija odgovarajućim operacijama nad domenima. Interpretaciju programa `P` zapisanog u apstraknoj sintaksi u denotacionoj semantici označavamo zagradama, $[\![\texttt{P}]\!]$.

Opisivanje imperativnih jezika je nešto složenije, jer je potrebno modelirati bočni efekat i pojam *stanja* programa. Ako trenutno stanje programa uzima vrednosti iz domena S, onda je domen za vrednosti naredbi $C = [S \to S]$, jer naredba specificira vrednost stanja posle izvršenja naredbe u funkciji od stanja pre izvršenja naredbe. Ako je, primera radi, `x` jedina promenljiva koja čini stanje programa, tada važi $[\![\texttt{x:=x+1}]\!] = \lambda s.s + 1$. Znak `;` kojim se razdvajaju naredbe se tumači kao asocijativan operator tipa $[C \to [C \to C]]$, pa je tako $[\![s_1; s_2]\!] = \text{seq } s_1\ s_2$ gde je funkcija seq definisana sa

$$\text{seq } f\ g = \lambda s.g(f\,s)$$

tj. predstavlja kompoziciju funkcija kojima su predstavljene naredbe.

Izrazi koji računaju vrednost iz domena A i pri tome mogu imati bočni efekat se predstavljaju vrednostima iz $[S \to (A \times S)]$. Pri tom je ključno pitanje definicije kompozicije takvih bočnih efekata. Tako je interpretacija operacije sabiranja u slučaju kada operandi mogu menjati trenutno stanje sledeća:

$$[\![E_1\texttt{+}E_2]\!] = \lambda s_0.\langle e_1 + e_2, s_2\rangle,$$

gde je $\langle e_1, s_1\rangle = [\![E_1]\!]s_0$ i $\langle e_2, s_2\rangle = [\![E_2]\!]s_1$. Ova definicija opisuje proces u toku kojeg se izračuna izraz E_1 i izvrše promene stanja s_0 usled bočnih efekata da bi se dobilo stanje s_1. Zatim se u stanju s_1 izvrši računanje izraza E_2, a novi bočni efekti dovode do stanja s_2. Krajnji rezultat je zbir izračunatih vrednosti, a krajnje stanje je s_2. Vidimo da se već ove definicije prilično komplikuju: bez upotrebe skraćenica bismo imali

$$[\![E_1\texttt{+}E_2]\!] = \lambda s_0.\langle \pi_1([\![E_1]\!]s_0) + \pi_1([\![E_2]\!](\pi_2([\![E_1]\!]s_0))), \pi_2([\![E_2]\!](\pi_2([\![E_1]\!]s_0)))\rangle$$

gde su π_1 i π_2 projekcije. Ovakve definicije su osnova monad transformera za stanje opisanog u 3.2.1. Pisanjem ovih definicija u Haskell-u videćemo prednosti notacije funkcionalnih programskih jezika.

Dalje se može definiše značenje uslovne naredbe `if`, a značenje naredbe `while` i sličnih konstrukcija struktuiranog programiranja se definiše upotrebom rekurzije.

Problemi nastaju kada se pokušaju modelirati složenije kontrolne strukture, kao što je `goto`. Zato se uvode *kontinuacije*. I pored toga što je struktuirano programiranje dominantan pristup u današnjim imperativnim jezicima, modeliranje direktnih skokova je od teorijskog značaja jer ilustruje principijelne mogućnosti denotacione semantike. Osim toga, kontinuacije se mogu koristiti i za modeliranje složenijih oblika obrade grešaka, korutina i konkurentnih procesa. Ideju kontinuacija ćemo pokušati da ilustrujemo na primeru naredbi imperativnog jezika. U prethodnom slučaju (tzv. direktna semantika) naredbe su uzimale vrednosti iz domena C, dok je operator kompozicije seq bio tipa $[C \to [C \to C]]$. Prevedimo niz naredbi $S_1; S_2; \ldots; S_{n-1}; S_n$ u denotacionu semantiku. U pitanju je kompozicija funkcija koja je asocijativna, pa to možemo napisati kao $S_1; (S_2; \ldots; (S_{n-1}; S_n)\ldots)$, što se prevodi u

$$\text{seq}\ \ s_1\ (\text{seq}\, s_2\ (\ldots(\text{seq}\ \ s_{n-1}\ s_n)\ldots))$$

gde je $s_i = [\![S_i]\!]$. U semantici baziranoj na kontinuacijama interpretacija naredbe S_i nije samo s_i već seq s_i. Zato je naredbe imaju domen $[C \to C]$. Unošenje parcijalne aplikacije funkcije seq u semantiku naredbe daje naredbama slobodu da utiču na način povezivanja sa ostalim naredbama. Naredbe koje ne menjaju tok kontrole imaće semantiku seq s_i, gde je s_i semantika naredbe data kao u slučaju direktne semantike. Ako sa $[\![S_i]\!]_c$ označimo interpretaciju u slučaju kontinuacija, tada važi $[\![\texttt{x:=x+1}]\!]_c = \lambda k\lambda s.k(s+1)$. U odnosu na prethodnu definiciju javlja se jedan dodatni parametar k, iz domena C. Ovaj parametar se naziva *kontinuacija*. Kontinuacija predstavlja izvršenje niza svih onih naredbi koje u nizu slede iza naredbe koja se interpretira, i može se posmatrati kao denotacioni analogon programskog brojača. Naredba kao što je `goto` ignoriše tekuću kontinuaciju i vraća kontinuaciju određenu labelom: $[\![\texttt{goto}\ \ l_1]\!]_c = \lambda k.l_1$. Tako je denotaciona semantika programa

```
L1:   S1;
```

```
S2;
goto L1;
S4;
S5
```

jednaka vrednosti l_1 koja je rekurzivno definisana sa

$$\begin{aligned} l_1 \;=\; & \text{seq } s_1 \\ & (\text{seq } s_2 \\ & (goto\, l_1 \\ & (\text{seq } s_4\ s_5))) \end{aligned}$$

Ovaj pristup se bez problema generalizuje na proizvoljan broj skokova.

Pokušajmo sada da kontinuacije proširimo i na izraze tako što ćemo kompoziciju delova izraza posmatrati u istoj formi kao i kompoziciju naredbi. Posmatrajmo u programu naredbu za uvećavanje brojača iza koje sledi ostatak programa p. Interpretacija takvog programa ima oblik $(\lambda k \lambda s.k(s+1))p$. Pretpostavimo da želimo da proširimo funkcionalnost naredbe uvećavanja brojača tako da pored svog bočnog efekta vrati i vrednost 7, i da tu vrednost možemo dalje koristiti u programu p. Tada program p pretvorimo u lambda apstrakciju $\lambda x.p$, a željenu vrednost 7 u naredbi uvećanja brojača vraćamo tako što na nju primenimo kontinuaciju k. Tako naš program postaje

$$(\lambda k \lambda s.k\ 7\ (s+1))(\lambda x.p).$$

Domen izraza sa vrednostima iz skupa A, koji ima bočni efekat i može da izazove promenu toka kontrole u semantici sa kontinuacijama postaje $\text{Cont}\, A = [A \to C] \to C$. Konstantna vrednost a se interpretira kao $\lambda k.ka$. Neka sada računanje izraza E_1 vraća vrednost a, dok računanje izraza E_2 vraća vrednost b. Želimo da njihovom kompozicijom dobijemo izraz koji vraća vrednost $a+b$. Možemo krenuti od izraza

$$\begin{aligned} & (\lambda k_1.k_1)(\\ & (\lambda k_2.k_2)(\\ & ?+?)) \end{aligned}$$

i primeniti prethodni postupak prenošenja vrednosti jednom,

$$\begin{aligned} & (\lambda k_1.k_1)(\\ & (\lambda k_2.k_2 b)(\lambda y. \\ & ?+y)) \end{aligned}$$

a zatim još jednom:

$$\begin{aligned} & (\lambda k_1.k_1 a)(\lambda x. \\ & (\lambda k_2.k_2 b)(\lambda y. \\ & x+y)) \end{aligned}$$

Sada možemo izvršiti generalizaciju na izraze E_1 i E_2, tako da dobijamo

$$\begin{array}{l}[\![E_1]\!]_c(\lambda x.\\ [\![E_2]\!]_c(\lambda y.\\ x+y))\end{array}$$

Ako računanje vrednosti $x+y$ nije poslednja naredba, onda i ona ima svoju kontinuaciju k, pa je potrebno primeniti je na dobijeni zbir. Tako dobijamo pravilo za prevođenje zbira dva izraza:

$$\begin{array}{lll}[\![E_1\texttt{+}E_2]\!]_c = \lambda k & . & [\![E_1]\!]_c(\lambda x.\\ & & [\![E_2]\!]_c(\lambda y.\\ & & k(x+y)\,))\end{array}$$

Jasno je da celo ovo razmatranje nije ni po čemu specifično za operaciju +, već da važi i za ostale binarne operacije. Ako je $\cdot$ oznaka binarne operacije u apstraktnoj sintaksi, i $\bullet$ njena interpretacija tipa $A \to [A \to A]$, onda definišemo $[\![E_1 \cdot E_2]\!] = \text{lift}\,\bullet [\![E_1]\!]_c [\![E_2]\!]_c$, gde je

$$\text{lift}\, f\, p\, q = \lambda k.p(\lambda x.q(\lambda y.k(fxy))).$$

Videćemo kasnije da se operacije kao što je lift mogu definisati i u opštijem kontekstu monada. Po svojoj izražajnosti kontinuacije su najbliže monadama i predstavljaju veoma bitan primer ovog koncepta.

Slično kao što je u slučaju naredbi `goto dest` naredba bila zadata izrazom $\lambda k.dest$, kada radimo sa kontinuacijama u izrazima pogodno je posmatrati naredbu `gotoWith dest val`, čija je semantika zadata funkcijom $\text{gw}\, dv = \lambda k.dv$. Ova naredba ignoriše tekuću kontinuaciju i prelazi na kontinuaciju d prosleđujući joj vrednost v. Pokušaćemo sada da damo i motivaciju za funkciju `callcc` koja se koristi da bi se u funkcionalnim jezicima koji koriste kontinuacije mogli zadavati složeniji oblici kontrole. Takvi oblici kontrole u imperativnim jezicima koji podržavaju `goto` su zasnovani na *grananju* i *spajanju* tokova kontrole. Grananje je u lambda računu lako realizovati jer imamo funkciju `if`. Spajanje tokova kontrole u imperativnim jezicima znači da se do jednog mesta u programu može doći na više načina. Mesto u programu odgovara kontinuaciji u denotacionoj semantici. Mogućnost da se do istog mesta dođe na dva načina znači mogućnost da se ista kontinuacija koristi na više mesta u lambda izrazu. Zato možemo prvo definisati funkciju koja duplicira tekuću kontinuaciju: $\text{dup}\, f = \lambda k.fkk$. Ako sada uzmemo $f = \lambda\text{out}.(\ldots(\text{gw out val})\ldots)$, u izrazu f smo dobili mogućnost da efektivno iskočimo iz tekućeg bloka u spoljašnji, vraćajući pri tome vrednost val. Sada ovu kombinaciju operacije dup i operacije gw možemo skratiti da bismo izraz f mogli pisati jednostavno $f = \lambda\text{ou}.(\ldots(\text{ou val})\ldots)$, tako što ćemo primenu gw premestiti iz f u definiciju operacije koja duplira tekuću kontinuaciju. Tako dobijamo operaciju

$$\text{callcc}f = \lambda k.f(\text{gw}\, k)k$$

ili, posle zamene definicije za gw,

$$\text{callcc}f = \lambda k.f(\lambda v.\lambda k'.kv)k$$

Oznaka callcc je skraćenica od "call with current continuation" (pozovi sa tekućom kontinuacijom). Operacija vraća vrednost tipa $\mathrm{Cont}\, A = [A \to C] \to C$, kao i ostali izrazi, a njen argument f se primenjuje na $\mathrm{gw}\, k : A \to \mathrm{Cont}\, B$. Zato sledi da je

$$\mathrm{callcc} : [[A \to \mathrm{Cont}\, B] \to \mathrm{Cont}\, A] \to \mathrm{Cont}\, A$$

Napominjemo na kraju da se zbog načina kompozicije izraza koji se interpretiraju u kontinuacijama tip naredbe $C = [S \to S]$ može zameniti tipom $C = [S \to \mathrm{Ans}]$ gde je Ans proizvoljan tip. Ovakav tip se dobija ako na kraj svakog programa dodamo naredbu koja "posmatra" sadržaj rezultujućeg stanja, i tipa je $S \to \mathrm{Ans}$.

Kontinuacije se koriste prilikom kompajliranja funkcionalnih programskih jezika (pre svega jezika Scheme i ML), jer se pomoću njih na eksplicitan način predstavlja tok kontrole u programu. To omogućava sprovođenje dodatnih analiza i optimizacija koje su prisutne i u kompajliranju imperativnih programskih jezika.

2.3.5 Ostali pristupi semantici programskih jezika

Akciona semantika ([86], [87]) se trudi da prevaziđe problem otežane čitljivosti formalne semantike od strane programera kojima nisu bliski formalni metodi. To se postiže pomoću notacije koja liči na opis semantike prirodnim jezikom, a istovremeno ima i preciznu formalnu semantiku. Karakteriše je modularnost i postojanje upotrebljivih alata za automatsku konstrukciju kompajlera. Njen glavni nedostatak je odsustvo dobro razvijene teorije koja bi omogućila dokazivanje svojstava programa zapisanih ovom notacijom.

Evoluirajuće algebre (eng. evolving algebras) koriste ažuriranja stanja (konfiguracije) i prelaze stanja zadate uslovnim izrazima. Primenjene su za specifikaciju implementacije jezika kao što je Prolog i C, ali su rezultujuće specifikacije prilično obimne.

VDM (Vienna Development Method) je postupak baziran na denotacionoj semantici. Poseduje slične probleme kao i denotaciona semantika, pa je osetljiv na promenu domena računanja.

RAISE (Rigorous Approach to Industrial Software Engineering) je sistem koji podržava denotacionu semantiku, ali sadrži i konstrukcije za opis rada sa stanjem i konkurentnim procesima koji međusobno komuniciraju. To je komercijalni proizvod za koji su razvijeni mnogobrojni alati i može se povezati sa akcionom semantikom.

2.4 Lenjo izračunavanje

U ovom odeljku opisujemo neke prednosti lenjog izračunavanja kao bitnog aspekta čisto funkcionalnih jezika poput Haskell-a. Ova osobina čini izražajne mogućnosti funkcionalnih jezika bližim mogućnostima lambda računa i denotacione semantike.

Lenjo izračunavanje predstavlja jedan način realizacije nestriktne semantike funkcionalnih programskih jezika. Dok se striktna semantika realizuje tako što se argumenti funkcije

izračunaju pre nego što se funkcija primeni, lenja semantika se realizuje na sledeći način. Argumenti funkcija su predstavljeni referencama koje u početku sadrže neizračunate podizraze. Kada se prvi put zahteva vrednost tog podizraza, on se izračuna, a izračunata vrednost se prepiše preko podizraza. Time je obezbeđeno da se sledeći put izraz ne računa ponovo, već se iskoristi izračunata vrednost. Na ovaj način se argumenti funkcija izračunavaju najviše jednom. To znači da se vrednost izraza neće izračunati ako neće biti potrebna unutar funkcije, a ako se izračuna, računaće se samo jednom.

U semantičkom smislu, ako $\perp$ označava nedefinisanu vrednost tj. izraz čije se računanje nikad ne završava, striktna funkcija je funkcija f sa osobinom $f\perp = \perp$. U jeziku sa striktnom semantikom su sve funkcije striktne, jer se pri računanju izraza $f\perp$ prvo pokuša sa računanjem $\perp$, a to računanje se nikad ne završava. Nasuprot tome, jezici nestriktne semantike omogućavaju definisanje funkcija za koje ovo ne važi. Tako za funkciju `one` definisanu sa `one x = 1` u jeziku nestriktne semantike važi `one` $\perp$ `= 1`. Nestriktna semantika zahteva složeniju tehniku implementacije i može rezultovati u slabijim performansama programa, ali pruža veće mogućnosti za programiranje. Pored toga, rezonovanje o semantici programa postaje jednostavnije. Na primer, izraz `one e`, gde je `e` proizvoljni izraz, može se u nestriktnoj semantici uprostiti na izraz `1`, što nije slučaj u striktnoj semantici gde je `one` $\perp$ `=` $\perp \neq$ `1`. Zaključujemo da je nestriktna semantika neophodna za jednakosno zaključivanje (eng. equational reasoning), što je osnovni pristup za dokazivanje osobina funkcionalnih programa. U prilog adekvatnosti nestriktne semantike govori i naredba

`if` *uslov* `then` e_1 `else` e_2

koja se u različitim oblicima javlja u svim funkcionalnim jezicima. Ako bi `if` bila striktna funkcija, uvek bi se računala oba izraza e_1 i e_2, što bi rezultovalo u beskonačnoj rekurziji čak i kod najjednostavnijih programa. To znači da se `if` u jezicima striktne semantike mora tretirati na poseban način.

Nestriktna semantika je u tesnoj vezi sa odsustvom bočnih efekata. Nestriktna semantika podrazumeva da tačno vreme računanja vrednosti izraza ne utiče na rezultat računanja. Ukoliko bi izrazi imali bočne efekte, rezultat izvršavanja programa bi zavisio od toga u kojem trenutku je izračunata vrednost izraza. Sa druge strane, teorijski je pokazano ([95]) da jezici bez bočnih efekata koji imaju striktnu semantiku u nekim slučajevima ne mogu da reše probleme jednako efikasno kao jezici sa bočnim efektima. Ovo razmatranje ne važi za jezike sa lenjim izračunavanjem. Šta više, isti argument se može koristiti da bi se pokazalo da su jezici bez bočnih efekata koji imaju striktnu semantiku u nekim slučajevima principijelno neefikasniji od jezika sa nestriktnom semantikom.

Postoje mnoge prednosti koje pruža lenjo izračunavanje u funkcionalnim jezicima. Ono omogućava generalizaciju pojma toka (eng. stream) koji se koristi za apstrakciju ulaza i izlaza na proizvoljne strukture podataka, što se često javlja u primenama. Lenjim izračunavanjem je jedan moćan mehanizam za sinhronizaciju ugrađen u jezik. Kao što je napomenuto u [49], lenjo izračunavanje omogućava jedan novi stepen modularnosti. Naime, često je potrebno izvršiti čitav niz različitih operacija nad nekom strukturom podataka. Umesto da se u jednoj funkciji zadaju sve te operacije, moguće je obradu ovih podataka razdvojiti na faze od kojih svaka proizvodi lenju strukturu podataka koju koristi naredna faza. Rezultujući program je jasniji i postoji veća mogućnost za ponovno korišćenje neke od njegovih funkcija. Pri tome se, zahvaljujući lenjom izračunavanju, program izvršava u osnovi na isti način kao da su te

faze sažete u jednu: struktura podataka se računa po potrebi, deo po deo. To dozvoljava da strukture podataka budu i beskonačne. Ova razmatranja ilustrujemo sledećim primerom.

Primer Funkcija `fsum` računa sumu prvih `n` Fibonačijevih brojeva. Bez primene upravo opisane modularnosti dobijamo sledeću funkciju.

```
fsum n = faccum 1 1 0 n
faccum f0 f1 sum 0     = sum
faccum f0 f1 sum (n+1) = faccum f1 (f0+f1) (sum+f0) n
```

Primenom modularnosti dobijamo funkciju

```
fsum n = sum (take n fib)
```

gde je `sum` funkcija koja računa sumu elemenata liste, `take` je funkcija koja izdvaja prvih `n` elemenata liste, a `fib` je beskonačna lista Fibonačijevih brojeva definisana sa

```
fib = 1 : 1 : sumfirst fib
sumfirst (x1:x2:xs) = x1+x2 : sumfirst (x2:xs)
```

U većini implementacija funkcionalnih jezika je prva verzija programa efikasnija, ali je razlika samo u konstantnom faktoru. Transformacije kao što su obešumljavanje (eng. deforestation, program fusion) i analiza striktnosti mogu smanjiti gubitak vremena usled lenjog izračunavanja.[1] Druga verzija koristi funkcije koje se mogu upotrebiti i u mnogo širem kontekstu i jednostavnije je utvrditi njenu korektnost.

Ovakva primena modularnosti dolazi do izražaja i prilikom pisanja interpretera ili kompajlera u funkcionalnom jeziku. Lenjo izračunavanje omogućava da se leksički analizator, sintaksni analizator i interpreter realizuju kao posebne faze interpretera. Svaka od njih se može testirati nezavisno, a prilikom izrvršavanja interpretera zahtev za vrednošću izraza inicira korak sintaksne analize koji pokreće korak leksičke analize, što je upravo situacija koju imamo i kod klasičnih jednoprolaznih jezičkih procesora.

Moć lenjog izračunavanja dolazi do izražaja i prilikom modeliranja nedeterminizma tehnikom "liste uspeha" (eng. list of successes, [126]). To je postupak kojim se relacija između tipova a i b realizuje kao funkcija $a \to [b]$. Ako funkcija predstavlja traženje elementa koji zadovoljava neki uslov, a taj element se ne pronađe, funkcija vraća praznu listu. Ako postoji više elemenata sa tim svojstvom, vraća listu tih elemenata. Kompozicija relacije $a \to [b]$ je sa relacijom $b \to [c]$ može se realizovati pomoću naredbe

```
compose :: (a -> [b]) -> (b -> [c]) -> (a -> [c])
compose f g = \a -> concat (map g (f a))
```

ili drugačije zapisano

```
compose f g = concat . map g . f
```

[1] Hugs interpreter čak daje nešto bolje rezultate za drugu verziju, ali kompajler za Clean koji vrši mnoge optimizacije, ali ne i obešumljavanje, izvršava prvu verziju znatno efikasnije zbog smanjene alokacije na heap-u.

gde je

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

funkcija koja listu listi pretvara u listu vrednosti, te je, na primer,

```
concat [[1,2,3],[],[4,5],[6]] = [1,2,3,4,5,6]
```

Ova tehnika dozvoljava da program pišemo kao kompoziciju relacija. Ako nas na kraju zanima samo jedan od rezultata relacije `r : a -> [b]` primenjene na argument `x`, možemo ga dobiti kao `head (r x)`. Ono što dobijamo lenjim izračunavanjem je da se u tom slučaju ne računa cela lista `r x`, već samo one vrednosti koje su potrebne za računanje prvog elementa te liste. To praktično odgovara primeni pretraživanja stabla rešenja u dubinu. U odeljku 3.2 ćemo videti kako se ova tehnika može posmatrati kao primena monade za nedeterminizam.

2.5 Memoizacija

Memoizaciju je prvi put uveo Michie 1968. godine kao tehniku kojom se višestruko izračunavanje vrednosti neke složene funkcije izbegava pamćenjem prethodno izračunatih vrednosti. Razmatranja memoizacije koja ovde dajemo su bazirana na [27], poglavlje 19. U odeljku 3.5.1 pokazaćemo kako bi se memoizacija mogla primeniti u postupku lenje konstrukcije stanja i prelaza konačnog automata.

Kada se prvi put primeni na neki argument, memo funkcija izračuna njegovu vrednost kao i obična funkcija, a zatim tu vrednost zapamti u *memo tabeli*. Kada se sledeći put primeni na neki argument tada se prvo proverava da li se argument već nalazi u memo tabeli. Ukoliko je to slučaj, onda se jednostavno vraća ranije zapamćena vrednost umesto da se funkcija računa ponovo. Memo funkcija u čisto funkcionalnom jeziku vraća isti rezultat kao i obična funkcija, jer izračunavanje vrednosti nema nikakav bočni efekat. Pored lenjog izračunavanja, to je još jedan pokazatelj da odsustvo bočnih efekata pruža veću slobodu u načinu implementacije funkcija.

Primer Da bismo pokazali prednost memo funkcija poslužićemo se opet funkcijom za računanje Fibonačijevih brojeva. Način na koji se obično piše rekurentna veza kojom se definišu Fibonačijevi brojevi odgovara sledećem funkcionalnom programu.

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Nedostatak ovako zapisane definicija funkcije `fib` je što u klasičnoj implementaciji broj rekurzivnih poziva pri računanju `fib n` eksponencijalno raste sa `n`. Ukoliko je, međutim, funkcija `fib` realizovana kao memo funkcija, tada se svaka od vrednosti `fib i` za $1 \leq i \leq n$ računa po jednom, pa je broj poziva funkcije linearan. Na ovom primeru vidimo da memo funkcije omogućavaju da se programi pišu bliže specifikaciji umesto da se transformišu u manje čitljiv oblik. Tako se tehnike kao što su dinamičko programiranje mogu izbeći, a

da rezultujući program ima istu asimptotsku složenost kao i da su primenjene. I dok je u slučaju računanja Fibonačijevih brojeva jednostavno napisati i verziju funkcije koja se izvršava u linearnom vremenu, u složenijim slučajevima nije moguće unapred znati da li će neka vrednost funkcije biti potrebna u programu i kojim će se redosledom računati vrednosti funkcije za različite argumente. U tom slučaju bismo bili prinuđeni da svaki put ponovo računamo vrednost, ili da transformišemo funkciju tako da koristi memo tabelu realizovanu u funkcionalnom jeziku. Poslednji postupak transformacije izvornog programa se može i automatizovati, ali se u svakom slučaju pri njegovoj primeni javlja sledeći problem: efikasna realizacija memoizacije počiva na destruktivnom ažuriranju memo tabele, što nije moguće realizovati u čisto funkcionalnom jeziku. (Postoje slučajevi kada je moguće destruktivno ažuriranje struktura u čisto funkcionalnim programskim jezicima, ali su oni obično zasnovani na pretpostavci da postoji najviše jedan pokazivač na strukturu. Ograničenje na najviše jedan pokazivač na memo tabelu bi smanjilo upotrebnu vrednost memo funkcija. Suština transformisane funkcije je upravo u tome što ona računa istu vrednost bez obzira na trenutnu verziju njene memo tabele, ali ako je transformacija izvršena u izvornom kodu, ne može se od kompajlera očekivati da to uzme u obzir.) Primeri kod kojih memoizacija može doneti znatnu korist su interpreteri i kompajleri. U [27] je dat primer interpretera koji je napisan kao da implementira "call by name" semantiku, ali ako se sam interpreter rezlizuje pomoću memo funkcija, ponaša se kao da implementira lenjo izračunavanje. Dalje, kompajler koji bi se realizovao pomoću memo funkcija bi se ponašao kao *inkrementalni kompajler* koji kompajlira samo one delove programa koji su promenjeni.

Ono što sprečava širu primenu memo funkcija kao opšteg mehanizma su dva problema u njihovoj implementaciji: memoizacija vrednosti složenih struktura podataka i kontrola veličine memo tabela.

Ukoliko se želi izvršiti memoizacija vrednosti složenih struktura javlja se problem njihovog poređenja. Potpuna provera jednakosti (tzv. puna memoizacija) zahteva vreme koje raste sa povećanjem veličine strukture, a na beskonačne strukture se ne može ni primeniti. Dalje, takvo poređenje dovodi do potpunog izračunavanja strukture koja se poredi, pa memo funkcije postaju obavezno striktne i to čak hiperstriktne (pokušavaju da izračunaju svoj argument do normalne forme, a ne samo slabe glavene normalne forme WHNF). Ovi problemi se mogu rešiti primenom *lenje memoizacije*. Suština ovog postupka je da se umesto poređenja vrednosti složenih struktura porede njihovi pokazivači. Time se rešavaju svi problemi u vezi kompleksnih struktura podataka, ali se potencijalno smanjuje broj slučajeva u kojima je moguće uštedeti računanje funkcije jer će se neki jednaki argumenti smatrati različitim samo zato što su smešteni na različitim mestima u memoriji. Ukoliko se, međutim, memoizacija primeni i na sve konstruktore struktura podataka, tada se iste strukture smeštaju na isto mesto u memoriji, pa se postiže efekat potpune memoizacije. Ako, na primer, definišemo

```
hCons = memo cons
unique = foldr hCons [ ]
```

tada funkcija `unique` vraća jedinstvenu kopiju liste koja joj je prosleđena: ako se `unique` primeni dva puta na listu koja ima istu vrednost, (bez obzira da li su pokazivači argumenata isti) rezultat će uvek biti isti pokazivač. Primena memoizacije na sve konstruktore nije praktična, ali pokazuje da ovim postupkom možemo postići i efakat pune memoiza-

cije. Stepen primene memoizacije određuje programer odabirom funkcija (i konstruktora) koje proglašava za memo funkcije. Jedan primer ovakve selektivne upotrebe memoizacije dat je u odeljku 3.5.1 kao mogući način za realizaciju lenje konstrukcije prelaza i stanja konačnog automata. Lepa osobina lenje memoizacije je što ciklične strukture pretvara u ciklične. Ako je, na primer, definisana struktura `ones = 1 : ones`, koja se u memoriji realizuje kružnom listom, tada se kao rezultat izračunavanja memoizovane funkcije `map double` gde je `double x = 2*x`, posle drugog koraka dobija ciklična struktura kao i ona koju bismo dobili iz definicije `twos = 2 : twos`. Dalje računanje ne zahteva primenu `map` niti primenu bilo kakve memoizacije, što znači da je praktično izračunata beskonačna struktura u dva koraka. Lenja memoizacija se takođe pokazuje korisnom u rešavanju sledećeg problema koji ograničava primenu memo funkcija: kontrole veličine memo tabela.

Jednostavan način da se kontroliše zauzimanje prostora od strane memo tabela je da se memo funkcije deklarišu kao lokalne ukoliko je bitna njihova memoizacija samo u toku lokalnog računanja (npr. da bi se očuvale ciklične strukture). U tom slučaju se memo tabele dealociraju pri izlasku iz odgovarajućeg bloka. Sledeći bitan postupak za kontrolisanje veličine memo tabele je upotreba funkcije koja upravlja tabelom (eng. table manager). Ova funkcija prilikom svakog novog ubacivanja u memo tabelu određuje koji se prethodni elementi mogu izbrisati. U nekim slučajevima, kada su funkcije definisane šemom rekurzije specijalnog oblika, kompajler može sam da generiše funkcije za upravljanje memo tabelom. Druga mogućnost je da korisnik sam definiše funkciju za upravljanje tabelom.

Osim navedenih problema sa memo funkcijama, potrebno je i garbage collector (deo sistema za izvršavanje funkcionalnih programa koji upravlja memorijom) prilagoditi da bi ispravno radio sa pokazivačima u memo tabelama. Jedan način da se to uradi je opisan u [27].

2.6 Elementi programskog jezika Haskell

U ovom delu dajemo pregled osnovnih osobina programskog jezika Haskell. Haskell je čisto funkcionalni jezik nestriktne semantike sa Curry-jevim funkcijama višeg reda. Drugim rečima, to znači da su funkcije u Haskell-u funkcije u smislu lambda računa. Nastao je 1987. godine kao pokušaj da se ujedini mnoštvo jezika zasnovanih na principima nestriktne semantike i odustva bočnih efekata, kako bi se olakšala međusobna komunikacija u ovoj oblasti i time pospešio opstanak ovog pravca razvoja programskih jezika. Namenjen je za obrazovne, istraživačke i praktične primene. Od prvobitne verzije on je evoluirao, pretežno se šireći, te je pretila opasnost da postane nerazumljiv za sve osim uske grupe istraživača. Zato je početkom 1999. godine napisana definicija jezika Haskell98. Haskell98 bi trebalo da bude stabilan jezik koji bi se koristio kao minimalni standard za implementaciju Haskell-a, kao i u obrazovne svrhe. Ovde dajemo pregled jezika definisanog tim standardom, a zainteresovanog čitaoca upućujemo na [65], [45], [13], [14] ili [112]. Implementacije Haskell-a su u javnom vlasništvu i mogu se preuzeti sa adrese `www.haskell.org`.

Haskell je definisan svojim jezgrom (eng. kernel), i skupom pravila kojim se određene sintaksne konstrukcije prevode u konstrukcije iz jezgra. Ovaj pristup znatno pojednostavljuje specifikaciju semantike. Mnoštvo sintaksnih konstrukcija koje formalno ne povećavaju izražajnost jezika (tzv. sintaksni šećer, eng. syntax sugar) u Haskell-u je prisutno da bi se

programi pisali konciznije i intuitivnije.

Programi u Haskell-u su organizovani u module koji predstavljaju mehanizam za kontrolu vidljivosti imena. Moduli sadrže deklaracije u kojima se javljaju izrazi. Haskell je statički tipiziran jezik, sa mogućnošću za parametarski polimorfizam kao i za ad-hoc polimorfizam (overloading). Poslednja mogućnost je realizovana klasama (type classes), koje se razlikuju od klasa u objektno-orijentisanim jezicima i opisane su u 2.7.

2.6.1 Leksička struktura

Haskell program je niz leksema odvojenih prazninama. Lekseme se prepoznaju tako što se uvek uzima najduži niz znakova koji predstavlja leksemu (eng. maximal munch rule, longest match rule). Neugnježdeni komentar počinje leksemom `--` i traje do kraja linije. Ugnježdeni komentar počinje leksemom `{-` i završava se leksemom `-}`.

Postoje dve vrste identifikatora: oni koji počinju velikim slovom i oni koji počinju malim slovom. Znak `_` se tretira kao malo slovo, a ako stoji sam, to je poseban neimenovan identifikator koji se koristi ako ime identifikatora nije bitno.

Operatori predstavljaju niz specijalnih simbola. Operatori koji počinju sa `:` su konstruktori, ostali su funkcije. Operatori su infiksni ukoliko se drugačije ne specificira. Operatori se tretiraju kao identifikatori ako se stave u zagrade, npr. `(+) x y = x + y`. Identifikatori se tretiraju kao operatori ako se pišu unutar obrnutih navodnika, npr. `x ‘mod‘ y = mod x y`.

Identifikatorima i operatorima se označavaju imena u programu. Postoji 6 vrsta imena: promenljive, konstruktori, promenljive tipa, konstruktori tipa, klase i moduli. Identifikatori moraju početi malim slovom ako se njima označavaju promenljive ili promenjive tipa, u suprotnom počinju velikim slovom. Ispred imena se može pisati ime modula sa tačkom ukoliko je to ime uvezeno iz nekog modula kvalifikovanim uvozom.

U cilju povećanja čitljivosti programa, oznake interpunkcije `{`, `}` i `;` se mogu izostaviti ukoliko se koristi sledeća konvencija o uvlačenju konstrukcija programa i rasporedu sintaksnih konstrukcija po linijama (layout rule). Ukoliko se iza neke od ključnih reči `where`, `let`, `do` ili `of` izostavi `{`, tada se izostavljena zagrada umetne i zapamti se pozicija naredne lekseme u redu. Za svaku sledeću nepraznu liniju se umeće `;` ukoliko ona počinje na zapamćenoj poziciji, a umeće se `}` ukoliko počinje na manjoj poziciji. Ovakva definicija dozvoljava i da se koristi mešavina znakova interpunkcije i uvlačenja.

2.6.2 Izrazi

Izrazi predstavljaju niz imena (identifikatora i operatora), znakova interpunkcije i ključnih reči `let`, `in`, `if`, `then`, `else`, `case`, `of`, `do`. Mogu se posmatrati kao lambda izrazi sa sintaksnim šećerom. Izrazi koji označavaju vrednosti mogu da sadrže sledeće konstrukcije.

- primenu funkcija: primena funkcije `f` na argument `x` se piše jednostavno `f x` i ima najviši prioritet;

- infiksne operatore, koji mogu biti levo ili desno asocijativni i imaju prioritete u opsegu 0 do 9; na način parsiranja izraza utiču i deklaracije prioriteta korisnički definisanih operatora;

- lambda apstrakcije oblika `\ p1 p2 ... pn -> e` koje označavaju (anonimnu) funkciju koja primenjena na argumente oblika `p1 p2` ... `pn` vraća vrednost izraza `e`;
- uzorke (patterns), kojima se implicitno definišu neke promenljive, kao `n+1` u definiciji `fact (n+1) = n * fact n` ili `x:xs` u definiciji `cdr (x:xs) = xs`;
- `let`-izraze oblika

```
let x1 = e1
    x2 = e2
      ...
    xn = en
in e
```

  kojima se uvode vrednosti koje promenljive imaju u izrazu `e`;
- `case`-izraze oblika

```
case e of
  p1 -> e1
  p2 -> e2
    ...
  pn -> en
```

  koji definiše vrednost u zavisnosti od oblika izraza `e`;
- uslovne izraze oblika `if c then e1 else e2` koji predstavljaju skraćenicu za `case`-izraz;
- liste, koje predstavljaju skraćenicu za primenu konstruktora `:` i `[]`; tako je, na primer, `[1,3,7] = 1 : (3 : (7 : []))`
- torke (eng. tuples), kao što je uređena trojka `(1,'w',8.3)`
- aritmetičke nizove, kao što je `[10,12..17]`, što je skraćeni oblik za `enumFromThenTo 10 12 17` i označava listu `[10,12,14,16]`;
- ZF-izrazi (list comprehensions) koji označavaju liste, a njihovo precizno značenje je dato prevodom u funkcije `concat` i `map`; tako `[ x + y | x <- xs, y <- ys, x < y ]` označava listu zbirova takvih parova brojeva iz listi `xs` i `ys`, da je prva komponenta para manja od druge. Zapis funkcije `compose` iz odeljka 2.4 pomoću ZF-izraza je

```
compose :: (a -> [b]) -> (b -> [c]) -> (a -> [c])
compose f g a = [c | b <- f a, c <- g b]
```

- `do`-izraze oblika

```
do s1
   s2
   ...
   sn
```

  koji predstavljaju skraćenu primenu operatora `>>` i `>>=` iz klase `Monad`, prema sledećim pravilima:

```
do {e} = e
do {e;es} = e >> do {es}
do {p <- e; es} = e >>= (\p -> do {es})
```

  pri čemu se poslednje pravilo komplikuje ako uzorak `p` nije promenljiva, videti [65];

- konstrukcije i ažuriranje vrednosti algebarskih tipova podataka sa imenovanim poljima, tako se uz deklaraciju

```
data List a = Nil | {hd :: a, tl :: List a}
```

  može pisati `x = List{hd=1,tl=Nil}` da bi se označila nova vrednost tipa `List Int`, ili `x{hd=2}` kao oznaka za listu koja se od `x` razlikuje samo po vrednosti polja `hd`;

- deklaracije tipova podizraza koji označavaju vrednost, kao `::Float` u definiciji funkcije `f x = x * (2::Float)`.

Izrazima koji označavaju vrednost se statički dodeljuje tip Hindley-Milner-ovim postupkom. Ukoliko je za neki podizraz eksplicitno zadat tip u programu, onda se vrši unifikacija specificiranog tipa sa izvedenim tipom, čime je moguće izbeći dvosmislenosti koje kompajler ne može sam da odredi, ili suziti najopštiji tip koji je izrazu dodelio kompajler. U slučaju rekurzivnih definicija funkcija moguće je zadati i opštiji tip nego što bi postupak izvođenja tipova našao, što omogućava upotrebu polimorfne rekurzije (videti [65], odeljak 4.4.1, kao i [92], glava 10). Izrazi koji označavaju tipove mogu da sadrže sledeće konstrukcije.

- primenu jednog izraza na drugi, što se označava na isti način kao i primena funkcija, na primer, `m a`;
- funkcijski tip, kao na primer `a -> b`;
- konstruktor liste, `[ ]` koji označava tip listi datog elementa kao u deklaraciji tipa `map :: a -> b -> [a] -> [b]`;
- konstruktor torke, kao `(a,b)` u deklaraciji tipa funkcije `first :: (a,b) -> a`;

- specifikacija konteksta u deklaraciji tipa, koja označava da određeni podizraz koji označava tip mora predstavljati instancu date klase, kao `Ord a =>` u primeru

```
sort :: Ord a => [a] -> [a]
```

Izrazima koji označavaju tipove se statički prilikom provere tipova dodeljuje vrsta (kind). Vrsta može imati vrednost `*` ili vrednost `(p -> q)` gde su `p` i `q` vrste. Izraz označava tip koji može uzimati vrednosti samo ako ima vrstu `*`, u suprotnom je u pitanju konstruktor.

Dobar deo sintaksnog šećera se oslanja na primenu uparivanja uzoraka (prepoznavanje oblika, eng. pattern matching). Uzorak označava formu koju vrednost izraza može da ima. Sastoji se od promenljivih koje uparivanjem treba da dobiju vrednost, kao i konstruktora, listi, prirodnih brojeva i znaka `+`, imena polja označenih algebarskih tipova, kao i oznaka `~` (uzorak koji uvek uspeva) i oznake `@` kojom se dodeljuje ime poduzorku nekog uzorka. Prilikom uparivanja se vrši uporedo računanje vrednosti izraza i poređenje sa uzorkom. Rezultat ovog procesa može biti

1. izraz odgovara uzorku; u tom slučaju promenljive iz uzorkka dobijaju vrednosti odgovarajućih komponenti vrednosti izraza;

2. izraz ne ogovara uzorku (fail);

3. računanje se nikad ne završava: ovo se dešava ako pokušaj određivanja da li vrednost može biti u obliku koji određuje uzorak divergira.

Uzorci u nizu uzoraka se uparuju s leva na desno, spolja prema unutra. Precizniji opis ovog postupka je dat u [65]. Uparivanje uzoraka predstavlja alternativu korišćenja destruktora, kojima bi se pristupalo elementima složenih struktura podataka. Prednost uparivanja uzoraka je u konciznosti, što posebno dolazi do izražaja prilikom dokazivanja osobina programa. Kao primer funkcije koja koristi uparivanje uzoraka dajemo funkciju koja obrće binarno stablo.

```
data Tree a = Nil | Node a (Tree a) (Tree a)
swaptT :: Tree a -> Tree a
swaptT Nil = Nil
swaptT (Node x left right) = Node x (swapT right) (swapT left)
```

2.6.3 Deklaracije

Razlikujemo daklaracije tipova (`type`, `newtype`, `data`), deklaracije klasa (`class`, `instance`, `default`) i ugnježdene deklaracije: deklaracije vrednosti, pridruživanje tipa nekoj vrednosti i deklaracije prioriteta operatora. Ugnježdene deklaracije se mogu javiti i unutar `let` i `where` blokova.

Deklaracija `data` Ovom deklaracijom se uvode algebarski tipovi. Tako se deklaracijom

```
data Tree a = Leaf a | Node a  (Tree a) (Tree a)
```

definiše binarno stablo parametrizovano tipom `a`. Tom deklaracijom se zadaje da je `Tree` konstruktor tipa vrste `* -> *`, dok su `Leaf` i `Node` konstruktori vrednosti čiji su tipovi:

```
Leaf :: a -> Tree a
Node :: a -> Tree a -> Tree a -> Tree a
```

Ovakva definicija odgovara sledećoj rekurzivnoj definiciji binarnog stabla sa elementima `a`:

- čvor sa elementom tipa `a` je stablo;
- ako su `t1` i `t2` stabla, a `x` element tipa `a`, onda je i `Node x t1 t2` stablo.

Formalno, algebarski tipovi se dobijaju primenom proizvoda i sume domena iz denotacione semantike, uz korišćenje rekurzije. Polja algebarskih tipova se mogu imenovati, pa je prethodna deklaracija mogla da glasi i ovako.

```
data Tree a = Leaf {getLeaf :: a}
            | Node {treeElem :: a, left :: Tree a, right :: Tree a}
```

Pri tome su

```
  getLeaf, treeElem :: Tree a -> a
  left, right       :: Tree a -> Tree a
```

destruktori koji predstavljaju skraćeni zapis (parcijalnih) funkcija datih uparivanjem vrednosti. Tako je npr. funkcija `left` definisana sa `left (Node x le ri) = le`.

Konstruktori tipova se mogu označiti kao striktne funkcije znakom `!`, što može znatno da utiče na performanse programa. Može se zadati i kontekst kojim se ograničavaju promenljive tipova za koje deklaracija algebarskog tipa važi. Tako

```
data Ord a => Tree a = Leaf a | Node a  (Tree a) (Tree a)
```

dozvoljava definisanje tipova `Tree a` samo za one tipove `a` koji su intance klase `Ord`.

Deklaracija `newtype` Ova deklaracija uvodi novu reprezentaciju za postojeći tip. Slična je deklaraciji `data` koja ima samo jednu alternativu.

```
newtype N = N {noN :: Int}
```

Pri tome je, za razliku od `data` deklaracije, `N` $\perp$ `=` $\perp$. Pri proveri tipova `N` i `Int` se tretiraju kao različiti te je potrebno izvršiti ekplicitnu konverziju pomoću konstruktora `N` ili destruktora `noN`.

Deklaracija `type` Ovom deklaracijom se uvodi sinonim za postojeći tip. Tako se deklaracijom

```
type Command a s = a -> (a, s)
```

uvodi identifikator `Command` koji predstavlja samo skraćenicu za izraz sa njegove desne strane. Sinonim se može koristiti u drugim izrazima samo ako se navedu svi njegovi argumenti.

Deklaracije operatora Moguće je zadati da je određeni operator levo asocijativan (`infixl`), desno asocijativan (`infixr`) ili neasocijativan (`infix`). Zadaje se i prioritet operatora kao broj od 0 do 9.

Definicije funkcija Definicija funkcije vezuje vrednost imena za vrednost koja predstavlja funkciju. Najopštiji oblik definicije funkcije `f` je sledeći.

```
f p11 p12 ... p1k  m1
f p21 p22 ... p2k  m2
  ...
f pn1 pn2 ... pnk  mn
```

Pri tome su `pij` za $1 \leq i \leq n$, $1 \leq j \leq k$ uzorci, a `mi` za $1 \leq i \leq n$ su parovi (eng. match) koji određuju vrednosti funkcija kada argumenti funkcije `f` zadovoljavaju sve uzorke `pij` za $1 \leq j \leq k$. Opšti oblik parova `mi` je

```
| gi1 = ei1
| gi2 = ei2
 ...
| gis = eis
   where ...
```

Pri tome je svaki `gir` ($1 \leq r \leq s$) izraz tipa `Bool` koji se naziva *stražar* (eng. *guard*), dok je `eir` izraz koji definiše vrednost funkcije kada su argumenti oblika datim sa `pij`, $1 \leq j \leq k$, a zadovoljen je uslov `gir`. Broj `s` zavisi od `i`, tj. broj stražara u svakom paru može biti različit.

Primetimo da arnost funkcije, `k`, mora biti ista za svaku jednačinu kojom se definiše vrednost funkcije (to je samo sintaksni zahtev u definiciji funkcije i ne ograničava broj argumenata na koji se funkcija može primeniti, jer su sve funkcije Curry-jeve).

Uzorci i stražari se mogu međusobno preklapati tj. može se desiti da je zadovoljeno više njih, i proveravaju se od vrha na dole, s leva na desno. Opšta preporuka je ipak da se uzorci pišu tako da ne zavise od redosleda kojim su navedeni ukoliko to znatno ne umanjuje čitljivost.

Kako su u mnogim slučajevima stražari nepotrebni, uvodi se mogućnost da par bude jednostavno oblika

```
= e   where ...
```

što je ekvivalentno sa

```
| True = e where ...
```

Ključna reč `where` se koristi za uvođenje lokalnih definicija koje važe za dati par. Ukoliko nema lokalnih definicija može se izostaviti i ključna reč `where`.

Vezivanje uzorka Vezivanje uzorka (pattern binding) ima oblik `p m`, gde je `p` uzorak, a `m` je par, kao i u prethodnom odeljku. Vezivanjem uzorka se promenljive koje učestvuju u uzorku vezuju za vrednost, po istom mehanizmu kao i kod definicije funkcije.

2.6.4 Moduli

Modul predstavlja niz deklaracija (opisanih u prethodnom odeljku). Modul definiše nova imena u kontekstu imena uvezenih ključnom rečju `import` i izvozi neka od tih imena, što se označava listom imena u zagradama iza imena modula. Dajemo primer modula koji implementira apstraktni tip stek.

```
module Stack(StkType, push, pop, empty) where
data StkType a = EmptyStk | Stk a (StkType a)
push x s = Stk x s
pop (Stk _ s) = s
empty = EmptyStk
```

Program u Haskell-u je skup modula od kojih se jedan proglašava za glavni modul, naziva se `Main` i izvozi ime `main` koje mora biti tipa `IO a` za neki tip `a`. Moduli su međusobno povezani `import` deklaracijama koje mogu definisati uzajamno rekurzivne veze.[2] Moduli su čisto sintaksne konstrukcije i svaki program napisan u više modula se može jednostavno transformisati u jedan modul. Svi moduli se nalaze na jednom globalnom nivou, nema ugnježdenih modula. Specijalni modul `Prelude` se uvozi u svaki modul ukoliko se ne navede drugačije.

Ako se u listi izvoza navede samo ime algebarskog tipa, onda se njegova struktura ne izvozi, pa on postaje apstraktni tip podataka. Ako se žele izvesti i neki od konstruktora tipa, oni se mogu staviti u zagradu iza imena tipa, npr. `StkType(EmptyStk)`. Oznaka `(..)` se koristi kao skraćenica za listu svih konstruktora algebarskog tipa. Slično važi i za izvoz klasa u odnosu na funkcije definisane u klasi. Mogu se takođe izvesti i moduli koji su uvezeni. Ukoliko se izostavi lista izvoza, izvoze se sva imena definisana u tom modulu (ali ne i ona koja su uvezena).

Imena se uvoze u modul ključnom rečju `import`, iza koje može slediti lista imena koja se iz tog modula uvoze. Ukoliko se ispred imena modula navede `qualified`, ispred uvezenih imena se mora pisati ime modula i tačka. Pomoću ključne reči `as` može se promeniti prefiks kojim se pristupa imenima modula koji je uvezen sa `quialified`. Ako se ne navede lista imena koja se uvoze iz modula, podrazumeva se da se uvoze sva imena koja taj modul izvozi, ali se iz tog skupa mogu izostaviti željena imena njihovim navođenjem u zagradama iza ključne reči `hiding`.

[2] Ova mogućnost nije implementirana u Hugs interpreteru od Februara 2000.

2.7 Klase u Haskell-u

U ovom odeljku opisane su *klase tipova* (eng. type classes) u programskom jeziku Haskell sa proširenjima implementacije Hugs98 koja dozvoljava upotrebu multiparametarskih klasa. Ova mogućnost jezika je od velikog značaja za prenošenje konstrukcija iz terorije kategorija u jedan funkcionalni programski jezik. Opis klasa u verziji Haskell98 dat je u [65], opis implementacije jezika Gofer koji je prvi uveo multiparametarske klase dat je u [57], a neka razmatranja multiparametarskih klasa su data u [61]. Odmah napominjemo da postoji bitna razlika između pojma klase u Haskell-u i pojma klase u objektno-orijentisanim ili hibridnim imperativnim jezicima kao što su Java ([35]) ili Oberon-2 ([88]).

Klase su inicijalno uvedene u Haskell u cilju sistematskog rešavanja mogućnosti programskih jezika koja se naziva "ad hoc polymorphism" ([119], [118]) ili popularno "overloading". Ova mogućnost je prisutna u operacijama nad brojevima u gotovo svim programskim jezicima i omogućava da se koriste isti simboli za operacije nad celim i realnim brojevima različitih unutrašnjih reprezentacija. Sastavni deo provere tipova u tom slučaju predstavlja izbor odgovarajuće operacije iz skupa operacija koje se mogu označiti datom sintaksnom konstrukcijom. Ovaj izbor se vrši na osnovu tipova operanada. U najjednostavnijem slučaju, klasa u Haskell-u predstavlja kolekciju tipova. Definiše se `class` deklaracijom, dok se `instance` deklaracije koriste da bi se specificiralo da dati tip ili elementi familije tipova pripadaju klasi. Pojam tipa pridruženog promenljivoj se proširuje tako da, pored najopštijeg tipa koji se dobija kao rezultat Hindley-Milner postupka, sadrži i *kontekst*. Kontekst zadaje ograničenja za promenljive u tipu.

Primer Neka klasa `Numer` predstavlja brojeve nad kojima je moguće vršiti aritmetičke operacije sabiranja i množenja. Tada njena deklaracija izgleda ovako.

```
class Numer a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
```

Da bismo iskazali činjenicu da tipovi `Int` i `Float` pripada familiji tipova koja čini klasu `Numer`, koristimo sledeće deklaracije.

```
instance Numer Int where
  (+) = primIntPlus
  (*) = primIntTimes
instance Numer Float where
  (+) = primFloatPlus
  (*) = primFloatTimes
```

Pri tome, na primer, `primIntPlus` predstavlja operaciju nad celim brojevima, tipa `Int -> Int -> Int`.

Sistem tipova pronalazi minimalni kontekst koji je neophodan da bi se odgovarajuće operacije mogle primeniti. Tako je za funkciju definisanu sa `addDigit n d = 10*n+d` najoštiji tip glasi

```
addDigit :: Numer a => a -> a -> a
```

Klase predstavljaju mehanizam za specifikaciju apstraktnih tipova podataka. Njima se definiše samo signatura tj. funkcije i tipovi funkcija koje čine tu klasu. Ovako definisane klase odgovaraju interfejsu u programskom jeziku Java ([35]), dok `instance` deklaracije odgovaraju `implements` specifikaciji u Javi kojom se zadaje da data klasa implementira navedene interfejse.

Primer Apstraktni tip skupa celih brojeva se može zadati na sledeći način.

```
class Set s where
  emptySet :: s
  addElem  :: Int -> s -> s
  member   :: Int -> s -> Bool
```

U Javi je moguće definisati nasleđivanje kako klasa, tako i interfejsa. Osim toga, termin klasa u Javi se koristi u značenju tipa podataka. U Haskell-u klase nisu tipovi, i nema smisla govoriti o pripadnosti vrednosti nekoj klasi, već samo pripadnosti vrednosti nekom tipu i pripadnosti nekog tipa klasi. Nasleđivanje u Haskell-u je moguće samo za klase, ne i za tipove. Pošto klase predstavljaju samo signaturu, bez implementacije, nema smetnji za realizaciju višestrukog nasleđivanja. `instance` deklaracije u Haskell-u se mogu pozivati na egzistenciju drugih instanci, pa i instanci iste klase, te je moguće specificirati složena pravila o pripadnosti tipova klasama.

Primer Jednakost nad listama moguće je specificirati sledećom deklaracijom.

```
instance Eq a => Eq [a] where
  (==) xs ys = and [x == y | x <- xs, y <- ys]
```

Specifikacijom konteksta u deklaraciji klase se ta klasa deklariše kao potklasa date klase. Tako naredna deklaracija označava da je `Ord` potklasa klase `Eq`.

```
class Eq a => Ord a where
  compare                 :: a -> a -> Ordering
  (<), (<=), (>=), (>)    :: a -> a -> Bool
  max, min                :: a -> a -> a
```

Praktična implikacija navođenja konteksta `Ord a` u definiciji klase `Ord a` je da pri svakom navođenju `instance` deklaracije kojom se određuje da tip `a` pripada klasi `Ord`, mora da postoji i deklaracija instance kojom se se taj tip deklariše kao pripadnik klase `Eq`. To znači da kad želimo da neki tip deklarišemo kao instancu neke klase, moramo ga prethodno deklarisati kao instancu svih njenih nadklasa.

Sa druge strane, deklaracija instance koja sadrži kontekst se može shvatiti kao implikacija koja za svaku instancu sa leve strane znaka `=>` generiše instancu sa desne strane znaka `=>`. I u slučaju klasa i u slučaju instanci kontekst može da sadrži više klasa koje se pišu u zagradi odvojeni zarezima. U slučaju definicije klase to označava višestruko nasleđivanje, a u ostalim slučajevima označava da moraju biti zadovoljena sva navedena ograničenja. Sledeća klasa ilustruje višestruko nasleđivanje.

```
class (Real a, Enum a) => Integral a where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    even, odd           :: a -> Bool
    toInteger           :: a -> Integer
    toInt               :: a -> Int
```

Kasnije su u Haskell iz Gofer-a ([57]) preuzete *konstruktorske klase*. Tako klasa sada može biti ne samo kolekcija tipova, već i kolekcija konstruktora tipova.

Primer Tipičan primer upotrebe konstruktorskih klasa u Haskell-u je definisanja pojma funktora. Promenljiva `f` u narednoj deklaraciji ima vrstu (kind) `* -> *`, što znači da preslikava jedan tip u drugi.

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Klasa `Functor` definiše funktor kategorije tipova. Naime, `f` predstavlja deo funktora koji preslikava objekte, a `fmap` preslikava morfizme među tipovima (funkcije). Polimorfne strukture podataka kao što su (homogene) liste ili stabla se mogu deklarisati kao instance klase `Functor`, pa se mogu tretirati na uniforman način.

Za neke klase, kao što su `Eq` i `Show`, kompajler može sam da generiše instance algebarskih tipova podataka. To se postiže tako što se iza definicije algebarskog tipa navede ključna reč `deriving` iza koje sledi ime klase. Radi se na proširivanju ove mogućnosti i na klase kao što je `Functor` (videti predstojeći Haskell Workshop 2000, Montreal, kao i sistem PolyP u [54]). Takođe postoje i `default` deklaracije koje se koriste za razrešavanje dvosmislenosti vezanih za klasu `Num` ([65], odeljak 4.3.4).

Dodatnu pogodnost klase pružaju u kombinaciji sa proširenjem Haskell-a *egzistencijalnim tipovima*. Tako je moguće deklarisati nehomogene strukture podataka koje odgovaraju strukturama u klasičnim objektno-orijentisanim jezicima.

Primer Sledeći fragment definiše heterogenu kolekciju objekata sa signaturom klase `Polygon`.

```
class Polygon a where
  area :: a -> Int
data AnyPolygon = D (forall a . Polygon a => a)
type Drawing = [AnyPolygon]
```

Intuitivno, tip `forall a . Polygon a => a` predstavlja uniju svih tipova koji su instanca klase `Polygon a`.

Dok u Javi odnose nasleđivanja definišemo nad tipovima i možemo deklarisati instance promenljivih, u Haskell-u se hijerarhija nasleđivanja definiše nad kolekcijama tipova (i konstruktora tipova), zatim se sofisticiranim `instance` deklaracijama određuje pripadnost tipova klasama, a `forall` konstrukcijom se po potrebi mogu dobiti tipovi čiji je skup vrednosti unija vrednosti svih tipova klase. Jasno je da na ovaj način možemo postići isti efekat

kao i u objektno-orijentisanim jezicima, a može se naslutiti i da je preciznost specifikacije tipova u ovakvom sistemu veća.

Značajnu generalizaciju sistema klasa predstavljaju *multiparametarske klase*. One nisu deo Haskell98 standarda, ali su podržane u tekućim Hugs i GHC (Glasgow Haskell Compiler) implementacijama. Multiparametarska klasa je klasa koja ima više parametara. Ako su klase bile kolekcije tipova i konstruktora tipova, tada su multiparametarske klase relacije nad tipovima i konstruktorima tipova. Ovo je veoma moćan mehanizam koji ćemo intenzivno koristiti u nastavku. Ovde ilustrujemo nove mogućnosti koje pružaju multiparametarske klase u specifikaciji apstraktnih tipova podataka. Tako apstraktni tip parametrizovanog skupa možemo zadati sa

```
class Set s a where
  emptySet :: s a
  isEmptyS :: s a -> Bool
  addElem  :: a -> s a -> s a
  member   :: a -> s a -> Bool
```

Pri tome je `s` konstruktor vrste `* -> *`, dok je `a` tip. Apstrakni tip liste možemo definisati ovako

```
class List l a where
  emptyList :: l a
  cons      :: a -> l a -> l a
  isEmptyL  :: l a -> Bool
  listHead  :: l a -> a
  listTail  :: l a -> l a
```

Konačno, možemo deklaracijom instance iskazati činjenicu da svaka lista nad elementima nad kojim je definisana jednakost može da se iskoristi za implementaciju steka.

```
instance (Eq a, List l a) => Set l a where
  emptySet = emptyList
  isEmptyS = isEmptyL
  addElem  = cons
  member e xs | isEmptyL xs     = False
  member e xs | listHead xs==e  = True
              | otherwise       = member e (listTail xs)
```

Jezici u kojima je moguće zadati ovako opšte odnose između apstraktnih tipova podataka su vrlo retki. Dodatni primeri upotrebe klasa za specifikaciju apstraktnih tipova kolekcija objekata mogu se naći u [92]. Još više mogućnosti za zadavanje ovakvih specifikacija pruža proširenje Hugs implementacije *funkcionalnim zavisnostima* između parametara multiparametarskih klasa ([58]).

Klase kao deo sistema tipova u funkcionalnim jezicima su još uvek u eksperimentalnoj fazi. Postoje problemi vezani za odlučivost i efikasnost postupka izvođenja tipova u prisustvu klasa. Posebni problemi se javljaju kod multiparametarskih klasa. Može se reći da sintaksna

forma klase sa više parametara omogućava da se specificira mnogo više nego što se realno može očekivati da kompajler analizira. Takođe se u opštem slučaju pri primeni klasa javlja dodatna neefikasnost u toku izvršavanja programa. Analizom programa u toku kompajliranja ovaj problem se može eliminisati u mnogim slučajeva od praktičnog interesa ([59], [96]). Sam sistem tipova nije korektan sa teorijskog stanovišta, pa postoje predlozi da se on ograniči. Za diskusiju mogućnosti daljeg razvoja sistema klasa, videti [61].

2.8 Monade i monad transformeri

Ovde se daje motivacija i definicije pojma monade i monad transformera, uglavnom iz [85], a zatim se razmatra primena monada u funkcionalnom programiranju i pisanju modularnih interpretera.

2.8.1 Monada u kategoriji

Osnovna ideja pri konstrukciji monada je da se za svaki tip a koji može biti rezultat funkcije, definiše tip Ta koji označava izračunavanje funkcija tog tipa. Umesto da funkcije budu morfizmi $a \to b$, sada ih posmatramo kao morfizme $a \to Tb$. Različitim izborom preslikavanja T možemo opisati različite oblike izračunavanja. Tako dolazimo do pojma *Kleisli-jeve trojke*.

Klejslijeva trojka u kategoriji $\mathbf{C}$ je $(T, \eta, ^*)$ gde je T preslikavanje koje objekte kategorije slika u objekte, η je je preslikavanje koje svakom objektu A pridružuje morfizam $\eta_A : A \to TA$, i * je preslikavanje koje svakom morfizmu $f : A \to TB$ pridružuje morfizam $f^* : TA \to TB$, tako da važe sledeći zakoni:

1. $\eta_A^* = \mathrm{id}_{TA}$
2. $f^* \circ \eta_A = f$
3. $g^* \circ f^* = (g^* \circ f)^*$

Ovi zakoni govore upravo da morfizmi $f : A \to TB$ čine kategoriju. To je *Kleisli-jeva kategorija* $\mathbf{C}_T$. Objekti ove kategorije su objekti polazne kategorije $\mathbf{C}$, a morfizmi iz A u B su morfizmi $f : A \to TB$ polazne kategorije. Identički morfizmi su η_A za sve objekte A, dok se kompozicija morfizama $f : A \to TB$ i $g : B \to TD$ definiše kao $g^* \circ f$.

Monada u kategoriji $\mathbf{C}$ je trojka (T, η, μ) gde je $T : \mathbf{C} \to \mathbf{C}$ endofunktor, $\eta : \mathrm{id}_C \dot{\to} T$ prirodna transformacija identičkog funktora u funktor T, i $\mu : T \circ T \dot{\to} T$ prirodna transformacija $T \circ T$ u T, tako da važi

1. $\mu \circ \eta T = \mathrm{id}_T = \mu \circ T\eta$
2. $\mu \circ T\mu = \mu \circ \mu T$

Pri tome je ηT oznaka za preslikavanje koje objektu A pridružuje η_{TA}, dok $T\eta$ objektu A pridružuje $T\eta_A$. Oznaka id_T predstavlja identičnu prirodnu transformaciju funktora T.

Postoji bijekcija između Kleisli-jevih trojki i monada ([85], [125]). Neka je, na primer, data Kleisli-jeva trojka $(T, \eta, ^*)$. Tada η već predstavlja prirodnu transoformaciju, T se može dopuniti da preslikava morfizme ako se za $f : A \to B$ definiše $T(f) = (\eta_B \circ f)^*$, a μ se dobija sa $\mu_A = \mathrm{id}_{TA}^*$.

2.8.2 Monade u funkcionalnom programiranju

Dok je Moggi ukazao na značaj monada za struktuiranje denotacione semantike, Wadler je u [124], [125] i mnogim drugim popularnim člancima demonstrirao kako se ovaj koncept može iskorititi za struktuiranje funkcionalnih programa. Monade su potom korišćene kao teorijska osnova za ulaz i izlaz u programskom jeziku Haskell, kao teorijska osnova za implementaciju nizova, referenci i kao način za proširenja jezika ([123]).

Strukturu monada je moguće izraziti u Haskell-u korišćenjem konstruktorskih klasa na sledeći način.

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

Ovako se monade standardno definišu u Haskell-u. Ova formalizacija odgovara Kleisli-jevoj trojci. `m` je preslikavanje T, `return` predstavlja preslikavanje η, dok `>>=` odgovara operaciji *. Naime, ako prvi i drugi parametar u `>>=` zamene mesta, uzimajući u obzir da su sve funkcije Curry-jeve, dobijamo tip `(a -> m b) -> (m a -> m b)`. Između * i `>>=` važi relacija $f^*m = m$`>>=`f. Direktno iz zakona za Kleisli-jevu trojku dobijamo zakone za monade izražene u notaciji Haskell-a:

```
m >>= return = m
(return x) >>= f = f x
m >>= (\a -> (f a >>= g)) = (m >>= f) >>= g
```

Postoje različite notacije za monade. Wadler je u [124] uočio da struktura monada odgovara strukturi ZF-izraza u Haskell-u (list comprehension). Ovakva semantika za ZF-izraze bazirana na monadama je potom uvedena u Haskell, ali je u poslednjoj verziji Haskell98 vraćeno osnovno jednostavno značenje ZF-izraza, a za monade je uvedena tzv. `do`-notacija. Ova notacija omogućava skraćeni zapis kompozicije funkcija zadatih lambda apstrakcijama. Rezultujući programi podsećaju na imperativne, ali imaju jasnu denotacionu semantiku. Tako je, na primer, moguće definisati monadu koja implementira pojam stanja, a zatim pisati funkcije sledećeg oblika.

```
prog = do {x <- fetch;
           store (x*2);
           y <- fetch;
           store (x+y);
           res <- fetch;
           return res}
```

Prethodni `do` izraz je ekvivalentan sa

```
fetch       >>= (\x ->
store (x*2) >>= (\dummy1 ->
fetch       >>= (\y ->
store (x+y) >>= (\dummy2 ->
```

```
fetch       >>= (\res ->
return res
)))))
```

gde `\x` označava apstrakciju po promenljivoj x. `do`-notacija predstavlja samo skraćenicu (sintaksni šećer) za kompoziciju odgovarajućih funkcija. Tačna pravila za njegovu eliminaciju su data u [65], ali je njihova suština data vezom

```
do {x <- m; e} = m >>= (\x -> e)
```

Zakoni za monade u ovoj notaciji postaju

```
do {x <- m; return x} = m
do {y <- return x; f y} = f y
do {a <- m; do {b <- f a; g b}} = do {b <- do {a <- m; f a}; g b}
```

Oba izraza u poslednjem zakonu zapisujemo u obliku

```
do a <- m
   b <- f a
   g b
```

čineći tako primenu tog zakona transparentnom.

Sve pomenute notacije bazirane su na Kleisli-jevoj trojci. Pristup baziran na definiciji monade u kategoriji koristi polimorfne funkcije

```
map    :: (x -> y) -> (M x -> M y)
return :: x -> M x
join   :: M (M x) -> M x
```

gde `map` je deo funktora koji preslikava morfizme, `return` je prirodna transformacija η, a `join` je prirodna transformacija μ. Naravno, i u ovom slučaju dobijamo odgovarajuću formu zakona za monade ([124], [125]).

Monade pružaju mogućnost za opisivanje različitih efekata procesa računanja kao što su izuzeci, nedeterminizam, okruženje, stanje, kontinuacije. Tehnike koje se pri tome koriste su bile poznate i ranije iz denotacione semantike. Ono što su monade donele je uniforman način tretiranja tih efekata. Definicije monada za različite efekte računanja date su u odeljku 3.2 koji se odnosi na modularnu semantiku.

Sa programerskog stanovišta monade su apstraktni tip podataka, sa odgovarajućim operacijama i zakonima koje te operacije moraju da zadovoljavaju. Ono što je za monade specifično (i što je možda razlog što se nisu ranije pojavile) je što se suštinski oslanjaju na postojanje funkcija višeg reda ([124]).

Ukoliko ostanemo na osnovnoj definiciji monade u proizvoljnoj kategoriji, sve što možemo definisati jeste kompozicija niza morfizama. Jasno je da to nije dovoljno za opis semantike funkcionalnih jezika. Zato je potrebno proširiti pojam monade na Kartezijanski zatvorene kategorije (CCC). Proširenje na CCC zahteva da za monadu (T, η, μ) postoji i prirodna transformacija $t_{A,B} : A \times TB \to T(A \times B)$ koja zadovoljava određene zakone ([85]). Ovakva monada se zova *jaka monada*, a prirodna transformacija t se naziva *tenzorska snaga* (eng. tensorial

strength). Ova prirodna transformacija daje i egzistenciju morfizma $TA \times TB \to T(A \times B)$, koji intuitivno odgovara računanju vrednosti $A \times B$ tako što se prvo izračuna vrednost tipa A, a zatim vrednosti tipa B. Kada se u kategoriju uvedu i stepeni, moguće je dati prevode lambda računa u monade CCC kategorije ([124]). To praktično znači da se svaki funkcionalni program može napisati kao kompozicija funkcija koje za rezultat daju monade. Zavisno od izbora monade, ovaj program može da označava različita izračunavanja. Zato takav način pisanja programa znatno olakšava kasnije modifikacije. U [125] je ilustrovana pogodnost koju pružaju monade pri postepenom razvoju jednog interpretera. Treba napomenuti da pristup modularnih interpretera koji će biti predstavljen u ovom radu podrazumeva više od pisanja interpretera pomoću jedne monade koja se modifikuje po potrebi.

Pogodnost koju imamo kada radimo u CCC kategoriji funkcionalnog programskog jezika je da tenzorsku snagu `t : (a, m b) -> m (a, b)` možemo konstruisati. Ona odgovara funkciji

```
t (a, mb) = b <- mb
            return (a, b)
```

u Haskell-u.

Jedan od problema sa primenom monada je upravo to da se ceo program mora izraziti u obliku kompozicije funkcija oblika `a -> m b`. To znači da programi imaju linearnu strukturu umesto strukturu opšteg stabla. No upravo zahvaljujući ovoj linearnoj strukturi je moguće pomoću monada formulisati apstraktne tipove podataka koji se mogu implementirati pomoću destruktivnog ažuriranja, kao što su nizovi. Iz istog razloga je moguće monade u čisto funkcionalnim programskim jezicima koristiti za realizaciju ulaza i izlaza i povezivanja sa imperativnim programskim jezicima ([123]). (U pogledu strukture dozvoljenih programa i mogućnosti rada sa ulazom i izlazom čini se da nešto veću fleksibilnost pruža pruža sistem jedinstvenih tipova funkcionalnog jezika Clean ([96]), ali se on ne može koristiti za za uvođenje proizvoljnih efekata procesa računanja od strane korisnika.)

2.8.3 Monad transformeri

Sledeći problem vezan sa monadama je kombinovanje više različitih monada. Pri tome treba istaći da je Moggi uveo monade upravo da bi definisao efekte procesa računanja na takav način, da se različiti efekti mogu međusobno kombinovati sa predvidljivom rezultujućom semantikom. Ideja je da se krene od monade koja opisuje jednostavan proces računanja, da bi se zatim primenom *monad transformera* ova monada proširila dodatnom semantikom. Različiti efekti računanja opisuju se tako kao različiti monad transformeri za koje važe određeni zakoni. Predloženi pristup semantici se sastoji u proučavanju osnovnih osobina koje monad transformeri imaju, kao i zahteva koji moraju biti zadovoljeni kako bi se primenom monad transformera sačuvali zakoni koji su važili na polaznoj monadi. U ovom odeljku se daju osnovne definicije i formulišu problemi, dok se konkretni monad transformeri obrađuju u delu 3.2.

Prema jednoj od definicija, monad transformeri predstavljaju endofunktore u kategoriji **K** čiji su objekti monade neke kategorije **C**. Morfizmi u **K** se nazivaju morfizmi monada (eng. monad morphism). Tzv. *pojednostavljeni morfizam monada* između dve jake monade

(T, η^T, μ^T, t^T) i (S, η^S, μ^S, t^S) je prirodna transformacija $\sigma : T \dot{\to} S$ koja komutira sa prirodnim transformacijama koje čine monadu tj.

1. $\sigma_A \circ \eta^T_A = \eta^S_A$
2. $\mu^S_A \circ (\sigma; \sigma)_A = \sigma_A \circ \mu^T_A$
3. $\sigma_{A\times B} \circ t^T_{A,B} = t^S_{A,B} \circ (\mathrm{id}_A \times \sigma_B)$

Ova definicija je u skladu sa definicijom morfizma između dve monade kao funktora između Kleisli-jevih kategorija koje odgovaraju tim monadama ([26]). Morfizmi monada omogućavaju da dovedemo u vezu dve monade koje su definisane na različite načine. Tako postoji morfizam monade koja čita trenutno stanje (state reader) u monadu koja može da čita i piše trenutno stanje ([125]).

Ovde napominjemo da u opštem slučaju nije moguće izvršiti kompoziciju monada kao kompoziciju njihovih funktora, jer rezultat ne mora biti monada. Drugim rečima, ako za kategoriju $\mathbf{C}$ formiramo Kleisli-jevu kategoriju $\mathbf{C}_T$ za funktor T, a zatim u novodobijenoj kategoriji $\mathbf{C}_T$ formiramo Kleisli-jevu kategoriju $\mathbf{C}_{TS}$ za funktor S, rezultatujuća kategorija sa morfizmima oblika $f : A \to S(TB)$ ne mora biti Kleisli-jeva kategorija u $\mathbf{C}$. Postoji više načina da se pristupi ovom problemu ([26]), a mi se ovde opredeljujemo za pristup pomoću monad transformera.

Monad transformeri od postojeće monade prave novu. Zato oni u najmanju ruku predstavljaju funkcije koje preslikavaju monade u monade. Pri tom je jedno od osnovnih pitanja kada se neko preslikavanje nad monadama može proširiti do funktora. Ova razmatranja dovode do klasifikacije transformera ([26]). Važan primer monad transformera, kontinuacije, po [85] ne može da se proširi do funktora, iako Espinoza predlaže da se svi transformeri posmatraju kao pre-monade (funktori sa jediničnom prirodnom transformacijom) ili čak monade. Zahtev da transformer bude funktor obezbeđuje da se očuva i struktura morfizama između monada. Zahtev za jedinicom znači sledeće. Ako je T monad transformer, onda za njega postoji prirodna transformacija $\eta^T : 1 \dot{\to} T$ između jediničnog endofunktora i endofunktora T. To znači da za svaku monadu M postoji morfizam monada $\eta^T_M : M \to T(M)$ koji omogućava da se monada M posmatra "unutar monade" $T(M)$. To je očekivano svojstvo za monad transformere koji imaju ulogu da dodaju neku novu mogućnost računanja. Možemo ići i toliko daleko da od transformera T zahtevamo da i sam predstavlja monadu u kategoriji monada sa morfizmima monada. U tom slučaju imamo garantovani morfizam monada iz $T(T(M))$ u $T(M)$, što znači da je moguće izvesnom smislu eliminisati višestruku primenu monad transformera. Ovaj uslov nije ispunjen za transformer stanja ([26]). Za nas će **monad transformer** biti par (T, lift^T), gde je T preslikavanje koje monadi M pridružuje monadu $T(M)$, a $\mathrm{lift}^M : M \dot{\to} T(M)$ je prirodna transformacija koja je morfizam monade M u monadu $T(M)$.

Smisao transformera je da monada dobijena transformacijom omogućava i dodatne operacije. Tako, na primer, monad transformer `StateT` (opisan u odeljku 3.2.1) dodaje operaciju `update` koja ažurira stanje. Kada jednom imamo monadu M koja podržava operaciju `update` i na nju primenimo neki drugi transformer T, postavlja se pitanje kako definisati `update` nad novodobijenom monadom TM. Analogna pitanja se javljaju i za ostale operacije. To je suštinsko pitanje celog pristupa: mesto gde se manifestuje modularnost semantike

(ili njeno odsustvo) i mesto gde se opisuje interakcija između različitih efekata računanja. Mi ćemo se ovde opredeliti za postupak zasnovan na *liftingu*. Lifting je uveo Moggi, a prisutan je i u radu Liang-a i Hudak-a ([76], [75]). Espinoza ([26]) ga takođe obrađuje, ali daje prednost tzv. stratifikaciji. Moggi je posmatrao monade unutar teorije kategorija i zahtevi koje je on postavio za lifting su prilično strogi, jer odgovaraju onim slučajevima kada su efekti opisani monadama međusobno nezavisni. Liang i Hudak razmatraju i netrivijalne slučajeve liftinga, kao što su `calcc` i `inEnv`. Ovi rezultati su primenjeni u 3.2 za izgradnju semantičke komponente našeg modularnog interpretera.

2.8.4 Strelice

Ovde je objašnjen pojam stelica (eng. arrows) koji je značajan kao moguća generalizacija monada. U isto vreme ovaj koncept pojašnjava vezu između teorije kategorija i Haskell-a. Hughes je u [51] predložio strelice kao generalizaciju monada, navodeći primere kombinatora za parsiranje koje je razvio Swierstra ([108]), gde su strelice primenljive, dok monade nisu. Pošto se sličan pristup statičke analize koji je primenio Swierstra može primeniti i u širem kontekstu, može se očekivati da će i strelice biti od značaja za funkcionalno programiranje. Strelice se jednostavno mogu posmatrati kao pokušaj definisanja pojma kategorije kao apstraktog tipa podataka. Ovo je moguće uraditi u Haskell-u pomoću sledeće definicije klase `Arrow`.

```
class Arrow a where
  arr   :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
```

Vidimo da se kategorija reprezentuje pomoću binfunktora tj. konstruktora tipa `a` sa dva parametra. Instance klase `Arrow` su objekti (tipovi) koji reprezentuju morfizme unutar neke kategorije formirane unutar CCC tipova Haskell-a. Pošto smo u prilici da definišemo proizvoljne kategorije, moguće je izvesti i konstrukciju Kleisli-jeve kategorije, na sledeći način.

```
newtype Kleisli m a b = K (a -> m b)
instance Monad m => Arrow (Kleisli m) where
  arr f = K (\b -> return f b)
  K f >>> K g = K (\b -> (f b >>= g))
```

Ovo znači da se i unutar Haskell-a monade mogu posmatrati kao uopštenje strelica. Prethodna deklaracija instance predstavlja formalni zapis konstrukcije Kleisli-jeve kategorije. Mogućnost da tako opšte konstrukcije zadamo u programskom jeziku Haskell čini ga moćnim sredstvom za istraživanje konstruktivnih aspekata teorije programskih jezika.

Klasa `Arrow` opisuje neku opštu kategoriju, ali ono što nama treba za rad je kategorija sa dodatnom strukturom. Hughes opisuje ovu dodatnu strukturu i daje zakone koje ona treba da zadovoljava. Operatori koje pri tom dodajemo signaturi `Arrow` odgovaraju proizvodu i sumi u kategoriji, ali zakoni ne zahtevaju da važe sve osobine koje ti operatori u kategoriji imaju. Rezultujući sistem je slabiji, da bi se postigla veća opštost apstraktnog tipa podataka.

Dalje je objašnjena generalizacija monad transformera, za koje se ispostavlja da odgovaraju funktorima u kategoriji. Konačno se zaključuje da dodavanjem stepenovanja dobijamo

apstraktni tip koji može da podrži monade. Značaj strelica je u tome što predlažu da se i kategorije bez stepenovanja ali sa proizvodima i sumama mogu koristiti kao inspiracija za upotrebljiv apstraktni tip podataka.

Možda još ambiciozniji pokušaj uvođenja teorije kategorija u funkcionalni jezik je Categorical Prelude iz [24], baziran na jeziku Gofer koji je blizak Haskell-u.

2.9 Parsiranje

U ovom delu se daju osnovni pojmovi parsiranja uopšte, a zatim se obrađuju specifičnosti ovog postupka u kontekstu funkcionalnog programiranja. Opširnije o teorijskim aspektima parsiranja se može naći npr. u [74] i [79], a primena u kompajliranju se razmatra npr. u [2], [114], [4].

Posmatramo konačan skup koji nazivamo *azbuka*. Azbuku u slučaju programskih jezika mogu činiti znakovi koji čine program, ili tokeni ukoliko je već izvršena leksička analiza. Reči su konačni nizovi slova, a svaki skup reči zovemo *jezik*. Programski jezici su po pravilu beskonačni, pa se postavlja problem njihovog opisivanja. Za tu svrhu su se kao najpogodnije pokazale *generativne gramatike*. Gramatike su konačni skupovi pravila (produkcija) čijom se primenom, polazeći od nekog fiksiranog simbola, izvode reči jezika. Najčešće se koriste kontekstno slobodne gramatike, koje predstavljaju jednu od klasa u hijerarhiji gramatika koje je uveo Noam Chomsky u pokušaju da opiše gramatiku prirodnih jezika. Kontekstno slobodna gramatika kojom opisujemo jezike nad azbukom T je uređena četvorka $G = (S, V, T, P)$ gde je V je skup pomoćnih simbola (neterminala), $S \in V$ početni neterminalni simbol, a P je relacija između skupa V i skupa reči nad azbukom $V \cup T$. Elemente skupa T zovemo terminali. Parove $(X, \alpha) \in P$ zapisujemo u obliku

$$X \to \alpha.$$

Ovo pravilo označava da se neterminal X može zameniti rečju α. Može postojati više pravila koja sa leve strane imaju X. Skup reči koje sadrže samo simbole iz T, a koje možemo dobiti primenom pravila iz P polazeći od neterminala S čine *jezik određen gramatikom* G.

Opšte je prihvaćena praksa da se sintaksa programskog jezika opiše pomoću kontekstno slobodne gramatike. Prilikom pisanja kompajlera ili interpretera za taj jezik postavlja se pitanje kako proveriti da li je program sintaksno ispravan, što se svodi na pitanje da li data reč pripada jeziku određenom datom gramatikom. Ovaj postupak se naziva *parsiranje*. Njegov značaj je ne samo u proveri ispravnosti programa, već i u tome što se njime program raščlanjuje na sastavne delove, a to je od značaja za kasniju obradu programa. Iako postoje algoritmi za parsiranje kontekstno slobodnih gramatika u polinomijalnom vremenu, oni nisu dovoljno praktični za primenu u kompajlerima. Stoga se za opis jezika koriste specijalne klase gramatika za koje postoje efikasni algoritmi parsiranja linearne složenosti. Dve najčešće korišćene tehnike za efikasno parsiranje su LL(1) i LALR(1) parsiranje.

LL(1) parsiranje se ističe sličnošću između strukture programa i strukture LL(1) kontekstno slobodne gramatike. Za opis kontekstno slobodne gramatike se tada najčešće koristi EBNF notacija, u kojoj su jednostavni oblici rekurzije zamenjeni zagradama za označavanje iteracije i opcionih delova izraza. U postupku *rekurzivnog spusta* ove konstrukcije EBNF notacije se mogu direktno prevesti u odgovarajuće WHILE i IF naredbe. Svakom neterminalu

se može pridružiti procedura koja parsira taj neterminal. Rekurzija u gramatici se parsira primenom rekurzije među procedurama. To je top-down postupak koji započinje pozivom procedure za parsiranje početnog neterminalnog simbola. Izbor između različitih alternativa u LL(1) postupku parsiranja se vrši na osnovu sledećeg simbola koji sledi u ulaznom nizu. Ograničenja koja se postavljaju na LL(1) gramatike upravo odgovaraju zahtevu da je na osnovu jednog simbola moguće odrediti koju alternativu (produkciju sa istom levom stranom) treba odabrati. Ovaj postupak se može realizovati i pomoću tabele sa prelazima koja daje push-down automata za parsiranje date gramatike, što rezultuje u manje čitljivom programu za parsiranje, ali pruža veću mogućnost za obradu grešaka.

LALR(1) parsiranje je nešto ograničeni oblik LR(1) parsiranja. LR(1) parsiranje parsira pravi nadskup gramatika koje je moguće parsirati LL(1) postupkom. To je bottom-up postupak u kojem se odluka o tome koja se produkcija primenjuje donosi tek pošto su pročitani svi simboli koji čine tu produkciju. Problem sa LR(1) postupkom je u velikoj tabeli koja se koristi za opis prelaza. Zato se obično koristi LALR(1) postupak koji ima znatno manji broj stanja, i parsira nešto užu klasu gramatika. Ovi postupci nisu pogodni za ručnu implementaciju već se najčešće realizuju pomoću generatora parsera.

Danas se parseri za kompajlere i interpretere najčešće prave pomoću generatora parsera ([56], [89]). Ovi alati postoje i za funkcionalne programske jezike, ali u ovim jezicim znatnu popularnost uživa i parsiranje pomoću kombinatora ([53], [64], [108], [109], [51]). U parsiranju pomoću kombinatora se, korišćenjem struktura podataka koje sadrže funkcije, realizuje apstraktni tip podataka `Parser`. Operacije apstraktnog tipa `Parser` odgovaraju operacijama nadovezivanja i alternative kojima se definišu i gramatike. Rekurzija između neterminalnih simbola se svodi na rekurziju između definicija funkcija. Zahvaljujući korisničkim definicijama prioriteta moguće je praktično pretvoriti definicije gramatika u definicije funkcija za parsiranja. Sličnost između gramatika i postupka parsiranja je ovde još veća nego pri rekurzivnom spustu u imperativnim programskim jezicima, kao što pokazuje sledeći primer. Fragment gramatike

```
Statement = AssignmentStat | StatSequence.
AssignmentStat = "LET" Variable ":=" Expression.
StatSequence = "BEGIN" Statement {";" Statement} "END".
```

se pomoću kombinatora iz [108] zapisuje na sledeći način.

```
stat = assStat <|> statSeq
assStat = mkAssign <$> pSym "LET" <*> var <*> pSym ":=" <*> expr
statSeq = mkSeq    <$> pSym "BEGIN" <*> pList stat (pSym ";") <*> pSym "END"
```

Kombinatori su članovi klase `Parser`.

```
class Parser p where
  pEmpty :: a -> p s a
  pSym   :: Eq s => s -> p s s
  (<|>)  :: p s a      -> p s a -> p s a
  (<*>)  :: p s (b->a) -> p s b -> p s a
```

```
  (<$>)  :: (b -> a) -> p s b -> p s a
  (<$>) f p = pEmpty f <*> p
```

Konstruktor `p` vrste `* -> * -> *` je član klase `Parser` ako su za njega definisane navedene funkcije. Kombinator `<|>` označava alternativu, a `<*>` sekvencu. `pEmpty` je parser koji vraća datu vrednost ne uzimajući ni jedan znak sa ulaza, dok `pSym` parsira navedeni token. Semantičke akcije (kao što je funkcija `mkAssign` iz prethodnog primera) se mogu zadati pomoću `<$>` koji predstavlja skraćenicu za kombinaciju kombinatora `pEmpty` i `<*>`.

Dok je ovde prisutna samo velika sličnost između gramatike i programa za parsiranje, u generatorima kompajlera je moguće direktno specificirati gramatiku. Ono što je pomalo problematično u primeni kompajler-kompajlera je specifikacija semantike. Ona se obično izražava u ciljnom jeziku, a rezultat prevođenja programa je definisan sintaksnim nadovezivanjem delova definisanih od strane korisnika i delova koji se generišu od strane kompajler-kompajlera. Nasuprot tome, primenom kombinatora ostajemo u funkcionalnom programskom jeziku, što rezultuje u predvidljivosti i sigurnosti koju pruža provera tipova. Osim toga, parseri su građani prvog reda, pa ih možemo prosleđivati drugim funkcijama i definisati svoje vlastite kombinatore. Tako se mogu definisati kombinatori `pList` i `pMany`.

```
pMany :: Parser p => p s a -> p s [a]
pMany p = pm where pm = (:) <$> p <*> pm <|> pEmpty []
pList :: Parser p => p s a -> p s b -> p s [a]
pList item sep = (:) <$> item <*> (pMany (k2 <$> sep <*> item))
               where k2 x y = y
```

U svom najjednostavnijem slučaju kombinatori za parsiranje se mogu realizovati primenom bektreka. (To odgovara realizaciji parsera za vrednost `a` kao monade koja se dobija od monade za nedeterminizam primenom transformera stanja, pri čemu je stanje niz preostalih ulaznih znakova parsera.)

```
newtype BParser s a = P {unP :: [s] -> [(a, [s])]}
instance Parser BParser where
  pEmpty x = P (\s -> [(x,s)])
  pSym sym = P (\s -> case s of
                        []      -> []
                        (hs:ts) -> if hs==sym then [(sym,ts)]
                                              else [])
  (<|>) (P p) (P q) = P (\s -> p s ++ q s)
  (<*>) (P p) (P q) = P (\s -> [(f x, s2) | (f,s1) <- p s, (x,s2) <- q s1])
```

Swierstra ([108]) pokazuje kako je moguće poboljšati efikasnost ove naivne implementacije LL(1) parsiranja računanjem $first$ i $follow$ simbola, a u [109] objašnjava i implementaciju LR(1) parsera zasnovanog na istim principima. Hughes ([51]) objašnjava zašto je ove efikasnije verzije parsere nemoguće posmatrati kao monade, i to koristi kao motivaciju za uvođenje strelica koje smo pomenuli u odeljku 2.8.4.

Pored opisanih postupaka, koristi se i tehnika parsiranja pomoću prioriteta ([114], [2]). Ova tehnika se može shvatiti kao postupak za eliminisanje shift-reduce konflikta u bottom-up parseru. Varijaciju jednog takvog postupka kao i razlog njegove primene u modularnim interpreterima dajemo u odeljku 3.4.

Rezultat procesa parsiranja je apstraktno sintaksno stablo (eng. abstract syntax tree, AST). AST ima istu rekurzivnu strukturu kao i gramatika koja se parsira, ali sadrži samo one elemente koji su bitni za značenje programa. Zato AST ne sadrži interpunkcijske znake i ključne reči čija je svrha jednoznačno parsiranje programa. Tipičan primer predstavljaju zagrade u izrazima. One služe da bi se na jedinstven način mogao rekustruisati izraz kao jedna rekurzivno definisana struktura. U AST zagrade nisu potrebne jer je rekurzivna struktura direktno izražena pomoću strukture stabla u memoriji. Opis strukture apstraktnog sintaksnog stabla se naziva *apstraktna sintaksa* i ona se koristi kada nam je bitna samo struktura programa, a ne i način na koji se on zapisuje pomoću niza simbola (*konkretna sintaksa*). Zavisno od načina implementacije, AST se ne mora eksplicitno kreirati u toku parsiranja, ali njegovo korišćenje povećava modularnost jezičkog procesora, jer jasno definiše izlaz faze sintaksne analize. Pored toga, postojanje sintaksnog stabla daje veću slobodu u implementaciji ostalih delova jezičkog procesora.

3 Modularni interpreteri

Ovde je opisana realizacija modularnog interpretera. Mnogi od pojmova teorije kategorija opisani u uvodu ovde se pojavljuju u kontekstu Haskell-a. Korišćenjem njegove notacije definicije i konstrukcije postaju ne samo neposredno primenljive, već često i razumljivije. Pri tome konstruktorske klase Haskell-a značajno povećavaju broj koncepata koje možemo izraziti ([24]).

3.1 Ideja i realizacija modularnih interpretera

Modularni interpreteri predstavljaju pokušaj primene ideja koje je 1990. godine razvio Moggi ([85]) za struktuiranja denotacione semantike. Moggi koristi aparat teorije kategorija, podižući na viši nivo apstrakcije rezonovanje sa konstrukcijama koje su razvijene u kontekstu denotacione semantike ([106]), da bi zatim predložio način da se različite osobine programskih jezika proučavaju nezavisno. Ovaj pristup je prihvaćen i kao metod za razvoj modularnih interpretera i kompajlera u programskom jeziku Haskell u [26] i [76]. U [127] je opisan način za povezivanje ovog pristupa sa akcionom semantikom. Duponcheel ([23]) je pokazao kako je moguće modularnost primeniti i na apstraktnu sintaksu korišćenjem tzv. katamorfizama (`fold` operacija nad stablima) kojima se definiše algebra čija je signatura data apstraktnim sintaksnim stablom. Ovde se razmatraju i mogućnosti za modularno zadavanje konkretne sintakse i leksike, što, po svemu sudeći, nije privuklo veliku pažnju u ranijim radovima.

Rezultat je interpreter koji je modularan na dva načina. S jedne strane, leksička analiza, sintaksna analiza i semantička analiza se zadaju nezavisno jedna od druge, što je široko prihvaćen način za struktuiranje kompajlera i interpretera. Sa druge strane, jezik je sastavljen od komponenti od kojih svaka opisuje jednu osobinu (eng. feature) jezika kao što su rad sa memorijom, definisanje funkcija, upravljanje greškama ili nedeterminizam. Svaka od ovih komponenti specificira svoj deo leksičke, sintaksne analize i semantičke analize, nezavisno od ostalih.

Primeri komponenti jednostavnog interpretera su dati u dodatku. Ovakav opis možemo tretirati kao specifikaciju jedne osobine jezika, ali je bitno naglasiti da to predstavlja običan modul u verziji programskog jezika Haskell koja podržava multiparametarske klase. Fleksibilnost i visoki nivo programskog jezika Haskell omogućili su nam da izbegnemo korišćenje generatora koda kao što su kompajler-kompajleri u upotrebi ([89], [56]), već da ceo proces sklapanja komponenti opišemo u Haskell-u. Upotrebom *kombinatora* kao funkcija višeg reda realizovanih u vidu korisnički definisanih operatora, Haskell se pretvara u jezik za specifikaciju komponenti interpretera. Prednost Haskell-a posebno dolazi do izražaja u opisu semantike: pošto ima ne-striktnu semantiku, moguće je direktno pisati matematičke definicije koje je razvila denotaciona semantika ([106]), a pošto podržava algebarske tipove, možemo direktno raditi sa domenima denotacione semantike. (Ova adekvatnost Haskell-a nije toliko neočekivana, s obzirom da su na nastanak jezika kao što je Haskell uticale ideje denotacione semantike.)

Za ovu realizaciju od ključnog značaja su apstraktni tipovi podataka. Osnovni apstraktni tip za realizaciju semantike je monada. Semantika svake od osobina jezika realizovana je kao podklasa klase `Monad` koja dodaje neke nove operacije. Sklapanje semantičkih komponenti

jezika je realizovano formalizacijom pojma monad transformera unutar sistema konstruktorskih klasa Haskell-a. Definisanje proširivih tipova je bilo značajno za realizaciju domena računanja u interpretaciji, kao i za realizaciju sintaksne i leksičke analize. Prošrivi tipovi su realizovani pomoću multiparametarskih klasa, kao što je opisano u 3.3.2. Sintaksna analiza se vrši pomoću apstraknog parsera koji prihvata specifikaciju operacija, njihovog prioriteta i arnosti i kreira odgovarajuće sintaksno stablo. Leksička analiza zadaje se pomoću regularnih izraza čiji apstraktni tip skriva prevođenje u konačni automat.

3.2 Modularna denotaciona semantika

Ovde dajemo opis realizacije modularne denotacione semantike u Haskell-u. Prikazujemo način implementacije monada i monad transformera, a zatim opisujemo osnovne semantičke komponente i njihovu interakciju sa ostalim komponentama.

Polazna i osnovna kategorija sa kojom radimo je CCC kategorija tipova Haskell-a, sa `()` kao terminalnim objektom, `(,)` kao označenim proizvodom i `->` kao označenim stepenom. Da bismo neki konstruktor tipa proglasili za funktor, deklarišemo ga kao instancu klase `Functor`. Tako je, na primer, konstruktor listi funktor.

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
instance Functor [ ] where
  fmap = map
```

Jezgro modularne semantike čini pojam monade, date sledećom konstruktorskom klasom.

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

Monad transformer je konstruktor `t` vrste `(* -> *) -> (* -> *)` koji od monade pravi monadu, a ima i prirodnu transformaciju lift koja predstavlja morfizam iz monade `m` u monadu `t m`.

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
```

Zahtev da `lift` predstavlja ne samo prirodnu transformaciju već i morfizam monada ne možemo izraziti u Haskell-u, već moramo proveriti sami kada pišemo odgovarajuće deklaracije instanci. `MonadT` je multiparametarska klasa, jer promenljiva `m`, koja se javila sa leve strane, mora da se pojavi i sa desne strane deklaracije klase. Ono što bismo želeli da uradimo je da univerzalno kvantifikujemo `m` sa leve strane "implikacije", ali sistem tipova to ne omogućava (razlozi za to su dati u [61]).

Struktura monade u kategoriji tipova Haskell-a se može posmatrati kao generalizacija pojma aplikacije funkcija. To izražavamo definicijom monade `IdMon`.

```
newtype IdMon x = Id { unId :: x }
instance Monad IdMon where
  return = Id
  (Id x) >>= f = f x
```

Efektivno je `IdMon x = x`, ali da bismo mogli da deklarišemo novu instancu za `x`, moramo ovu vrednost da označimo, za šta koristimo konstruktor `Id`. (Ukoliko se ovakva ograničenja ne bi postavljala za deklaracije instanci izvođenje tipova bi u opštem slučaju postalo neodlučivo, videti [61]).

Sledeća bitna monada je lista. Ona predstavlja sistematski način za primenu tehnike liste uspeha (odeljak 2.4). Pri tome `ndfail` označava neuspešan pokušaj računanja, dok `ndmerge` listu računanja spaja u jedno računanje.[3]

```
class Monad m => NondetMonad m where
  ndfail  :: m a
  ndmerge :: [m a] -> m a
instance NondetMonad List where
  ndfail  = [ ]
  ndmerge = concat
```

3.2.1 Monad transformeri

Sada ćemo preći na opise monad transformera kojima se zadaju različiti efekti računanja. Opšti oblik ovakve specifikacije sastoji se od sledećih deklaracija u Haskell-u.

1. deklaracija specijalne monade koja uvodi nove operacije i time pruža nove mogućnosti za računanje;

2. deklaracija konstruktora tipa, obično vrste `* -> (* -> *) -> (* -> *)`. Prvi argument konstruktora je parametar čija svrha zavisi od konkretnog transformera, drugi argument je monada koja se transformiše, a rezultat je nova monada;

3. deklarisanje instance klase `Monad` za tip transformisan novodobijenim konstruktorom, čime se zadaje način realizacije novih operacija monade nad transformisanim tipom;

4. deklarisanje instance specijalne monade, čime se pomoću dodatne strukture koju pruža transformisani tip realizuju nove operacije;

5. deklarisanje konstruktora tipa uvedenog u 2. koraku kao instance klase `MonadT`, zadavanjem funkcije lift kojom se vrednost polazne monade preslikava u vrednost nove monade.

[3]Pojam paralelnog spajanja nedeterminističih procesa ovde uzimamo intuitivno. Preciznija razmatranja pokazuju da postoje ozbiljni problemi sa uvođenjem nedeterminizma u funkcionalne jezike i lambda račun [50].

S obzirom da su svi monad transformeri koje navodimo poznati od ranije, nećemo dokazivati da oni preslikavaju monade u monade niti da je prirodna transformacija lift zaista morfizam monada. Ono što ćemo lako primetiti u narednim definicijama je da važi $\text{return}_{t\,m} = \text{return}_m \circ \text{lift}$. To je jedan od uslova za morfizam monade (2.8), koji možemo koristiti i da izvedemo $\text{return}_{t\,m}$ ako znamo lift.

Same definicije monad transformera bazirane su na odgovarajućoj tehnici iz denotacione semantike, što smo ukratko razmotrili u 2.3.3.

Transformer za stanje Ovaj transformer je motivisan predstavljanjem računanje izraza tipa `a` vrednošću `s -> (a,s)`. Uvodi novu operaciju `update` čiji su specijalni slučajevi `fetch` i `store`. Operacija `update` primenjuje zadatu funkciju da bi ažurirala tekuće stanje i vraća to novo stanje kao rezultat računanja. Konstruktor tipa `StateT` je parametrizovan tipom koji predstavlja stanje i za datu monadu vraća novu monadu koja preslikava prethodno stanje u računanje (u osnovnoj monadi `m`) uređenog para vrednosti i novog stanja.

```
class Monad m => StateMonad s m where
  update :: (s -> s) -> m s
  -- derived methods:
  fetch :: m s
  fetch = update id
  store :: s -> m s
  store s = update (\_ -> s)
newtype StateT s m a = ST {unST :: s -> m (a, s)}
instance Monad m => Monad (StateT s m) where
  return a = ST (\s -> return (a,s))
  (ST f) >>= k = ST (\s -> do (a,s') <- f s
                              unST (k a) s'
                    )
instance Monad m => StateMonad s (StateT s m) where
  update upd = ST (\s -> return (s, upd s))
instance Monad m => MonadT (StateT s) m where
  lift m = ST (\s -> do a <- m
                        return (a,s)
              )
```

Kompozicija dva procesa računanja sa stanjem vrši se operacijom `>>=` tako što se izračuna vrednost prvog izraza i stanje promenjeno računanjem tog izraza, a zatim se računanje drugog izraza izvrši u tom promenjenom stanju. Funkcija lift izvršava zadati proces računanja ne menjajući stanje.

Transformer za kontinuacije Ovaj transformer pravi monade klase `ContMonad`, koje podržavaju operaciju `callcc`, razmotrenu u 2.3.3. Definicija za `callcc` u drugoj deklaraciji instance se od definicije koju smo dali u lambda računu razlikuje samo po dodatnoj primeni konstruktora `Cont` i destruktora `unCont`. Ovi konstruktori i destruktori izražavaju

izomorfizam novouvedenog tipa i njegove definicije te se mogu zanemariti pri razmatranju semantike.

```
class Monad m => ContMonad m where
  callcc :: ((a -> m b) -> m a) -> m a
newtype ContT c m a = Cont {unCont :: (a -> m c) -> m c}
instance Monad m => Monad (ContT c m) where
  return a = Cont (\k -> k a)
  (Cont m) >>= f = Cont (\k -> m (\a -> unCont (f a) k))
instance Monad m => ContMonad (ContT c m) where
  callcc f = Cont (\k -> unCont (f (\v -> Cont (\k' -> k v))) k)
instance Monad m => MonadT (ContT c) m where
  lift = Cont . (>>=)
```

Operacija `return` primenjuje tekuću kontinuaciju na zadatu vrednost, dok operacija `>>=` pokreće računanje prve vrednosti sa kontinuacijom koja uključuje računanje druge vrednosti. Pri izvođenju ovih operacija oslanjamo se na smisao pojma kontinuacije, a veliku pomoć nam pruža i sistem tipova. Tako znamo da je u ovom slučaju:

```
  (>>=) : ContT c m a -> (a -> ContT c m b) -> ContT c m b
```

pa rešenje tražimo kao jednostavni lambda izraz željenog tipa. Operacija lift se (do na konstruktor `Cont`) poklapa sa operacijom `>>=` nad polaznom monadom `m`. To rasvetljava neformalni postupak u odeljku 2.3.3 gde smo kontinuacije izveli parcijalnom aplikacijom funkcije seq. Primenom zakona monada i definicije `do` notacije lako se pokazuje da je definicija za lift ekvivalentna sa

```
lift m = Cont (\k -> do x <- m
                        return k x)
```

Transformer za okruženje Ovaj transformer ima slične mogućnosti kao i transformer za stanje, ali je njegova namena da podrži okruženje u toku interpretacije ili kompajliranja (tabelu simbola), a ne stanje u programu koji se izvršava. Monada `EnvMonad` podržava operaciju `rdEnv` koja vraća trenutnu vrednost okruženja i operaciju `inEnv` koja pokreće računanje u novom okruženju.

```
class Monad m => EnvMonad e m where
  rdEnv :: m e
  inEnv :: e -> m a -> m a
newtype EnvT e m a = Env {unEnv :: e -> m a}
instance Monad m => Monad (EnvT e m) where
  return x = Env (\_ -> return x)
  (Env m) >>= k = Env (\e -> do x <- m e
                                let (Env f) = k x in f e)
instance Monad m => EnvMonad e (EnvT e m) where
  rdEnv = Env (\e -> return e)
```

```
  inEnv e (Env m) = Env (\_ -> m e)
instance Monad m => MonadT (EnvT e) m where
  lift m = Env (\_ -> m)
```

Operacija `return` ignoriše okruženje, dok `>>=` pokreće prvo jedan, a potom drugi proces računanja u istom okruženju. Zbog toga ovaj transformer čuva komutativnost ukoliko je ona prisutna u polaznoj monadi. Liang ([76]) daje aksiome koje zadovoljavaju operacije `return`, `>>=`, `inEnv` i `rdEnv` ove monade. Bitna razlika u primeni `inEnv` u ovoj monadi u odnosu na `update` u monadi za stanje je što promena okruženja pomoću `inEnv` ima dejstvo samo u monadi koja predstavlja njen argument, a ne i u monadama koje slede. Zato je `inEnv` pogodna za realizaciju potprograma i lokalnih definicija u programu.

Transformer za greške Ovaj transformer omogućava zadavanje izračunavanja koje uspeva ako nigde ne dođe do greške, ili se prekida prvi put kada se pojavi neka greška. Ne postoji mogućnost da se unutar samog programa obrađuju greške nastale prilikom računanja, kao što je to npr. u implementaciji izuzetaka u programskom jeziku Java.

```
class Monad m => ErrMonad e m where
  eThrow :: e -> m a
data Exc e a = OK a | Raise e
newtype ErrT e m a = Err {unErr :: m (Exc e a)}
instance Monad m => Monad (ErrT e m) where
  return = Err . return . OK
  (Err m) >>= f = Err (do ae <- m
                          case ae of
                            OK a      -> unErr (f a)
                            Raise err -> return (Raise err)
                      )
instance Monad m => ErrMonad e (ErrT e m) where
  eThrow = Err . return . Raise
instance Monad m => MonadT (ErrT e) m where
  lift m = Err (m >>= return . OK)
```

Operacija lift se sprovodi tako što se izračunata vrednost označi sa `OK`. Greška nastala pri računanju prve vrednosti u `>>=` se prenosi bez računanja druge vrednosti. Ukoliko je prva vrednost uspešno izračunata, računa se i druga, koristeći rezultat prve.

3.2.2 Struktuiranje funkcionalnih programa

Wadler ([124], [125]) je u svojem nastojanju da popularizuje primenu monada za struktuiranje funkcionalnih programa praktično zanemario upotrebu monad transformera. To je dovelo do zaključaka da u opštem slučaju ne možemo govoriti o kompoziciji monada. Monad transformeri rešavaju problem kompozicije monada sistematično tretirajući problem njihove interakcije. Tako je, na primer, tip parsera iz odeljka 2.9:

```
newtype BParser s a = P {unP :: [s] -> [(a, [s])]}
```

izomorfan sa tipom

```
type MParser s a = StateT [s] [ ] a
```

tj. primeni transformera stanja na listu kao instancu monade za realizaciju nedeterminizma. U našem interpreteru možemo koristiti monadu dobijenu kompozicijom svih navedenih monada.

```
type Compute = EnvT   EnvTable
               (ContT  Answer
               (StateT State
               (ErrT   Errors
                [ ])))
```

Sve operacije koje su definisane nad nekim transformerom i za koje postoji lifting kroz operacije navedene prethodno u nizu, definisane su i za monadu `Compute`. Na taj način monad transformeri realizuju modularnost. Zanimljivo bi bilo ovaj pristup uporediti sa pokušajem da se isti efekat postigne nasleđivanjem polazne klase za nedeterminizam u nekom hibridnom objektno-orijentisanom jeziku. Pri tome bi svaki korak fizičkom nasleđivanja dodavao realizacije novih operacija. Jasno je da pristup pomoću transformera ima znatnih prednosti. Upotreba konstruktora tipova omogućava da se izvrši mnogo složenija transformacija strukture prilikom dodavanja novih mogućnosti računanja nego što je to slučaj pri nasleđivanju u objektno-orijentisanim jezicima. Dalje, pomoću pojma liftinga je ovde precizno određena propagacija ne samo operacija, već i zakona koji među njima važe. Tako je pojmu proširivosti (extensibility) ovde dat precizan matematički tretman. Na osnovu ovoga sugerišemo da je problem proširivosti mnogo složeniji nego što se ponekad predstavlja i da se ne može na zadovoljavajući način rešiti nekom jednostavnom univerzalnom tehnikom kao što je višestruko nasleđivanje.

Čini se da je osnovni problem koji sprečava širu neposrednu upotrebu monad transformera potreba za multiparametarskim klasama, a to je koncept koji, čini se, još uvek nije potpuno razrađen. Ali čak i ako nismo u mogućnosti da monad transformere u Haskell-u podržimo onako kako bismo želeli, možemo ih koristiti za sistematski dizajn monada koje ćemo koristiti za struktuiranje funkcionalnih programa.

3.2.3 Lifting

Deklaracije monad transformera u prethodnim odeljcima obezbeđuju samo da novodobijeni tip poseduje i dalje strukturu monade i da podržava neke nove operacije. Pomoću funkcije `lift` dobijamo još i mogućnost da vrednost polazne monade pretvorimo u vrednost nove monade. Ako su za polaznu monadu bile definisane još neke dodatne operacije (možda realizovane primenom prethodnih monad transformera), sada se postavlja pitanje kako te operacije definisati nad transformisanom monadom. To je problem *liftinga*. Rešenje koje ovde primenjujemo je bazirano na [75] i [76]. Osim same definicije operacija nad transformisanim tipom, postavlja se i pitanje očuvanja zakona koji za njih važe. Ovde se time nećemo detaljno baviti. U [76] se dokazuje da se osobine za okruženje čuvaju prilikom liftinga kroz

`EnvT`, `StateT` i `ErrT` transformere i za očekivati je da se slično razmatranja može sprovesti ako se i operacije ostalih monada aksiomatizuju.

Neka je T tip operacije f pri čemu T sadrži konstante, parametre tipa, kao i konstruktore tipa: `->` (za funkcije), `(,)` (za parove), i `m` (konstruktor polazne monade). Ako je `t` monad transformer, tada se liftingom operaciji f pridružuje operacija $\mathcal{L}_T f$, u čijem tipu su sve pojave konstruktora monade `m` zamenjene konstruktorom transformisane monade `t m`. U [75], [76] se daje uslov koji treba da zadovoljava $\mathcal{L}$ da bi smo dobijeni lifting zvali *prirodni lifting* (natural lifting).

Lifting operacija `eThrow`, `update` i `rdEnv`, čiji domen ne sadrži monade, može se uraditi jednostavnom kompozicijom sa operacijom lift sa leve strane. Tako za transformer za greške i transformer za stanje koristimo sledeće deklaracije.

```
instance (ErrMonad e m, MonadT t m) => ErrMonad e (t m) where
  eThrow = lift . eThrow
instance (StateMonad s m, MonadT t m) => StateMonad s (t m) where
  update = lift . update
```

Ovim je rešen problem interakcije transformera za greške i transformera za stanje sa svim ostalim monadama. Ono što ostaje je da se razmotri interakcija `inEnv` i `callcc` sa svim ostalim transformerima.

Dajemo deklaracije u Haskell-u za lifting `rdEnv` kroz sve monad transformere. Lifting `rdEnv` kroz `ContT` je isti kao u [75].

```
instance EnvMonad e m => EnvMonad e (EnvT f m) where
  rdEnv = lift rdEnv
  inEnv e (Env m) = Env (\e1 -> inEnv e (m e1))
instance EnvMonad e m => EnvMonad e (StateT s m) where
  rdEnv = lift rdEnv
  inEnv e (ST m) = ST (\s -> inEnv e (m s))
instance EnvMonad e m => EnvMonad e (ErrT er m) where
  rdEnv = lift rdEnv
  inEnv e (Err m) = Err (inEnv e m)
instance EnvMonad e m => EnvMonad e (ContT c m) where
  rdEnv = lift rdEnv
  inEnv e ma = Cont (\k ->
       do old <- rdEnv
          inEnv e (unCont ma (inEnv (old `asTypeOf` e) . k)))
```

Funkcija `asTypeOf` koja se javlja u liftingu za `inEnv` služi da bi tip prvog argumenta ograničila tipom drugog argumenta i tako pomogla interpreteru u proveri tipova. Njena definicija je

```
asTypeOf :: a -> a -> a
asTypeOf x _ = x
```

Lifting operacije `callcc` takođe moramo treitirati za svaki transformer posebno.

```
instance ContMonad m => ContMonad (EnvT e m) where
  callcc f = Env (\e -> callcc (\k -> unEnv
              (f (\a -> Env (\e1 -> k a))) e))
instance ContMonad m => ContMonad (StateT s m) where
  callcc f = ST (\s0 -> callcc (\k -> unST
              (f (\a -> ST (\s1 -> k (a,s1)))) s0))
instance ContMonad m => ContMonad (ErrT er m) where
  callcc f = Err (callcc (\k -> unErr
              (f (\a -> Err (k (OK a))))))
```

Lifting `callcc` kroz `ContT` nam nije poznat, a postavlja se pitanje i smislenosti višestrukog korišćenja transformera za kontinuacije.

Razmotrimo sada ukratko lifting monade za nedeterminizam. U [76] se navodi da lista ne može da se predstavi kao monad transformer jer ne zavodoljava odgovarajuće zakone. (Kao što se ističe u [127] i [26], ovo je posledica toga što bi nedeterminizam u osnovi trebalo zadati *skupom*, a ne *listom* vrednosti.) Zbog toga mi nedeterminizam (kao i [76] i [23]) uvek uvodimo u polaznoj monadi, i posebno zadajemo lifting za ovu monadu. To rezultuje u deklaracijama oblika

```
instance MonadT t List => NondetMonad (t List) where
  ndfail  = lift [ ]
  ndmerge = join . lift

instance (MonadT t1 List,
          MonadT t2 (t1 List))
       => NondetMonad (t2 (t1 List)) where
  ndfail = lift (lift [ ])
  ndmerge = join . lift . lift
...
```

Broj deklaracija ovakve prirode zavisi od broja transformera monada koje želimo da primenimo na polaznu monadu.

3.2.4 Semantički gradivni blokovi

U prethodnim odeljcima smo opisali kako je različite vrste procesa računanja moguće opisati pomoću monad transformera i kako je zahvaljujući liftingu moguće sistematski opisati međusobnu interakciju tih vrsta računanja.

U [75] se predlaže da se ovako izgrađeno jezgro za specifikaciju računanja koristi za izgradnju semantičkih gradivnih blokova za programske jezike. Navode se definicije blokova za aritmetiku, funkcije, reference, lenjo izračunavanje, praćenje toka izvršavanja programa (program tracing), kontinuacije i nedeterminizam. Svaki od tih blokova definiše se pomoću operacija iz nekog od monad transformera, a definicije gradivnih blokova su međusobno nezavisne. Izborom odgovarajućih blokova i izborom transformera koji podržavaju korišćene operacije dobija se kompletan interpreter sa predvidljivom semantikom.

Prednosti modularnosti na nivou semantike su višestruke. Pre svega, proučavanjem odgovarajućih osobina operacija monad transformera, a potom i gradivnih blokova, olakšava se rezonovanje o programima. Omogućava se da se matematička svojstva programa pokazuju na nivou koji je bliži konstrukcijama samog programa. Čak i ako se struktura domena računanja promeni, aksiome gradivnih blokova ostaju da važe, što znatno olakšava ciklus dizajna programskog jezika. Na taj način modularna denotaciona semantika dobija slične prednosti nad direktnom primenom denotacione semantike kao i akciona semantika ([86], [87]). Ono što ostaje kao prednost modularne denotacione semantike je čvrsta matematička osnova koja znatno olakšava aksiomatizaciju gradivnih blokova.

Sledeća prednost leži u povećanoj efikasnosti sistema sistema za generisanje jezičkih procesora na osnovu specifikacija visokog nivoa. Upotreba konstrukcija višeg nivoa apstrakcije umesto čistog lambda računa pruža znatno veće mogućnosti za implementaciju. Pri tome denotaciona semantika i dalje predstavlja kriterijum korektnosti implementacije, ali ne nužno i metodu realizacije, jer se konstrukcije jezika mogu realizovati direktnim prevođenjem operacija gradivnih blokova. Sličan pristup doveo je i do efikasne implementacije ulaza i izlaza u Haskell-u. Tako je monadama koje prepoznaje Haskell kompajler moguće realizovati direktno ažuriranje struktura podataka i interakciju funkcionalnog programa sa okruženjem. Realizacija ovih konstrukija kao monada daje im čisto funkcionalnu semantičku podlogu, a činjenica da su izražene posebnim konstrukcijama u jeziku olakšava posao kako kompajleru, tako i programeru koji može da postigne efekte imperativnog programiranja.

Moglo bi se zaključiti da modularni interpreteri stavljaju denotacionu semantiku u njenu pravu ulogu: ulogu matematičke osnove za apstraktno objašnjenje procesa računanja na najnižem nivou. Ona se može se posmatrati kao baza za izgradnju teorija višeg nivoa i njihovo međusobno povezivanje.

Koristeći ideju Duponcheel-a ([23]), semantičke blokove definišemo kao algebru koja predstavlja interpretaciju komponente apstraktnog sintaksnog stabla. U narednom odeljku objašnjavamo tehniku pomoću koje je ostvarena modularnost apstraktne sintakse, a zatim dajemo odabrane semantičke komponente kao interpretaciju komponenti apstraktnog sintaksnog stabla.

3.3 Modularna apstraktna sintaksa

U ovom odeljku opisujemo kako je moguće na modularan način definisati apstraktna sintaksna stabala da bi se njihovom interpretacijom realizovali semantički gradivni blokovi. Dok modularna semantika daje modularnost u pogledu upotrebe različitih efekata računanja, modularna apstraktna sintaksa će omogućiti da se različiti efekti računanja predstave različitim komponentama apstraktnog sintaksnog stabla, što je pretpostavka za modularnu specifikaciju interpretera. Ovaj problem nije razmatran u radovima [26], [76], gde se za realizaciju interpretera koristi jedno fiksirano sintaksno stablo koje se mora redefinisati prilikom dodavanja novih komponenti. Rešenje modularnosti za sintaksna stabla koje ovde opisujemo preuzeto je iz [23].

3.3.1 Algebre i modularnost

Apstraktno sintaksno stablo izraza (odeljak 2.9) se u Haskell-u može zadati definicijom oblika

```
data Expr = Const Int
          | Add Expr Expr
          | Sub Expr Expr
          | Div Expr Expr
          | If Expr Expr Expr
            ...
          | Seq Expr Expr
```

Primećujemo da je stablo zadato rekurzivno, korišćenjem proizvoda tipova (više polja unutar jedne alternative) i sume tipova (više alternativa). Alternative opisuju različite osobine jezika. Tako alternative `Add`, `Sub` i `Div` pokazuju da je u jeziku moguće koristiti sabiranje, oduzimanje i deljenje, dok `If` označava da postoje uslovni izrazi. Da bismo postigli modularnost sintakse, želimo da nezavisno specificiramo različite alternative ili nizove alternativa u stablu. Kada definicija stabla ne bi bila rekurzivna, mogli bismo to uraditi pomoću koncepta podtipa, ali rekurzivna struktura nam to ne dozvoljava. Zato ćemo rekurzivno stablo definisati u dva koraka:

1. definišemo konstruktor tipova `ExprDef x`
2. definišemo tip `Expr` kao najmanju nepokretnu tačku konstruktora `ExprDef` korišćenjem konstruktora nepokretne tačke `Fix`.

Prethodno stablo tako možemo napisati u obliku

```
data ExprDef x = Const Int
               | Add x x
               | Sub x x
               | Div x x
                 ...
               | If x x x
                 ...
               | Seq x x
type Expr = Fix ExprDef
newtype Fix f = In {out :: f (Fix f)}
```

Vrste pomenutih konstruktora u prethodnoj definiciji su

```
ExprDef :: * -> *
Fix     :: (* -> *) -> *
Expr    :: *
```

Ovaj postupak je analogan eliminaciji rekurzije upotrebom kombinatora Y, i ova analogija nije slučajna jer se i semantički domeni definiču kao najmanje nepokretne tačke preslikavanja ([106]).

Ono što smo ovim postupkom dobili je da sada definiciju konstruktora `ExprDef` možemo izraziti kao sumu nezavisnih tipova.

Za svaki od tih tipova definišemo interpretaciju, a potom interpretacije tih delova kombinujemo. Svaku od alternativa, kao što je `Add x x` ili `Div x x` možemo posmatrati kao definiciju signature operacije. Zato se niz alternativa može posmatrati kao signatura algebre. Sama algebra je data interpretacijom, koja imenu operacije kao što je `Add` pridružuje operaciju nad ogovarajućim domenom. To opravdava definiciju klase `Algebra`.

```
class Functor f => Algebra f a where
  phi :: f a -> a
```

U prethodnoj definiciji funktor `f` predstavlja signaturu algebre, a tip `a` njen domen. Funkcija `phi` istovremeno zadaje interpretaciju svih operacija iz signature algebre. Ovo odgovara definiciji pojma F-algebre u teoriji kategorija. Ako znamo interpretaciju svih operacija, postavlja se pitanje kako definisati interpretaciju proizvoljnih izraza. Ispostavlja se da je tip izraza (termova) sa operacijama iz signature date funktorom `f` upravo nepokretna tačka onog dela funktora `f` koji deluje nad objektima. Funkcija `eval` se dobija se kao vrednost `phi . f phi . f f phi . f f f phi` ... To izražavamo pomoću deklaracija

```
fix :: Algebra f a => Fix f -> a
eval = phi . fmap eval . out
```

Pošto smo definisali pojam algebre, vratimo se modularnosti. Sve što treba da uradimo je da definišemo sumu dve algebre, što činimo pomoću sledećih deklaracija.

```
newtype Sum f g x = Sum {unSum :: Either (f x) (g x) }
instance (Functor f, Functor g) => Functor (Sum f g) where
  fmap h (Sum (Left  a)) = Sum (Left  (fmap h a))
  fmap h (Sum (Right b)) = Sum (Right (fmap h b))
instance (Algebra f a, Algebra g a) => Algebra (Sum f g) a where
  phi (Sum (Left  ef)) = phi ef
  phi (Sum (Right eg)) = phi eg
```

Prvo smo definisali sumu funktora, a zatim i pravilo za interpretaciju sume. Time smo omogućili da svaku komponentu interpretera definišemo nezavisno kao algebru, a ceo interpreter kao sumu algebri.

Prethodni posupak ilustrujemo na primeru nezavisnog definisanja dela za sabiranje i oduzimanje i dela za deljenje.

```
-- Addition component
data Addit x = Const Int
             | Add x x
             | Sub x x
instance Functor Addit where
  fmap f (Const i) = Const i
  fmap f (Add x y) = Add (f x) (f y)
  fmap f (Sub x y) = Sub (f x) (f y)
```

```
instance Algebra Addit Int where
  phi (Const x) = x
  phi (Add x y) = x + y
  phi (Sub x y) = x - y

-- Division component
data Divis x = Div x x
instance Functor Divis where
  fmap f (Div x y) = Div (f x) (f y)
instance Algebra Divis Int where
  phi (Div x y) = div x y
```

Ove dve komponente kombinujemo deklaracijom

```
type Expr = Fix (Sum Addit Divis)
```

i sada na tip `Expr` možemo primeniti `eval`.

Napominjemo da je funkcija `eval` ono što se često naziva katamorfizmom (eng. catamorphism) ili generalizovani fold (eng. generalized fold). Postoji i dualni pojam anamorfizma (anamorphism, generalized unfold). Primena ovakvih funkcija omogućava da se u mnogim slučejevima izbegne eksplicitna upotreba rekurzije u programu. Ideja da se programi izražavaju kao kompozicija generalizovanih `fold` i `unfold` operacija nad odgovarajućim algebrama je detaljnije razrađena u [8]. Ovim pristupom se daje potpuna teorijska zasnovanost za korespondenciju između struktura podataka sa jedne strane i algoritamskih struktura sa druge. (Značaj ove korespondencije uočio je i primenio N. Wirth u jezicima Pascal i Modula-2.) Smisao ovog pristupa je ne samo u tome što ovako napisani programi postaju kraći, već što se tako eksplicitno zadaje suštinska struktura programa i dobijaju opštiji algoritmi. To olakšava matematičku analizu svojstava programa, a pruža i mogućnost za eliminisanje privremenih struktura podataka. Poslednji postupak je poznat pod nazivom *deforestation* (obešumljavanje) [121], jer se pomoćne strukture najčešće mogu posmatrati kao razne vrste stabala. Ovaj postupak se naziva i *program fusion*. Razvijeni su i posebni sistemi koji favorizuju pisanje programa u ovom stilu ([73], [94]). Postoje predlozi da se bolja podrška za ovakav stil programiranja uvede i u Haskell ([42], videti takođe i predstojeći Haskell Workshop 2000, Montreal).

3.3.2 Podtipovi

Primer u prethodnom odeljku je koristio pretpostavku da obe komponente interpretera koriste isti domen vrednosti sa kojima računaju (algebre koje se sabiraju su bile nad istim domenom). Pri realizaciji različitih komponenti često će biti neophodno da nova komponenta proširi skup vrednosti sa kojima se računa. Tako dodavanje komponente za uslovne izraze zahteva da se domen proširi vrednostima `True` i `False`. To dovodi do potrebe za realizacijom podtipova u Haskell-u (subtyping).

Podtipovi se mogu realizovati pomoću multiparametarskih klasa na sledeći način. Prvo deklarišemo klasu koja označava relaciju “biti podtip”.

```
class Subtype sub sup where
  inj :: sub -> sup
  prj :: sup -> Maybe sub
```

Funkcija `inj` pretvara podtip u nadtip, dok funkcija `prj` pretvara nadtip u podtip, ili vraća `Nothing` ukoliko to nije moguće. (Tip `Maybe` u Haskell-u je definisan sa `Maybe a = Just a | Nothing`.) Pomoću deklaracija instanci sada možemo definisati da je sabirak podtip algebarske sume tipova. Koristimo algebarski tip `Either` dat sa

```
Either a b = Left a | Right b
```

Činjenicu da je `a` podtip od `Either a b` izražavamo na sledeći način.

```
instance Subtype a (Either a b) where
  inj   = Left
  prj x = case x of
            (Left v) -> Just v
            _        -> Nothing
```

dok tranzitivnost relacije biti podtip izražavamo sa

```
instance Subtype a x => Subtype a (Either b x) where
  inj = Right . inj
  prj x = case x of
            (Right y) -> prj y
            _         -> Nothing
```

Napomena Moglo bi se postaviti pitanje zašto nismo definisali i instancu

```
Subtype b (Either a b)
```

Razlozi su povezani sa mehanizmom izvođenja instanci u našoj imlpementaciji Haskell-a. Naime, instance `Subtype a (Either a b)` i `Subtype b (Either a b)` se preklapaju (overlapping instances). Zajednička instanca je `Subtype a (Either a a)`. To znači da postoje dva načina na koje je tip `a` podtip tipa `Either a a`, pa mehanizam ne može znati koji od parova funkcija `prj` i `inj` da primeni. Doduše, i deklaracije koje smo mi prihvatili sadrže preklapajuće instance. Implementacija Hugs-a koju koristimo (slično implementaciji Gofer-a na kojoj je bazirana), koristi pravilo da je preklapanje dve instance dozvoljeno ako je jedna instanca pojava druge instance. U našem primeru, instance `Subtype a (Either a b)` je pojava instance `Subtype a' (Either b' x')`. U takvoj situaciji sistem za izvođenje tipova daje prednost specifičnoj instanci, u ovom slučaju prvoj. Drugim rečima, bazni slučaj se primenjuje ako je to moguće, u suprotnom se pokušava sa rekurzivnom deklaracijom instance.

Definicije koje smo mi usvojili se mogu posmatrati po analogiji sa listom realizovanom pomoću binarnog stabla. Naime, `Either` je bifunktor i njegovom primenom možemo izgraditi binarna stabla tipova. Instance koje smo mi usvojili definišu relaciju podtipa duž leve "kičme" stabla, tretirajući ga tako kao listu. Važi sledeća paralela između deklaracije instanci Subtype i definicije predikata `member` nad listom.

```
member a (Cons a x)                    Subtype a (Either a b)
member a x => member a (Cons b x)      Subtype a x => Subtype a (Either b x)
```

Kao ilustraciju primene podtipova navodimo definiciju komponente za uslovne izraze. I ostale komponente se moraju promeniti da bi radile sa podtipovima, što ilustrujemo na primeru komponente za deljenje.

```
-- Conditionals
data Condit x = If x x x
instance Functor Condit where
  fmap f (If x1 x2 x3) = If (f x1) (f x2) (f x3)
instance Subtype Bool dom => Algebra Condit dom
  phi (If x1 x2 x3) = if c then x2 else x3
    where
     c = case (prj x1) of
           Just x   -> x
           Nothing  -> error "Boolean expected in if statement"

-- Division
data Divis x = Div x x
instance Functor Divis where
  fmap f (Div x y) = Div (f x) (f y)
instance Sybtype Int dom => Algebra Divis dom where
  phi (Div x y) = inj (div (prj0 x) (prj0 y))
    where prj0 x = case x of
                     Just v  -> v
                     Nothing -> error "Integer expected in division"
```

Primetimo da prijava greške predstavlja sistemsku konstrukciju Haskell-a koja je semantički ekvivalentna sa $\perp$. Sa ovakvom definicijom domena praktično ne postoji način da se označi proces računanja u kojem je došlo do greške.

3.3.3 Interpratacija pomoću monada

Nedostatak prethodnog interpretera u pogledu prijavljivanja grešaka možemo otkloniti ako za domen algebre uzmemo monade i primenimo semantičke komponente iz prethodnog dela. Tada bi interpretacija komponente za deljenje izgledala ovako.

```
instance (Subtype Int dom, ErrMonad m) => Algebra Divis (m dom) where
  phi (Div xm ym) = do x <- mprj xm
                       y <- mprj ym
                       if y==0 then eThrow "Division by zero"
                               else returnInj (div x y)
    where
      mprj m = do x <- m
                  return (prj x)
      returnInj = return . inj
```

3.4 Modularna konkretna sintaksa

U prethodnom delu je opisano kako možemo napisati modularni inerpreter za apstraktna sintaksna stabla. Ovde se razmatraju mogućnosti za realizaciju modularne analize konkretne sintakse. Rezultat sintaksne analize će biti apstraktno sintaksno stablo.

3.4.1 Prethodni pristupi

Parsiranje se u [26] i [76] ne razmatra. Jedan od mogućih načina za realizaciju parsera je da se on pravi kao monolitična faza za jedan fiksirani skup za semantičkih komponenti. Ovo rešenje nama nije zanimljivo jer je naš cilj da modularnost dosledno sprovedemo kroz sve faze jezičkog procesora. Duponcheel ([23]) uvodi modularnost u apstraktnu sintaksu, ali prilikom pokušaja da se sličan postupak primeni i na parsiranje konkretne sintakse nastaju problemi, koje ne rešava ni Labra ([72]). Pomenuti pristupi su zasnovani na pokušaju da se parsiranje alternative apstraktnog stabla, kao što je

```
...
 | Add x x
...
```

realizuje pomoću parsera koji ima analognu rekurzivnu strukturu oblika

```
pSym '(' <*> parseX <*> pSym '+' <*> parseX <*> pSym ')'
```

Kombinacija ovakvih parsera rezultuje u parseru u kojem gotovo sve alternative započinju istim simbolom. Dobijamo, dakle, parser koji odgovara gramatici koja nije LL(1). Iako nedeterministički parseri mogu da izvrše parsiranje ovakve gramatike, proces se svodi na intenzivni bektrek, te je krajnje neefikasan. Ono što se obično radi da bi se sa top-down parserima izbegli ovakvi problemi je leva faktorizacija gramatike ([2], [114]). U našem slučaju to bi značilo da je potrebno naći šemu za levu faktorizaciju gramatike zadate modularnom specifikacijom interpretera. Čini se izvesnim da se primenom funkcija višeg reda ovo može realizovati. Tako se može poboljšati efikasnost top-down parsera, ali ostaje problem što se izrazi moraju zadavati sa svim zagradama. Pošto nije jasno kako ovaj problem rešiti u opštem slučaju, kada je broj različitih prioriteta izraza unapred nepoznat, u ovom radu smo se opredelili za jednostavnu tehniku bottom-up parsiranja. Bottom-up parsiranje je u funkcionalnom jeziku takođe moguće realizovati pomoću kombinatora, ali ako su ti kombinatori zasnovani na kontekstno slobodnoj gramatici, onda i dalje nije jasno kako zadati prioritete ukoliko ne želimo da se primena kombinatora pretvori u potpunu analizu gramatike u toku rada parsera.

To je razlog zbog kojeg ovde za specifikaciju sintakse ne koristimo gramatiku već deklaracije operatora. Za svaki operator se zadaje njegova arnost, prioritet, da li je prefiksni, infiksni, ili postfiksni, da li je levo asocijativan, desno asocijativan ili neasocijativan. Zadaje se i interpretacija operatora u vidu konstruktora apstraktnog sintaksnog stabla. Pošto ovakve deklaracije ne sadrže nikakvu formu rekurzije, nije teško specifikaciju učiniti modularnom. Ova specifikacija se vrši pomoću funkcija iz modula `PrecPar`.

3.4.2 Modularna specifikacija konkretne sintakse

Dajemo primer specifikacije konkretne sintakse semantičkog bloka za aritmetičke izraze.

```
par (N x)   = literal  (Const x)
par Plus    = infixOpL 502 Add
par Minus   = infixOpL 502 Sub
par Times   = infixOpL 503 Mul
par Divided = infixOpL 503 Div
par Power   = infixOpR 504 Pow
```

Pri tome je `Token` tip tokena koje uvodi ova komponenta,

```
data Token = N Int | Plus | Minus | Times | Divided | Power
```

dok je `Tree` deo apstraktnog stabla koji određuje signaturu algebre sa operacijama ove komponente (videti prethodni odeljak 3.3).

```
data Tree x = Const Int
            | Add x x | Sub x x
            | Mul x x | Div x x
            | Pow x x
```

Specifikacija parsera data je funkcijom `par` koja tokenima uvedenim u posmatranoj komponenti pridružuje odgovarajuću "ulogu u parsiranju". Poznavanje strukture "uloge u parsiranju" nije neophodno za pisanje specifikacije parsera, jer je ova struktura sakrivena u primeni funkcija kao što su `literal`, `infixOpL` i `infixOpR` koje su definisane u modulu `PrecPar`. Poslednje dve funkcije omogućavaju definisanje levo, odnosno desno asocijativnih infiksnih operatora zadatog prioriteta. Prvi argument predstavlja prioritet operacije, a drugi je odgovarajući konstruktor apstraktnog binarnog stabla. Funkcija `literal` definiše list apstraktnog sintaksnog stabla koji se kreira na osnovu datog tokena. Pored ovih operatora, modul `PrecPar` definiše i operacije `infixOp` za neasocijativne operatora, `prefixOp` za unarne prefiksne operatore, `prefixBinOp` za binarne prefiksne operatora, `ternary` za ternarne operatore, kao i operatore `marker`, `open`, `close`, `newLine` koje služe za rad sa zagradama i znakovima interpunkcije.

Parser definisan na ovaj način u funkciji `par` može se pokrenuti funkcijom `runParser`. Izraz `runParser par` je tipa `[Token] -> Fix Tree`, tj. predstavlja funkciju koja na osnovu liste tokena generiše apstraktno sintaksno stablo (dobijeno pomoću tipa konstruktora `In` koji ostvaruje rekurzivnu primenu konstruktora `Tree`, kao što je opisano u odeljku 3.3).

Prednost ovakvog pristupa, pored visokog nivoa apstrakcije specifikacije sintakse, je u mogućnosti kombinovanja nezavisno definisanih parsera. Modul `Glue` sadrži binarnu operaciju `cpar` koja od dva parsera pravi novi. Skup tokena koji se prepoznaju novim parserom je disjunktna suma polaznih skupova tokena, a stablo koje se dobija kao rezultat parsiranja sadrži alternative određene sa oba stabla. Tako korisniku preostaje da specificira komponente i da ih sklapa, dok sistem vodi računa o tome da se svi nivoi specifikacije kombinuju na odgovarajući način. Postupak kojim je ovo postignuto opisan je u narednom odeljku.

3.4.3 Implementacija analize konkretne sintakse

U osnovi tekuće realizacije sintaksnog analizatora baziranog na specifikaciji prioriteta je verzija "precedence parsing" tehnike ([2], [114]). Ovaj postupak je realizovan u modulu `PrecPar` koji izvozi operacije za parsiranje pomenute u prethodnom odeljku. Sam postupak parsiranja predstavlja spoj algoritma za prevođenje infiksnog izraza sa operatorima u postfiksni oblik i postupka za računanje vrednosti postfiksnog izraza pomoću steka. Parsiranje izraza se vrši pomoću dva steka. Prvi stek odgovara steku za operatore u postupku prevođenja iz infiksnog zapisa u postfiksni (`opStack`), a drugi odgovara steku kojim se računaju postfiksni izrazi (`resStack`).

```
data PPState tok a = PP { opStack   :: [(Int,ParsingItem tok a)],
                          resStack  :: [a] }
type ParsingItem tok a = PPState tok a -> PPState tok a
```

Prvi stek sadrži uređene parove (prioritet, operacija), pri čemu se operacija smešta na stek već interpretirana kao prelaz stanja. Drugi stek sadrži apstraktna stabla konstruisana za parsirane delove izraza.

Tip funkcije `par` iz prethodnog odeljka koja specificira parsiranje niza tokena `Token` i konstruiše stablo `Fix Tree` je

```
par :: Token -> (Tree x -> x) -> ParsingItem b x
```

Prvi argument funkcije `par` je token za koji se zadaje akcija prilikom parsiranja. Rezultat funkcije `par` je prelaz stanja parsera. Drugi argument, tipa `Tree x -> x`, je potreban da bi se moglo ostvariti kombinovanje različitih specifikacija parsera, i njegovu ulogu ćemo videti u nastavku. Za sada primetimo da se taj argument u specifikaciji

```
par Plus   = infixOpL 502 Add
```

ne pojavljuje. To je moguće zbog toga što su u Haskell-u sve funkcije Curry-jeve. Prethodna definicija je ekvivalentna sa

```
par Plus r = infixOpL 502 Add r
```

gde je `r` pomenuti argument, uveden iz tehničkih razloga.

Značajno je primetiti da tip funkcije `par` zavisi od tipova `Token` i `Tree`. Ovi tipovi su definisani unutar komponente i nisu poznati modulu `PrecPar`. To znači da se upotreba ovog parsera esencijalno oslanja na parametarski polimofizam. Ovakav pristup pruža znatnu slobodu prilikom pisanja komponente, a ide i u prilog efikasnosti parsera. Naime, funkcija `par` je zadata pomoću uparivanja uzoraka, što se prevodi u odgovarajuću `case` naredbu, pa se dobija parser koji na osnovu tekućeg tokena direktno izvršava zadati prelaz.

Način na koji se apstraktne operacije koriste za specifikaciju prelaza parsera pokazaćemo na primeru funkcije `infixOpL`.

```
infixOpL :: Int -> (b -> b -> a) -> (a -> b) -> ParsingItem tok b
infixOpL pri constr r = step    -- specify left associative operator
  where
```

```
    step st@PP{opStack=[]} = st{opStack=[(pri,applyBinOp r constr)]}
    step st@PP{opStack=os@((pri2,app2):ops)}
      | pri > pri2 = st{opStack=(pri,applyBinOp r constr):os}
      | otherwise  = step (pop st)
```

Rezultat ove funkcije je prelaz stanja koji poredi prioritet operatora na vrhu steka sa prioritetom novog operatora. Operatori većeg prioriteta na steku se izvršavaju operacijom `pop`.

```
pop st@PP{opStack=(_,op):ops} = op st{opStack=ops}
```

Zatim se pomoću `applyBinOp` na stek stavlja prelaz stanja definisan na osnovu konstruktora stabla `constr` i funkcije `r`.

```
applyBinOp r op st@PP{resStack=a1:a2:as} = st{resStack=r (op a2 a1):as}
```

Tako se dodatni argument `r` može posmatrati kao operacija konverzije koja se uvek primenjuje na rezultat primene konstruktora. Ako je je `constr` konstruktor apstraktnog stabla, kao što je `Add` u posmatranom primeru, i ako su `a2` i `a1` tipa `Fix Tree`, onda je `constr a2 a1` tipa `Tree (Fix Tree)`, pa stavljanjem `r = In` prema

```
newtype Fix f = In {out :: f (Fix f)}
```

dobijamo da je `r (op a2 a1)` opet tipa `Fix Tree`. Zahvaljujući tome, stek operanada `resStack` može biti homogenog tipa `[Fix Tree]`. Osim što omogućava da se radi sa stablima koja se zadaju pomoću `Fix`, dodatni argument `r` omogućava i kombinovanje dve funkcije kao što je `par`. To je ostvareno funkcijom `cpar` iz modula `Glue`:

```
cpar :: (tokL -> (treeL a -> b) -> c) ->
        (tokR -> (treeR a -> b) -> c) ->
        Either tokL tokR -> (Sum treeL treeR a -> b) -> c

cpar parL parR = f
  where
    f (Left  x) r = parL x (r . Sum . Left)
    f (Right x) r = parR x (r . Sum . Right)
```

Ako parser `parL` prihvata tokene `tokL` i generiše apstraktna stabla `treeR`, dok parser `parR` prihvata tokene `parR` i generiše apstraktna stabla `treeR`, tada rezultujući parser prihvata sumu tipova `tokL` i `tokR`, a generiše sumu stabala. Kreiranje sume stabala zahteva upotrebu dodatnih međukonstruktora `Sum`, `Left` i `Right`, što zahteva modifikaciju argumenta `r`.

Na sličan način su realizovane i ostale operacije za zadavanje infiksnih i prefiksnih operatora. To omogućava definisanje parsera za mnoge konstrukcije koje se pojavljuju u sintaksi programskih jezika. Rezultujući parser je moguće pokrenuti funkcijom

```
runParser :: (tok -> (b (Fix b) -> Fix b) -> ParsingItem c tree)
          -> [a] -> tree
```

koja postavlja inijalno stanje parsera, a zatim pokreće čitav proces parsiranja primenom operacije `foldl` na `par`.

U parser koji je realizovan su dodate i mogućnosti za prijavljivanje grešaka. To je zahtevalo uvođenje nove komponente u stanje parsera koja vodi računa o tekućoj liniji.

```
data PPState tok a = PP { opStack   :: [(Int,ParsingItem tok a)],
                          resStack  :: [a],
                          line      :: Int }
```

Pošto su svi prelazi stanja definisani pomoću imenovanih polja, dodavanje novog polja nije zahtevalo promene u delovima koda koji nisu povezani sa obradom grešaka. Tekuća linija se kao komponenta stanja implicitno prenosi. Da smo koristili uparivanje uzoraka prema njihovoj poziciji, morali bismo menjati sve definicije prelaza. Ovo ilustruje kako jednostavan mehanizam imenovanja može znatno da utiče na jednostavnost održavanja programa.

3.4.4 Dalja proširenja parsera

Parser realizovan u modulu `PrecPar` omogućava efikasno parsiranje jednostavnih izraza. U trenutnoj verziji on još uvek ne vrši zadovoljavajuću proveru grešaka i nije pogodan za zadavanje svih konstrukcija sintakse programskih jezika. Ovde ćemo ukratko razmotriti mogućnosti za njegovo proširenje.

Prva, pomalo iznenađujuća osobina ovako realizovanog parsera je da parser dopušta da operatori koji su definisati sa npr. `infixOpL` budu zadati ne samo u infiksnom, veći i u prefiksnom, pa čak i postfiksnom obliku. Ovo svojstvo se direktno prenosi iz algoritma za prevođenje infiksnih izraza u postfiksne. Rad ovog algoritma se zasniva na jednostavnom odgađanju primene operatora na argumente dok ti argumenti ne budu dostupni. Sam redosled stavljanja operatora i operanda na stek se ne proverava. U [114] je opisan jedan postupak za proveru greške zasnovan na relaciji koja određuje koji simboli u izrazu mogu biti susedni. Ovde predlažemo nešto opštiji postupak koji se može primeniti i na izraze koji sadrže infiksne i prefiksne operatora, kao i operatore sa više argumenata i pomoćnim interpunkcijskim tokenima.

Ovakve mogućnosti za proširenje su posledica odabranog načina zadavanja i realizacije sintaksne analize. Parser se može posmatrati kao deterministički push-down automat, a njegovo proširivanje se ostvaruje definisanjem apstraktnih operacija koje se mogu prevesti u prelaze automata.

Postupak za poboljšanje koji mi predlažemo je da se prilikom svakog stavljanja operatora na stek za operatore, stavi i odgovarajući marker na stek za operande. Proverom položaja markera u odnosu na operande se jednostavno detektuje razlika između prefiksne, infiksne i postiksne primene operatora, te se može prijaviti greška ako ta primena nije u skladu sa sintaksom. Ako se obezbedi i zadavanje različitih vrsta markera, omogućila bi se i upotreba dodatne interpunkcije i ključnih reči kao što su `then` i `else`. Tako se `if` može realizovati kao ternarni prefiksni operator koji prilikom izvršavanja proverava i da li su markeri `then` i `else` na odgovarajućim mestima na steku. Ovaj postupak bi omogućio i ispravno parsiranje prefiksnih operatora različitih prioriteta, što trenutno nije korektno podržano.

Napominjemo da nam ovde nije od preteranog interesa primena opštih oblika parsiranja sa prioritetima, jer specifikacija ne koristi neterminalne simbole već samo operatore. Dalje proširenje bi verovatno bilo zasnovano na LR(1) ili LALR(1) gramatici sa mogućnošću specifikacije shift-reduce konflikata, a efikasna realizacija toga bi verovatno zahtevala primenu generisanja izvornog koda. Jedan pristup generisanju Haskell programa koji implementira LR(1) parser za datu gramatiku je dat u [25].

3.5 Leksička analiza

U ovom odeljku je dat opis postupka koji se koristi za realizaciju leksičke analize našeg modularnog interpretera. Baziran je na [2], ali sadrži i neka razmatranja specifična za leksičku analizu kod modularnih interpretera. Videćemo da je modularnost leksičke analize relativno lako postići, a razmotrićemo mogućnosti memoizacije za realizaciju lenje konstrukcije prelaza i stanja automata.

3.5.1 Lenja konstrukcija prelaza

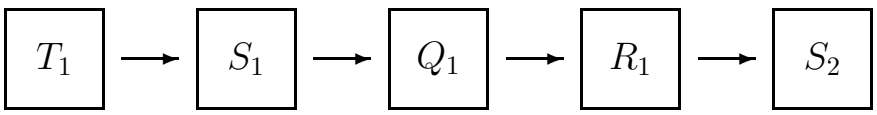
Leksička analiza je jednostavan, ali značajan deo interpretera i kompajlera. Teorijsku osnovu ovih postupaka čine deterministički konačni automati (DFA) i nedeterministički konačni automati (NDFA). Većina generatora leksičkih analizatora koriste regularne izraze za zadavanje tokena, što je i ovde usvojeno. Regularni izrazi su pogodni za specifikaciju tokena, ali ne i za direktnu implementaciju. Zato se pri implementaciji regularni izraz pretvara u NDFA. NDFA je moguće simulirati tokom leksičke analize tako što se vodi računa o skupu stanja u kome se automat trenutno može nalaziti. Naredni skup stanja se računa tako što se izračunaju prelazi iz svakog od skupa trenutnih stanja. Alternativni pristup je da se NDFA prevede u DFA. Ideja ovog prevođenja je da svakom stanju DFA odgovara skup stanja NDFA, a prelaz između od *skupa stanja* A do *skupa stanja* B je skup svih prelaza iz elemenata $a \in A$ u elemente $b \in A$. Dobijenim DFA je moguće vrlo efikasno vršiti leksičku analizu. Problem koji se javlja pri prevođenju NDFA u DFA je što u najgorem slučaju broj stanja DFA može da bude eksponencijalna funkcija broja stanja polaznog NDFA. To sa jedne strane zadaje probleme sa smeštanjem DFA u memoriji, a sa druge strane ukazuje na eksponencijalnu vremensku složenost procesa prevođenja NDFA u DFA.

U specifikaciji modularnog interpretera zadaju se regularni izrazi, i potrebno je od njih konstruisati leksički analizator. To znači da se i sve transformacije regularnog izraza u automate obavljaju u toku ili pre leksičke analize, pa efikasnost čitavog postupka zavisi kako od transformacije regularnog izraza u automat, tako i od izvršavanja automata. Ovakav problem u kompajler-kompajlerima nije toliko bitan (jer se specifikacija kompajlera obrađuje znatno manji broj puta nego što se pokreće generisani program), ali se javlja kod editora teksta koji koriste regularne izraze. Jedan od pristupa koji izbegava eksploziju stanja DFA, a efikasniji je od simulacije NDFA je tzv. *lenjo izračunavanje prelaza*, koje se može posmatrati na sledeći način. Uočimo prvo da se i simulacija izvršavanja NDFA može posmatrati kao izvršavanje DFA, jedino što prelazi nisu dati eksplicitnom tabelom već se računaju u toku rada automata. Ako se isti skup stanja javi više puta, više puta će se izračunati. Sa druge strane, pristup sa prevođenjem u DFA garantuje da će se izračunavanje svakog prelaza vršiti

tačno jednom. Time se izbegava višestruko izračunavanje istog stanja, ali se može nepotrebno izgubiti vreme na računanje prelaza koji se neće koristiti u leksičkoj analizi konkretnog niza znakova. Prirodno se nameće ideja da se prelaz stanja izračuna prvi put kada je potreban, a zatim zapamti, tako da se sledeći put ne mora ponovo računati. Osim toga, može se rezervisati unapred određena količina memorije za smeštanje prelaza izračunatih "u letu". Ako se ova količina prepuni, neki prelazi se mogu izbaciti i kasnije po potrebi ponovo računati.

Uočavamo da je ideja lenje konstrukcije stanja automata analogna sa problemom strategije izračunavanja funkcionalnih programa. Tako simulacija NDFA odgovara *call by name* izračunavanju, prevođenje u DFA je slično *call by value* izračunavanju, a ideja lenje konstrukcije stanja automata odgovara lenjom izračunavanju. Zato bi se moglo pomisliti da će sam mehanizam lenjog izračunavanja Haskell-a obezbediti ponašanje simulacije koje odgovara lenjoj konstrukciji prelaza. Nažalost, to nije slučaj. Lenjo izračunavanje omogućava da se izbegne višestruko računanje samo onih primena funkcija koje su kreirane statički na istom mestu unutar funkcionalnog programa. Prelaz stanja je, međutim, određen nizom ulaznih znakova automata, tako da se primena funkcija koje dovode do izračunavanja prelaza kreira dinamički u toku izvršavanja programa. Svaki prelaz stanja predstavljen je novim čvorom na heap-u, pa se mora ponovo računati.

Lenjo izračunavanje je pak samo jedan stepen u primeni ideje memoizacije [27]. Ukoliko bi funkcija za izračunavanje sledećeg stanja u simulaciji NDFA bila memo funkcija, rezultat bi upravo bila lenja konstrukcija prelaza automata. Kao što je pomenuto u odeljku 2.5, tehnika memoizacije koja se čini najprimenljivija je lenja memoizacija. Ispitajmo šta bi se desilo kada bismo lenju memoizaciju primenili direktno na simulaciju NDFA u kojoj je skup stanja predstavljen listom. Neka S predstavlja skup stanja, a S_i pokazivače na liste na heap-u čije su vrednosti S. Memoizacija ima efekta samo onda kada DFA više puta prolazi kroz ista stanja. Neka je S prvi skup stanja NDFA koji se ponavlja tokom analize niza znakova. Neka je memo funkcija zapamtila vrednost za S_1 kada je prvi put izračunat prelaz iz stanja S i neka je S_2 pokazivač koji se kreira kada se to stanje drugi put konstruiše (po izboru skupa S to je prvi put da se neki skup stanja ponovio u toku rada NDFA).



Kako su svi pokazivači T_1, S_1, Q_1, R_1 kreirani do ovog trenutka različiti, memo funkcija nema zapamćen prelaz iz R_1 u S_2, pa će i pokazivač S_2 biti nov.

Zaključujemo da lenja memoizacija bez prethodne transformacije u ovom slučaju nema efekta. Problem je kako prepoznati složeni objekat kada se on prvi put ponovi. Rešenje je da prvo sortiramo listu koja predstavlja niz stanja, a zatim primenimo funkciju `unique` iz odeljka 2.5. U prethodnom slučaju, kada se generiše S_2, lista se sortira, tako da se dobija pokazivač S_3. Konačno se pomoću `unique` dobija upravo pokazivač S_1. Kada se jednom kreira prelaz od R_1 do S_1, sledeći put se on direktno primenjuje, bez sortiranja liste stanja i primene funkcije `unique`.

Ovo razmatranje pokazuje značaj koji napredni mehanizmi evaluacije imaju za podizanje nivoa apstrakcije u programiranju: različite varijacije leksičke analize mogu se posmatrati

kao rezultat različitih strategija izračunavanja funkcija, pri čemu je izvorni program isti. Nažalost, implementacije Haskell-a ne podržavaju izračunavanje memo funkcija, tako da smo postupak memoizacije morali da sprovedemo eksplicitno.

3.5.2 Realizacija opšteg leksičkog analizatora u Haskell-u

Leksička analiza se pri pisanju parsera u funkcionalnim jezicima uglavnom realizuje ili opštim kombinatorima za parsiranje, ili se piše analizator specijalno za skup tokena konkretnog jezika.

Drugi pristup nije primenljiv za modularne interpretere, jer se od dva nezavisna leksička analizatora u opštem slučaju ne može napraviti novi. Naime, uobičajena pretpostavka je da će leksički analizator vratiti najduži niz znakova koji predstavlja token, pa dolazi do problema ako komponenta leksičkog analizatora koja se prva primeni prepozna kratku leksemu i ne pruži priliku sledećoj komponenti da eventualno prepozna dužu leksemu.

Primena opštih kombinatora za parsiranje je moguće rešenje, ali s obzirom da regularni izrazi predstavljaju vrlo specijalnu klasu gramatika, može se očekivati postojanje znatno efikasnijeg postupka. Pored toga, direktna primena kombinatora za parseranje baziranih na rekurzivnom spustu rezultuje u bektreku, dok LL(1) kobinatori nisu primenljivi jer regularni izrazi ne zadovoljavaju uslov LL(1) gramatike.

S obzirom na nedostatke prethodnih postupaka, za konstrukciju modularnog leksičkog analizatora je usvojen pristup sa generisanjem konačnog automata, kao postupak koji daje teorijski najefikasniji algoritam za leksičku analizu.

Pored toga, primenjeno je i poboljšanje postupka sinteze konačnog automata, inspirisano konstrukcijom iz [2]. Standardna konstrukcija NDFA iz regularnih izraza se sprovodi rekurzivno po strukturi regularnog izraza, pri čemu se koriste prazni prelazi. To olakšava dokaz korektnosti postupka, ali generiše veliki broj stanja. Takođe se generišu i prazni prelazi, pa je kasnije potrebno tražiti tranzitivno zatvorenje skupova stanja. Efikasnija konstrukcija, kojom se od datog regularnog izraza dobija direktno DFA, opisana je u [2]. Prvo se izraz proširi konkatenacijom sa znakom za kraj `#`. Zatim se svakom znaku azbuke u regularnom izrazu (uključujući `#`) pridruži različita pozicija. Pozicije se zatim mogu posmatrati kao stanja nedeterminističkog konačnog automata koji prati do koje tačke je stigao u regularnom izrazu. Automat vrši nedeterministički izbor pri nailasku na alternativu (koju će granu izabrati), kao i pri nailasku na zvezdu (da li će još jednom proći kroz izraz pod zvezdom ili će produžiti na sledeći deo izraza). Efikasan način za nalaženje prelaza ovog nedeterminističkog konačnog automata je obilazak u dubinu drveta kojim je predstavljen regularni izraz. Pri tome se za svaki podizraz računaju sledeći skupovi pozicija.

- firstpos(i) – stanja u kojima automat može da otpočne analizu niza znakova koji odgovara izrazu i;
- lastpos(i) – stanja u kojima automat može da se nađe na kraju analize niza znakova.

Funkcije firstpos i lastpos se lako rekurzivno računaju iz strukture stabla. Za njihovo računanje je potrebna funkcija nullable, koja određuje da li dati podizraz prihvata praznu reč. Računanjem ove tri funkcije se eliminiše potreba za praznim prelazima. Prelazi NDFA

su određeni relacijom nad pozicijama followpos, koja se definiše kao najmanja relacija sa sledeća dva svojstva.

- za svaki čvor konkatenacije st važi lastpos(s) $\times$ firstpos(t) $\subseteq$ followpos;
- za svaki zvezda-čvor $s*$ važi lastpos(s) $\times$ firstpos(s) $\subseteq$ followpos.

Tako dobijamo NDFA.

Pri samom izvršavanju NDFA koristi se lenja konstrukcija prelaza automata. Memo funkcije nisu implementirane u Haskell-u, kao ni u ostalim funkcionalnim jezicima koji su u široj upotrebi. Zato je ovde memoizacija implementirana eksplicitno. To je zahtevalo uključivanje zapamćenih prelaza u stanje simuliranog automata. Struktura funkcija za simulaciju automata je zasnovana na prelazima stanja, kao i realizacija parsera. Zato su se i ovde kao korisna pokazala imenovana polja u deklaracijama algebarskih tipova podataka.

Pri nalaženju tokena putem DFA koristi se pravilo najdužeg prefiksa (eng. longest match, maximal munch rule). To znači da se pri svakom ulasku u završno stanje automata pamti tekuća pozicija u ulaznom nizu. Kada automat pređe u stanje iz kojeg nema prelaza, vraća se pozicija i token koji odgovaraju poslednjem prihvaćenom stanju. Time se ostvaruje prepoznavanje najduže lekseme koja sledi od tekuće pozicije u nizu znakova.

Eksplicitna implementacija memoizacije unutar Haskell-a se pokazala kao prilično neefikasna. Osnovni problem je potreba za propagacijom stanja memo tabele. Zanimljivo je primetiti da se korektno implementirana memo funkcija ponaša u skladu sa zakonima referencijalne transparentnosti, ali da kompajler to nije u stanju da detektuje. Može se razmišljati o definisanju mehanizama u jeziku koji bi dozvolili bočni efekat i lokalno stanje u funkcijama ako je moguće pokazati da rezultat funkcije ne zavisi od tog stanja. To bi omogućilo implementaciju memo funkcija, i uopšte struktura podataka sa većim mogućnostima za samomodifikaciju nego što pruža samo lenjo izračunavanje. Sadašnji stepen razvoja kompajlera za čisto funkcionalne programske jezike dozvoljava destruktivno ažuriranje, ali se stanje mora propagirati (bilo eksplicitno, kao u programskom jeziku Clean, bilo putem monada, kao u Haskell-u).

3.5.3 Modul `RegExps`

Signatura modula RegExps koji implementira modularni interpreter prema opisanom postupku data je sledeća.

```
data RegExp a
rNull          :: RegExp a
rSym           :: a -> RegExp a
rMany          :: RegExp a -> RegExp a
(<|>), (<&>)   :: RegExp a -> RegExp a -> RegExp a

rLit   s = foldr1 (<&>) (map rSym s)
rAnyOf s = foldr1 (<|>) (map rSym s)

data Automaton a b
```

```
type Lexeme a b = (RegExp a,[a] -> b)
makeAutomaton :: Bounded a => [Lexeme a b] -> Automaton a b
runAutomaton  :: (Eq (Maybe a), Eq (Int,a)) =>
                    Automaton a b -> [a] -> [b]
```

`RegExp`, `rNull`, `rSym`, `rMany`, `<|>`, `<&>` realizuju apstraktni tip podataka za definisanje regularnih izraza. `RegExp` je tip regularnog izraza, `rNull` označava prazan izraz, `rSym` služi za zadavanje pojedinačnih znakova, `rMany` je iteracija, `<|>` je alternativa, a `<&>` je kontatenacija regularnih izraza. `rLit` i `rMany` su često korišćene skraćenice za zadavanje niza znakova odnosno jednog od datog skupa znakova.

Leksema se zadaje regularnim izrazom i preslikavanjem koje niz ulaznih znakova pretvara u token. `makeAutomaton` prihvata niz specifikacija leksema i generiše automat. Ograničenje `Bounded a` se koristi jer je potreban tip koji ima jedan fiksirani element (pointed set). Taj tip se koristi kao znak `#` za označavanje poslednje pozicije i bitna je samo njegova egzistencija.

`runAutomaton` pretvara konačni automat u funkciju koja niz znakova pretvara u niz tokena. Ograničenja za klase su posledica načina implementacije. Ograničenja `Eq` su potrebna zbog poređenja pri implementaciji memo funkcije.

Implementacija modula `RegExps` uz minimalne komentare ima oko 180 linija u Haskell-u. (Tu je ukljuceno prevođenje regularnih izraza u NDFA, kao i lenja konstrukcija prelaza automata sa implementacijom memoizacije.) Modul `RegExps` uvozi module `FRels` za zadavanje relacija nad skupom, što se koristi za prelaze nedeterminističkog automata, `FMaps` koja implementira preslikavanja, što se koristi za generisane prelaze determinističkog automata, `Hashing` za implementaciju heširanja za pamćenje veze između skupova stanja NDFA i stanja koje tom skupu odgovara u DFA, i `Sorting` koji implementira sortiranje potrebno za memoizaciju. Svaki od ovih pomoćnih modula je realizovan na trivijalan način pomoću listi (u manje od 20 linija), ali je njihovo dalje poboljšavanje jednostavno i nezavisno od ostatka programa jer su predstavljeni kao apstraktni tipovi podataka.

3.5.4 Sklapanje specifikacija leksike

Sklapanje specifikacija leksike je jednostavno iz dva razloga. Prvi je to što, za razliku od sintakse, leksička struktura ne koristi rekurziju. Zato je moguće dva regularna izraza koji predstavljaju specifikacije leksike dve komponente spojiti operacijom `<|>`. Sledeći razlog je što je specifikacija leksike zadata običnom listom tipa `[Lexeme a b]`, što omogućava jednostavnu manipulaciju ovom strukturom. Sklapanje se vrši u modulu `Glue` operacijom `clex`.

```
clex :: [Lexeme c tokL] -> [Lexeme c tokR] -> [Lexeme c (Either tokL tokR)]
clex lexL lexR = (map (m Left) lexL) ++ (map (m Right) lexR)
  where
    m f (exp,val) = (exp, f . val)
```

Pošto se domeni leksema sabiraju, bilo je potrebno modifikovati i preslikavanja koja od leksema prave tokene. Operacija `clex` je definisana tako da se njenom primenom na specifikacije leksema istim redosledom kao što se primenjuje `cpar` za sklapanje specifikacija parsera, dobija leksički analizator koji se može koristiti uz odgovarajući parser.

3.5.5 Primer specifikacije leksike

Sledi primer leksičkog dela aritmetičke komponente jezika.

```
data Token = N Int | Plus | Minus | Times | Divided | Power
lexemes = [(pInt, makeInt),
           (rSym '+', \_ -> Plus),
           (rSym '-', \_ -> Minus),
           (rSym '*', \_ -> Times),
           (rSym '/', \_ -> Divided),
           (rSym '^', \_ -> Power)]
pInt = digit <&> (rMany digit)
  where digit = rAnyOf "0123456789"
makeInt ds = N (foldl op 0 ds)
  where op n d = 10*n + (ord d - ord '0')
```

Uočavamo da je tip `Token` definisan kao korisnički tip u komponenti. Visoki stepen parametrizacije omogućio je specifikaciju komponente sa čitavom klasom tipova ulaznih podataka i klasom tipova izlaznih podataka, što povećava fleksibilnost primene modula `RegExps`.

3.5.6 Parcijalno izračunavanje za regularne izraze

Postavlja se pitanje da li je i u slučaju zadavanja regularnih izraza moguće primeniti ideju parcijalnog izračunavanja kao kod opštih kombinatora za parsiranje ([109], [108]). To bi značilo da operatori za kreiranje regularnih izraza istovremeno računaju i followpos simbole i same prelaze. Pošto su i followpos simboli definisani rekurzivno nad strukturom regularnih izraza, ovo je u principu moguće uraditi, ali je nepraktično. Naime, problem se javlja sa numeracijom pozicija unutar regularnog izraza, jer je pri spajanju izraza potrebno obezbediti da različite pozicije unutar izraza budu označene različim brojevima. To pak znači ili kopiranje kompletne strukture jednog izraza da bi se brojevi pomerili, ili uvođenje globalnog parametra koji bi se koristio kao brojač. Prvo rešenje je neefikasno. Drugo rešenje uništava kompozicionalnost kombinatora uvodeći sekvencijalnost, bez obzira da li je sama funkcija napisana u vidu monade ili eksplicitnim prenosom parametra.

Zaključujemo da postupak konstrukcije automata iz regularnog izraza nema strukturu koja pogoduje parcijalnom računanju u toku same konstrukcije izraza. Jedini deo koji se može izračunati u toku konstrukcije izraza je funkcija nullable, jer je ona zadat kao fold nad drvetom regularnog izraza.

3.6 Sklapanje komponenti

Ovaj odeljak je posvećen sumiranju postupaka zahvaljujući kojima je ostvarena modularnost svih slojeva interpretera i pokazuje kako se ovi slojevi povezuju u konačan interpreter.

Sklapanje komponenti modularnog interpretera je predstavljalo posebnu teškoću s obzirom na odluku da se sva aktivnost uradi isključivo pisanjem funkcija u Haskell-u, bez automatskog generisanja Haskell koda. To znači da smo bili ograničeni mogućnostima sistema tipova. Koristili smo ne samo standardne mogućnosti Haskell-a, već i eksperimentalne

mogućnosti interpretera Hugs, pre svega multiparametarske klase i rešavanje konflikta pri preklapanju instanci. Čak i ove mogućnosti sistema tipova u nekim slučajevima su se pokazale nedovoljne.

Konačni modularni interpreter je sastavljen od 8 komponenti i njegova globalna struktura je sledeća.

```
module Main where  -- Glavni modul modularnog interpretera
 -- komponente:
import qualified Aritm  as A      -- aritmetika
import qualified Condit as C      -- if naredba i relacije
import qualified Enviro as E      -- lokalne definicije
import qualified Excep  as X      -- izuzeci
import qualified Funcs  as F      -- funkcije
import qualified Loops  as L      -- petlje
import qualified Nondet as N      -- nedeterminizam
import qualified State  as S      -- stanje
import qualified PUtils as U      -- pomocni modul za parsiranje

import Glue         -- sklapanje komponenti

import FMaps        -- konacna preslikavanja
import Subtypes     -- podtipovi
import Algebras     -- algebre za interpretaciju
import Ormers       -- monad transformeri
import Showmers     -- prikaz monada
import PrecPar      -- parser
import RegExps      -- leksicki analizator
```

Sklapanje leksike i sintakse komponenti se vrši pomoću operacija `clex` i `cpar` iz modula `Glue`.

```
lexemes = A.lexemes `clex` C.lexemes `clex`
          E.lexemes `clex` X.lexemes `clex`
          F.lexemes `clex` L.lexemes `clex`
          N.lexemes `clex` S.lexemes `clex`
          U.lexemes

par     = A.par `cpar` C.par `cpar`
          E.par `cpar` X.par `cpar`
          F.par `cpar` L.par `cpar`
          N.par `cpar` S.par `addpar`
          U.par
```

Time se ujedno sklapaju i komponente apstraktne sintakse. Pri tome koristimo skraćena imena modula koji definišu komponente (reimenovanjem prilikom uvoza). Modul `U` (`PUtils`)

definiše parsiranje interpunkcije i nema svoje sintaksno stablo, pa ga sa ostalim parserima povezujemo posebnim operatorom `addpar`.

Sklapanje monad transformera moguće je zahvaljujući liftingu. Na kraju još preostaje definisanje domena računanja, za šta smo koristili konstruktor `Either` oslanjajući se na implementaciju podtipova pomoću klase `Subtype`.

```
-- monada za opis racunanja interpretera
type Compute = EnvT   NTable                  -- okruzenje
               (ContT Ans                     -- kontinuacije
               (StateT (FMap String Value)    -- stanje
               (ErrT   String                 -- greske
                [ ])))                        -- nedeterminizam
-- vrednost koja se racuna sa Compute
type Value =  Either Int  -- celi brojevi
              (Either Bool -- logicki tip
              (Either Fun  -- funkcije
                     ()))
```

Prilikom zadavanja domena računanja (`Value`) nailazimo na teškoće jer su definicije tipova često rekurzivne, pa ih nije moguće zadati sa `type`. Zato se koristi `newtype`, što zahteva primenu dodatnog konstruktora tipa. Ovi novi konstruktori dalje zahtevaju dodatne deklaracije instanci koje donekle narušavaju modularnost jer zavise od konkretno izabranih domena i monada. Oni ne nose nikakvu semantiku, već izražavaju posledice izomorfizma između argumenta `newtype` deklaracije i njene definicije.

Kada tim deklaracijama obezbedimo željeni domen računanja, možemo definisati sam interpreter kao funkciju `calc` tipa `String -> String`. Ovde do punog izražaja dolazi modularnost s obzirom na faze kompajlera.

```
auto = runAutomaton (makeAutomaton lexemes)
execute :: String -> Compute Value
execute = eval . runParser par . auto
calc :: String -> String
calc = mShow . test
```

Pri tome je `mShow` metod pomoćne klase koja se koristi za prikazivanje monada.

3.7 Primer rada interpretera

Ovde prikazujemo rezultate rada malog interpretera koji je realizovan kao ilustracija prethodno opisanih postupaka. Interpreter je rezultat sklapanja 8 komponenti na način opisan u prethodnom odeljku. Kompletna specifikacija ovih komponenti i kratak opis njihove funkcionalnosti dat je u dodatku.

Funkciju `calc` pozivamo direktno iz promta Hugs iterpretera sa stringom koji predstavlja program. Ovu funkciju možemo koristiti kao jednostavan kalkulator. Tako `calc "3*10^2-9"` vraća 291. Izraz `calc "1+4/0` vraća grešku pomoću mehanizma za prijavu greške realizovanog `ErrT` monad transformerom.

Moguće je uneti i složenije izraze. Tako izraz

```
calc "s0:=3; (&n (s0:=?s0+1; n+?s0))@7 + ?s0"
```

kao rezultat vraća broj 15. Naime, prvo se sadržaj lokacije `s0` inicijalizuje na vrednost 3. Zatim se definisana funkcija po promenljivoj `n` primeni na argument 7, i tom prilikom se izvrši bočni efekat uvećanja sadržaja lokacije `s0` sa 3 na 4. Rezultat primene funkcije je broj 11, a konačni rezultat se dobija sabiranjem sa trenutnom vrednošću 4 lokacije s0.

Pozivom ove funkcije na ekranu se ispisuje i konstruktor `Left` jer je rezultat tipa `Value`, a ovaj tip je definisan kao suma tipova `Int`, `Bool` i `Fun`. Rezultat je lista vrednosti, jer se monad transformeri primenjuju na listu kao monadu za nedeterminizam.

Za ispitivanje većih programa je napisana i funkcija `testf` koja učitava program koji treba interpretirati iz navededne datoteke, kao i funkcija `main` koja zahteva da se program unese sa terminala.

Petlje su ilustrovane sledećim primerom, koji računa sumu prvih 10 kvadrata prirodnih brojeva.

```
:A (&x (x*x))
s0:=1;
s1:=0;
while (?s0 <= 10) (
  s1:=?s1 + A@?s0;
  s0:=?s0 + 1
);
?s1
```

Primenu funkcija višeg reda ilustruje program

```
:X (&f f@3)
:C (&x x*7)
X@C
```

koji definiše funkciju `X` koja svoj argument primenjuje na broj 3 i funkciju `C` koja dati broj množi sa 7. Rezultat izvršavanja programa je 21.

Sledeći program definiše `C` kao lambda izraz za računanje faktorijela, a zatim taj izraz primenjuje na vrednost 9.

```
:C (&n (if (n=1) (1#(0-2)) (n*C@(n-1))))
C@9
```

Specifičnost ovog primera je u tome što se ne vraća jedan rezultat, već se u baznom koraku upotrebom operatora `#` nedeterministički vraćaju vrednosti 1 i -2. Zbog toga i program ima dva rezultata: `362880` i `-725760`.

Mogućnosti nedeterminističkog računanja ilustruje i sledeći program.

```
:A (&x
   (s0:= if (x < 0) (0-x) x;
    s1:= (16 # (0-3));
    s1:= ?s0 + ?s1))
when (s3:= A@(5 # (1-2) # 10 # 21)) < 20 ;
?s3
```

Ovaj program primenjuje funkciju `A` na 5 različitih argumenata. U prvoj naredbi funkcije se promenljivoj `s0` dodeljuje apsolutna vrednost argumenta x. Funkcija za jedan argument vraća dva rezultata, jer se promenljiva `s1` inicijalizuje ili na 16 ili na -3. Od 10 potencijalnih rezultata izdvajaju se oni koji su manji od 20 i oni čine rezultat programa: 2, 17, -2, 7, 18.

Sledeći primer je sličan prethodnom i ilustruje mogućnost za obradu grešaka.

```
try
  (&y
  (:A (&x
     (s0:= if (x < 0) y@999 x;
      s1:= (16 # (0-3));
      s1:= ?s0 + ?s1))
  when (s3:= A@(5 # (1-2) # 10 # 21)) < 20;
  ?s3))
catch
  (&z (z+1))
```

Iza ključne reči `try` sledi lambda apstrakcija. Promenljiva `y` služi za prekid normalnog toka kontrole i prijavu greške. Greška se prijavljuje njenom primenom na argument koji predstavlja informaciju o prirodi greške. Ukoliko do greške ne dođe, ova promenljiva nema nikakvu ulogu. U slučaju greške (kada je $x < 0$), prekida se normalan tok kontrole, a vrednost prosleđena `y` postaje argument lambda apstrakcije navedene iza `catch`. To omogućava da se ustanovi priroda greške. Rezultat izvršavanja ovog programa je lista 2, 1000, 7, 18. Umesto dva rezultata 17 i -2 koja su se javila u prethodnoj verziji programa, sada se javlja samo jedan rezultat 1000 dobijen dodavanjem 1 na 999.

4 Diskusija

4.1 Rezultati

Konačan rezultat ovog rada je jednostavni modularni interpreter, dobijen sastavljanjem nezavisnih komponenti od kojih svaka opisuje neku od osobina programskog jezika koji se interpretira.

Interpreter je napisan u Haskell-u korišćenjem proširenja implementacije Hugs98. Korišćen je Hugs98 interpreter od februara 2000. godine. Program se sastoji od modula koji realizuju apstraktne tipove za specifikaciju semantike, apstraktne sintakse, konkretne sintakse i leksike komponenti modularnog interpretera. Takođe je napisano i nekoliko komponenti koje koriste ove apstraktne tipove, realizujući tako mali interpreter. Opis ovih komponenti dat je u dodatku. Ukupna veličina ovih Haskell programa je oko 40KB.

I pored svih ograničenja, ovaj interpreter funkcioniše i predstavlja dokaz da se u principu navedenim postupkom mogu pisati jezički procesori. Ipak, glavni rezultat rada je iskustvo stečeno u funkcionalnom programiranju u Haskell-u i u razumevanju odnosa njegovih konstrukcija sa matematičkom teorijom na kojoj je baziran.

4.2 Prednosti pisanja u Haskell-u

Programski jezik Haskell je odabran kao jezik koji ilustruje savremeni trend u funkcionalnom programiranju. Složenost jezika donekle preti njegovoj popularnosti, ali se čini izvesno da će on imati značajnu ulogu u naučnoj komunikaciji, pa bi se u neku ruku mogao uporediti sa programskim jezikom Algol. Opštost implementiranih jezičkih konstrukcija ga čine svojevrsnom laboratorijom iz koje vremenom mogu nastati konstrukti koji će uticati i na programske jezike koji su u široj upotrebi.

U praktičnoj primeni Haskell se pokazuje kao jezik koji je teže naučiti, ali kada se nauči efikasnost programiranja je veća nego u imperativnim jezicima.

Ovde ističemo one osobine Haskell-a koje su se pokazale kao posebno korisne u realizaciji modularnih interpretera.

Konciznost Specifikacije komponenti interpretera su tipične dužine od oko 70 linija. Sve monade i monad transformeri sa liftingom zauzimaju oko 170 linija. Modul `PrecPar` ima oko 120 linija, a modul `RegExps` oko 180. Programi ove dužine koji se mogu izvršavati makar i za male ulazne podatke su od velikog značaja za proveru algoritamskih ideja. Mnoge greške je moguće otkriti proverom funkcija za male podatke. Konciznost je takođe element koji Haskell čini pogodnim za komunikaciju algoritamskih ideja.

Lako nalaženje grešaka Stiče se utisak da je glavni razlog lakšeg nalaženja grešaka u Haskell-u uniformno tretiranje vrednosti sa kojima se računa u programu. Pokazivači su skriveni kao i upravljanje memorijom. Vrednosti složenih tipova se lako i koncizno mogu eksplicitno zadati što pomaže u proveri korektnosti funkcija. Interpretativni karakter okruženja kao što je Hugs dodatno doprinosi lakoći ispravljanja grešaka. Stil pisanja programa koji zasnovan na intenzivnoj upotrebi međustruktura podataka koji favorizuje Haskell rezultuje u većoj modularnosti što olašava lokalizaciju grešaka.

Ovde treba ipak priznati da je većina grešaka koje Haskell prijavljuje greške tipa. Za razumevanje uzroka ovih grešaka je potrebno imati iskustvo, a često je potrebno i poznavati pravila za izvođenje tipova. Dalji razvoj dijagnostičkog dela sistema za proveru tipova bi znatno doprineo upotrebljivosti implementacija Haskell-a.

Matematička zasnovanost Haskell programi imaju jednostavnu semantiku zasnovanu na lambda računu. Šta više, zapisi u Haskell-u su često mnogo koncizniji i jasniji nego čista matematička notacija, a zakoni koji važe za sintaksne konstrukcije omogućavaju zaključivanje o osobinama programa bez prevođenja u neku drugu notaciju. Ono što se pokazalo kao posebno korisno je činjenica da je moguće direktno pisati denotacionu semantiku sa tipovima koji predstavljaju semantičke domene, i što je mnoge opšte konstrukcije teorije kategorija moguće realizovati u Haskell-u.

Korisnički operatori Mogućnost definisanja korisničkih operatora se pokazuje pogodnom za definisanje kombinatora. Kombinatori efektivno mogu da pretvore Haskell u jezik posebne namene. Deklaracije prioriteta prilagođavaju sintaksu datim potrebama, favorizujući još više upotrebu funkcija višeg reda.

Upotreba imenovanih destruktora Ova jednostavna mogućnost se pokazuje kao vrlo pogodno proširenje stila pisanja zasnovanog na definisanju funkcija putem uparivanja uzoraka. Njenom upotrebom se u nekim slučajevima znatno smanjuje potreba za modifikacijom funkcija. Ovde su se pokazali posebno pogodni za realizaciju sistema zasnovanih na prelazu stanja kao što je konačni automat ili parser. Programi koji definišu prelaze stanja u Haskell-u odgovaraju strogim matematičkim opisima ovakvih sistema.

Klase Klase su se pokazale kao veoma opšti mehanizam, posebno ako se dozvole multiparametarske konstruktorske klase. Njihova moć proizilazi iz velike opštosti. Iako je malo verovatno da će ikada biti široko korišćene u svom punom obliku, multiparametarske klase omogućavaju da se eksperimentiše sa mnogim mogućnostima jezika koji se kasnije mogu pojaviti kao posebne konstrukcije. Klase smo koristili za definisanje monada i njihovih transformera, za podtipove, algebre kao i za apstraktne tipove podataka. Bez klasa, veza između naših programa i teorije koja iza programa stoji bila bi mnogo manje eksplicitna. To je koncept nad kojim se vredi zamisliti pre bezuslovnog prihvatanja pojma klase kako je on definisan u objektno-orijentisanim jezicima.

4.3 Poređenje sa ostalim pristupima

Ono što odlikuje ovaj način realizacije modularnog interpretera je pre svega jednostavnost i neefikasnost. To je u suprotnosti sa pristupom u kojem se specifikacija zadaje u nekom drugom jeziku, kao što su atributivne gramatike. Problemi kod atributivnih gramatika nastaju pri definiciji semantike, gde je potrebna puna izražajnost programskog jezika opšte namene. U slučaju kada se značenje jezika zadaje denotacionom semantikom, upotreba ne-striktnih čisto funkcionalnih jezika ima jasnih prednosti jer nismo ograničeni samo na striktne funkcije. Specifikacije koje su u denotacionoj semantici stajale samo na papiru možemo izvršavati, što je od značaj čak i ako je rezultujući interpreter neefikasan.

Ovaj pristup je prihvaćen između ostalog i zbog toga što je cilj bio da se demonstrira ekspresivnost Haskell-a, a pre svega njegovog sistema tipova. Sistem tipova se pokazao kao fleksibilan i precizan u isto vreme, ali ipak nedovoljno moćan za neke konstrukcije. Mnogo više se od sistema tipova zasnovanog pretežno na Hindley-Milnerovom postupku ne može ni očekivati. Znatno veću izražajnost daje sistem zavisnih tipova (dependent types), ali on ima neodlučiv problem izvođenja tipova te zahteva eksplicitne deklaracije.

Zato se stiče utisak da je za praktičnu primenu pogodnije koristi generisanje koda od strane programa (metacomputation). Za ovakav postupak se obično vezuju problemi u vezi nedostatka garancije ispravnosti generisanog programa. Iako je ovaj problem prisutan u klasičnim kompajler-kompajlerima, on nije nužna odlika ovakvog pristupa. To ilustruju i proširenja funkcionalnog jezika ML koja omogućavaju da se elementi generisanog koda tretiraju kao struktuirane i tipizirane vrednosti. Ovakav pristup je obećavajući jer je složene operacije sklapanja komponenti moguće izolovati od samog procesa kreiranja modularnog jezičkog procesora. Važno pitanje prilikom primene generisanja koda iz opisa gramatike je način specifikacije semantike. U kompajler-kompajlerima se za tu svrhu pretežno koriste atributivne gramatike. U [2] se atributi klasifikuju na nasleđene i sintetizovane. Sintetizovani atributi čvora u stablu se izračunavaju primenom neke funkcije na atribute sinova datog

čvora. Primena isključivo sintetizovanih atributa odgovara pristupu denotacione semantike. Značenje se u denotacionoj semantici dodeljuje svakom elementu sintakse nezavisno od ostalih elemenata ([106]), što predstavlja jedan od razloga intenzivne upotrebe funkcija višeg reda: svaki oblik interakcije između značenja elemenata neke sintaksne konstrukcije se izražava pomoću aplikacije funkcija. Činjenica da je ovim pristupom uvek moguće zadati semantiku jezika odgovara rezultatu da se svi atributi atributivne gramatike mogu pretvoriti u sintetizovane ([2]). Prirodan način da se to ostvari je upravo primenom funkcija kao vrednosti atributa. Interpretacija programa u modularnim interpreterima je zadata putem funkcije `eval` koja se može shvatiti i kao opis rekurzivnog postupak izračunavanja sintetizovanih atributa od dna prema vrhu (bottom-up). Primena monada je ipak omogućila da se ovim postupkom mogu implementirati i jezici koji imaju bočne efekte. Prema tome, monade u sebi skrivaju mehanizam kojim je moguće prevesti nasleđene atribute u sintetizovane. Tako uz definiciju operacija `getCount` i `putCount` u okviru monade tranformera stanja, dobijamo mogućnost za pisanje programa

```
phi (Add mx my) -> do x <- mx
                      y <- my
                      count <- getCount
                      putCount (count+1)
                      return (plus x y)
```

koji pored izvršavanja operacija broji koliko puta su one izvršene. Brojač, koji bi se pomoću atributivnih gramatika predstavio upotrebom nasleđenog atributa (inherited attribute), realizovan je kao komponenta stanja monade. Navođenja kvalifikatora `x <- mx` ispred `y <- my` specificira da se apstrakto sintaksno stablo obilazi "preorder" postupkom, pri čemu se prvo obilazi levo, a potom desno podstablo.

Postoji, međutim, principijelna mogućnost da se i u postupku pisanja specifikacije direktno u Haskell-u dobije na efikasnosti ako se primeni *specijalizacija* i *parcijalno izračunavanje* (partial evaluation) ([60], [105]). Ideja postupka je da se nad datim programom u Haskell-u izvrše sva ona računanja koja ne zavise od ulaznih podataka. U slučaju modularnih interpretera, postupak bi mogao biti sledeći:

1. pisati interpretere modularno opisanim postupkom i izvršiti testiranje;
2. za konkretan izbor komponenti interpretera izvršiti specijalizaciju da bi se eliminisalo računanje koje nastaje usled samog procesa sklapanja;
3. rezultujući program u Haskell-u kompajlirati optimizirajućim kompajlerom da bi se dobio konačan interpreter.

I pored konceptualne jednostavnosti cilja koji žele da postignu, tehnike specijalizacije su složene i sam pristup je za sada ograničenog dometa. Može se, međutim, primetiti da je funkcionalno programiranje zbog apstraknog načina predstavljanja procesa računanja posebno pogodna platforma za istaživanje u ovoj oblasti. Značaj ovog pristupa je veliki, posebno u kombinaciji sa modularnošću, jer obećava programiranje u kojem se neće mnogo žrtvovati na modularnosti da bi se dobilo na efikasnosti. Parcijalno izračunavanje i razdvajanje prolaza

(pass separation) je takođe od velikog potencijalnog značaja za proširivanje interpretera do kompajlera. Ovo je poznata tehnika, ali po svemu sudeći još uvek nedovoljno praktična za širu upotrebu.

4.4 Nedostaci

U ovom odeljku se navode nedostaci realizovanog pristupa modularnim intepreterima i diskutuje njihova relevantnost.

U prethodnom poglavlju je napomenuto da se sistem tipova sa multiparametarskim klasama u nekim slučajevima pokazao kao neadekvatan. Dvosmislenost koja nastaje usled primene klasa ponekad se mora rešavati dodatnim deklaracijama instanci koje narušavaju modularnost. Ograničenje da se sve promenljive u kontekstu instance moraju javiti i sa desne često onemogućava definisanje mnogih instanci čiji kontekst čine multiparametarske klase.

Opredeljenje da se modularnost realizuje isključivo mehanizmima Haskell-a u nekim slučajevima je rezultovalo u složenijim i manje efikasnim rešenjima. Ovakav pristup takođe otežava upotrebu složenijih postupaka analize specifikacije komponenti. To, naime, zahteva da biblioteka za realizaciju određenog nivoa modularnog interpretera praktično analizira strukturu programa koji koristi tu biblioteku. To je u izvesnoj meri moguće uraditi, ali često rezultuje u dodatnom računanju. Uklanjanje ovog računanja tehnikom specijalizacije i parcijalnog izračunavanja nije ispitano u ovom radu. Pri tome treba imati u vidu da je specijalizacija uspešna samo ako je program pisan na odgovarajući način u kojem je relativno lako identifikovati delove koji se mogu statički izračunati.

Ideja o doslednoj primeni dvodimenzionalne modularnosti na ceo jezički procesor se takođe može dovesti u pitanje. Postoje osobine jezika koje zahtevaju proširenje samo u sintaksi, samo u semantici, ili samo u leksici, te nema potrebe da se za njih zadaju sve komponente. S obzirom na neefikasnost čitavog pristupa, problem leksičke analize se mogao potpuno zaobići i realizovati parsiranje direktno nad karakterima. Pristup koji je ovde realizovan predstavlja stoga više predlog kako se u principu može generisati efikasan modularan interpreter nego pokušaj da se raspoloživim alatima modularni interpreter realizuje na najefikasniji način. Dalje treba primetiti da je modularnost o kojoj je ovde reč pre svega semantičkog, a ne sintaksnog karaktera. Njen rezultat je pre modularnost specifikacije nego modularnost realizacije. Iako se rezultujuće komponente zadaju u posebnim modulima, Hugs implementacija ne podržava odvojeno kompajliranje, tako da se spajanje komponenti efektivno obavlja već prilikom kompajliranja. Da je modularnost pre svega vezana za specifikaciju ilustruje i proces generisanja leksičkog analizatora. Leksički analizator se specificira regularnim izrazima koji se lako kombinuju operatorom alternative. Kompajliran program sadrži specifikaciju izraza celog modularnog interpretera, ali ne i sam konačni automat. Pri prvoj upotrebi leksičkog analizatora vrši se prevođenje celog regularnog izraza u nedeterministički konačni automat i njegovo postepeno pretvaranje u deterministički automat tokom procesa računanja. Nemodularnost realizacije leksičke analize je posledica primene automata: za generisanje leksičkog analizatora koji u principu daje optimalan algoritam potrebno je znati sve lekseme lezika.

Sama realizacija ne rezultuje u praktično upotrebljivom interpreteru. Interpreter nema zadovoljavajuću obradu grešaka, i ne podržava programe sa ulazom i izlazom. Programi koji se mogu interpretirati su vrlo ograničene veličine. Specifikacija leksičkog analizatora rezul-

tuje u velikom broju stanja što dovodi do problema sa memorijom. Poznato je da primena kontrolnih struktura tipa `case` rezultuje u efikasnijem leksičkom analizatoru nego interpretacija tabele, a to posebno dolazi do izražaja ako tabela nije implementirana kao statički niz. Pomenuta neefikasnost je delom posledica i jednostavnosti primenjenih struktura podataka i može se donekle eliminisati oznakama za striktnost, upotrebom struktura podataka sa destruktivnim ažuriranjem i korišćenjem boljeg kompajlera. Ostaje međutim činjenica da su ovako konstruisani interpreteri samo idealizacija jezika sa kojima se obično radi i postavlja se pitanje da li će se doslednom primenom ovakvog postupka ikad moći realizovati interpreteri za jezike u široj upotrebi. Tako, na primer, pojednostavljeno viđenje procesa interpretacije ne dozvoljava upotrebu identifikatora pre nego što su on definiše, što je ograničenje koje se sve ređe pojavljuje u programskim jezicima.

Ovim ne želimo da sugerišemo besmislenost čitavog pristupa, u najmanju ruku on predstavlja verifikaciju teorije modularne denotacione semantike i može se koristiti u eksperimentalne svrhe. Raniji sistemi ovakve vrste zasnovani na denotacionoj semantici generisali su jezičke procesore oko 1000 puta sporije od ručno generisanih ([15]). U [15] je dat opis trenutnog stanja jednog generatora jezičkih procesora zasnovanog na akcionoj semantici. Ovaj generator (ACTRESS) daje jezičke procesore oko 70 puta sporije od ručno generisanih, ali autori predviđaju da se tim pristupom može postići generisanje jezičkih procesora koji su svega 2 puta neefikasniji od ručno generisanih. Apstraktan karakter modularne denotacione semantike je čini bliskom akcionoj semantici (a to je i eksplicitno pokazano u [127]), što znači da izgradnjom sistema baziranih na modularnoj denotacionoj semantici takođe mogu očekivati bolji rezultati.

Parcijalno izračunavanje predstavlja samo jednu od tehnika transformacije programa (program transformation). Ovaj postupak se ne mora izvoditi potpuno automatizovano već se može posmatrati kao sistematski pristup implementaciji programskih jezika. Pisanje modularnih interpretera postupkom opisanim u radu bi stoga predstavljalo samo polaznu fazu (fazu specifikacije) u sistematskom proučavanju implementacije različitih osobina programskih jezika. Sledeći korak predstavlja proučavanje veza između postupaka do kojih se u računarstvu došlo dugogodišnjim iskustvom u pisanju kompajlera s jedne strane, i specifikacije u modularnim interpreterima s druge strane. Takav pristup može doprineti i razvoju novih postupaka za implementaciju, a postojanje formalne specifikacije omogućava uspostavljanje preciznih kriterijuma optimalnosti.

Na kraju se možemo osvrnuti i na samu modularnu denotacionu semantiku koja je pokrenula niz radova o modularnim interpreterima, uključujući i naš pokušaj. Modularnost semantičkih komponenti se realizuje definisanjem interakcije između tih komponenti, što je najčešće rađeno pomoću liftinga. I sam Moggi je zaključio da nije moguće lifting sprovesti na uniforman način. Dok je Moggi u takvim slučajevima odustao od liftinga, radovi na modularnim interpreterima su ove kritične slučajeve identifikovali kao mesta na kojima se definiše interakcija između različitih semantičkih osobina jezika. To je mesto na kojem je doneti odluke koje utiču na konačnu semantiku celog jezika. To u isto vreme znači da je u opštem slučaju potrebno ispitati interakciju svake dve semantičke komponente, a broj takvih interakcija raste kvadratno sa brojem komponenti. To se konstatuje u [75], uz napomenu da, po svemu sudeći, broj semantičkih komponenti koje ima smisla razmatrati i nije tako velik. Čini se da ovakav zaključak relativizira polazni pojam modularnosti semantike. Naše

je mišljenje da rezultujuća specifikacija ipak ima prednosti u odnosu na globalnu specifikaciju svih mogućnosti odjednom, jer su interakcije pojedinačnih komponenti jasno lokalizovane, čime se dobilo na sistematičnosti i razumljivosti specifikacije.

4.5 Dalja proširenja

Semantičke komponente Komponenta za greške je jednostavna i ne omogućava obradu izuzetaka (ne implementira naredbu `catch`). Ovu mogućnost nije teško dodati, ali se onda javljaju novi problemi sa liftingom te operacije.

U [76] navodi se aksiomatizacija samo semantičke komponente za okruženja. Dosadašnji napori su, čini se, više bili usmereni na ispitivanje mogućnosti za realizaciju modularnosti nego na detaljno proučavanje pojedinačnih komponenti. Detaljno proučavanje aksiomatizacije i složenijih osobina ovih komponenti je ono što bi trebalo da opravda dobijenu apstrakciju. Ukoliko se ovi apstraktni tipovi pokažu kao pogodni za razumevanje značenja programa, rezultat bi bila teorija značenja programa nezavisna od konkretnog jezika, pa čak i od same denotacione semantike koja je poslužila za aksiomatizaciju. Apstrakcija je takođe bitna za implementaciju, pa se mogu očekivati upotrebljiviji alati nego u slučaju direktne primene denotacione semantike.

Još jedna mogućnost za dalje proučavanje je razvoj monade za modeliranje konkurentnih procesa. Ovaj problem nije rešen u [76], [26]. Pošto Moggi ([85]) koristi monad transformer za CCS (Calculus of Communicating Processes), trebalo bi ispitati da li postoje smetnje da se ova konstrukcija sprovede i u funkcionalnim programskim jezicima. Jedno rešenje zasnovano na tehnici zvanoj "resumptions" je dato u [127].

Efikasnost Kao što je obrazloženo u prehodnom delu, efikasnost nije bila osnovni cilj ovakvog pristupa. Značaj pristupa bi ipak bio mnogo veći ako bi se mogli zadavati i složeniji interpreteri koji bi prihvatali veće programe. To je moguće postići poboljšanjima na različitim nivoima.

Jedan od najvećih izvora neefikasnosti je leksički analizator. To je pomalo ironično s obzirom da je upravo ta faza bazirana na teorijski optimalnom postupku. Ovo predstavlja poseban problem s obzirom da veliki deo vremena rada jezičkih procesora odlazi upravo na leksičku analizu. Problem je rezultat uniformnosti specifikacije leksičkog analizatora pomoću regularnih izraza. Tako se identifikatori zadaju kao nizovi slova ili cifara, pri čemu se slova definišu kao alternativa velikog broja znakova. To generiše prevelik broj prelaza. Ovo je moguće prevazići na isti način kao u generatorima leksičkih analizatora: definisanjem intervala karaktera, upotrebom negacije, ili upotrebom specijalnih korisnički definisanih funkcija. Svi ovi pristupi su zasnovani na specijalizaciji opšteg leksičkog analizatora tako da radi sa ulaznim znakovima fiksiranog tipa, što i samo po sebi može doneti efikasniju implementaciju. Takođe je moguće koristiti dodatni korak preprocesiranja kojim se eliminišu komentari i praznine kako bi se skratio ulaz u modularno konstruisan leksički analizator. Konačno, s obzirom na jednostavnost i irelevantnost pitanja leksičke analize sa stanovišta semantike, moguće je odustati od pune modularnosti leksičkog analizatora i realizovati fiksiran leksički analizator sa opšteprihvaćenim konvencijama o predstavljanju identifikatora, brojeva, uz neku implementaciju komentara. Leksički analizator bi mogao da sadrži konvencije o zapisu

operatora, slično kao i o zapisu identifikatora. To bi omogućilo da leksički analizator ostane fiksan, a da sintaksni analizator dodatne operatore koje prihvata prepoznaje preko njihove reprezentacije u vidu niza znakova.

Ukoliko se pak ostane pri leksičkom analizatoru sa lenjom konstrukcijom stanja, može se na njegovom primeru dalje ispitivati memoizacija u Haskell-u. Zanimljivom se čini mogućnost za korisnika transparentne implementacije memoizacije pomoću funkcije `unsafePerformIO` kojom se zaobilazi referencijalno transparentno korišćenje IO monada.

Efikasnost se dalje može poboljšati upotrebom pogodnijih struktura podataka. Ovo je jednostavno realizovati jer su te strukture pretežno korišćene kao apstraktni tipovi u onim situacijama kada je pretpostavljeno da bi složenija implementacija u budućnosti bila od koristi. Posebnu mogućnost predstavljaju nizovi. Ako se program transformiše da koristi monade onda se nizovi mogu destruktivno ažurirati i time poboljšati performanse.

Konačno, pre bilo kakvog ozbiljnijeg razmatranja u pogledu efikasnosti uputno bi bilo preći na potpuni Haskell kompajler kao što je GHC (Glasgow Haskell Compiler). Hugs je namenjen samo za eksperimentisanje i efikasnost izvršavanja programa u njemu nije merilo performansi programa kompajliranih do izvršnog koda. GHC sadrži i alate za ispitivanje kritičnih delova koda koji mogu pomoći da se ne umanjuje čitljivost delova programa koji nisu od značaja za efikasnost.

Upotrebljivost Pored efikasnosti, na upotrebljivost interpretera utiče i način prijave grešaka tokom sintaksne i leksičke analize. Trenutno se greške prijavljuju pretežno upotrebom sistemske funkcije `error`. Adekvatniji način bi bila upotreba odgovarajućih domena u funkcijama ili primena kontinuacija u samom interpreteru. Obe ove mogućnosti se mogu realizovati pomoću monada, što će zahtevati restruktuiranje programa, ali će kasnije modifikacije biti jednostavnije. Ističemo da je ovo razmatranje prijave grešaka prilikom leksičke i sintaksne analize potpuno nezavisno od rada sa greškama koje se javljaju u toku interpretacije sintaksnih konstrukcija i koje se realizuju odgovarajućom semantičkom komponentom (monad transformerom) odgovornom za greške, odnosno kontinuacije. Konkretno, u tekućoj verziji su greške koje nastaju prilikom leksičke analize (npr. nepoznat znak u ulazu) ili prilikom leksičke analize (npr. višak zatvorenih zagrada) sistemske prirode i detektuju se izvan procesa interpretacije apstraktnog sintaksnog stabla, te se prijavljuju sa `error`. Greške prilikom provere tipova u toku izvršavanja se vrše kao sastavni deo interpretacije te se obrađuju odgovarajućim vrednostima domena algebre koja daje semantiku apstraktnom sintaksnom stablu.

Leksički analizator trenutno ne prijavljuje konflikte ukoliko postoje dve maksimalne lekseme u tekućem ulaznom nizu. Ova situacija označava preklapanje tokena i najsigurnije bi bilo prijaviti je kao grešku, dok se ovde proizvoljno bira jedna od leksema. Rešavanje ovog problema je od posebnog značaja u kontekstu modularnosti jer bi od dela za sklapanje komponenti trebalo očekivati da izvrši što više statičkih provera radi izbegavanja neočekivanog ponašanja rezultujućeg interpretera.

Kompajliranje Dok interpreteri predstavljaju prirodan način za specifikaciju semantike, kompajleri su osnovni oblik realizacije praktično upotrebljivih jezičkih procesora. Suština kompajlera je u razdvajanju akcija koje vrši interpreter na statički deo, koji postaje zadatak

kompajlera, i dinamički deo, koji izvršava kompajliran program. Ovaj proces se naziva razdvajanje prolaza (pass separation). Primena parcijalnog izračunavanja na definisanje kompajlera i kompajler-kompajlera data je u [60], dok je kompajliranje modularno specificiranih jezika opisano u [76] i [38]. Sistematski karakter definicije modularnih interpretera sugeriše ovakav pristup pisanju kompajlera jer on daje teorijsku motivaciju postupaka kompajliranja. Kakvi su njegovi praktični dometi ostaje da se ispita.

Složeniji jezici Semantičke komponente koje su ovde realizovane mogu se koristiti za opis suštine imperativnih i funkcionalnih jezika. Neka praktična pitanja, kao što je odvojeno kompajliranje, ostaju otvorena. Takođe se postavlja pitanje sintaksne analize složenijih konstrukcija. Tehnike zasnovane na specifikacije operatora se u principu mogu koristiti za zadavanje proizvoljnih prelaza stanja push-down automata, ali je pitanje koliko je takva specifikacija prirodna i koncizna. Čini se da bi najadekvatnija bila implementaija LALR(1) parsiranja sa specifikacijom prioriteta i rešavanja shift-reduce konflikata. To bi dozvolilo kako upotrebu deklaracija operatora, tako i upotrebu gramatika. Smatramo da je pri implementaciji složenijih oblika parsiranja najpogodnije prihvatiti implementaciju sa generisanjem Haskell koda, ali na način koji je transparentan za korisnika. To bi se moglo ostvariti definisanjem atributivnih gramatika koje opisuju vezu između apstraktnog stabla zasnovanog na algebarskom tipu podataka i konkretne sintakse. Prethodno je potrebno proučiti mogućnosti postojećih generatora parsera i sistema zasnovanim na atributivnim gramatikama.

Može se razmatrati proširenje leksičkog analizatora tako da implementira i neku vrstu konvencije o uvlačenju (layout rule). Opis jedne implementacije ovog pravila (offside rule) je dat u [53]).

Provera tipova Provera tipova predstavlja jedan od najvažnijih vidova semantičke analize prilikom kompajliranja. Ona se može relativno jednostavno ostvariti i u tekućoj implementaciji ako se koriste "aktivni konstruktori". To bi značilo da se parseru prilikom specifikacije ne prosleđuju jednostavno konstruktori apstraktnog stabla, već funkcije istog tipa kao i konstruktori koje prilikom konstrukcije stabla od njegovih delova vrše i statičku analizu, kao što je provera tipova. Ono što bi bilo još više u duhu modularnosti je primena više faza interpretacije. Prva faza interpretacije bi mogla raditi proveru tipova, a zatim stablo proslediti sledećoj fazi. Tako je moguće izbeći da se greške tipova otkriju tek prilikom izvršavanja programa, ali ne i potrebu za primenom operacija projekcije. Naime, trenutna šema apstraktne sintakse, urađena po uzoru na Duponcheel-ov rad ([23]) koristi jednosortnu algebru kao osnovu interpretacije, pa se svi tipovi realizovanog jezika moraju sabrati u jedan tip, da bi se po potrebi vršila projekcija i injekcija. Mogla bi se posmatrati generalizacija ovog pristupa na višesortne algebre kako bi se omogućila definicija apstraktnih sintaksnih stabala koja preciznije opisuju statičku semantiku jezika, ali se čini da u postojećem sistemu tipova nije moguće izraziti tipiziranu strukturu interpretiranih programa, pogotovo ne modularno. To bi trebalo detaljnije ispitati, a ukoliko to jeste slučaj može se razmišljati o mogućnosti upotrebe jezika sa zavisnim tipovima ([6]), ili upotrebi MetaML-a.

Primena višestrukog obilaska sintaksnog stabla se može koristiti i za eliminisanje sintaksnog šećera, čime se omogućava definisanje složenijih sintaksnih konstrukcija bez opterećivanja kasnijih faza kompajliranja.

Kada okruženje za specifikaciju modularnih interpretera postigne određeni nivo upotrebljivosti, dalje se može koristiti jednostavno kao alat za razvoj interpretera, kompajlera i drugih jezičkih procesora. Kao primer analize koja ima primenu u kompajliranju može se implementirati apstraktna interpretacija funkcionalnih programa, u cilju određivanja striktnosti funkcija ([63]), analize vremenske složenosti funkcija ([102]), ili specijalizacije i parcijalnog izračunavanja. Ovakva platforma bi bila pogodna i za razvoj sofisticiranih tehnika kompajliranja, jer potencijalna neefikasnost izvršavanja generisanog modularnog jezičkog procesora ne ograničava kvalitet koda koji jezički procesor može da generiše.

4.6 Zaključak

U ovom radu smo pokazali kako je u programskom jeziku Haskell moguće pisati interpretere čija se semantika, apstraktna sintaksa, konkretna sintaksa i leksika zadaju na *modularan način*. Modularnost rezultujućeg interpretera se ogleda u sledećem:

- različite osobine jezika (aritmetika, uslovni izrazi, lokalna imena, stanje, petlje, funkcije, nedeterminizam, obrada grešaka) zadaju se nezavisno jedna od druge;
- različite faze interpretacije (leksička, sintaksna, semantička analiza) zadaju se (za svaku osobinu jezika) nezavisno jedna od druge;
- opis osobina jezika je zadat nezavisno od univerzalnog jezgra koje implementira procese leksičke, sintaksne i semantičke analize.

Ovo je ostvareno odabirom odgovarajućih tehnika za realizaciju faza i struktura podataka jezičkog procesora:

- regularni izrazi za zadavanje leksike;
- konačni automati za realizaciju leksičkog analizatora;
- specifikacija operatora sa prioritetima za konkretnu sintaksu;
- prevođenje specifikacije operatora u prelaze push-down automata za sintaksnu analizu;
- apstraktna stabla za apstraktnu sintaksu;
- pojam algebre i sume algebri za interpretaciju apstraktnog sintaksnog stabla;
- podtipovi i monad transformeri za semantiku;
- lifting za specifikaciju interakcije između monad transformera.

Pri tome je modularnost koja se ogleda u stvarnom rasporedu delova specifikacije po datotekama od sekundarnog značaja, jer je to samo prividan i formalan oblik modularnosti. Ono što je suštinsko jeste mogućnost za kombinovanje delova u celinu. Programski jezik Haskell se po takvim mogućnostima posebno ističe, pre svega zahvaljujući:

- sofisticiranom sistemu tipova;

- funkcijama višeg reda;
- lenjom izračunavanju.

Pomenute mogućnosti dozvoljavaju definisanje vrlo opštih fragmenata programa, a posledica toga su veće mogućnosti za njihovo kombinovanje. Posebno ističemo značaj modularnosti semantike, jer se realizacijom neke semantičke osobine ujedno ukazuje i na, makar principijelnu, mogućnost da se određeni način programiranja (paradigma) podrži u čisto funkcionalnom programskom jeziku.

Sami moduli koji implementiraju apstraktne tipove podataka za specifikaciju interpretera ovim postupkom su realizovani za relativno kratko vreme zahvaljujući visokom nivou apstrakcije programskog jezika Haskell. Ovakva realizacija predstavlja dokaz da je iz modularnih specifikacija ove vrste moguće generisati jezičke procesore. Za praktičnu primenu ovog pristupa potrebno je povećati fleksibilnost u specifikaciji jezika i poboljšati efikasnost generisanih modularnih interpretera. Mišljenja smo da je moguće izvršiti poboljšanja u oba ova, međusobno suprotstavljena pravca. Za realizaciju ovih ciljeva predlažemo postupak generisanja koda iz specifikacije opisa komponenti. Smatramo da je pri tome veću pažnju potrebno posvetiti specifikacije semantike nego što je slučaj u kompajler-kompajlerima u dosadašnjoj upotrebi. To bi se moglo ostvariti ako bi se specifikacija zadavala u tipiziranom jeziku koji bi predstavljao proširenje Haskell-a (ili njegovog relevantnog podskupa) dodatnom sintaksom koja čini specifikaciju modularnih interpretera još konciznijom i preglednijom, a u isto vreme olakšava pronalaženje onih konstrukcija koje se mogu izračunati prilikom generisanja kompajlera. Drugim rečima, predlažemo upotrebu pre-procesora za funkcionalni programski jezik koji bi se ponašao kao program za parcijalno izračunavanje (partial evaluator) posebne namene. I pored velikog broja sistema zasnovanih na generisanju implementacija jezika iz opisa denotacione semantike, efikasan sistem koji bi bio zasnovan na modularnoj denotacionoj semantici specificiranoj u tipiziranom jeziku se, prema našem znanju, još nije pojavio.

Smatramo da je pisanje specifikacija visokog nivoa oblast od velikog značaja za primenu funkcionalnih programskih jezika. Tome u prilog govori ne samo implementacija jezika MetaML, već i programski jezik Mondrian, koji je blizak Haskell-u, a namenjen je sklapanju aplikacija napisanih u različitim jezicima od postojećih komponenti. U slične svrhe se može koristiti i sam Haskell ([66], [70]).

Literatura

[1] H. Aït-Kaci, J. Garrigue. *Label-Selective Lambda Calculus*, Research Report 31, Digital Equipment Corporation, Paris 1993.

[2] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Adison Wesley, 1986.

[3] P. B. Andrews, D. A. Miller, E. L. Choen, F. Phenning. Automating higher-order logic. *Contemporary Mathematics, Volume 29*, 1984.

[4] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[5] A. W. Appel. *Semantics-Directed Code Generation*, available online, 1984.

[6] L. Augustsson, M. Carlsson. *An exercise in dependent types: A well typed interpreter*, `www.cs.chalmers.se/augusts`.

[7] R. C. Backhouse. *An Exploration of the Bird-Meertend Formalism*, Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningem, 1988.

[8] R. Backhouse et al. *Generic Programming - An Introduction*, `www.win.tue.nl/~rolandb/`

[9] H. P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics,* 2nd edn. North-Holland, Amsterdam, 1984.

[10] H. P. Barendregt. Lambda Calculi with Types. In: S. Abramsky, D. M. Gabbay, T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, Oxford, 1992, pp.117–309

[11] C. Beierle. *Algebraic Implementations in an Integrated Software Development and Verification System*, PhD Thesis, University of Kaiserslautern, 1985.

[12] R. S. Bird. A Formal Development of an Efficient Supercombinator Compiler. *Science of Computer Programming*, 8(2):113-137, 1987.

[13] R. Bird, P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[14] R. Bird. *Introduction to functional programming using Haskell*, 2nd Edn. Prentice Hall, 1998.

[15] D. F. Brown, H. Moura, D. A. Watt. Actress: an Action Semantics Directed Compiler Generator. *Workshop on Compiler Construction, Paderborn, Germany*, 1992.

[16] N. G. de Bruijn. A Survey of the AUTOMATH project, in [41], pp.580–606.

[17] Z. Budimac, M. Ivanović, Z. Putnik, D. Tošić. *LISP kroz primere.* Univerzitet u Novom Sadu, 1994.

[18] Z. Budimac. *Prilog teoriji funkcionalnih jezika i implementaciji njihovih procesora.* Doktorska disertacija, Univerzitet u Novom Sadu, Prirodno-matematički fakultet, Institut za matematiku, 1994.

[19] N. Chomsky. On certain formal properties of grammars. *Information and Control 2 : 2*, 1959, 137–167.

[20] G. Cousineau, The Categorical Abstract Machine, In G. Huet, ed., *Logical Foundations of Functional Programming*, Addison-Wesley Publishing Company, Reading, MA, 1990, pp.25–46.

[21] N. Dershowitz, J. P. Jouannaud. Rewrite Systems. In J. van Leeuwen, ed. *Handbook of Theoretical Computer Science*, Elsevier Science Publishers, 1990.

[22] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT 8*, 1968, pp174–186.

[23] L. Duponcheel. *Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters*, `http://cs.ruu.nl/people/luc`, 1995.

[24] L. Duponcheel, E. Meijer. *On the expressive power of Constructor Classes.*, Research Report RUU, `www.cs.ruu.nl/erik`, 1994.

[25] L. Duponcheel, D. Swierstra. A case study in functional programming: generating efficient functional LR(1) parsers. `www.cs.ruu.nl/luc`

[26] D. Espinosa. *Semantic Lego*, PhD Thesis, Columbia University. `www.cs.columbia.edu`, 1995.

[27] A. J. Field, P. G. Harrison: *Functional Programming*, Addison-Wesley Publishers Ltd., 1988.

[28] R. Floyd. Assigning meanings to programs. In Schwartz, ed. *Proc. Symp. in Applied Math.*, 1967.

[29] J. Gallier. Typing untyped λ-terms, or reducibility strikes again! *Annals of Pure and Applied Logic* **91**, 1998, pp231–270.

[30] S. J. Garland, J. V. Guttag. An overview of LP, the Larch Prover. Proceedings of the Third International Conference on Rewriting Techniques and Applications, Chapel Hill, N.C., *Lecture Notes in Computer Science* 355, Springer-Verlag, 1989, pp.137–151.

[31] J. Gibbons, G. Jones. The Under-Appreciated Unfold, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Baltimore, 1998.

[32] J. A. Goguen et al. *Initial Algebra Semantics and Continuous Algebras*, JACM, Vol. 24, No. 1, January 1977, pp.68-95.

[33] D. I. Good. The proof of a distributed system in Gypsy. In M. J. Elphick, *Formal Specification - Proceedings of the Joint IBM/University of Newcastle-upon-Tyne Seminar*, pp.443–489.

[34] M. C. Gordon. *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[35] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Sun Microsystems, 1996.

[36] B. Hansen. *Brinch Hansen on Pascal Compilers.* Prentice-Hall, Inc. 1985.

[37] W. L. Harrison, S. N. Kamin. Metacomputation-based Compiler Architecture. `www.cs.uiuc.edu/harrison`, 1998.

[38] W. L. Harrison, S. N. Kamin. Modular Compilers Based on Monad Transformers. *IEEE Computer Society International Conference on Computer Languages.* Loyola University, Chicago, 1998.

[39] W. L. Harrison, S. N. Kamin. Compilation as Partial Evaluation of the Functor Category Semantics. `www.cs.uiuc.edu/harrison`, 1997.

[40] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming 22*, Elsevier Science B.V., 1994, pp.197-230

[41] J. R. Hindley, J. P. Seldin, eds. *To H. B. Curry: Essays on Combinatory Logic, lambda calculus and formalism.* Academic Press, 1986.

[42] R. Hinze. A Generic Programming Extension for Haskell. `www.informatik.uni-bonn.de/~ralf`, 1999.

[43] C. A. R. Hoare. An axiomatic base for computer programming. *Communications of the ACM*, 12, 1969.

[44] P. Hudak et al. Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, Volume 27, No.5, May 1992.

[45] P. Hudak, J. Peterson, J. Fasel. Gentle Introduction to Haskell. `www.haskell.org`

[46] P. Hudak, How to Have Your State and Munge it Too. Yale Research Report YALEU/DCS/RR-914, 1993.

[47] G. Huet. Cartesian closed categories and lambda-calculus. In G. Huet, ed., *Logical Foundations of Functional Programming*, Addison-Wesley Publishing Company, Reading, MA, 1990, pp.7–24.

[48] G. Huet. *Formal Structures for Computation and Deduction*. Notes for graduate level course at CMU, Pitsburg, May 1986.

[49] J. Hughes. Why Functional Programming Matters. *Computer Journal 32(2)*, 1989.

[50] J. Hughes, J. O'Donnell. Expressing and Reasoning About Non-deterministic Functional Programs. *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1989.

[51] J. Hughes. Generalising Monads to Arrows, *Fourth International Conference on Mathematics of Program Construction*, Baastad, Sweden, 15-17 June, 1998.

[52] G. Hutton. Fold and Unfold for Program Semantics. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Baltimore, 1998.

[53] G. Hutton. Higher-Order Functions for Parsing. *Journal of Functional Programming 2(3):323-343*, July 1992.

[54] J. Jeuring, P. Jansson. Polytypic Programming. In: J. Launchbury, E. Meijer, T. Sheard, eds. *Lecture Notes in Computing Science 1129, Advanced Functional Programming*, 1996, pp.68–114

[55] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Compiler Architecture*, volume 274 of *Lecture Notes in Computer Science* 154-173. Springer-Verlag, 1987

[56] S. C. Johnson. *Yacc - yet another compiler-compiler.* Technical Report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ.

[57] M. P. Jones. *The implementation of the Gofer functional programmins system.* Research Report YALEU/DCS/RR-1030, May 1994.

[58] M. P. Jones, J. C. Peterson. *Hugs98 User Manual, Revised version: September 1999*, `http://haskell.org/hugs`.

[59] M. P. Jones. Partial evaluation for Dictionary-free overloading. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-959, 1993.

[60] N. D. Jones, P. Sestoft, H. Sondergaard. An Experiment in Partial Evaluation: Generation of a Compiler Generator. In: J. P. Jouannaud, ed. *Rewriting Techniques and Applications*, Lecture Notes in Computer Science 202, Springer-Verlag, 1985.

[61] S. L. Peyton Jones, M. P. Jones, E. Meijer, *Type Classes: An Exploration of the Design Space*, `www.cs.uu.nl/erik`.

[62] S. L. Peyton Jones, E. Meijer, Henk: a typed intermediate language, `www.cs.uu.nl/erik`.

[63] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall International, 1987.

[64] S. L. Peyton Jones, D. R. Lester. *Implementing Functional Languages*, Prentice Hall, 1992.

[65] S. L. Peyton Jones, J. Hughes. *Haskell 98: A Non-strict, Purely Functional Language.* February 1999, language report available from `http://haskell.org/report`.

[66] S. L. Peyton Jones, E. Meijer, D. Leijer. Scripting COM Components in Haskell. *Proceedings of the 5th International Conference on Software Reuse*, Victoria, British Columbia, June 1998.

[67] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, *Journal of Functional Programming 2(2)*, April 1992, pp.127–202.

[68] S. Kamin. Standard ML as a Meta-Programming Language. `www.cs.uiuc.edu/kamin`, 1996.

[69] P. Koopman. R. Plasmeijer. Efficient Combinator Parsers.

[70] D. Leijen, E. Meijer. Domain Specific Embedded Compiler. `www.cs.uu.nl/erik`, 1999.

[71] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[72] J. E. Labra, J. M. Cueva, M. C. Luengo Diez. Language Prototyping using Modular Monadic Semantics. *3rd Latin-American Conference on Functional Programming*, 8-9, March, 1999, Recife - Brasil

[73] J. Launchbury, T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture, La Jolla, California,* 1995, pp. 314–323.

[74] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall International, Inc., 1981.

[75] S. Liang, P. Hudak. Modular denotational semantics for compiler construction. *ESOP'96: 6th European Symposium on Programming, Linkoping, Sweden.* Springer-Verlag, 1996.

[76] S. Liang. Modular Monadic Semantics and Compilation. PhD thesis, Yale University, 1998.

[77] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[78] D. Maćoš, Z. Budimac, M. Ivanović. An Interpreter of Extended λ-calculus using monads. *XI Conference on Applied Mathematics*, Novi Sad, 1997.

[79] R. Sz. Madarász, S. Crvenković. *Uvod u teoriju automata i formalnih jezika.* Univerzitet u Novom Sadu, 1995.

[80] I. A. Mason. *The Semantics of Destructive LISP.* Center for Study of Language and Information, Leland Stanford Junior University, 1986.

[81] E. Meijer, M. Fokkinga, R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire, in *Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)*, Cambridge, Massachuetts, 1991, pp.124–144.

[82] E. Meijer. Calculating Compilers. PhD Thesis, Nijmegen University, 1992.

[83] E. Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, 1964.

[84] G. J. Michaelson. *Interpreter prototypes from formal language definitions.* PhD Thesis, Heriot-Watt University, 1993.

[85] E. Moggi. An abstract view of programming languages. Technical Report, ECS-LFCS-90-113, University of Edinburgh. `theory.doc.ic.ac.uk`, 1990.

[86] P. D. Mosses. A tutorial on Action Semantics. `www.brics.dk/pdm`, 1996.

[87] P. D. Mosses. Theory and Practice of Action Semantics. `www.brics.dk/pdm`, 1996.

[88] H. Mössenböck, N. Wirth. The Programming Language Oberon-2. Technical Report, ETH Zürich, 1995.

[89] H. Mössenböck. Cocor/R – A Generator for Fast Compiler Front-Ends. Technical Report, ETH Zürich, 1990.

[90] J. Mountjoy. The Spineless Tagless G-machine, naturally. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, Baltimore, Maryland, USA, 1998.

[91] M. Oderski, D. Rabin, P. Hudak. Call by Name, Assignment, and the Lambda Calculus. *Proceedings of the 20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993.

[92] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[93] W. Olthoff. The Connection Between Applicative and Procedural Languages in an Integrated Software Development and Verification System. PhD Thesis, University of Kaiserslautern, 1987.

[94] Y. Onoue et al. A Calculational Fusion System HYLO, *IFIP 1996.* Chapman & Hall, 1996.

[95] N. Pippenger. Pure versus Impure LISP. *ACM Transactions on Programming Languages and Systems,* Vol. 19, No. 2, March 1997, pp.223–238.

[96] R. Plasmeijer, M. van Eekelen. *Clean Language Report, Version 1.3.*. `www.cs.kun.nl/~clean`, 1998.

[97] G. D. Plotkin. A Structural Approach to Operational Semantics. *Lecture Notes DAIMI FN-19*, Dept. of Computer Science, Unviersity of Arhus, 1981.

[98] A. Reid. Designing Data Structures. *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1989.

[99] D. E. Rydeheard, R. M. Burstall. *Computational Category Theory*. Prentice Hall, 1988.

[100] J. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *JACM 12, 23-41*, 1965.

[101] T. Rus et al. An Algebraic Language Processing Environment, `www.cs.uiowa.edu/~rus`, 1998.

[102] D. Sands. Complexity Analysis for a lazy, Higher-Oder Language. *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1989.

[103] D. S. Scott. Data types as lattices. *SIAM J. Comput. 5*, 1976, pp.522–587

[104] Z. Shao. The FLINT compiler system. Presentation at IFIP Working Group 2.8, 1996.

[105] T. Sheard. Type-directed, On-line, Partial Evaluator for a Polymorphic Language. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, 1997.

[106] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[107] A. Suarez. Compiling ML into CAM. In G. Huet, ed., *Logical Foundations of Functional Programming*, Addison-Wesley Publishing Company, Reading, MA, 1990, pp.47–73.

[108] S. D. Swierstra, L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, T. Sheard, eds. *Advanced Functional Programming*, volume 1129 of LNCS-Tutorial. Springer-Verlag, 1996, pp.184–207.

[109] S. D. Swierstra. Fast Error-Correcting Parser Combinators. Utrecht University, `www.cs.uu.nl`, 1999.

[110] L. Szarapka, Z. Budimac. The best of two worlds: Using memo functions to implement higher order functions. *XIII Conference on Applied Mathematics*, Novi Sad, 1998.

[111] S. Hu. *Osnovi opšte topologije*. Savremena administracija, Beograd, 1973.

[112] S. Thompson. *Haskell: the craft of functional programming*, 2nd Edn. Addison Wesley, 1999.

[113] R. D. Tennent. The Denotational Semantics of Programming Languages. *Communication of the ACM*, Volume 19, Number 8, August 1976.

[114] J. P. Tremblay, P. G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill Inc., 1985.

[115] P. Trinder, Referentially Transparent Database Languages. *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1989.

[116] D. A. Turner. Recursion Equations as a Programming Language. In J. Darlington, P. Henderson, D. A. Turner: *Functional Programming and its Applications.* Cambridge University Press, Cambridge, 1982.

[117] A. Voß. Algebraic Specifications in an Integrated Software Development and Verification System. PhD Thesis, University of Kaiserslautern, 1985.

[118] M. Odersky, P. Wadler, M. Wehr. A Second Look at Overloading. *7'th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.

[119] P. Wadler. How to make ad-hoc polymorphism less ad hoc, *Proceedings of the 16'th ACM Symposium on the Principles of Programming Languages*, Austin, Texas, January 1989.

[120] P. Wadler. How to Declare Imperative. In J. Loyd, ed. *International Logic Programming Symposium*, MIT Press, 1995.

[121] P. Wadler. Deforestation: Transforming programs to eliminate trees, *Theoretical Computer Science*, 73:231-248, 1990.

[122] P. Wadler. The Essence of Functional Programming. *19'th Annual Symposium on Principles of Programming Languages,* Albuquerque, New Mexico, January 1992.

[123] S. L. Peyton Jones, P. Wadler. Imperative Functional Programming. *ACM Symposium on Principles of Programing Languages (POPL)*, Charleston, January 1993, pp.71-84.

[124] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science* Volume 2, Cambridge University Press, 1992, pp.461–493.

[125] P. Wadler. Monads for functional programming. In J. Jeuring, E. Meijer, eds.: *Advanced Functional Programming*, Proceedings of the Bastad Spring School, May 1995, Springer-Verlag Lecture Notes in Computer Science 925, 1995.

[126] P. Wadler. How to replace failure by a list of successes: a method for exception handling, backtracking and pattern matching in lazy functional languages. In J. P. Jouannand, ed.: *Functional Programming Languages and Computer Architecture.* LNCS 201, 1985, pp.113–128.

[127] K. Wansbrough. A Modular Monadic Action Semantics. Masters Thesis, University of Auckland, New Zeland, 1997.

[128] E. Van Wyk. Domain Specific Meta Languages. Oxford University Computing Laboratory, `http://web.comlab.ox.ac.uk/oucl/`, 2000.

Biografija autora

Viktor Kunčak je rođen u Novom Sadu 1977. godine. Osnovnu školu je završio u Somboru 1992. godine sa odličnim uspehom (5.00). Gimnaziju "Veljko Petrović" u Somboru je završio 1996. godine sa odličnim uspehom 5.00 (maturski rad "Formalni sistem verovatnosne logike" je pisao iz matematike). Učestvovao je na republičkim, saveznim i međunarodnim tamičenjima učenika osnovnih i srednjih škola iz fizike, informatike, matematike, hemije. Osvojio je tri puta prvo i jednom treće mesto na saveznom takmičenju iz fizike kao i drugo mesto na saveznom takmičenju informatike. Dobio je pohvalu na međunarodnoj olimpijadi iz fizike u Oslu (Norveška) 1996. godine. Studije informatike je započeo 1996. godine i sve prethodne ispite je položio sa ocenom 10.

Sa računarima je u kontaktu od 1990. godine. Boravio je u informatičkom kampu Palić 1991. godine, a u informatičkom i matematičkom kampu Đerdap 1995. godine. Na seminarima istraživačke stanice Petnica je boravio 1991., 1993., 1994., 1995. i 1996. godine. Na letnjim seminarima u Petnici je realizovao i prezentovao radove "PLS, Programski jezik za pisanje simulacija" (objavljen u Petničkoj svesci 33: Radovi polaznika obrazovnih programa ISP u 1993. godini), zatim "TPL Transformacioni programski jezik", "TPL-2 Compiler-compiler", kao i "Formalni sistemi i primena". 1996. godine je boravio u Vajcmanovom institutu u Izraelu gde je radio na implementaciji rešavanja konturnog problema diferencijalnih jednačina multigrid postupkom (rad Janis Voigtländer, Viktor Kunčak: Developing a Multigrid Solver for Standing Wave Equation, štampan u zborniku radova učesnika 28. međunarodnog instituta Dr. Bessie F. Lawrence, 1996.).

Godine 1998. je primio nagradu za temat "Tvrđenje Erbrana i postupak rezolucije", a 1999. godine nagradu za temat "Rano otkrivanje zaglavljivanja". 1999. godine je primio nagradu Mileva Marić-Anštajn za uspeh u oblasti informatike.

Trenutno radi na pripremi rada o numeričkim reprezentacijama čisto funkcionalnih struktura podataka, primeni Tait-ovog metoda reducibilnosti za dokazivanje svojstava tipiziranog lambda računa i prebrojavanju kombinatornih konfiguracija "polimino".

Bibliografski podaci

UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj RBR	
Identifikacioni broj IBR	
Tip dokumentacije TD	Monografska dokumentacija
Tip zapisa TZ	Tekstualni štampani materijal
Vrsta rada VR	Diplomski rad
Autor AU	Viktor Kunčak
Mentor MN	dr Mirjana Ivanović
Naslov rada NR	Modularni interpreteri u Haskell-u
Jezik publikacije JP	srpski (latinica)
Jezik izvoda JI	srpski/engleski
Zemlja publikovanja ZP	SR Jugoslavija
Uže geografsko područje UGP	Vojvodina
Godina GO	2000.
Izdavač IZ	autorski reprint
Mesto i adresa MA	Novi Sad
Fizički opis rada (broj poglavlja / strana / lit.citata / slika / priloga) FO	(4, 119, 0, 1, 1)
Naučna oblast NO	Računarske nauke

Naučna disciplina ND	Funkcionalno programiranje
Predmet odbrane / Ključne reči PO	interpreter, modularnost, denotaciona semantika
UDK	
Čuva se ČU	
Važna napomena VN	
Izvod IZ	U radu je opisana realizacija skupa modula u čisto funkcionalnom programskom jeziku Haskell koji omogućavaju modularnu specifikaciju semantike, apstraktne sintakse, konkretne sintakse i leksike interpretera. Modularna specifikacija semantike je zasnovana na transformerima monada i liftingu. Modularna specifikacija apstrakne sintakse se zadaje realizacijom koncepta algebre i sume algebri. Modularna konkretna sintaksa je realizovana postupkom parsiranja operacija različitih prioriteta. Leksička analiza je realizovana pretvaranjem regularnih izraza u nedeterministički automat na osnovu kojeg se lenjom konstrukcijom prelaza i stanja generiše deterministički konačan automat. Rezultat je sistem u kojem je moguće nezavisno zadati specifikaciju različitih osobina jezika, kao što su aritmetika, uslovni izrazi, lokalne definicije, izuzeci, funkcije, petlje, nedeterminizam i stanje.
Datum prihvatanja teme od strane NN Veća Instituta za matematiku DP	Maj 2000.
Datum odbrane	8. Jun 2000.
Članovi komisije (naučni stepen / ime i prezime / fakultet) KO	
Predsednik	dr Zoran Budimac, vanredni profesor, Prirodno-matematički fakultet, Novi Sad
Član	dr Mirjana Ivanović, vanredni profesor, Prirodno-matematički fakultet, Novi Sad
Član	dr Đura Paunić, redovni profesor, Prirodno-matematički fakultet, Novi Sad

UNIVERSITY OF NOVI SAD
FACULTY OF NATURAL SCIENCES
KEY WORDS DOCUMENTATION

Accession number ANO	
Identification number INO	
Documentation type DT	Monograph documentation
Type of record TR	Textual printed material
Contents code CC	Graduation thesis
Author AU	Viktor Kunčak
Mentor MN	Dr. Mirjana Ivanović
Title TI	Modular interpreters in Haskell
Language of text LT	Serbian (Latin)
Language of abstract JI	Serbian/English
Country of publication CP	FR Yugoslavia
Location of publication LP	Vojvodina
Publication year PY	2000.
Publisher IZ	Author's reprint
Publisher place PP	Novi Sad
Physical description PD	(4, 119, 0, 1, 1)
Scientific field SD	Computer Science

Scientific discipline SD	Functional programming
Subject/Keywords SKW	interpreter, modularity, Denotational Semantics
UC	
Holding data HD	
Note N	
Abstract AB	The work describes an implementation of a framework for modular specification of semantics, abstract syntax, concrete syntax and lexical structure of interpreters. The framework consits of modules written in a non-strict, purely functional programming language Haskell. Modular semantics is based on monad transformers and lifting. Modular abstract syntax is specified using algebras and sum of algebras. Modular concrete semantics is implemented using a variation of operator precedence parsing. The implementation of lexical analysis is based on nondeterministic finite automaton generated from regular expressions. Lazy transition evaluation is used to create a deterministic finite automaton from nondeterministic finite automaton during lexical analysis. The resulting system permits independent specification of language features such as arithmetics, conditional expressions, local definitions, exceptions, functions, loops, nondeterminism, and state.
Accepted on the Scientific Board of the Institue of Mathematics AS	May 2000
Defended DB	June 8, 2000
President	Dr. Zoran Budimac, associate professor, Faculty of Sciences and Mathematics, Novi Sad
Member	Dr. Mirjana Ivanović, associate professor, Faculty of Sciences and Mathematics, Novi Sad
Member	Dr. Đura Paunić, full professor, Faculty of Sciences and Mathematics, Novi Sad

Dodatak: Komponente malog interpretera

Rezultujući sistem je realizovan u Haskell-u uz korišćenje proširenja koja dopušta Hugs interpreter, verzija od Februara 2000. Jezgro sistema čine moduli `RegExps` za leksičku analizu, `PrecPar` za sintaksnu analizu, `Glue` za spajanje komponenti u celinu, kao i `Ormers` koji realizuje monad transformere. Ovi moduli realizovani su na način opisan u delu 3. Da bismo istakli jednostavnost specifikacije interpretera u ovakvom okruženju, dajemo sada pregled 8 realizovanih komponenti modularnog interpretera. Mnoga od ograničenja na sintaksu i semantiku komponenti bi se mogla eliminisati onako kako je to opisano u delu 4.

Komponenta za celobrojnu aritmetiku

Ova komponenta realizuje 4 osnovne računske operacije kao i stepenovanje za brojeve tipa `Int`. Brojevi se zadaju kao konačni neprazni nizovi cifara, a operacije znacima `+`, `-`, `*`, `/` i `^`.

Konstante se interpretiraju pomoću funkcije `returnInj` koja datu vrednost ugrađuje u sumu tipova, a zatim od nje pravi monadu.

```
returnInj :: (Monad m, Subtype sub sup) => sub -> m sup
returnInj = return . inj
```

Operacije `+`, `-`, `*` i `^` se realizuju pomoću funkcije `lift2sub`.

```
lift2sub :: (ErrMonad String m, Subtype a a1, Subtype b b1, Subtype c c1)
               => (a -> b -> c) -> (m a1 -> m b1 -> m c1)
lift2sub f ma mb = do a <- mprj ma
                      b <- mprj mb
                      returnInj (f a b)
```

Ova funkcija višeg reda prihvata funkciju od 2 argumenta i od nje pravi funkciju koja je definisana nad monadama koje računaju vrednosti iz šireg domena. Njena definicija koristi i funkciju `mprj` koja vrši projekciju unutar monade, prijavljujući grešku ukoliko projekcija ne uspe.

```
mprj :: (ErrMonad String m, Subtype sub sup) => m sup -> m sub
mprj ma = do x <- ma
             case prj x of
               Just a  -> return a
               Nothing -> eThrow "Projection failed"
```

Operacija `/` se definiše posebno jer se prethodno proverava da li je delilac 0.

```
module Aritm where -- arithmetic component
import Subtypes
import Algebras
import RegExps
import PrecPar
import Ormers
-- Lexical analysis ------------------------------------------------------
data Token = N Int | Plus | Minus | Times | Divided | Power
           deriving Show
```

```
lexemes = [(pInt, makeInt),
           (rSym '+', \_ -> Plus),  (rSym '-', \_ -> Minus),
           (rSym '*', \_ -> Times), (rSym '/', \_ -> Divided),
           (rSym '^', \_ -> Power)]
pInt = digit <&> (rMany digit)        -- digit digit*
  where digit = rAnyOf "0123456789"
makeInt ds = N (foldl op 0 ds)
  where op n d = 10*n + (ord d - ord '0')
-- Syntax analysis -----------------------------------------------------
data Tree x = Const Int
            | Add x x | Sub x x
            | Mul x x | Div x x
            | Pow x x
par :: Token -> (Tree x -> x) -> ParsingItem b x
par (N x)   = literal  (Const x)
par Plus    = infixOpL 502 Add
par Minus   = infixOpL 502 Sub
par Times   = infixOpL 503 Mul
par Divided = infixOpL 503 Div
par Power   = infixOpR 504 Pow
-- Semantics -----------------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Const x    -> Const x
               Add x y    -> Add (f x) (f y)
               Sub x y    -> Sub (f x) (f y)
               Mul x y    -> Mul (f x) (f y)
               Div x y    -> Div (f x) (f y)
               Pow x y    -> Pow (f x) (f y)
instance (Subtype Int v, ErrMonad String m) => Algebra Tree (m v) where
  phi e = case e of
            (Const x)    -> returnInj x
            (Add xm ym)  -> lift2sub plus xm ym
            (Sub xm ym)  -> lift2sub minu xm ym
            (Mul xm ym)  -> lift2sub tims xm ym
            (Div xm ym)  -> do x <- mprj xm
                               y <- mprj ym
                               if y==0 then eThrow "Division by zero"
                                       else returnInj (divi x y)
            (Pow xm ym)  -> lift2sub pow xm ym
          where -- resolve arithmetic overloading
            plus, minu, tims, divi, pow :: Int -> Int -> Int
            plus = (+); minu = (-)
            tims = (*); divi = div
            pow  = (^)
-- end of module Aritm
```

Komponenta za uslovne izraze

Ova komponenta realizuje logičke operacije, konstante `True` i `False`, operacije poređenja celih brojeva, kao i funkciju `if`. Interpretacija se i ovde vrši uglavnom sa `lift2sub`. Posebno se tretiraju samo konstante i naredba `if`.

```
module Condit where -- conditionals and relations
import Subtypes
import Algebras
import RegExps
import PrecPar
import Ormers
-- Lexical analysis ------------------------------------------------------
data Token = B Bool
           | OrT  | AndT
           | EqT  | LtT   | GtT   | LeqT | GeqT
           | IfT  | ThenT | ElseT
           deriving Show
lexemes = [(rLit "True",  \_ -> B True),
           (rLit "False", \_ -> B False),
           (rLit "&", \_ ->  AndT),
           (rLit "|", \_ ->  OrT),
           (rLit "=", \_ ->  EqT),
           (rLit "<", \_ ->  LtT),
           (rLit ">", \_ ->  GtT),
           (rLit "<=", \_ -> LeqT),
           (rLit ">=", \_ -> GeqT),
           (rLit "if", \_ -> IfT),
           (rLit "then", \_ -> ThenT),
           (rLit "else", \_ -> ElseT)
           ]
-- Syntax analysis -------------------------------------------------------
data Tree x = BConst Bool
            | And x x | Or x x
            | Equ x x | Lth x x | Gth x x       -- =   <   >
            | LEQ x x | GEQ x x                 -- <=  >=
            | If x x x
par (B x) = literal  (BConst x)
par OrT   = infixOpL 405 Or
par AndT  = infixOpL 406 And
par EqT   = infixOp  410 Equ
par LtT   = infixOp  410 Lth
par GtT   = infixOp  410 Gth
par LeqT  = infixOp  410 LEQ
par GeqT  = infixOp  410 GEQ
par IfT   = ternary  400 If
par ThenT = marker   400
par ElseT = marker   400
```

```
-- Semantics analysis ------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               BConst x   -> BConst x
               And x y    -> And (f x) (f y)
               Or  x y    -> Or  (f x) (f y)
               Equ x y    -> Equ (f x) (f y)
               Lth x y    -> Lth (f x) (f y)
               Gth x y    -> Gth (f x) (f y)
               LEQ x y    -> LEQ (f x) (f y)
               GEQ x y    -> GEQ (f x) (f y)
               If x y z   -> If  (f x) (f y) (f z)
  -- interpretation
instance (Subtype Int v, Subtype Bool v, ErrMonad String m)
      => Algebra Tree (m v) where
  phi e = case e of
            (BConst x)   -> returnInj x
            (And xm ym)  -> lift2sub (&&) xm ym
            (Or xm ym)   -> lift2sub (||) xm ym
            (Equ xm ym)  -> lift2sub  eq  xm ym
            (Lth xm ym)  -> lift2sub  ls  xm ym
            (Gth xm ym)  -> lift2sub  gr  xm ym
            (LEQ xm ym)  -> lift2sub  le  xm ym
            (GEQ xm ym)  -> lift2sub  ge  xm ym
            (If cm tm em) -> do c <- mprj cm
                                if c then tm
                                     else em
         where eq, ls, gr, le, ge :: Int -> Int -> Bool
               eq = (==); ls = (<); gr = (>);
               le = (<=); ge = (>=)
-- end of module Condit
```

Komponenta za lokalna imena

Ova komponenta realizuje lokalne definicije. Lokalna definicija se uvodi znakom `:` iza kojeg sledi ime identifikatora. Ovakva deklaracija se tretira kao unarni operator koji čini definiciju vidljivu u izrazu na koji se primenjuje. Identifikatori su u ovoj verziji pojedinačna slova iz liste `letters`. Lokalne definicije se interpretiraju pomoću operacija `rdEnv` i `inEnv` iz konstruktora `EnvT`.

```
module Enviro where -- environments for local names
import Ormers
import Subtypes
import Algebras
import RegExps
import PrecPar
import FMaps
-- Lexical analysis --------------------------------------------------
```

```
data Token = IdentT String | IdentDefT String
           deriving Show
lexemes = [(ident, \s -> IdentT s),
           (rSym ':' <&> ident, \s -> IdentDefT (tail s))]
letters = "XYZABCDEFMNPQ"
ident = rAnyOf letters
-- Syntax analysis -----------------------------------------------------
data Tree x = Ident String
            | Let String x x
            deriving Show
par (IdentT s)    = literal (Ident s)
par (IdentDefT s) = prefixBinOp 200 (Let s)
-- Semantics analysis --------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Ident s     -> Ident s
               Let s e1 e2 -> Let s (f e1) (f e2)
  -- interpretation
type Loc = String
instance (ErrMonad String m, EnvMonad (FMap Loc (m v)) m)
       => Algebra Tree (m v) where
  phi e = case e of
            Ident s -> getValue s
            Let s def exp -> do env <- rdEnv
                                let newEnv = mInsert (s,inEnv newEnv def) env
                                inEnv newEnv exp
getValue :: (ErrMonad String m, EnvMonad (FMap String (m v)) m)
               => String -> m v
getValue s = do env <- rdEnv
                extract (mVal env s)
                        ("Unknown variable "++s++" in "++show env)
-- end of module Enviro
```

Komponenta za funkcije

Ova komponenta imlementira lambda izraze i primenu funkcija postupkom "call by value". Lambda apstrakcija se uvodi unarnim operatorom koji počinje znakom `&` iza kojeg sledi identifikator. Identifikatori koji se pri tome koriste su pojedinačna slova iz liste `letters`. Primena jednog izraza na drugi se zadaje operatorom `@`.

```
module Funcs where -- functions as lambda abstractions
import Ormers
import Subtypes
import Algebras
import Refl
import RegExps
import PrecPar
```

```
import FMaps
-- Lexical analysis ------------------------------------------------------
data Token = IdentT String | LambdaT String | AppT
           deriving Show
lexemes = [(ident, \s -> IdentT s),
           (lident, \s -> LambdaT (tail s)),
           (rSym '@', \_ -> AppT)]
letters = "xyzabcdefmnpq"  -- better than 'Out of control stack'
ident = rAnyOf letters
lident = rSym '&' <&> ident
-- Syntax analysis -------------------------------------------------------
data Tree x = Ident String
            | App x x
            | Lambda String x
            deriving Show
par (IdentT s)    = literal  (Ident s)
par AppT          = infixOpL 800 App
par (LambdaT s)   = prefixOp 300 (Lambda s)
-- Semantics analysis ----------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Ident s    -> Ident s
               App x1 x2  -> App (f x1) (f x2)
               Lambda s x -> Lambda s (f x)
  -- interpretation
instance (ErrMonad String m, Reflexive v m, EnvMonad (FMap String (m v)) m)
           => Algebra Tree (m v) where
  phi e = case e of
            Ident s     -> getValue s
            App mf mx   -> rApply mf mx
            Lambda s mx -> do env <- rdEnv
                              rMkFun (\rv->inEnv (mInsert (s,rv) env) mx)
getValue :: (ErrMonad String m, EnvMonad (FMap String (m v)) m)
               => String -> m v
getValue s = do env <- rdEnv
                extract (mVal env s) "Unknown variable"
-- end of module Funcs
```

Ove operacije se implementiraju pomoću operacija `rApply` i `rMkFun` iz klase `Reflexive`. Rezultat je "call by value" semantika zbog sledeće definicije.

```
instance (EnvMonad (FMap String (m dom)) m, ErrMonad String m,
          Subtype (m dom -> m dom) dom)
       => Reflexive dom m where
  rApply mf mx = do (f :: m a -> m a) <- mprj mf
                    x <- mx          -- call by value
                    f (return x)
```

Komponenta za izuzetke

Ova komponenta ilustruje mogućnost upotrebe kontinuacija, ovde realizovanih u `ContT` transformeru. Uvodi ključnu reč `try` koja se tretira kao prefiksna binarna operacija. Ova operacija očekuje dve lambda apstrakcije kao svoje argumente: jedna predstavlja program koji se izvršava, a druga funkciju za obradu greške. Argument prve lambda apstrakcije služi za prekid normalnog izvršavanja programa. Ukoliko se izvrši njegova primena na neku vrednost `msg`, tada se rezultat čitave konstrukcije `try` dobija primenom druge lambda apstrakcije na vrednost `msg`.

```
module Excep where -- exceptions via continuations
import Subtypes
import Algebras
import RegExps
import PrecPar
import Ormers
import Refl
-- Lexical analysis ----------------------------------------------------
data Token = TryT | CatchT
           deriving Show
lexemes = [(rLit "try",  \_ -> TryT),
           (rLit "catch", \_ -> CatchT)]
-- Syntax analysis -----------------------------------------------------
data Tree x = Try x x

par TryT   = prefixBinOp 350 Try
par CatchT = marker      350
-- Semantics analysis --------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Try x y -> Try (f x) (f y)
  -- interpretation
instance (ContMonad m, Reflexive v m) => Algebra Tree (m v) where
  phi e = case e of
            Try (mf :: m v) (mg :: m v) ->
              do (f :: m v -> m v) <- mprj mf
                 (g :: m v -> m v) <- mprj mg
                 callcc (\(k :: a -> m v) -> f (returnInj (\mmsg ->
                           (do (handled :: v) <- g mmsg
                               (k handled) :: m v))))
-- end of module Excep
```

Interpretacija operacije `try` se izvodi pomoću `callcc`. Njena definicija u lambda računu koja odgovara onoj u deklaraciji instance klase `Algebra` bila bi

$$\texttt{try}\ f\ g = \text{callcc}(\lambda k.f(k \circ g))$$

gde je $k \circ g = \lambda x.k(gx)$. Definicija u Haskell-u je nešto složenija zbog načina reprezentacije funkcija kao podtipa `Fun` vrednosti `Value` koji predstavlja funkcije tipa `Compute Value -> Compute Value`.

Komponenta za petlje

Ova jednostavna komponenta uvodi naredbu `while` kao prefiksni binarni operator koji izvršava drugi argument dok god je rezultat izračunavanja prvog `True`.

```
module Loops where    -- structured loops
import Ormers
import Subtypes
import Algebras
import RegExps
import PrecPar
import FMaps
-- Lexical analysis ----------------------------------------------------
data Token = WhileT
           deriving Show
lexemes = [(rLit "while", \_ -> WhileT)]
-- Syntax analysis -----------------------------------------------------
data Tree x = While x x
par WhileT = prefixBinOp 305 While
-- Semantic analysis ---------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               While x y -> While (f x) (f y)
instance (Subtype Bool v, ErrMonad String m) => Algebra Tree (m v) where
  phi e = case e of
            While mc ms -> p where
              p = do c <- mprj mc
                     if c then do _ <- ms
                                  p
                          else returnInj False
-- end of module Loops
```

Komponenta za nedeterminizam

Ova komponenta omogućava nedeterminističko računanje. Pomoću ključne reči `fail` se zadaje proces računanja koji ne uspeva, a pomoću operatora `#` se zadaju alternativni rezultati izraza koji dovode do nedeterminizma. Unarna operacija `when` vraća `fail` ako je rezultat izraza `False`. To omogućava da se eliminišu neželjene grane u bektreku.

```
module Nondet where -- nondeterministic computations
import Ormers
import Subtypes
import Algebras
import RegExps
import PrecPar
-- Lexical analysis ----------------------------------------------------
data Token = FailT | ParT | WhenT
           deriving Show
```

```
lexemes = [(rLit "fail", \_ -> FailT),
           (rSym '#',    \_ -> ParT),
           (rLit "when", \_ -> WhenT)]
           -- when e p = if e then p else fail
-- Syntax analysis -----------------------------------------------------
data Tree x = Fail
            | Par x x
            | When x
par FailT = literal  Fail
par ParT  = infixOpL 300 Par
par WhenT = prefixOp 310 When
-- Semantics analysis --------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Fail     -> Fail
               Par  x y -> Par  (f x) (f y)
               When x   -> When (f x)
  -- interpretation
instance (Subtype Bool v, ErrMonad String m, NondetMonad m)
       => Algebra Tree (m v) where
  phi e = case e of
            Fail       -> ndfail
            Par  xm ym -> ndmerge [xm,ym]
            When xm    -> do c <- mprj xm
                             if c then returnInj True
                                  else ndfail
-- end of module Nondet
```

Komponenta za stanje

Ova komponenta omogućava rad sa promenljivim kao u imperativnom jeziku. Promenljive se u ovoj verziji označavaju početnim malim slovom `s` iza kojeg sledi cifra od 0 do 5. Vrednost promenljive `si` se menja izrazom oblika `si:= e`, a sadržaj promenljive se čita sa `?si`. Uvodi se i binarni operator sekvence "`;`" koji ignoriše rezultat (ali ne i bočni efekat) prvog argumenta i vraća rezultat drugog argumenta.

```
module State where -- state component for assignable store
import Ormers
import Subtypes
import Algebras
import RegExps
import PrecPar
import FMaps
-- Lexical analysis ----------------------------------------------------
data Token = FetchT String | StoreT String | SeqT
           deriving Show
lexemes = [(storeVar <&> rLit ":=", StoreT . init . init),
```

```
          (rLit "?" <&> storeVar,  FetchT . tail),
          (rLit ";",  \_ -> SeqT)]
storeVar = rSym 's' <&> rAnyOf "012345"
-- Syntax analysis ------------------------------------------------------
data Tree x = Fetch String
            | Store String x
            | Seq x x
par (FetchT s) = literal (Fetch s)
par (StoreT s) = prefixOp 340 (Store s)
par SeqT       = infixOpL 300 Seq
-- Semantics analysis ---------------------------------------------------
instance Functor Tree where
  fmap f e = case e of
               Fetch s   -> Fetch s
               Store s x -> Store s (f x)
               Seq   x y -> Seq (f x) (f y)
  -- interpretation
instance (StateMonad (FMap String dom) m, ErrMonad String m)
       => Algebra Tree (m dom) where
  phi e = case e of
            Fetch s     -> do mem <- fetch
                              case mVal mem s of
                                Nothing -> eThrow ("Unknown store var "++s)
                                Just v  -> return v
            Store s xm  -> do x <- xm
                              mem <- fetch
                              store (mInsert (s,x) mem)
                              return x
            Seq xm ym   -> do _ <- xm
                              ym
-- end of module State
```