UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA U NOVOM SADU

Alen Arslanagić

TIPSKI SISTEM SA RESTRIKCIJAMA ZA ODREĐIVANJE TIPOVA ML JEZIKA SA IMPLEMENTACIJOM U HASKELL-U

MASTER RAD

Novi Sad, 2017.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографски рад
Тип записа, **ТЗ**:		Штампа
Врста рада, **ВР**:		Мастер рад
Аутор, **АУ**:		Ален Арсланагић
Ментор, **МН**:		др Силвиа Гхилезан
Наслов рада, **НР**:		Типски систем са рестрикцијама за одређивање типова *ML* језика са имплементацијом у Хаскелу
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски/Енглески
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2017.
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Факултет Техничких Наука (ФТН), Д. Обрадовића 6, 21000 Нови Сад
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		7/67/23/0/0/0/1
Научна област, **НО**:		Математика
Научна дисциплина, **НД**:		Примењена математика
Предметна одредница/Кључне речи, **ПО**:		Крипкеове семантике, интуиционистичка логика, ламбда рачун
УДК		
Чува се, **ЧУ**:		Библиотека ФТН, Д. Обрадовића 6, 21000 Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Типски системи са рестрикцијама представљају генерализацију Дамас-Милнеровог типског система који се налази у основи *ML* програмског језика. *HM(X)* представља фамилију типских система параметризованих са синтаксом и интерпретацијом рестрикција. Ови системи су погодни за одређивање типова програмских језика. Рестрикције као међу-језик омогућавају модуларну презентацију одређивања типова. У овом раду смо представили поменуте системе и модуларну имплементацију одређивања типова у програмском језику Хаскел.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		27.10.2017.
Чланови комисије, **КО**:	Председник:	др Тибор Лукић
	Члан:	др Срђан Попов
	Члан, ментор:	др Силвиа Гхилезан

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic type	
Type of record, **TR**:		Printed text	
Contents code, **CC**:		Master thesis	
Author, **AU**:		Alen Arslanagić	
Mentor, **MN**:		Silvia Ghilezan, PhD	
Title, **TI**:		Type system with constraints for ML type inference with implementation in Haskell	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian/English	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2017.	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		7/67/23/0/0/0/1	
Scientific field, **SF**:		Mathematics	
Scientific discipline, **SD**:		Applied mathematics	
Subject/Key words, **S**/**KW**:		Damas-Milner type system, type inference, ML	
UC			
Holding data, **HD**:		Library of the Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad	
Note, **N**:			
Abstract, **AB**:		Type systems with constraints are generalization of Damas-Milner type system implementation was part of the type system of the programming language ML. HM(X) represents a family of constraint-based type sustems parametrized with respect to the syntax and interpretation of constraints. These systems are designed for type inference. Constraints as intermediate representation offer modular representation of type inference. In this thesis we presented Damas-Milner and HM(X) type systems and modular type inference implementation in programming language Haskell.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:		27. October 2017.	
Defended Board, **DB**:	President:	Tibor Lukić, PhD	
	Member:	Srđan Popov, PhD	Menthor's sign
	Member, Mentor:	Silvia Ghilezan, PhD	

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА МАСТЕР РАД	Датум:

(Податке уноси предметни наставник - ментор)

Студијски програм:	Математика у техници
Руководилац студијског програма:	Проф. др Маја Недовић

Студент:	Ален Арсланагић	Број индекса:	V1 2/2015
Област:	Примењена математика		
Ментор:	Проф. др Силвиа Гхилезан		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА МАСТЕР РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ МАСТЕР РАДА:

ТИПСКИ СИСТЕМ СА РЕСТРИКЦИЈАМА ЗА ОДРЕЂИВАЊЕ ТИПОВА
”ML” ЈЕЗИКА СА ИМПЛЕМЕНТАЦИЈОМ У ХАСКЕЛУ

ТЕКСТ ЗАДАТКА:

Задатак мастер рада је дати преглед типских система са рестрикцијама погодних за одређивање типова језика из “ML” фамилије и функционалну имплементацију приказаних теоријских резултата. Потребно је прво дефинисати типски систем “ML” програмског језика и његове карактеристике, рестрикције, а затим и генерализацију овог система у виду “HM(X)” типског система. За потребе имплементације потребно је дати основне појмове теорије категорија. На крају потребно је приказати имплементацију програмског решења.

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

Predgovor

Teorija tipova je originalno predložena od strane *Russel*-a početkom dvadesetog veka da bi je kasnije dalje razvili *Ramsey*, *Chwiste*, *Church* i drugi. Teorija je prvobitno nastala kao rešenje *Russel*-ovog paradoksa u teoriji skupova. Osnovna ideja je napuštanje principa da se bilo koji predikat ili funkcija može primeniti na bilo koji objekat. To je postignuto time što se objekti razlikuju po svojoj funkcionalnosti. Razlikuju su osnovni objekti, funkcije nad objektima, funkcije koje preslikavaju osnovne objekte na osnovne objekte, funkcije koje preslikavaju funkcije na funkcije itd. Funkcionalnosti objekta su opisane u tipu.

Pojam teorija tipova se danas najčešće odnosi na tipove u okviru nekog formalnog sistema prezapisivanja. Najpoznatiji primer je svakako tipizirani λ-račun koji je uveo *Church* kao rešenje *Kleene-Rosser* paradoksa. Pored λ-računa postoje druge teorije tipova kao što je račun konstrukcija koji je razvio *Thierry Coquand* i čije nadogradnje se koriste kao osnova za interaktivni dokazivač *Coq*.

Iako su teorija tipova i tipski sistemi nastali kao teorijski rezultat u potrazi za osnovama univerzalnog jezika matematike, pronašli su veoma važnu primenu u računarskim naukama. Međutim osim praktične primene, što je mnogo značnije, omogućili su otkriće veze odnosno korespodencije između matematičkih dokaza i računarskih programa odnosno algoritama. Ovo je čuvena *Curry-Howard*-ova korespodencija. Korespodencija govori o tome da svaka propozicija u logici ima svoj korespodentni tip u programskom jeziku i obrnuto, zbog toga je ovaj nivo korespodencije poznat pod sloganom: *propozicije kao tipovi*. Sledeći nivo jeste da za svaki dokaz date propozicije postoji program sa korespodentnim tipom, što je poznato kao: *dokazi kao programi*. Ova korespodencija je omogućila razvoj inteaktivnih ili polu-automatskih dokazivača kao sto su *Agda*, *Coq*, *Automath* i drugi.

Sa stanovišta računarskih nauka i praktičnog primene, osnovna uloga tipskog sistema je prevencija grešaka u izvršavanju programa. Tipizirani programski jezici koriste teorijske rezultate tipskih sistema. Što je tipski sistem složeniji, programski jezik je eskpresivniji odnosno "moćniji". Međutim, to ima cenu u pogledu navođenja tipova u programima što ume da bude veoma nepraktično. Cilj jeste automatsko određivanje tipova sa minimalnim navođenjem ili bez navođenja tipova u programima. Međutim, složeni tipski sistemi obično ne dozvoljavaju algoritme za automatsko određivanje. Jedan od najpogodnijih tipskih sistem za funkcionalne programski jezici koji su bazirani na λ-računu jeste Sistem F. Sistem F predstavlja tipizirani λ-račun

drugog reda. Nezavisno su ga otkrili matematičar *Jean-Yves Girard* (1972.) i *John C. Reynolds* (1974.). Neformalno, Sistem F omogućava veoma fleksibilan polimorfizam za funkcionalne programske jezike. Međutim, pokazano je da je određivanje tipova za Sistem F neodlucivo. ML predstavlja minimalno smanjenje izražajnosti jezika tako da određivanje tipova bude odlučivo. U ovom radu bavićemo se određivanjem tipova za ML i skorijim teorijskim rezultatima u pogledu tipskih sistema koji su dizajnirani za određivanje tipova.

U prvom delu rada "Tipski sistem sa restrikcijama" predstavljen je prvo Damas-Milnerov tipski sistem koji je u osnovi tipova ML jezika i varijacije tog sistema. Zatim su predstavljene restrikcije koje omogućavaju modularnu prezentaciju algoritma. Na kraju ovog dela predstavljen je skorašnji rezultat HM(X) tipski sistem koji restrikcije koristi u pravilima kao kontekst zaključivanja.

U drugom delu rada "Modularno određivanje tipova ML-jezika u Haskell-u" predstavljena je implementacija teorijskih rezultata prvog dela u Haskell-u. Na početku dela je dat sažet pregled Teorije kategorije čiji rezultati omogućavaju "čisto" funkcionalnu implementaciju u Haskell-u. U nastavku je dat opis ključnih delova implementacije.

Želim da se zahvalim kolegi Novaku Boškovu od koga sam prvi put čuo za funkcionalno programiranje i koleginici Simoni Kašterović na pomoći tokom rada na tezi.

Želim da izrazim zahvalnost profesoru i predsedniku komisije dr Tiboru Lukiću i profesoru dr Srđanu Popovu na razumevanju i profesionalnom pristupu.

Posebnu zahvalnost dugujem mentorki Prof. dr Silvii Ghilezan na upoznavanju sa svetom matematičke logike i teorijskog računarstva, na poverenju, na pomoći na mom putu učenja i istraživanja i na prilici za saradnju.

Iznad svega zelim da se zahvalim svojim roditeljima, babi i dedi na bezuslovnoj ljubavi i podršci u svakoj mojoj odluci.

Sadržaj

Uvod

Pojam tipa je dobro poznat u softverskom inženjerstvu. Većina programskih jezika ima tipove koje koristi i tretira na različite načine. Ovakvi jezici nazivaju se *tipizirani*, sa druge strane nalaze se *netipizirani* jezici. Uloga tipova u svim tipiziranim programskim jezicima je uglavnom ista - tipovi imaju ulogu specifikacije očekivanog ponašanja programa koja je deo programa i koju prevodilac (eng. compiler) na neki način treba da sledi. Iako je uloga slična, razlika u implementaciji i značaju tipova kod različitih programskih jezika je velika. Zato i postoje različite kategorizacije programskih jezika u odnosu na tipskih sistem kao što su statički i dinamički, nominalni i strukturalni itd.

Sa jedne strane nalaze se programski jezici kao što su C, $C++$. Tipovi kod ovih jezika služe da opišu *oblik* podataka. Ovo su jezici nižeg nivoa, i tip je uglavnom u direktnoj vezi sa manipulacijom memorije. Takođe, jezik kao što je C je *slabo* tipiziran, odnosno vrednosti mogu biti deklarisane sa jednim tipom, a zatim da se interpretiraju kao vrednosti nekog drugog tipa, ovo omogućava *aritmetika sa pokazivačima* i konkretnije *casting*. Kao što i ime kaže, kod ovakvih jezika tipovi se ne tretiraju veoma striktno.

Međutim, C je programski jezik nižeg nivoa kod koga su performanse izvršavanja programa veoma važne i napravljen je kompromis sa bezbednošću koju donose složeniji tipski sistemi. Provere tipova u toku izvršavanja koje su potrebne da bi se postigla bezbednost ponekad se smatraju skupim. Odnosno, bezbednost programa ima cenu. Dakle, za programski jezik kažemo da je *slabo tipiziran* ako prevodilac ne može da detektuje sve nebezbedne operacije. Jezici koji predstavljaju naslednike C kao što su $C++$ i $Java$ razvijaju se u smeru jačeg tipiziranja.

Sa druge strane postoje programski jezici sa sofisticiranim tipskim sistemima, kao što su npr. $Scala$, $Haskell$. I iz ugla korisnika može delovati da su sličnosti između npr. tipova u C i tipova *Haskell* beznačajne. Složenost tipskih sistema u ovim jezicima može nagovestiti da se ovi sistemi ne razvijaju samo u kontekstu softverskog inzenjerstva. Štaviše, tipski sistem ili *teorija tipova* podrazumeva mnogo šire polje u matematičkoj logici i filozofiji. Tipski sistemi koji su implementirani u jezicima su samo praktična primena nekih od rezultata razvijene teorije tipova.

Ukratko rečeno, *uloga tipskih sistema je garancija osobine programa da nemaju greške prilikom izvrsavanja.* Kada ova osobina važi za sve programe koji mogu biti izraženi na određenom programskom jeziku, onda za taj jezik kažemo da je *valjan u pogledu tipova ili tipski valjan (eng. type sound).* Ako želimo da konstruišemo takav sistem i da tvrdimo da određene osobine važe za njega, to svakako nagoveštava da je za to potreban matematički aparat. Tipski sistem mora biti pažljivo dizajniran i analiziran kako bi se moglo tvrditi da je valjan. Ovo sve predstavlja motive za razvijanje preciznih formalnih metoda. Formalizacija tipskih sistema zahteva preciznu notaciju i mehanizam za dokazivanje određenih osobina. Ponekad ove formalizacije mogu delovati veoma apstraktno, ili čak i nepotrebno složeno. Međutim, motivacija je za sve ista i veoma pragmatična, i sve apstrakcije koje se prave su ćesto neophodne i donose prednosti kao što je npr. jednostavnije dokazivanje.

Formalizacije i dokazivanje osobina tipskih sistema je prvi korak, drugi je implementacija koja često predstavlja problem za sebe. Razlog za to se može naći u tome što se tipski sistemi često ne mogu formulisati tako da nedvosmisleno određuju implementaciju, iako se tokom dizajniranja ima na umu potencijalna implementacija i određeni elementi se mogu prilagoditi tome.

Tipski sistemi nisu jedina formalna metoda koja obezbeđuje da se sistem ponaša u skladu sa specifikacijom. Postoje i veoma sofistiricarni sistemi kao što su Horova logika, modalna logika, denotaciona semantika itd. Treba napomenuti da odsustvo tipskog sistema ne znači da su jezici nebezbedni. Netipizirani jezici proveravaju programe i postižu bezbednost na druge načine. Na primer neki od njih proveravaju da li je pristup nizu dozvoljen, da li su sve operacije deljenja dozvoljene itd. Ovaj proces provere naziva se *dinamička provera.*

Deo I

Tipski sistem sa restrikcijama

U ovom delu biće predstavljen tipski jezik sa restrikcijama koji je nastao za potrebe određivanja tipova. Pretežno se oslanja na rad Rémy-ija i Pottier-ija prezentovan u [1]. Prvo će biti prikazan tipski sistem ML-a odnosno Damas-Milnerov sistem. Zatim će biti opisane restrikcije koje su razvijene kao među-jezik za određivanje tipova. Biće date restrikcije za tipizirani λ-račun, a zatim i njihovo proširenje za ML jezik. Na kraju je prikazan HM(X) sistem koji predstavlja uopštenje Damas-Milnerovog sistema koji uključuje direktno restrikcije kao deo sistema.

Glava 1

Damas-Milnerov sistem

Pojam *ML* se prvi put pojavio tokom kasnih sedamdesetih godina prošlog veka. Tada se taj pojam odnosio na funkcionalni programski jezik opšte namene. Robin Milner i drugi su razvili ML kao meta jezik (eng. meta-language) *LCF* dokazivaca teorema (otuda potiče i ime). Danas pojam ML ima dvostruko značenje. Iz semantičkog pogleda, ML je funkcionalni programski jezik opšte namene koji sadrži funkcije prve klase, strukture podata zasnovanih na proizvodima i sumama, reference, rukovanje izuzecima, automatsko upravljanje memorijom i *call-by-value* semantiku. Ovaj pogled obuhvata i *Standard ML* [2] i *Caml* (Leroy, 2000) [3] familiju programskih jezika.

Iz ugla teorije tipova, ML se odnosi na određenu granu sistema tipova zasnovanih na tipiziranom λ-računu.

ML tipski sistem je prvi definisao Milner (1978.) [4]. Verzija koja će biti data u ovom radu je definicija koju su nekoliko godina kasnije (1982.) [5] predložili Damas i Milner. Ova definicija data je kolekcijom pravila tipiziranja pomocu kojih se tvrđenje zaključuje induktivno. U nastavku ćemo Damas-Milnerov tipski sistem označavati sa *DM tipski sistem*.

U literaturi DM tipski sistem ima različito tumačenje - smatra se ili proširenjem tipiziranog λ-računu ili podskupom polimorfnog Sistema F.

DM tipski sistem predstavlja restrikciju polimorfizma tako da se izbegnu poteškoće vezane za određivanje tipova u Sistemu F. Izrazi tipiziranog λ-računa su prošireni primitivnom **let** deklaracijom.

```
let x = a1 in a2
```

Predstavićemo ulogu **let** deklaracije koja je opisana u [6]. Uloga **let** deklaracije je izbegavanje duplikacije koda izdvajanjem nekoliko pojavljivanja istog podizraza a_1 u izrazu forme $[\ x \to a_1]a_2$. Deklaracija **let** je samo drugačija sintaksna forma za β-redeks:

$$(\lambda x.a_2)\ a_1$$

i nema nikakvu novu semantičku ulogu. Međutim, u DM sistemu **let** deklaracija je primitiva ili objekat prve klase. DM sistem se može formulisati i samo na osnovu čistog λ-računa tako što bi se β-redeksi tipizirali kao što se **let** konstrukcija tipizira. Međutim, za pisanje programa i radi preglednosti koda prirodnije je koristiti deklarativnu konstrukciju kao što je **let**. Ukratko, uloga **let** konstrukcije je isključivo praktična.

Što se tiče tipiziranja β-redeksa korisno je spomenuti dva pristupa koji su predloženi za λ-račun. Prvi pristup koristi invarijantu određivanje tipova pod β-redukcijom odnosno redukciju subjekta (eng. subject reduction). Znamo da ako u nekom okruženju Γ važi $\Gamma \vdash (\lambda x.a_2)\ a_1 : \tau$ i ako $(\lambda x.a_2)\ a_1 \rightarrow_\beta [x \rightarrow a_1]\ a_2$ onda važi $\Gamma \vdash [x \rightarrow a_1]\ a_2 : \tau$. Zatim se problem svodi na određivanje tipa β-normalne forme. Međutim, dokazano je da algoritam ne postoji zato što je problem odlučivanja da li se može odrediti tip za dati term polu-odlučiv [7].

Drugi pristup takođe koristi redukciju subjekta. Zasnovan je na proširenju pojma tipa funkcije tako da podrži i apstrakcije u kojima se argument koristi polimorfno. U ovom sistemu, tipovi funkcija imaju formu:

$$[\tau_1, ..., \tau_n] \rightarrow \tau$$

.

Ovaj tip označava funkcije koje vraćaju term tipa τ za dati term koji ima jedan od tipova $\tau_1 ... \tau_n$. Međutim, autori su pokazali da je određivanje tipova i za ovaj sistem neodlučivo.

Slično kao i za β-redeks, **let**-normalna forma se dobija primenjivanjem sledećeg pravila prezapisivanja u bilo kom kontekstu:

$$\text{let } x = a_1 \text{ in } a_2 \longrightarrow a_1; [\ x \rightarrow a_1]a_2$$

Na početku sekvence nalazi se a_1 da bi se obezbedila dobra tipiziranost terma a_1 u patološkim slučajevima kada se x ne pojavljuje slobodno u a_2. Ukoliko dobra tipiziranost zahteva da se x uvek pojavljuje slobodno u a_2 onda se može koristiti sledeće pravilo prezapisivanja:

$$\text{let } x = a_1 \text{ in } a_2 \longrightarrow [\ x \rightarrow a_1]a_2$$

Stanovište da se DM tipski sistem posmatra kao proširenje tipiziranog λ-računu podržan je time da se **let** može posmatrati samo kao mehanizam za tekstualno proširenje. Međutim, ovo stanovište ne uspeva da izrazi svojstvo postojanja opštih tipova koje poseduje DM tipski sistem. Intuitivno, zatvoren izraz ima tip τ ako i samo ako njegova let-normalna forma ima tip τ u tipiziranom λ-računu. Ovakva intuicija vodi ka naivnim algoritimima za proveru i određivanje tipova. Međutim, ovi algoritmi imaju eksponencijalnu kompleksnost.

U nastavku biće data direktna prezentacija Damas i Milnerovog sistema tipova koja ne uključuje let-normal formu. Ova prezentacija je praktična zato što vodi ka efikasnom algoritmu za određivanje tipova.

1.1 Implicitna prezentacija

Kao što je prethodno navedeno, osnovni jezik DM tipskog sistem je λ-račun sa dodatnom **let** konstrukcijom. Termi su u implicitnom obliku i dati su sa:

$$a ::= x \mid c \mid \lambda x.a \mid \text{let } x = a \text{ in } a \mid . \ ..$$

Jezik tipova je između tipiziranog λ -računa i Sistema F. U okviru DM sistema nalaze se kategorije za tipove i sheme tipova. Sintaksa tipova je kao i u tipiziranom λ - računu, a sheme tipova predstavljaju polimorfne tipove:

$$\tau ::= \alpha \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall \alpha.\sigma$$

Tipovi DM sistema se mogu smatrati restrikcijom odnosno podskupom tipova Sistema F. Primarna kategorija su sheme tipova, a tipovi su njihov podskup. Sheme tipovi su polimorfni tipovi kod kojih se kvantifikatori nalaze na preneks poziciji tako da su manje izražajni u odnosu na polimorfne tipove Sistema F.

DM kontekst tipiziranja vezuje promenljive programa za sheme tipova. U implicitnoj verziji DM sistema promenljive tipova se koriste implicitno i ne deklarišu se u kontekstu. Međutim, u ovom radu data je ekvivalentna prezentacija gde se tipske promenljive ekspilicitno deklarišu u Γ.

$$\text{IDM-VAR}\ \ \Gamma \vdash x : \Gamma(x) \qquad \text{IDM-CST}\ \ \Gamma \vdash c : \Delta(c)$$

$$\text{IDM-ABS}\ \ \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.a : \tau_0 \to \tau} \qquad \text{IDM-APP}\ \ \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\text{IDM-LET}\ \ \frac{\Gamma \vdash a_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash a_2 : \sigma_2}{\Gamma \vdash \textbf{let } x = a_1 \textbf{ in } a_2 : \sigma_2} \qquad \text{IDM-GEN}\ \ \frac{\Gamma, \alpha \vdash a : \sigma}{\Gamma \vdash a : \forall \alpha.\sigma} \qquad \text{IDM-INST}\ \ \frac{\Gamma \vdash a : \forall \alpha.\sigma}{\Gamma \vdash a : [\alpha \mapsto \tau]\sigma}$$

Slika 1.1: DM pravila tipiziranja

Na slici 1.1 data je standardna prezentacija **DM** sistema koja nije sintaksno usmerena. Pravilo IDM-VAR omogućava pristupanje shemi tipa za identifikator x iz okruženja. Pravilo IDM-LET odslikava operacionu semantiku u kojoj se tokom izvršenja programa u **let** $x = a_1$ **in** a_2 identifikator x vezuje za izraz a_1 u okruženju pre evualuiranja izraza t_2. Pravilo IDM-CST pristupa shemi tipa za konstantu c iz

okruženja za konstante Δ. Pravilo za apstrakciju IDM-ABS takođe proširuje okruženje, ali za razliku od pravila IDM-LET sa monotipom τ_0. U pravilu IDM-ABS premisa zahteva da telo funkcije bude valjano tipizirano pod uslovom da svako slobodno pojavljivanje x u a ima isti tip τ_0. Pravilo IDM-GEN od tipa pravi shemu tipa pomoću univerzalnog kvantifikatora nad skupom tipskih promenljivih koje se ne pojavljuju slobodno u okruženju. Sa druge strane, pravila IDM-INST specijalizuje shemu tipa za arbitrarni tip i dobija se jedna od instanci.

Ovde se vidi osnovno ograničenje koje DM sistem uvodi u odnosu na Sistem F. U sintaksi samo identifikatori koje uvodi **let** konstrukcija mogu imati polimorfne tipove odnosno sheme tipove koji su podskup polimorfnih tipova. Sa druge strane, identifikator λ apstrakcije može imati samo monotip. Ovo je ključan detalj ML jezika. Ukoliko bi se dozvolilo da argument apstrakcije ima i shemu tipa σ onda bi se mogao izgraditi tip oblika $\sigma \to \tau$. Međutim, tip ovog oblika ne mora pripadati sintaksi DM tipova zato što to može biti polimorfni tip, ali se univerzalni kvantifikator ne mora nalaziti na preneks poziciji. Na primer, na ovaj način može da se izgradi tip $(\forall\alpha.\alpha \to \alpha) \to bool$. Onda univerzalni kvantifikator može se javiti na bilo kojem nivou u tipu i tako se dobija implicitna verzija Sistema F. Ova dizajnerska odluka pravi značajnu razliku pošto je određivanje tipova u implicitnom Sistemu F neodlučivo, dok je određivanje tipova u ML sistemu odlučivo. Na primer, izraz **let** $f = \lambda z.z$ **in** $(f\ 0, f\ \mathbf{true})$ je valjano tipiziran i prihvaćen u DM sistemu. Sa druge strane, semantički ekvivalentni izraz $\lambda f.(f0, f\ \mathbf{true})$ ne može da se tipizira. Da bi izraz $\lambda f.(f\ 0, f\ \mathbf{true})$ mogao da se tipizira f mora istovremeno da dozvoljava tipove $int \to \tau_1$ i $bool \to \tau_2$ odnosno trebalo bi da bude polimorfan. Međutim, pošto je pod λ apstrakcijom provera tipa ista kao i u tipiziranom λ računu gde f mora imati monotip i zato ovaj izraz ne može da se tipizira. Sa druge strane, u implicintnom Sistemu F oba mogu da se tipiziraju i da im se odredi npr. polimorfni tip $\forall\alpha.\alpha \to \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha, z : \alpha \vdash z : \alpha}\,\text{VAR}}{\alpha \vdash \lambda z.z : \alpha \to \alpha}\,\text{ABS}}{\varnothing \vdash \lambda z.z : \forall\alpha.\alpha \to \alpha}\,\text{GEN} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}\,\text{VAR}}{\Gamma \vdash : \mathbf{int} \times \mathbf{int}}\,\text{INST}}{\Gamma \vdash f\ 0 : \mathbf{int}}\,\text{APP} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}\,\text{VAR}}{\Gamma \vdash f : \mathbf{bool} \times \mathbf{bool}}\,\text{INST}}{\Gamma \vdash f\ \mathbf{true} : \mathbf{bool}}\,\text{APP}}{\Gamma \vdash (f\ 0, f\ \mathbf{true}) : \mathbf{int} \times \mathbf{bool}}\,\text{PAIR}}{\varnothing \vdash \mathbf{let} f = \lambda z.z\ \mathbf{in}\ (f\ 0, f\ \mathbf{true}) : \mathbf{int} \times \mathbf{bool}}\,\text{LET}$$

Dat je primer provere tipa za izraz $\Gamma \vdash (f, f\ \mathbf{true}) : \mathbf{int} \times \mathbf{bool}$.

1.2 Eksplicitna prezentacija

Iako se implicitna verzija koristi u programima, eksplicitna prezentacija DM sistema je pogodnija za dokaze. Može se odrediti podskup terma Sistema F koji je

nakon brisanja tipova ekvivalentan implicitnom DM sistemu. Ovaj podskup terma dat je sledećom gramatikom:

$$M ::= x \mid c \mid \lambda x : \tau.M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid \text{let } x : \sigma = M \text{ in } M$$

gde τ i σ nisu proizvoljni tipovi Sistema F, već tipovi koji prate ograničenja data u DM sistemu: τ su monotipovi, a σ su sheme tipova. Ekspliticni DM koji je dat navedenom gramatikom ćemo označiti sa eDM.

DM sistem je restrikcija Sistema F, odavde sledi da ako $\Gamma \vdash_{eDM} M : \sigma$ onda $\Gamma \vdash_F M : \sigma$. Za razliku od implicitne verzije iDM kojim smo se do sad bavili, termi u eksplicitnom eDM sistemu imaju jedinstveno stablo tipiziranja i jedinstvene tipove, kao i u eksplicitnom Sistemu F.

Međutim, drugi smer ne važi - term M može biti i sintaksno u DM sistemu i imati tip σ valjano formiran u eDM sistemu tako da $\Gamma \vdash_F M : \sigma$ važi, ali da $\Gamma \vdash_{eDM} M : \sigma$ ne važi.

Da bismo ovo videli na primeru, dovoljno je pronaći term M koji ima shemu tipa σ, ali takav da se za njegovo tipiziranje koristi pravilo Sistema F koje je restriktovano u DM sistemu. Očigledna restrikcija DM sistema je kod pravila tipiziranja apstrakcije gde i argument i telo funkcije mogu imati samo monotip. Ako pogledamo sintaksu eDM sistema vidimo da se ne može konstruisati apstrakcija koja ima polimorfni argument, ali može apstrakcija koja ima polimorfno telo. Odnosno, ne postoji ograničenje u sintaksi koja ovo onemogućuje. Koristeći ovo, za term M možemo uzeti sledeći term $(\lambda x : \tau_0.\Lambda\alpha.\lambda y : \alpha.y)\ M_0$ gde $\Gamma \vdash M_0 : \tau_0$. 'Međutim, tip celog izraza pripada sintaksi tipova DM sistema, odnosno $\Gamma \vdash M : \forall\alpha.\alpha \to \alpha$. Kao što se iz konstrukcije i vidi, ovaj term može da se tipizira samo u Sistemu F, ali ne može u DM sistemu - pravila za tipiziranje apstrakcije pretpostavlja polimorfno telo funkcije što je nemoguće tipizirati u DM sistemu. Ukratko, važi $M \in eDM$ i $\Gamma \vdash_F : M : \sigma$, ali ne važi $\Gamma \vdash_{eDM} : \sigma$.

Sintaksno usmerena prezentacija Eksplicitni eDM termi imaju jedinstveno stablo tipiziranja i jedinstvene tipove - kao i u Sistemu F. eDM sistem je sintaksno usmeren: ne postoji izbor za upotrebu pravila tipiziranja odnosno za svaki term tj. svaki korak se uvek samo jedno pravilo može primeniti. Sa druge strane, implicitni iDM termi nemaju jedinstven tip, već mogu imati nekoliko različitih tipova i takođe nekoliko stabala tipiziranja. U iDM sistemu postoji izbor oko upotrebe pravila, konkretno pravila za generalizaciju i instanciranje tipskih promenljivih, pošto mesta za upotrebu ovih pravila nisu specifikovana u sintaksi terma.

Kao što je već rečeno, za programe je pogodnija implicitna prezentacija poput iDM sistema i algoritam za određivanje tipova umesto eksplicitne prezentacije. Za određivanje tipova neophodna je sintaksno usmerena prezentacija koju nažalost ne poseduje iDM. Odnosno, potrebna je sintaksno usmerena implicitna verzija DM sistema. Zapravo, uz određena ograničenja, moguće je konstruisati ovakav sistem.

U nastavku je dat implicitni sistem DM sistema u kom je stablo tipiziranja u potpunosti određeno termom i time je jedinstveno.

Sa xDM je označen podskup eksplicitnih terma i DM sistema dat je sledećom gramatikom:

$$N ::= \Lambda\vec{\alpha}.Q$$
$$Q ::= x\ \vec{\tau} \mid Q\ Q \mid \lambda x:\tau.Q \mid \text{let } x:\sigma = N \text{ in } Q$$

Zahtev je da su sve tipske promenljive u potpunosti instancirane, odnosno da u izrazu $x\ \vec{\tau}$ arnost $\vec{\tau}$ bude jednaka arnosti sheme tipa $\forall\vec{\alpha}.\tau$ promenljive x. Vidimo da Q termi mogu imati samo monotip τ, dok termi N imaju sheme tipova. Na ovaj način, određeno je mesto apstrakcija i aplikacija tipova - eksplicitna apstrakcija tipova može se naći samo u parametru **let** konstrucije.

$$\text{XML-TABS}\ \frac{\Gamma, \vec{\alpha} \vdash Q : \tau}{\Gamma \vdash \Lambda\vec{\alpha} : \forall\vec{\alpha}.\tau} \qquad \text{XML-ABS}\ \frac{\Gamma, x:\tau_0 \vdash Q : \tau}{\Gamma \vdash Q.a : \tau_0 \to \tau} \qquad \text{XML-APP}\ \frac{\Gamma \vdash Q_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash Q_2 : \tau_2}{\Gamma \vdash Q_1\ Q_2 : \tau_1}$$

$$\text{XML-LET-GEN}\ \frac{\Gamma, \vec{\alpha} \vdash Q_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\tau_1 \vdash Q_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x : \forall\vec{\alpha}.\tau_1 = \Lambda\vec{\alpha}.Q_1\ \textbf{in}\ Q_2 : \tau_2} \qquad \text{XML-VAR-INST}\ \frac{\forall\vec{\alpha}.\tau = \Gamma(x)}{\Gamma \vdash x\ \vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

$$\text{XML-CST-INST}\ \frac{\forall\vec{\alpha}.\tau = \Delta(c)}{\Gamma \vdash c\ \vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

Slika 1.2: Pravila tipiziranja za xDM

Po konstrukciji xDM termi su sintaksno podskup eDM terma. Iz toga i pravila tipiziranja se dobijaju restrikcijom pravila tipiziranja za xDM i data su 1.2 iz toga takođe važi da ako $\Gamma \vdash_{xDM} M : \sigma$ onda i $\Gamma \vdash_{eDM} M : \sigma$.

Želimo da pokažemo da svaki term M koji ima valjan tip u eDM može da se mapira na term N koji ima valjan tip u xDM i istu implicitnu formu nakon brisanja tipova. Drugim rečima, da se svaki program u eDM sistemu može prevesti u program u xDM, odnosno da imaju istu izražajnu moć. Pitanje je kako se može promeniti anotacija tipova terma M tako da se dobije term N koji sintaksno pripada i koji je valjano tipiziran u xDM sistemu.

Na slici 1.3 su data pravila normalizacije. Normalizacija ima dve uloge. Prva je η-ekspanzija svake promenljive u skladu sa arnošću sheme tipa pravilom NORM-VAR. Ovo osigurava da svako pojavljivanje tipske promenljive bude u potpunosti specijalizovano. Druga uloga je ι redukcija pomoću pravila NORM-TAPP - elaborirani termi ne sadrže ι -redekse.

Kao što je navedeno u gramatici xDM sistema, telo **let** konstrukcije ima monotip, dok u eDM ima shemu tipa. η ekspanzija tipskih promenljivih i ι redukcija omogućavaju da se u telu **let** konstrukcije i od sheme tipa dobije monotip na taj način što se tipovi

$$\text{NORM-VAR}\ \frac{\forall\vec{\alpha}.\tau = \Gamma(x)}{\Gamma \vdash x : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.x\ \vec{\alpha}} \qquad \text{NORM-TABS}\ \frac{\Gamma, \alpha \vdash M : \sigma \Rightarrow N}{\Gamma \vdash \Lambda.M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}$$

$$\text{NORM-TAPP}\ \frac{\Gamma \vdash M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}{\Gamma \vdash M\ \tau : [\alpha \mapsto \tau]\sigma \Rightarrow [\alpha \mapsto \tau]N} \qquad \text{NORM-CST}\ \frac{\forall\vec{\alpha}.\tau = \Delta(c)}{\Gamma \vdash c : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.c\ \vec{\alpha}}$$

$$\text{NORM-LET}\ \frac{\Gamma \vdash M_1 : \sigma_1 \Rightarrow N_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.Q \qquad \vec{\alpha} \neq N_1, \sigma_1}{\Gamma \vdash \textbf{let}\ x : \sigma_1 = M_1 \textbf{in} M_2 : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.\textbf{let}\ x : \sigma_1 = N_1 \textbf{in}\ Q}$$

$$\text{NORM-APP}\ \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \Rightarrow Q_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow Q_2}{\Gamma \vdash M_1\ M_2 : \tau_1 \Rightarrow Q_1\ Q_2} \qquad \text{NORM-ABS}\ \frac{\Gamma, x : \tau_0 \vdash M : \tau \Rightarrow Q}{\Gamma \vdash x : \tau_0.M : \tau_0 \to \tau \Rightarrow \lambda x : \tau_0.Q}$$

Slika 1.3: Normalizacija

u potpunosti specijalizuju, a veznici iz terma prelaze ispred **let** konstrukcije. Na primeru 1.4 je to ilustrovano. Prikazana je normalizacija terma koji ima polimorfan term u telu **let**-a.

$$\text{NORM-LET}\ \frac{\varnothing \vdash M : \forall\alpha\beta.\tau \Rightarrow N \qquad \dfrac{\dfrac{\forall\alpha\beta.\tau = \Gamma(x)}{x : \forall\alpha\beta.\tau \vdash x : \forall\alpha\beta.\tau \Rightarrow \Lambda\alpha\beta.x\ \alpha\ \beta}\ \text{NORM-VAR}}{x : \forall\alpha\beta.\tau \vdash x\ int : \forall\beta.\tau[\alpha \mapsto int] \Rightarrow \Lambda\beta.x\ int\ \alpha}\ \text{NORM-TAPP}}{\varnothing \vdash \textbf{let}\ x : \forall\alpha\beta.\tau = M\ \textbf{in}\ x\ int : \forall\beta.\tau[\alpha \mapsto int] \Rightarrow \Lambda\beta.\textbf{let}\ x : \forall\alpha\beta.\tau = M\ \textbf{in}\ x\ int\ \beta}$$

Slika 1.4: Primer normalizacije

gde je $\tau = (\alpha \to \beta) \to \alpha \to \beta$.

To se i uklapa u pogled po kom tipovi služe kao predikat za prihvatanje programa - sve što je ograničeno u xDM sintaksi u odnosu na eDM sintaksu vezano je za tipove (mesta aplikacija i apstrakcije tipskih promenljivih), što znači da ne bi trebalo da se menja izražajnost jezika. xDM se može konstruisati ograničenjem skupa terma eDM sistema tako da se izbegne mogućnost izbora za mesto tipske apstrakcije i tipske aplikacije.

Principijelni tip i tipiziranje Za definiciju principijelnog tipa ili tipiziranja potrebno je prvo definisati relaciju instance. Jedna od mogućnosti koja se najčešće

koristi je sintaksna definicija data sa:

$$\textsc{inst-gen}\ \frac{\vec{\beta} \neq \forall \vec{\alpha}.\tau}{\forall \bar{\alpha}.\tau \leq \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}].\tau}$$

Međutim, ovako definisana relacija ne uspeva da obuhvati tipove koji bi trebalo da budu relaciji. Da bi sistem imao karakteristiku principijelnog tipiziranja relacija instance bi trebalo da bude što veća. Jedna od definicija u ovom smeru je:

Definicija 1.2.1. *Tipiziranje σ_1 je opštije od tipiziranja σ_2 ako i samo ako svaki term koji dozvoljava σ_1 dozvoljava i σ_2.*

Ova definicija se koristi kako bi se pokazalo da neki sistemi nemaju principijelno tipiziranje bez obzira na konkretnu definiciju instance.

Razlikuju se pojmovi principijelnog tipiziranje i principijelnog tipa. Tipiziranje za izraz M je par Γ, τ tako da važi $\Gamma \vdash M : \tau$. Principijelno tipiziranje je par Γ, τ tako da su sva ostala tipiziranje koje dozvoljava M instance od Γ, τ. Slabija karakteristika tipskog sistema je postojanje principijelnog tipa gde se za dato okruženje Γ i izraz M postoji tip τ tako da su svi ostali tipovi za M instance od τ.

Sistem tipiziranog λ-računa ima principijelno tipiziranje, i za definiciju relacije se može koristiti INST-GEN. Sistem F nema principijelno tipiziranje i određivanje tipova je neodlučivo.

DM tipski sistem nema karakteristiku principijelnog tipiziranja, ali se odlikuje postojanjem principijelnog tipa i odlučivog određivanja tipova. U sledećem poglavlju dat algoritam koji je omogućila ova karakteristika DM sistema.

Glava 2

Određivanje tipova ML jezika

Originalni algoritam za određivanje tipova ML jezika je Milnerov čuveni algoritam predstavljen u radu 1978. $\mathcal{W}$ ili varijacija $\mathcal{J}$. Za algoritam pogledati [4].

Milnerov algoritam nije lako čitljiv. Pre svega zato što je dat u monadičkom stilu i zato što nije modularan - odnosno zato što se prepliću faze generisanja i rešavanja restrikcija. Iz tih razloga, razvijene su restrikcije kao među-jezik i modularan algoritam za određivanje tipova. Princip određivanja tipova je isti kao u originalnom Milnerovom algoritmu, ali su restrikcije i dodatna teorija omogućile jasniju prezentaciju i modularnu implementaciju.

2.1 Određivanje tipova tipiziranog λ-računa

Tipizirani λ-račun je sintaksno-usmeren tipski sistem - term koji nije anotiran tipovima određuje izomorfno stablo tipiziranja gde su tipovi promenljive. Odnosno, term određuje familiju stabla tipiziranja parametrizovanu tipskim promenljivama. Ove tipske promenljive nisu međusobno nezavisne, stablo tipiziranja uspostavlja restrikcije nad tipskim promenljivama.

Određivanje tipova tipiziranog λ-računa se može razdvojiti u fazu *generisanja restrikcija* i *rešavanje restrikcija*. Generisanje restrikcija prati pravila tipiziranja λ-računa. Za rešavanje jednačina nad tipskim promenljivama koristi se algoritam za *unifikaciju prvog reda* pošto su tipovi λ-računa prvog reda.

Sintaksa restrikcija Sintaksa restrikcija sadrži dve vrste izraza - tipove i restrikcije. Tipovi su tipske promenljive ili aplikacija tipskog konstruktora F u skladu sa njegovom arnošću. Restrikcije mogu biti atomičke kao što su $true$ i $false$ ili jednakost među tipovima. Složene restrikcije su konjunkcija i egzistencijalna kvantifikacija tipskih promenljivih.

Intepretacija restrikcija Model u kom se intepretiraju restrikcije je *Herbrand-ov* univerzum - skup konkretnih tipova dat sa:

$$\begin{aligned} \tau &::= \alpha \mid F\ \bar{\tau} \\ C &::= true \mid false \mid \tau = \tau \mid C \wedge C \mid \exists \alpha.C \end{aligned}$$

Slika 2.1: Jezik restrikcija

$$t ::= F\ \bar{t}$$

Konkretni tipovi su tipovi koji ne sadrže tipske promenljive. Da bi model bio neprazan skup potrebno je da bar jedan konstruktor ima nula arnost.

Interpetacija restrikcija ima formu tvrđenja $\phi \vdash C$ što se čita: ϕ zadovoljava C ili ϕ je rešenje za C, gde je ϕ je totalno preslikavanje tipskih promenljivih na konkretne tipove. Prelikavanje ϕ se proširuje i na tipove $\phi(F\ \tau_1\ ...\tau_n) = F(\phi(\tau_1), ..., \phi(T_n))$.

ϕ je totalno preslikavanje tipskih promenljivih na konkretne tipove. Tvrđenje $\phi \vdash C$ je induktivno definisano:

$$\phi \vdash true \qquad \frac{\phi\tau_1 = \phi\tau_2}{\phi \vdash \tau_1 = \tau_2} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi[\alpha \to t] \vdash C}{\phi \vdash \exists \alpha.C}$$

Za restrikciju C kažemo da je *zadovoljivo* ako postoji preslikavanje ϕ takvo da je ϕ rešenje za C. Svaka interpretacija ϕ zadovoljava trivijalno restrikciju **true**. U skupu restrikcija možemo definisati relaciju ekvivalencije $\equiv$ tako da važi $C_1 \equiv C_2$ kada C_1 i C_2 imaju ista rešenja. Pošto su restrikcije izrazi prvog reda (nije moguće kvantifikovati nad restrikcijama C u restrikciji C, već samo tipske promenljive $\exists \alpha.C$) i restrikcije su samo tipa jednakosti, problem zadovoljivosti restrikcija C je unifikacija prvog reda.

$$\begin{aligned} \langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle &= \Gamma(x) = \tau \\ \langle\!\langle \Gamma \vdash \lambda x.a : \tau \rangle\!\rangle &= \exists \alpha_1 \alpha_2.(\langle\!\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle\!\rangle \wedge \tau = \alpha_1 \to \alpha_2) \\ \langle\!\langle \Gamma \vdash a_1\ a_2 : \tau \rangle\!\rangle &= \exists \alpha.(\langle\!\langle \Gamma \vdash a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle \vdash a_2 : \alpha \rangle\!\rangle) \end{aligned}$$

Slika 2.2: Generisanje restrikcija za tipizirani λ-račun

Mehanizam *Hindley*-evog algoritma [8] za određivanje tipova je jednostavan - ukratko, za dati ne-anotirani term svaki čvor stabla uvodi nove nepoznate tipske promenljive i restrikcije među njima. Na primer, čvor apstrakcije uvodi novu promenljivu za tip argumenta α i za tip telo β i restrikciju da tip terma bude jednak tipu $\alpha \to \beta$.

Generisanje restrikcija za terme tipiziranog λ-računa dato u slici 2.2. Rešavanjem skupa svih restrikcija dobija se principijelna forma tipa za početni term.

Kao što je već rečeno, ovaj mehanizam podrazumeva fazu generisanja i rešavanja restrikcija. Radi modularnosti, ove dve faze su ćesto razdvojene i nezavisne. Moguće je formalizovati jezik restrikcija u formi logike tako da ima svoju *sintaksu* i *interpretaciju* u modelu.

Primer 1. Dato je određivanje tipa za izraz $\lambda fgx.g\ (f\ x)$ u praznom okruženju. Sa C ćemo označiti restrikcije. Radi ilustracije na ovom primeru se faze generisanja i resavanja prepliću. U svakom koraku izvršena je ili transformacija restrikcija u ekvivalentne pojednostavljene forme ili translacija.

$$\langle\!\langle \varnothing \vdash \lambda fgx.g\ (f\ x) \rangle\!\rangle = \exists \alpha_1 \alpha_2 . \begin{pmatrix} \langle\!\langle mf : \alpha_1 \vdash \lambda gx.g\ (f\ x) : \alpha_2 \rangle\!\rangle \\ \alpha_0 = \alpha_1 \to \alpha_2 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_2 . \begin{pmatrix} \exists \alpha_3 \alpha_4 . \begin{pmatrix} \langle\!\langle f : \alpha_1; g : \alpha_3 \vdash \lambda x.g\ (f\ x) : \alpha_4 \rangle\!\rangle \\ \alpha_2 = \alpha_3 \to \alpha_4 \end{pmatrix} \\ \alpha_0 = \alpha_1 \to \alpha_2 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_2 . \begin{pmatrix} \exists \alpha_3 \alpha_4 . \begin{pmatrix} \exists \alpha_5 \alpha_6 . \begin{pmatrix} \langle\!\langle f : \alpha_1; g : \alpha_3; x : \alpha_5 \vdash g\ (f\ x) : \alpha_6 \rangle\!\rangle \\ \alpha_4 = \alpha_5 \to \alpha_6 \end{pmatrix} \\ \alpha_2 = \alpha_3 \to \alpha_4 \end{pmatrix} \\ \alpha_0 = \alpha_1 \to \alpha_2 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 . \begin{pmatrix} \langle\!\langle \Gamma \vdash g\ (f\ x) \rangle\!\rangle \\ \alpha_0 = \alpha_1 \to \alpha_2 \ \wedge\ \alpha_2 = \alpha_3 \to \alpha_4 \ \wedge\ \alpha_4 = \alpha_5 \to \alpha_6 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 . \begin{pmatrix} \langle\!\langle \Gamma \vdash g\ (f\ x) \rangle\!\rangle \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 . \begin{pmatrix} \exists \alpha_7 . \begin{pmatrix} \langle\!\langle \Gamma \vdash g : \alpha_7 \to \alpha_6 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \end{pmatrix} \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 \end{pmatrix} C = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 \alpha_7 . \begin{pmatrix} \begin{pmatrix} \alpha_3 = \alpha_7 \to \alpha_6 \\ \exists \alpha_8 . \begin{pmatrix} \langle\!\langle \Gamma \vdash f : \alpha_8 \to \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash x : \alpha_8 \rangle\!\rangle \end{pmatrix} \end{pmatrix} \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 \end{pmatrix}$$

$$\mathrm{C} = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 \alpha_7 \alpha_8 . \begin{pmatrix} \alpha_3 = \alpha_7 \to \alpha_6 \\ \alpha_1 = \alpha_8 \to \alpha_7 \\ \alpha_5 = \alpha_8 \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 \end{pmatrix}$$

Dobili smo oblik restrikcija:

$$\langle\!\langle \lambda fgx.g\ (f\ x) : \alpha_0 \rangle\!\rangle \equiv \exists \alpha_6 \alpha_7 \alpha_8 . (\alpha_8 \to \alpha_7) \to (\alpha_7 \to \alpha_6) \to \alpha_8 \to \alpha_6$$

Ove restrikcije mozemo zapisati kao principijelni tip:

$$\tau = (a \to b) \to (b \to c) \to a \to c$$

Vidimo da izraz predstavlja kompoziciju i da tipovi funkcija moraju da se slažu.

2.2 Sintaksa restrikcija

Za određivanje tipova ML jezika nadogradićemo rezultate prethodnog poglavlja o određivanju tipova λ-računa.

Kao što je prikazano u prethodnom poglavlju, određivanje tipova tipiziranog λ-računa zasnovano je na unifikaciji prvog reda. DM je složeniji tipski sistem koji uključuje sheme tipova i unifikacija prvog reda više nije dovoljna. Zbog uvedenog polimorfizma i sheme tipova potrebne su dodatne operacija nad tipovima kao što su *generalizacija* i *instanciranje*. Jednakost tipova više nije dovoljna relacija u restrikcijama. Potrebno je uvesti restrikcije za relaciju instance.

U odnosu na λ-račun, ML-ov tipski sistem uvodi ograničenu formu polimorfizma - samo u okviru **let** deklaracije. Samo promenljive koje su vezane u okviru **let** kontrukcije se mogu koristiti polimorfno. To znači da promenljiva x umesto jednog tipa, sada dozvoljava skup tipova. Pitanje je kako u generaciji restrikcija predstaviti ovaj skup tipova. U DM su uvedene tipske sheme oblika $\forall\bar{\alpha}.\tau$ koje već predstavljaju skup tipova. *Da li se tipske sheme mogu direktno koristiti u sintaksi restrikcija?*

Treba primetiti da su u kontekstu određivanja tipova svi tipovi nepoznate, pa time i tip promenljive x deklarisane u let - da bi se koristio treba se prvo odrediti. To znači da su tipske sheme uvek principijalni tipovi koje su rezultat određivanja tipova. Odnosno, tipske sheme su rešenja za skup restrikcija. Odnosno, u okviru generisanja restrikcija tipske sheme se ne mogu direktno koristiti za predstavu polimorfnog tipa promenljive x pošto bi takva sintaksa zahtevala mešanje faza generisanja i rešavanja restrikcija. Pošto želimo da razdvojimo ove dve faze, skup tipova ćemo predstaviti novom sintaksnom formom za tipove koja će sadržati nerešenu formu restrikcija tj. skup restrikcija pod nazivom *tipske sheme sa restrikcijama*:

$$\sigma ::= \forall\bar{\alpha}[C].\tau$$

Pored sintakse tipova, potrebno je proširiti i sintaksu restrikcija tako da sadrži relaciju instance između tipova. Kao što je rečeno, polimorfan tip promenljive x se može smatrati skupom tipova, tako da u okviru tela **let** deklaracije x može imati tipove iz tog skupa. Na primeru *let x = id in (x 0, x true)* tip koji x može imati u proizvodu nije jednak njegovom tipu već je element skupa. U sintaksi restrikcija potrebno pored jednakosti uvesti i relaciju instance ili pripadanja skupu.

Data je proširena sintaksa restrikcija:

Uvedena je **def** deklaracija. Ova deklaracija vezuje promenljivu x za tipsku shemu σ u okviru restrikcija C. Može se smatrati formom eksplicitne supstitucije. Uloga ove deklaracije je da pristup promenljivama okruženja prebaci u deo rešavanja

$$C ::= true \mid false \mid \tau = \tau \mid C \wedge C \mid \exists \alpha.C \mid x \preceq \tau \mid \sigma \preceq \tau \mid \textbf{def}\ x : \sigma\ in\ C$$

Slika 2.3: Proširenje sintakse restrikcije

restrikcija umesto generisanja. Ova dizajnerska odluka omogućava optimizaciju algoritama za rešavanje restrikcija. Na primer, kao što ćemo videti u nastavku, efikasno je pre zamene u restrikciju C pojednostaviti tipsku shemu σ.

Uvedena su još dve nove sintatičke forme $x \preceq \tau$ i $\sigma \preceq \tau$ koje označavaju instanciranje. Prva je neubičajena pošto može delovati da predstavlja relacija između programskih promenljivih i tipova. Ona je potrebna zbog gore opisanog dizajna generacije restrikcija koja, za razliku od restrikcija λ-računa, ne pristupa promenljivama iz okruženja. Promenljive x se sada ne zamenjuju eksplicitno za svoj tip koji se čita iz okruženju kao što to radi pravilo $\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle = \Gamma(x) = \tau$. Već se promenljive vezuju u formi eksplicitne supstitucije **def** i zapravo u restrikcijama predstavlju promenljive koje stoje za tip, a ne programske promenljive. Drugim rečima, u **def** $x{:}\sigma\ in\ C$ ako C sadrži podrestrikciju forme $x \preceq \tau$ gde je pojavljivanje promenljive x slobodno u C onda ova podrestrikcija u ovom kontekstu ima značenje $\sigma \preceq \tau$. Zato se i **def** smatra formom eksplicitne supstitucije. Ukratko, $x \preceq \tau$ predstavlja relaciju instance među tipovima pošto je x samo referenca ka tipskoj shemi.

Treba primetiti da pojam instance $\preceq$ ima drugačije značenje u odnosu na instancu u DM sistemu. Naime, instanca ovde predstavlja restrikcije dok je u DM instanca binarna relacija. Odnosno, za datu shemu tipa σ i tip τ, tip τ jeste ili nije instanca od σ. Takođe, ova relacija je isključivo sintaksne prirode. Sa druge strane, instanca ovde predstavlja restrikciju na slobodnim promenljivama koje su sadržane u tipovima. Uloga instance u je šira - ona postavlja restrikcije nad slobodnim promenljivama koje sadrži. Na primer, izraz $\forall x.x \to x \preceq y \to z$ nije tvrđenje već restrikcija nad promenljivama y i z za koji se može pokazati da je ekvivalentno izrazu $y \preceq z$. Sa druge strane, u DM-u $y \to z$ nije instanca $\forall x.x \to x$ pošto su y i z dve sintaksno različite promenljive i izraz $\forall x.x \to x \preceq y \to z$ se ne može izvesti.

Kao što je rečeno, uvedene su dve forme za instanciranje $x \preceq \tau$ i $\sigma \preceq \tau$.

Po konvenciji, $\exists$ i **def** vezuju jače od veznika $\wedge$, odnosno $\exists\bar{\alpha}.C \wedge D$ označava $(\exists\bar{\alpha}.C) \wedge D$ i **def** $x : \sigma\ in\ C \wedge D$ oznava (**def** $x : \sigma\ in\ C$) $\wedge D$.

U $\forall\bar{\alpha}[C].\tau$ tipske promenljive $\bar{\alpha}$ su vezane u C i τ. U $\exists\bar{\alpha}.C$ tispke promenljive $\bar{\alpha}$ su vezane u C.

Sledeća definicija predstavlja osnovnu ideju kako se ovaj sistem može svesti na sistem restrikcija za λ-račun.

Definicija 2.2.1. *Neka σ bude $\forall\bar{\alpha}[C].\tau$. Ako $\bar{\alpha} \neq ftv(\tau')$, onda $\sigma \preceq \tau''''''$ označava restrikciju $\exists\bar{\alpha}.(C \wedge \tau \preceq \tau')$.*

Ova definicija nam daje interpretaciju restrikcija relacijom - $\sigma \preceq \tau$. Data je generalizovana verzija sa relacijom instance u DM sistemu, naime $\preceq$. Za potrebe određi-

vanja tipova, dovoljno je koristiti specijalizaciju ove interpretacija sa jednakošću umesto instancom. Takođe, možemo zapisati i kao zakon ekvivalencije:

$$(\forall \bar{\alpha}[C].\tau) \preceq \tau' \equiv \exists \bar{\alpha}.(C \wedge \tau = \tau') \; ako \; \bar{\alpha} \neq \tau'$$

Vidimo da je interpretacija $\sigma \preceq \tau$ u sintaksi restrikcija za tipizirani λ-račun. Dalje, i druga nova sintaksna konstrukcija **def** se interpretira u istom jeziku. Drugi zakon ekvivalencije koji se odnosi na interpretaciju **def** konstrukcije dat je sa:

$$\textbf{def} \; x : \sigma \in C \equiv [x \mapsto \sigma]C$$

Uz pomoć datih zakona ekvivalencije DM restrikcije se mogu transformisati u restrikcije tipiziranog λ-računa. Ove zakone će koristiti sistem za rešavanje restrikcija.

2.3 Interpretacija restrikcija

U odeljku restrikcija za λ-račun predstavili smo interpretaciju restrikcija. Za λ-račun intepretacija ima oblik tvrđenja $\phi \vdash C$. Odnosno, interpretacija je *preslikavanje na konkretne tipove* ϕ slobodnih tipskih promenljivih u restrikciju C.

Ova interpretacija mora biti redefinisana, pošto se u *DM* restrikcijama pored tipskih promenljivih nalaze i programske promenljive. Za interpretaciju DM restrikcija potrebno je i preslikavanje programskih promenljivih na tipove. Ovo preslikavanje obeležićemo sa ψ. Za razliku od ϕ preslikavanja, ψ preslikava na *konkretne tipske sheme* umesto konkretne tipove. Programska promeljiva x može se naći samo u okviru restrikcija oblika $x \preceq \tau$. To znači da njena interpretacija mora biti skup konkretnih tipova koje ćemo zvati *konkretne tipske sheme*.

Konkretna tipska shema s je skup konkretnih tipova koji je zatvoren odozgo, tj. ako $\tau \in s$ i $\tau \leq \tau'$ onda važi $\tau' \in s$.

Sintaksa ograničenih tipskih shema i restrikcija je uzajamno rekurzivna - sintaksa ograničenih tipskih shema sadrži restrikcije, a sintaksa restrikcija sadrži tipske sheme, samim tim i njihova interpretacija. Interpretacija tipske sheme σ pod preslikavanjem ϕ i ψ je konkretna tipska shema koju ćemo obelezati sa $\binom{\phi}{\psi}\sigma$. Tipsku shemu možemo posmatrati kao ograničenu tipsku shemu kad je skup restrikcija prazan, onda je intepretacija:

$$\binom{\phi}{\psi}(\forall \bar{\alpha}[C].\tau) \equiv \{(\phi[\bar{\alpha} \mapsto \bar{\mathbf{t}}])\tau\} \quad ako \; je \; C = \varnothing$$

$\bar{\mathbf{t}}$ je iz skupa konkretnih tipova. Vidimo da je interpretacija skup konkretnih tipova dobijen tako što se preslikavanje ϕ proširuje preslikavanjem vezanih promenljivih $\bar{\alpha}$ na konkretne tipove. Na primer, interpetacija tipske sheme $\forall \alpha.\alpha \to \beta$ pod preslikavanjem ϕ je skup čiji članovi imaju oblik $t \to \phi(\beta)$ gde t pripada skupu konkretnih promenljivih.

Tipska shema je specijalni slučaj ograničene tipske sheme, sad se možemo vratiti na opšti oblik.

$$\binom{\phi}{\psi}(\forall\bar{\alpha}[C].\tau) \equiv \{\phi'\tau | (\phi' \setminus \bar{\alpha} = \phi \setminus \bar{\alpha}) \wedge (\phi', \psi \vdash C)\}$$

Interpretacija ima isto značenje kao i za tipske sheme samo u drugačijem obliku zapisano. Preslikavanje ϕ' je svako preslikavanje koje zadovoljavaja uslove da slobodne tipske promenljive preslikava išto kao i ϕ i da vezane promenljive $\bar{\alpha}$ da preslikava na konkretne tipove. Prvi uslov dat je u specifikaciji skupa, a drugi je implicitno dat pošto je svako ϕ preslikavanje totalno znači da će i sve vezane promenljive biti u domenu. Dodatni uslov za preslikavanje ϕ' jeste da zadovoljava restrikcije C date u ograničenoj tipskoj shemi. Ovaj uslov je restrikcija vezana sa preslikavanje vezanih promenljivih. Drugim rečima, preslikavanje vezanih promenljivih mora biti takvo da zadovoljava i restrikcije C.

Primer 2. Na primer, za $\forall\alpha[\exists\beta.\alpha = F\ \beta].\alpha \to \alpha$ gde je F neki unarni tipski konstruktor (na primer *list* u ML jeziku), interpretacija će biti skup svih tipova oblika $(F\ t \to F\ t)$. Znači, restrikcija je restriktovala skup konkretnih tipova odnosno preslikavanje α na konkretne tipove oblika $F\ t$.

Interpetacija restrikcija je tvrđenje oblika $\phi, \psi \vdash C$ koje se čita: ϕ i ψ zadovoljavaju restrikcije C. Pravila za induktivno izvođenje $\phi, \psi \vdash C$ data su na slici 2.4 što je zapravo i definicija ternarnog predikata $\vdash$.

IR-TACNO
$$\phi, \psi \vdash true$$

IR-PRED
$$\frac{P(\phi(\tau_1), ..., \phi(\tau_n))}{\phi, \psi \vdash P\ \tau_1 \ldots \tau_n}$$

IR-KONJ
$$\frac{\phi, \psi \vdash C_1 \qquad \phi, \psi \vdash C_2}{\phi, \psi \vdash C_1 \wedge C_2}$$

IR-EGZ
$$\frac{\phi[\bar{\vec{\alpha}} \mapsto \vec{t}], \psi \vdash C}{\phi, \psi \vdash \exists\vec{\alpha}.C}$$

IR-DEF
$$\frac{\phi, \psi[x \mapsto \binom{\phi}{\psi}\sigma] \vdash C}{\phi, \psi \vdash \textbf{def}\ x : \sigma\ in\ C}$$

IR-INST
$$\frac{\phi(\tau) \in \psi(x)}{\phi, \psi \vdash x \preceq \tau}$$

IR-INST2
$$\frac{\psi(\tau) \in \binom{\phi}{\psi}\sigma}{\phi, \psi \vdash \sigma \preceq \tau}$$

Slika 2.4: Interpretacija restrikcija

Interpretacija obuhvata interpretaciju tipske sheme i interpretaciju restrikcija. U definiciji pravila se vidi da su ove dve interpretacije uzajamno rekurzivne; definicija intepretacije tipske sheme koristi interpetaciju restrikcija i obratno.

Ukratko ćemo objasniti pravila. Skup pravila datih na slici 2.4 je sintaksno usmeren. Pravilo IR-EGZ omogućava da tipske promenljive $\vec{\alpha}$ označavaju proizvoljne konkretne tipove $\vec{t}$ nezavisno od njihove slike u ϕ. Što sledi iz toga da su u lokalnom kontekstu restrikcija C promenljive $\vec{\alpha}$ vezane i njihovo značenje je nezavisno od okolnog konteksta. Pravilo IR-DEF vezuje promenljivu x u okviru C za konkretnu tipsku shemu σ. To je pravilo koje uvodi tipsku shemu u okruženje. Sa druge strane, pravilo IR-INST koristi tipske sheme iz okruženja pošto se odnosi na restrikciju koje ima formu instance. Preslikavanje ψ preslikava promenljive za konkretne tipske

sheme koje predstavljaju skup konkretnih tipova i zato je u premisi instance dovoljno proveriti da li konkretan tip pripada ovom skupu.

2.4 Generisanje restrikcija

U ovom delu daćemo translaciju izraza u restrikcije. Ova translacija za osnovu ima generisanje restrikcija za tipizirani λ-račun koje je dato u 2.2.

$$
\begin{aligned}
\langle\!\langle x : \tau \rangle\!\rangle &= x \preceq \tau \\
\langle\!\langle \lambda x.a : \tau \rangle\!\rangle &= \exists \alpha_1 \alpha_2 . (\textbf{def } x : \alpha_1 \textit{ in } \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
\langle\!\langle a_1\ a_2 \rangle\!\rangle &= \exists \alpha . (\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \textit{ ako } \alpha \neq \alpha_1, \alpha_2, \tau \\
\langle\!\langle \textbf{let } x = a_1 \textit{ in } a_2 : \tau \rangle\!\rangle &= \textbf{let } x : (\!|a_1|\!) \textit{ in } \langle\!\langle a_2 : \tau \rangle\!\rangle \\
(\!|a|\!) &= \forall \alpha [\langle\!\langle a : \alpha \rangle\!\rangle] . \alpha
\end{aligned}
$$

Slika 2.5: Generisanje restrikcija za ML

Više komentara o pravilima iz 2.5 nalazi se u delu o implementaciji u Haskell-u, tj. poglavlju 5.1.

2.5 Odnos tipskog sistema restrikcija sa DM sistemom

U prethodnom delu prikazali smo samo jednu intepretaciju i model za restrikcije koji je pogodan za određivanje tipova za ML. Međutim, moguće je definisati različite interpretacije i modele. Štaviše, moguće je dalje uopštiti sistem restrikcija i to uopštenje je sistem *HM(X)* dat u poglavlju *HM(X)*.

Zanimljivo je primetiti da u određenom modelu tzv. *modelu sintaksne jednakosti* tipski sistem restrikcija je u veoma bliskoj korespodenciji sa *DM* tipskim sistemom. Želimo da pokažemo da svaka zadovoljiva restrikcija C takvo da važi $fpi(C) = \varnothing$, gde $fpi(C)$ označava skup slobodnih programskih promenljivih restrikcija C, ima *kanoničku rešenu formu* i da je to u veoma bliskoj vezi sa konceptom *najgeneralnijeg unifikatora*.

Rešena forma je konjunkcija restrikcija gde se sa leve strane nalaze različite tipske promenljive koje se ne nalaze sa desne strane jednačina, uz moguću egzistencijalnu kvantifikaciju.

Definicija 2.5.1. ***Rešena forma restrikcija*** *je oblika* $\exists \bar{\beta} . (\vec{\alpha} = \vec{\tau})$*, gde* $\bar{\alpha} \neq ftv(\bar{\tau})$.

Lema 2.5.1. *Svaka restrikcija je ekvivalentna ili rešenoj formi ili je* ***false****.*

Lema 2.5.2. *Svaka tipska shema sa restrikcijama je ekvivalentna standardnoj tipskoj shemi.*

Rešena forma $\exists\bar{\beta}.(\vec{\alpha} = \vec{\tau})$ je kanonička ako i samo ako su njene slobodne tipske promenljive $\bar{\alpha}$. Drugim rečima, forma je kanonička ako τ ne sadrži slobodne promenljive.

Data je ekvivalentna definicija:

Definicija 2.5.2. ***Kanonicka rešena forma*** *je restrikcija oblika* $\exists\bar{\beta}.(\vec{\alpha} = \vec{\tau})$ *gde je* $ftv(\bar{\tau}) \subseteq \bar{\beta}$ *i* $\bar{\alpha} \neq \bar{\beta}$.

Lema 2.5.3. *Svaka rešena forma je ekvivalentna kanoničkoj resenoj formi.*

Kanonička rešena forma je korisna zato što je lako naći njeno rešenje, to je zamena $[\vec{\alpha} \mapsto \vec{\tau}]$. Iz toga sledi da je svaka kanonicka forma zadovoljiva.

Takođe, kanonicka rešena forma može se smatrati restrikcijom ili idempodentnom zamenom.

Definicija 2.5.3. *Ako je* $[\vec{\alpha} \mapsto \vec{\tau}]$ *idempodentna zamena* θ*, sa* $\exists[\vec{\alpha} \mapsto \vec{\tau}]$ *obeležićemo kanoničku rešenu formu* $\exists\bar{\beta}.(\vec{\alpha} = \vec{\tau})$*, gde je* $ftv(\bar{\tau}) = \bar{\beta}$*. Idempodentna zamena* θ *je* ***najgeneralniji unifikator*** *restrikcija C ako i samo ako su* $\exists\theta$ *i C ekvivalentni.*

Prema definiciji ekvivalentne restrikcije imaju iste najgeneralnije unifikatore. Mnoge osobine kanoničke rešene forme mogu biti formulisane pomoću najgeneralnijeg unifikatora.

Glava 3

HM(X)

Formulacija restrikcija nagovestila je mogućnost uopštenja tako da se dobije tipski sistem. HM(X) je parametrizovano proširenje Damas-Milnerovog tipskog sistema.

Odersky, Sulzman i Wehr su prvi uveli HM(X) sistem 1999. godine [9]. Od tada, nekoliko varijacija sistema je izloženo u radovima kao što što su Sulzmann, Muller i Zengder 1999. [10], Pottier 2001. [11]. U ovom radu biće izložena definicija HM(X) sistema zasnovana na jeziku restrikcija koji je prezentovan u prethodnom poglavlju.

3.1 Definicija HM(X) tipskog sistema

Tipski sistem HM(X) sastoji se od tvrđenja oblika $C, \Gamma \vdash t : \sigma$ gde je C restrikcija, Γ okruženje, t izraz i σ tipska shema. Tvrđenje se čita: pod pretpostavkama C i Γ izraz t je tipa σ. C predstavlja pretpostavke o tipskim promenljivama, a Γ pretpostavke o identifikatorima.

Važno je napomenuti da restrikcije koje imaju isto značenje mogu imati različitu sintaksnu prezentaciju. Međutim, sintaksa restrikcija ne bi trebalo da utiče na validnost *HM(X)* tvrđenja već samo njihovo značenje.

Ovo možemo postići tako što ćemo jednakost *HM(X)* tvrđenja posmatrati modulo ekvivalencije njihovih restrikcija tako da su tipska tvrđenja $C, \Gamma \vdash t : \sigma$ i $D, \Gamma \vdash t : \sigma$ jednaka kada je $C \equiv D$. Kao rezultat, apstrahuje se nad sintaksom restrikcija i ona se ne analizira u tvrđenju.

Takođe, sintaksa restrikcija se može smatrati prezentacijom restrikcija na nižem nivou i bilo bi poželjno da nema ulogu u dokazima logičkih osobina restrikcija. Ovo predstavlja motivaciju da se restrikcije posmatraju sa apstraktnijeg stanovišta - konkretnije, preko odnosa i relacija među njima, tako da rezonovanje bude nezavisno od njihove strukture. U nastavku ćemo prezentovati relacije i njihove osobine koji omogućava ovakav pogled na restrikcije.

Najznačajnija relacija je relacija zadovoljivosti $\Vdash$.

Definicija 3.1.1. *$C_1 \Vdash C_2$ se čita: C_1 podrazumeva C_2 i važi ako i samo ako za svako preslikavanje ϕ i za svako preslikavanje za okruženje ψ i ako $\phi, \psi \vdash C_1$ implicira $\phi, \psi \vdash C_2$. Restrikcije su ekvivalentna $C_1 \equiv C_2$ ako i samo ako važi $C_1 \Vdash C_2$ i $C_2 \Vdash C_1$.*

Intuitivno, relacija $\Vdash$ govori o jačini restrikcija: C_1 je *jača* restrikcija od C_2. Jačina se meri preko parova (ϕ, ψ) koje zadovoljavaju restrikciju; restrikcija C_1 je *jača* od C_2 ako manji broj parova (ϕ, ψ) zadovoljava C_1. Drugim rečima, C_1 postavlja striktnije zahteve nad slobodnim tipskim promenljivama, koje su sadržane u C_1 i C_2 tj. $\bar{\alpha} \subseteq ftv(C_1)$ i $\bar{\alpha} \subseteq ftv(C_2)$, i programskim identifikatorima $\bar{x} \subseteq fpv(C_1)$ i $\bar{x} \subseteq fpv(C_2)$ i od C_2. Relacija $\Vdash$ je refleksivna i tranzitivna, a relacija $\equiv$ je relacija ekvivalencije.

Za tvrđenje kažemo da je *validno* ako i samo ako se može izvesti pravilima tipiziranja datih u slici 3.1.

Treba napomenuti da valjana tipiziranost izraza t u okruženju Γ više nije u 1-1 korespodenciji sa validnošću tvrđenja $C, \Gamma \vdash t : \sigma$, kao što je to bio slučaj u *DM* tipskom sistemu. Razlog tome je taj što $C, \Gamma \vdash t : \sigma$ može biti validno tvrđenje i kad je C nezadovoljiva restrikcija. Međutim, ako je C nezadovoljiva restrikcija onda t nije valjano tipiziran. Izraz t je valjano tipiziran u okruženju Γ ako i samo ako je tvrđenje oblika $C, \Gamma \vdash t : \sigma$ validno za neko zadovoljiva restrikcija C.

HMX-VAR
$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma}$$

HMX-ABS
$$\frac{C, (\Gamma; x : \tau) \vdash t : \tau'}{C, \Gamma \vdash \lambda x.t : \tau \to \tau'}$$

HMX-APP
$$\frac{C, \Gamma \vdash t_1 : \tau \to \tau' \qquad C, \Gamma \vdash t_2 : \tau}{C, \Gamma \vdash t_1\ t_2 : \tau'}$$

HMX-LET
$$\frac{C, \Gamma \vdash t_1 : \sigma \qquad C, (\Gamma; z : \sigma) \vdash t_2 : \tau}{C, \Gamma \vdash \textbf{let}\ z = t_1\ \textbf{in}\ t_2 : \tau}$$

HMX-GEN
$$\frac{C \wedge D, \Gamma \vdash t : \tau \qquad \bar{\alpha} \neq ftv(C, \Gamma)}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash t : \forall\bar{\alpha}[D].\tau}$$

HMX-INST
$$\frac{C, \Gamma \vdash t : \forall\bar{\alpha}[D].\tau}{C \wedge D, \Gamma \vdash t : \tau}$$

HMX-SUB
$$\frac{C, \Gamma \vdash t : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash t : \tau'}$$

HMX-EXISTS
$$\frac{C, \Gamma \vdash t : \sigma \qquad \bar{\alpha} \neq ftv(\Gamma, \sigma)}{\exists\bar{\alpha}.C, \Gamma \vdash t : \sigma}$$

Slika 3.1: Pravila tipiziranja HM(X) tipskog sistema

Sad ćemo objasniti pravila tipiziranja data na slici 3.1. Samo pravilo HMX-SUB nema svoje odgovarajuće pravilo u DM sistemu. Videćemo u nastavku da je specijalizacijom HM(X) na DM sistem to pravilo postaje suvišno.

Pravilo HMX-VAR išto kao i DM-VAR pretražuje okruženje za identifikator x i vraća odgovarajuću tipsku shemu. Sa tim što ovo pravilo ima još jednu premisu koja je tehničke prirode i koja garantuje da pronađena tipska shema pod restrikcijom C ima instance.

Najzanimljivija pravila su HMX-GEN i HMX-INST. To su pravila uvođenja i eliminacije $\forall$ kvantifikatora. Pravilo generalizacije HMX-GEN.

Da bi bilo jasnije pravilo HMX-GEN pogledacemo prvo specijalizaciju pravila kad je $C \equiv \mathbf{true}$:

$$\text{HMX-GEN'}\quad \frac{D, \Gamma \vdash t : \tau \qquad \bar{\alpha} \neq ftv(\Gamma)}{\exists \bar{\alpha}.D, \Gamma \vdash t : \forall \bar{\alpha}[D].\tau}$$

HMX-GEN'

ima istu drugu premisu kao i DM-GEN - tipske promenljive koje su generalizovane ne smeju da se pojavljuju slobodno u okruženju Γ. Međutim, generalizovane tipske promenljive mogu da se jave u D restrikciji i zato restrikcija D mora biti zapisano u tipskoj shemi u zaključku. U zaključku je izgrađena tipska shema $\forall \bar{\alpha}[D].\tau$ univerzalnim kvanitifikovanjem nad promenljivama koja su restriktovana u restrikciji D. Iz ovog razloga u zaključku se nalazi dodatna pretpostavka - $\exists \bar{\alpha}.D$, koja garantuje da postoji bar jedan skup konkretnih tipova u slici za $\bar{\alpha}$ tipske promenljive koji neće narušiti zadovoljivost D, odnosno da tipska shema $\forall \bar{\alpha}[D].\tau$ ima bar jednu instancu. Primetimo da D pored $\bar{\alpha}$ može imati i druge slobodne tipske promenljive koje su obično slobodne i u Γ. Izvorno pravilo HMX-GEN se od HMX-GEN' razlikuje samo u jednom tehničkom detalju. Razlika je u tome što se u pravilu HMX-GEN restrikcije koje se ne odnose na generalizovane tipske promenljive C ne kopiraju u tipsku shemu. Ovaj detalj je deo optimizacije koja je direktno ugrađena u pravilo tipiziranja - nepotrebne restrikcije koja mogu biti složena se ne kopiraju. Ukratko, pravilo HMX-GEN gradi ograničenu tipsku shemu generalizacijom nad promenljivama $\bar{\alpha}$ i dodatno kopira deo restrikcija iz pretpostavki koja restriktuju promeniljive $\bar{\alpha}$ u tipsku shemu.

Pravilo HMX-INST omogućava instanciranje tipske sheme. Zanimljivo je da, za razliku od DM-INST, ne uključuje zamenu za tipove. Odnosno, vrši identičku zamenu $\bar{\alpha} \mapsto \bar{\alpha}$. Međutim, tipske sheme se smatraju jednakim modulo imena vezanih promenljivih tj. α-konverzije tako da se promenljive mogu preimenovati implicitno pre instanciranja. Takođe, restrikcije D tipske sheme ulaže u pretpostavke u zakljucku.

Pravilo HMX-SUB omogućava da se tip τ može zameniti svojim podtipom τ'. I τ i τ' mogu da sadrže slobodne tipske promenljive i zato je izraz $\tau \leq \tau'$ podtipska restrikcija (a ne relacija instance) koje treba da važi pod pretpostavkama restrikcije C i o tome govori druga premisa.

Pravilo HMX-EXISTS omogućava da tipske promenljive koje se pojavljuju samo u restrikciji budu egzistencijalno kvantifikovane. Na ovaj način one postaju lokalne.

Može se dokazati da ovo pravilo ne omogućava tipiziranje više izraza, već proširuje skup validnih tvrđenja. Uloga pravila je tehničke prirode. HM(X) tvrđenja se posmatraju moduo ekvivivalencije restrikcija. To znači da se restrikcije mogu pojednostaviti i da se njihova pojednostavljena verzija može koristiti umesto originalne. Ekvivalencija restrikcija je kongruentna relacija, tako da iz $C \equiv D$ sledi $\exists\bar{\alpha}.C \equiv \exists\bar{\alpha}.D$, ali obrnuto ne važi uvek. Ali upravo drugi izraz često može da se pojednostavi, dok prvi ne može. HM-EXISTS nam onda omogućava da iskoristimo pojednostavljene, koje inače samo na osnovu ekvivalencije restrikcija ne bismo mogli. Na primer, ne postoji način da se pojednostavi tvrđenje $\alpha \leq \beta \leq \gamma, \Gamma \vdash t : \sigma$. Međutim, ako se β ne pojavljuje kao slobodna promenljiva u Γ i u σ onda pomoću HM-EXISTS možemo izvesti tvrđenje $\exists\beta.(\alpha \leq \beta \leq \gamma), \Gamma \vdash t : \sigma$. $\alpha \leq \beta \leq \gamma \equiv \alpha \leq \gamma$ ne važi, ali $\exists\beta.(\alpha \leq \beta \leq \gamma) \equiv \alpha \leq \gamma$ važi. Tako da se tvrđenje može pojednostaviti na $\alpha \leq \gamma, \Gamma \vdash t : \sigma$.

3.2 Alternativna prezentacija HM(X) sistema

Prezentacija HM(X) sistema data na slici 3.1 ima samo četiri sintaksno usmerena pravila. To je dobro teorijska prezentacija tipskog sistema, ali nije pogodna kao opis algoritma. Daćemo alternativnu prezentaciju HM(X) sistema u kojoj se generalizacija može javiti samo u okviru **let** izraza, a instanciranje samo na referencama programskih identifikatora.

$$\text{HMD-VarInst}\ \frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau}{C \wedge D, \Gamma \vdash x : \tau} \qquad \text{HMD-ABS}\ \frac{C, (\Gamma; x : \tau) \vdash t : \tau'}{C, \Gamma \vdash \lambda x.t : \tau \to \tau'} \qquad \text{HMD-APP}\ \frac{C, \Gamma \vdash t_1 : \tau \to \tau' \qquad C, \Gamma \vdash t_2 : \tau}{C, \Gamma \vdash t_1\ t_2 : \tau'}$$

$$\text{HMD-letGen}\ \frac{C \wedge D, \Gamma \vdash t_1 : \tau_1 \qquad \bar{\alpha} \neq ftv(C, \Gamma) \qquad C \wedge \exists\bar{\alpha}.D, (\Gamma; z : \forall\bar{\alpha}[D].\tau_1) \vdash t_2 : \tau_2}{C \wedge \exists\bar{\alpha}, \Gamma \vdash \textbf{let}\ z = t_1\ in\ t_2 : \tau_2}$$

$$\text{HMD-SUB}\ \frac{C, \Gamma \vdash t : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash t : \tau'} \qquad \text{HMD-EXISTS}\ \frac{C, \Gamma \vdash t : \sigma \qquad \bar{\alpha} \neq ftv(\Gamma, \sigma)}{\exists\bar{\alpha}.C, \Gamma \vdash t : \sigma}$$

Slika 3.2: Pravila tipiziranja HMD(X)-a

Teorema 3.2.1. *$C, \Gamma \vdash t : \tau$ se može izvesti pravilima iz slike 3.2 ako i samo ako je validno HM(X) tvrđenje.*

3.3 DM i HM(X)

Motivacija za nastanak HM(X) tipskog sistema je generalizacija DM tipskog sistema sa naglaskom na određivanje tipova. U ovom poglavlju ćemo i pokazati da je HM(X) zapravo generalizovana verzija DM tipskog sistema. Na početku ćemo definisati šta znači da je generalizacija.

HM(X) označava familiju tipskih sistema parametrizovanu sa X koje označava sintaksu i interpretaciju restrikcija kao i relaciju instance. Da bi HM(X) bio generalizacija DM svaki član familije HM(X) mora da sadrži DM. U drugom smeru, DM mora da sadrži HM(=).

Prvo ćemo dokazati direktan smer. Dokaz se sastoji u preslikavanju DM tvrđenja na HM(X) tvrđenja. Očigledno je da je svako validno DM tvrđenje je zapravo HM(X) tvrđenje sa restrikcijom **true**. Ovo je potkrepljeno i time da je $\forall\bar{\alpha}.\tau$ išto što i $\forall\bar{\alpha}[\mathbf{true}].\tau$ tako da su DM tipske sheme podskup od HM(X) tipskih shema, a to onda išto važi i za okruženja koja vezuju tipske sheme. Dajemo formulaciju u obliku teoreme.

Teorema 3.3.1. *Ako* $\Gamma \vdash e : \sigma$ *onda i* $\mathbf{true}, \Gamma \vdash e : \sigma$.

Dokaz. Dokaz je indukcijom po strukturi izvođenja $\Gamma \vdash_{DM} e : \sigma$.

- *Slučaj* DM-VAR. Zaključak je $\Gamma \vdash_{DM} x : \sigma$, a premisa je $\Gamma(x) = \sigma$ (1). Treba pokazati da važi $\mathbf{true}, \Gamma \vdash_{HM(X)} x : \sigma$. Ovo tvrđenje se može izvesti pomoću pravila HMX-VAR - prva premisa je $\Gamma(x) = \sigma$, druga je $\mathbf{true} \Vdash \exists\sigma$. Prva važi zbog (1). Za drugu ćemo iskoristiti definiciju $\exists\sigma$, naime $\exists(\forall\bar{\alpha}[D].\tau) \equiv \exists\bar{\alpha}.D$ i ekvivalenciju $\exists\bar{\alpha}.C \equiv C$ kada $\bar{\alpha} \neq ftv(C)$. Pošto je restrikcija u σ **true**, dobijamo $\exists\sigma \equiv \mathbf{true}$ i odatle $\mathbf{true} \Vdash \mathbf{true}$ što trivijalno važi.

- *Slučaj* DM-GEN. Znamo da važi $\Gamma \vdash_{DM} e : \forall\bar{\alpha}.\tau$, i premise $\Gamma \vdash_{DM} e : \tau$ (1) i $\bar{\alpha} \neq ftv(\Gamma)$ (2). Treba pokazati da važi $\mathbf{true}, \Gamma \vdash_{HM(X)} e : \forall\bar{\alpha}[\mathbf{true}].\tau$. To se može pokazati primenjujući pravilo HMX-GEN ako važe premise $\mathbf{true}, \Gamma \vdash e : \tau$ i $\bar{\alpha} \neq ftv(\mathbf{true}, \Gamma)$. Prvu premisa se dobija primenjujući induktivnu hipotezu na (1), a druga važi zbog (2). Iskoristili smo $\exists\bar{\alpha}.\mathbf{true} \equiv \mathbf{true}$.

- *Slučajevi* DM-ABS, DM-APP i DM-LET. Dokazuju se inverzijom pravila, primenom induktivne hipoteze i primenom HMX-ABS, HMX-APP i HMX-LET respektivno.

- *Slučaj* DM-INST. Zaključak pravila je $\Gamma \vdash_{DM} e : [\bar{\alpha} \mapsto \bar{\tau}]\tau$. Inverzijom pravila dobijamo $\Gamma \vdash_{DM}: \forall\bar{\alpha}\tau(1).\tau$. Možemo pretpostaviti, bez gubitka generalnosti, $\bar{\alpha} \neq ftv(\Gamma, \bar{\tau})$. Treba pokazati da važi $\mathbf{true}, \Gamma \vdash_{HM(X)} e : [\bar{\alpha} \mapsto \bar{\tau}]\tau$. Primenom induktivne hipoteze na (1) dobijamo $\mathbf{true}, \Gamma \vdash_{HM(X)}: \forall\bar{\alpha}.\tau$. Primenom HM-INST dobijamo $\mathbf{true}, \Gamma \vdash_{HM(X)}: \forall\tau$. Možemo primeniti oslabljivanje tako da dobijemo $\vec{\alpha} = \vec{\tau}, \Gamma \vdash_{HM(X)}: \tau$. Koristeći pravila ekvivalencija restrikcija može se pokazati $\vec{\alpha} = \vec{\tau} \Vdash \tau = [\vec{\alpha} = \vec{\tau}]\tau$ (6). Sad koristeći HMX-SUB i (5) i (6) imamo $\vec{\alpha} = \vec{\tau}, \Gamma \vdash_{HM(X)}: [\vec{\alpha} = \vec{\tau}]\tau$. (7) implicira $\bar{\alpha} \neq ftv(\Gamma, [\bar{\alpha} \mapsto \bar{\tau}]\tau)$ (8). Sad možemo

primeniti HMX-EXISTS na (7) i (8) i dobiti $\exists\bar{\alpha}.(\vec{\alpha} = \vec{\tau}), \Gamma \vdash_{HM(X)}: [\vec{\alpha} = \vec{\tau}]\tau$. I na kraju ako iskoristimo ekvivalenciju $\exists\bar{\alpha}.(\vec{\alpha} = \vec{\tau}) \equiv \mathbf{true}$ i dobijamo traženo tvrđenje.

□

Za drugi smer treba pokazati da je HM(=) sadržan u DM. HM(=) je specijalizacija HM(X) sistema gde se restrikcije intepretiraju u sintaksnom modelu odnosno u skupu *zatvorenih* tipova, a konkretni tipovi su tipovi koje ne sadrže tipske promenljive i formiraju Herbandrov univerzum. Za ovaj dokaz dovoljno je dati korektnu translaciju HM(=) tvrđenja u DM tvrđenje.

Prvo ćemo dati translaciju HM(=) tipskih shema u DM tipske sheme. Za ovu translaciju veoma je bitan izbor sintaksnog modela sa jednakošću, pošto to pruža osobine koje su neophodne za translaciju. Naime, u ovom modelu svaka restrikcija ima najgeneralniji unifikator

Pored tipskih shema, potrebna je translacija i za tipske supstitucije, zato što je nakon transliranja potrebno zameniti tipove. Ovi delovi mogu biti razdvojeni, ali se mogu posmatrati i zajedno tako što ćemo translaciju parametrizovati tipskom sustitucijom. $[\![\sigma]\!]_\theta$. Intuicija je da se uz pomoć unifikatora sve dodatne pretpostavke o tipskim promenljivama prebace iz restrikcija direktno u tipsku shemu. Sledeća lema to formalizuje.

Lema 3.3.2. *Neka σ označava tipsku shemu $\forall\bar{\alpha}[D].\tau$, a θ tipsku zamenu takvu da $ftv(\theta) \subseteq dom(\theta)$ (1) i $\exists\theta \Vdash \exists\sigma$. Postoji tipska supstitucija θ' koja proširuju θ tako da $dom(\theta') = dom(\theta) \cup \bar{\alpha}$ i θ' je najgeneralniji unifikator restrikcija $\exists\theta \wedge D$. Sa $\bar{\beta}$ obeležićemo tipske promenljive principijelne shemi za $\bar{\alpha}$ na koje mapira unifikator odnosno $\bar{\beta} = ftv(\theta'(\bar{\alpha})) \setminus kodomen(\theta)$. Translacija $[\![\sigma]\!]_\theta$ je DM tipska shema $\forall\bar{\beta}.\theta'(\tau)$. Takođe, $ftv([\![\sigma]\!]_\theta) \subseteq kodomen(\sigma')$.*

Pre glavne teoreme, daćemo nekoliko tehničkih karakteristika translacije.

Lema 3.3.3. *Ako $C, \Gamma \vdash t : \sigma$ onda $C \Vdash \exists\sigma$.*

Lema 3.3.4. *Ako se θ_1 i θ_2 podudaraju u okviru $ftv(\sigma)$, onda su translacije $[\![\sigma]\!]_{\theta 1}$ i $[\![\sigma]\!]_{\theta 2}$ ili nedefinisane, ili definisane i iste.*

Ova lema govori da je translacija $[\![\sigma]\!]_\theta$ nezavisna od ponašanja σ izvan $ftv(\sigma)$.

Lema 3.3.5. *Neka je $ftv(\sigma, \tau') \subseteq dom(\theta)$ i $\exists\theta \Vdash \exists\sigma$. Neka je $\exists\theta \Vdash \sigma \preceq \tau'$. Onda je $\theta(\tau')$ instanca DM tipske sheme $[\![\sigma]\!]_\theta$*

Drugim rečima, ako $C \Vdash \sigma \preceq \tau'$ važi, onda su translacije σ i τ' pod najgeneralnijim unifikatorom tj. supsitucijom θ za C u DM relaciji instance.

Lema 3.3.6. $D \Vdash \forall\bar{x}.\tau \preceq \tau$.

Lema 3.3.7. *Neka je θ najgeneralniji unifikator za C. Ako $\bar{\gamma} \neq ftv(C)$ onda je $\theta \setminus \bar{\gamma}$ takođe najgeneralniji unifikator za C. Ako $\bar{\gamma} \neq \theta$ onda postoji najgeneralniji unifikator za C koji proširuje θ i čiji je domen $dom(\theta) \cup \bar{\gamma}$.*

Lema 3.3.8. *Iz $C \Vdash D$ sledi $\theta(C) \Vdash \theta(D)$.*

Lema 3.3.9. *$\theta(\exists\theta) \equiv true$.*

Lema 3.3.10. *Ako $true \Vdash \tau = \tau'$, onda τ i τ' su išti.*

Lema 3.3.11. *Ako je θ najgeneralniji unifikator za C, onda $\theta \setminus \alpha$ je najgeneralniji unifikator za $\exists\alpha.C$. U suprotnom, ako je θ najgeneralniji unifikator za $\exists\alpha.C$ i $\alpha \neq \theta$ i $ftv(\exists\alpha.C) \subseteq dom(\theta)$ onda postoji tipska supsitucija θ' takva da proširuje θ, θ' je najgeneralniji unifikator za C, i $dom(\theta') = dom(\theta) \cup \alpha$.*

Proširećemo translaciju na okruženje. $[\![\varnothing]\!]_\theta$ je $\varnothing$. $[\![\Gamma; x:\sigma]\!]_\theta$ je $[\![\Gamma]\!]_\theta; x : [\![\sigma]\!]$ ako $\exists\theta \Vdash \exists\sigma$, u drugom slučaju je $[\![\Gamma]\!]_\theta$.

Sad možemo dokazati glavnu teoremu.

Teorema 3.3.12. *Neka je $,\Gamma \vdash t : \sigma$ važi u HM(=). Neka je σ najgeneralniji unifikator za C takav da $ftv(\Gamma, \sigma) \subseteq dom(\theta)$. Onda $[\![\Gamma]\!]_\theta \vdash t : [\![\sigma]\!]_\theta$ važi u DM.*

Dokaz. Dokaz je indukcijom po strukturi izvođenja HM(=) tvrđenja. Primetimo samo da iz leme 3.3.3 znamo da $C \Vdash \exists\sigma$. Ovo može da se zapiše $\exists\theta \Vdash \exists\sigma$ iz čega znamo da je translacija $[\![\sigma]\!]_\theta$ definisana. Za dokaz ćemo koristiti HMD(X) pravila data u firugi 3.2. Ali ćemo razdvojiti pravilo HMD-LetGen na HMX-LET i HMX-GEN radi bolje citljivosti.

- *Slučaj* HMD-VarInst. Zaključak pravila je $C \wedge D, \Gamma \vdash x : \tau$. Prema hipotezi θ je najgeneralniji unifikator za $C \wedge D$ (1) i $ftv(\tau) \subseteq dom(\theta)$ (2). Premisa je $\Gamma(x) = \sigma$ (3) gde je sa σ označeno $\forall\bar{x}[D].\tau$. Treba pokazati da $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$. Iz (1) znamo da $\exists\sigma \equiv C \wedge D \Vdash D \Vdash \exists\bar{x}.D \equiv \exists\sigma$ odnosno $\exists\theta \Vdash \exists\sigma$ (4). Dalje, iz toga sledi da $ftv(\sigma) \subseteq ftv(\Gamma) \subseteq dom(\theta)$ (5). Na osnovu toga znamo da je $[\![\sigma]\!]_\theta$ definisano iz čega sledi $[\![\Gamma]\!]_\theta(x) = [\![\sigma]\!]_\theta$. Sada iz DM-VAR sledi $[\![\Gamma]\!]_\theta \vdash x : [\![\sigma]\!]_\theta$ (6). Preostalo je samo pokazati da je $\theta(\tau)$ instanca od $[\![\sigma]\!]_\theta$. Iz leme 3.3.6 sledi $D \Vdash \sigma \preceq \tau$, a pošto $\exists\theta \Vdash D$ onda i $\exists\theta \Vdash \sigma \preceq \tau$ (7). Sada iz (2),(4),(5),(7) i iz leme 3.3.5 dobijamo $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$.

- *Slučaj* HMD-ABS. Zaključak pravila je $C, \Gamma \vdash \lambda x.t\tau \to \tau'$. Treba pokazati $[\![\Gamma]\!]_\theta \lambda x.t : [\![\tau \to \tau']\!]_\theta$. Premisa je $C, (\Gamma; x : \tau) \vdash t : \tau'$ (1). Ako primenimo induktivno hipotezu na (1) dobijamo $[\![\Gamma]\!]_\theta; x : [\![\tau]\!]_\theta \vdash t : [\![\tau']\!]$. Primenom DM-ABS dobijamo $[\![\Gamma]\!]_\theta \vdash \lambda x.t : [\![\tau]\!]_\theta$, što je $[\![\Gamma]\!]_\theta \lambda x.t : [\![\tau \to \tau']\!]_\theta$.

- *Slučaj* HMX-GEN. Zaključak je $C \wedge \exists\sigma, \Gamma \vdash t : \sigma$ gde σ označava $\forall\bar{\alpha}[D].\tau$. Prema hipotezi θ je najgeneralniji unifikator za $C \wedge \exists\sigma$ (1) i $ftv(\Gamma, \sigma) \subseteq dom(\theta)$ (2).

Podsetimo se da važi $\exists\bar{\alpha}.D \equiv \exists\sigma$, pa zbog toga možemo zapisati kao u (1). Premise su $C \wedge D, \Gamma \vdash t : \tau$ (3) i $\bar{\alpha} \# ftv(C, \Gamma)$ (4). Treba pokazati da važi $[\![\Gamma]\!]_\theta \vdash t : [\![\sigma]\!]_\theta$. Bez gubitaka generalnosti možemo izabrati $\bar{\alpha}$ tako da $\bar{\alpha} \# \theta$ (5). Sad možemo definisati proširenje supstitucije θ' i $\bar{\beta}$ kao u lemi 3.3.2. θ' je najgeneralniji unifikator za $C \wedge D$. Dalje, prema lemi znamo da je $dom(\theta') \equiv dom(\theta) \cup \bar{\alpha}$. Iz (2) znamo da $ftv(\Gamma, \sigma) \subseteq dom(\theta')$ (6). Iz toga možemo primeniti induktivnu hipotezu sa θ' na (3) da dobijemo $[\![\Gamma]\!]_{\theta'} \vdash t : \theta'(\tau)$. Prema lemi 3.3.4 i (2) i (6) možemo zapisati i kao $[\![\Gamma]\!]_\theta \vdash t : \theta'(\tau)$. Iz leme (:reference:) znamo da $ftv([\![\Gamma]\!]_\theta \subseteq domen(\theta))$, a prema konstrukciji znamo i $\bar{\beta} \# ftv([\![\Gamma]\!]_\theta)$ (7). Sada ako primenimo DM-GEN, (6) i (7) dobijamo $[\![\Gamma]\!]_{\theta'} \vdash t : \forall\bar{\beta}.\theta'(\tau)$. Što je $[\![\Gamma]\!]_\theta \vdash t : [\![\sigma]\!]_\theta$.

□ *Slučaj* HMD-APP. Zaključak je $C, \Gamma \vdash t_1 t_2 : \tau'$. Premise su $C, \Gamma \vdash t_1 : \tau \to \tau'$ (1) i $C, \Gamma \vdash t_2 : \tau$ (2). Iz hipoteze θ je najgeneralniji unifikator takav da $ftv(\Gamma, \tau') \subseteq dom(\theta)$ (3). Možemo uzeti θ' proširenje od θ tako da $ftv(\Gamma, \tau', \tau) \subseteq dom(\theta')$. Primenom induktivne hipoteze na premise (1) i (2) i θ' dobijamo $[\![\Gamma]\!]_{\theta'} \vdash t1 : \theta'(\tau) \to \theta'(\tau')$ (4) i $[\![\Gamma]\!]_{\theta'} \vdash t_2 : \theta'(\tau)$ (5). Primenom DM-APP na (4) i (5) dobijamo $[\![\Gamma]\!]_{\theta'} \vdash t_1\ t_2 : \theta'(\tau')$ (6). Pošto $dom(\theta') \setminus dom(\theta) = ftv(\tau)$ i prema lemi 3.3.4 iz (6) možemo zaključiti traženo tvrđenje $[\![\Gamma]\!]_\theta \vdash t_1\ t_2 : \theta(\tau')$.

□ *Slučaj* HMX-LET. Slično kao u prethodnom slučaju, osim što se $dom(\theta)$ proširuje za $ftv(\sigma)$, primenjuje se induktivna hipoteza i zatim DM-LET.

□ *Slučaj* HMD-SUB. Zaključak je $C, \Gamma \vdash t : \tau'$. Prema hipotezi θ je najgeneralniji unifikator za C (1) i $ftv(\Gamma, \tau') \subseteq dom(\theta)$ važi. Treba pokazati da važi $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau')$ (3). Premisa je $C, \Gamma \vdash t : \tau$ (4) i $C \Vdash \tau \preceq \tau'$ (5). Bez gubitka generalnosti možemo pretpostaviti $ftv(\tau) \# domen(\theta)$ (6). Prema lemi 3.3.7 možemo prositiri domen θ tako da važi $ftv(\tau) \subseteq dom(\theta)$ (7). Iz (1), (2) i (7) i primenom induktivne hipoteze na (4) dobijamo $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$ (8). (5) je ekvivalentno sa $\exists\theta \Vdash \tau = tau'$. Iz lema 3.3.8 i 3.3.9 imamo $true \Vdash \theta(\tau) = \theta(\tau')$. Iz leme 3.3.10 sledi da su $\theta(\tau)$ i $\theta(\tau')$ isti tipovi. Iz ovoga i (8) dobijamo traženo tvrđenje $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau')$.

□ *Slučaj* HMD-EX. Zaključak je $\exists\bar{\alpha}.C, \Gamma \vdash t : \tau$. Prema hipotezi θ je najgeneralniji unifikator za $\exists\bar{\alpha}.C$ (1) i $ftv(\Gamma, \tau) \subseteq dom(\theta)$ (2). Treba pokazati da $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$. Premise su $C, \Gamma \vdash t : \tau$ (3) i $\bar{\alpha} \# ftv(\Gamma, \tau)$ (4). Bez gubitka generalnosti možemo pretpostaviti $\bar{\alpha} \# \theta$. Prema lemi 3.3.7 možemo proširi domen θ tako da važi $ftv(\exists\bar{\alpha}.C) \subseteq dom(\theta)$ (5). Iz (1), (4) i (5) i leme 3.3.11 , postoji tipska susptitucija θ' takva da θ' proširuje θ (6) i θ' je najgeneralniji unifikator za C. Primenom induktivne hipoteze na (3) i θ' dobijamo $[\![\Gamma]\!]'_\theta \vdash t : \theta'(\tau)$. Prema (2) i (6) i leme 3.3.4 to je isto što i $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$.

□

Deo II

Modularno određivanje tipova ML-jezika u Haskell-u

U ovom delu rada biće predstavljena Haskell implementacija određivanja tipova za podskup ML jezika. Prvo je dat pregled osnova teorije kategorije koje su potrebna za rad sa monadama u Haskell-u. Zatim je prikazana implementacija modularnog određivanja tipova koja se oslanja na teorijske rezultate izložene u prvom delu rada.

Glava 4

Teorija kategorija

4.1 Uvod

U ovom delu biće prikazana osnova teorija kategorija koja je potrebna za definisanje monada u *Haskell*-u. Izložena teorija preuzeta je pretežno iz [12], [13] i [14].

4.2 Osnove

Osnovne postavke teorije kategorija su jednostavne. Da bi se definisao pojam kategorije, potrebno je uvesti gradivne elemente: 1. objekte, 2. morfizme, 3. identitete, 4. kompozicije, i zakone koji treba da važe: a. zakon identiteta, b. asocijativnost.

Definicija 4.2.1. ***Kategorija*** $\mathcal{C}$ *sastoji se iz:*

1. *kolekcija objekata,* $Obj(\mathcal{C})$*;*
2. *kolekcije strelica odnosno morfizama,* $Hom_{\mathcal{C}}(x, y)$ *gde* $x, y \in Obj(\mathcal{C})$ *se naziva hom – skup i označava skup morfizama od* x *do* y*;*
3. *identički morfizam za svaki objekat koji se označava sa* $id_x \in Hom_{\mathcal{C}}(x, x)$ *gde* $x \in Obj(C)$*;*
4. *operator kompozicije koji svakom paru strelica* f *i* g *dodeljuje strelicu kompozicije* $g \circ f$*, odnosno za svaka tri objekta* x*,* y *i* z*, ako* $f \in Hom_{\mathcal{C}}(x, y)$*,* $g \in Hom_{\mathcal{C}}(y, z)$*, onda* $g \circ f \in Hom_{\mathcal{C}}(x, z)$*;*

   a *(zakon identiteta) za svaki morfizam* $f : x \to y$ *treba da važi* $f \circ \mathrm{id}_x = f$ *i* $\mathrm{id}_y \circ f = f$*;*

   b *(asocijativnost) za sve morfizme* $f : w \to x$*,* $g : x \to y$ *i* $h : y \to z$ *treba da važi* $(h \circ g) \circ f = h \circ (g \circ f)$*;*

Napomena Ovde su kategorije definisane u *teoriji skupova*. Međutim, $\mathrm{Obj}(\mathcal{C})$ nije skup već kolekcija objekata kategorije $\mathcal{C}$. U okviru ovih formalizama možemo pričati o kategoriji svih skupova, odnosno kategoriji u kojoj su objekti svi mogući skupovi. Ukoliko $\mathrm{Obj}(\mathcal{C})$ posmatramo kao skup objekata onda ovakvom konstrukcijom dolazimo do *Raselovog* paradoksa. Rešenje je da se uvede pojam *univerzuma* koje ćemo označiti sa k. Ovo će omogućiti da imamo pogled iz tog univerzuma i van - sve što je u tom svetu nazivamo k-malo, a kad je potrebno da radimo sa samim k onda možemo izabrati veći univerzum k' i raditi u njemu. Kategorija u kojoj je $\mathrm{Obj}(\mathcal{C})$ skup se naziva *mala kategorija*. U nastavku ćemo posmatrati samo ovakve kategorije. Takođe, *operacije* pomenute u definiciji su funkcije iz teorije skupova.

Sledi primer kategorije koja modeluje pojam skupa.

Primer 3 (Kategorija **Skup**)**.** (1) Objekti u kategoriji **Skup** su skupovi. (2) morfizmi su totalne funkcije $f : A \to B$ iz skupa A na skup B; (3) identički morfizam za objekat A označen sa id_A je totalna funkcija $\mathrm{id}_A : A \to A$; (4) kompozicija totalne funkcije $f : A \to B$ i $g : B \to C$ je totalna funkcija iz A na C koja preslikava svaki element $a \in A$ na $g(f(a)) \in C$; za ovako definisan identički morfizam i kompoziciju važe (a) $\mathrm{id}_B \circ f = f$ i $f \circ \mathrm{id}_A = f$ i (b) $(h \circ g) \circ f = h \circ (g \circ f)$.

Primer 4 (Kategorija **Monoid**)**.**

Definicija 4.2.2 (Monoid)**.** *Monoid je triplet $(M, e, \star)$ gde je M skup, $e \in M$ je jedinični element i $\star : M \times M \to M$ je funkcija takva da važe sledeći zakoni za $m, n, p \in M$:*

- *(M1)* $m \star e = m$
- *(M1')* $e \star m = m$
- *(M2)* $(m \star n) \star p = m \star (n \star p)$

Definicija 4.2.3. *Neka je $\mathcal{M} = (M, e, \star)$ i $\mathcal{M}' = (M', e', \star')$. Homeomorfizam je $f : M \to \mathcal{M}'$ je funkcija $f : M \to M'$ koja zadovoljava sledeće uslove:*

- $f(e) = e'$;
- $f(m_1 \star m_2) = f(m_1) \star f(m_2)$, *za svako* $m_1, m_2 \in \mathcal{M}$

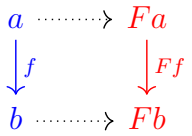
Monoid može biti predstavljen kao kategorija **Mon** sa samo jednim objektom. Elementi skupa M su strelice iz objekta ka samom sebi, identički element e je predstavljen sa identičkom strelicom. Operacija $\star$ je predstavljena kompozicijom strelica. Obeležimo objekat sa m. Svake dve strelice f i g iz **Mon** idu iz m u m, tako da se uvek može se naći njihova kompozicija $g \circ f$, što je strelica koja takođe predstavlja neki element u skupu M. Iz zakona identiteta i asocijativnosti dobijamo uslove binarnu operaciju $\star$. Obrnutno, svaka kategorija sa samo jednim objektom je monoid.

4.3 Funktori

Prvo ćemo funktore prikazati neformalnim opisom, a zatim ćemo dati definiju.

Kategorija je apstraktan mehanizam pomoću kog se mogu formulisati različiti matematički koncepti. Kategorija je takođe matematički koncept pa je moguće konstruisati kategoriju kategorija. U ovoj kategoriji objekti su kategorije, a strelice su specijalni morfizmi između kategorija koji se nazivaju *funktori*.

Za dve kategorije **C** i **D** neka $F : C \to D$ bude funktor. Funktor F preslikava svaki objekat $a \in Obj(C)$ u objekat $F\ a \in Obj(D)$. Međutim, kategorija pored objekata sadrži i morfizme, pa je za preslikavanje kategorija pored objekata potrebno preslikati i morfizme. Morfizmi se ne mogu preslikavati proizvoljno - povezanosti između objekata moraju biti očuvane. Odnosno, ako je f morfizam koji povezuje objekte a i b tj. $f : a \to b$ onda njegova slika je morfizam $F\ f$ koji povezuje slike tih objekata tj. $F\ f : F\ a \to F\ b$ dijagram 4.1.



Slika 4.1: Funktor

Funktor je morfizam koji održava strukturu kategorija. Struktura se oslikava u povezanosti objekata. Struktura podrazumeva i kompoziciju morfizama. Ako je $h = g \circ f$ onda treba da važi $F\ h = F\ g \circ F\ f$. Odnosno, funktor treba da bude takav morfizam koji održava kompoziciju. I na kraju funktor preslikava identičke morfizme jedne kategorije u identičke morfizme druge kategorije, $F\ id_a = id_{F\ a}$.

Definicija 4.3.1 (Funktor)**.** *Neka su C i D kategorije. Funktor F iz C u D, koji se obeležava sa $F : C \to D$ treba da sadrži:*

- *A. funkciju $Obj(F) : Obj(C) \to Obj(D)$, koja se može označiti sa $F : Ob(C) \to Ob(D)$;*
- *B. za svaki par objekata $a, b \in Ob(C)$ funkciju $Hom_F(a, b) : Hom_C(a, b) \to Hom_D(F\ a, F\ b)$, ili jednostavnije $F : Hom_C(a, b) \to Hom_D(F\ a, F\ b)$.*

Tako da važe fanktorski zakoni:

- *1. (očuvanje identiteta) za svaki objekat $a \in Obj(C)$ važi $F\ id_a = id_{F\ a}$;*
- *2. (očuvanje kompozicije) za sve objekte $a, b, c \in Obj(C)$ i morfizme $g : a \to b$ i $h : b \to d$ važi $F\ (h \circ g) = F\ h \circ F\ g$.*

U nastavku je dato nekoliko primera fanktora. Prvi primer je više teorijski, a druga dva primera ilustruju vezu teorije i implementacije u Haskellu.

Primer 5 (Monoid u skup)**.** Posmatramo preslikavanje kategorije **Monoid** u kategoriju **Skup**. Samo na osnovu definicije monoida možemo pronaći funktor U : **Monoid** → **Skup**. Ako je $\mathcal{M} = (M, e, \star)$ monoid onda je M skup i ako je $f : \mathcal{M} \to \mathcal{M}'$ homomorfizam, onda je $f : M \to M'$ funkcija nad skupom. Preslikavanje objekata fanktora U je $U(\mathcal{M}) = M$, a morfizama $U(f) = f$. Nije teško pokazati da za ovako definisan funktor važe uslovi (1) i (2).

Primer 6. Neka je **Skup** kategorija. Ako je s skup možemo definisati *Lista* s kao skup svih konačnih listi elemenata iz s. Na ovaj način smo već definisali funkciju nad objektima (A.). Ukoliko želimo da *Lista* bude funktor, potrebno je još definisati preslikavanje morfizama tako da važe uslovi 1. i 2. To je funkcija koja morfizam $f : s \to s'$ preslikava u morfizam $Lista\ f : Lista\ s \to List\ s'$. Odnosno funkcija treba samo treba da mapira morfizam f na elemente liste $l = [e_1, e_2, ..., e_n]$:

$$Lista\ f\ l = map\ f\ l = [f\ e_1, f\ e_2, ..., f\ e_n]$$

Sad ćemo proveriti da li su za ovako definisanu funkciju nad morfizmima ispunjeni uslovi (1) i (2). Što se tiče uslova (1), ako je id_s neki identički morfizam tako da $id_s : s \to s$ i koji je identitet nad elementima, odatle sledi da je

$$Lista\ id_s\ l = [id_s\ e_1, ..., id_s\ e_n] = [e_1, ..., e_n] = id_{Lista\ s}\ l$$

Međutim, identički morfizam u kategoriji **Skup** može biti bilo koja permutacija osim identiteta nad elementima. Obeležimo tu permutaciju sa prm_s. Nakon preslikavanja nad elementima svih konačnih listi skupa *Lista* s dobićemo permutaciju nad ovim skupom tj. $prm_{Lista\ s}$ što je identitet u ovoj kategoriji. Time smo pokazali (1). Što se tiče (2), neka je $f : s \to s'$ i $g : s' \to s''$ treba pokazati da važi $Lista\ (g \circ f) = Lista\ g \circ Lista\ f$:

$$(Lista\ g \circ Lista\ f)\ l = [g\ (f\ e_1), ..., g\ (f\ e_n)] = [(g \circ f)\ e_1, ..., (g \circ f)\ e_n] = Lista\ (g \circ f)\ l$$

Na ovom primeru možemo videti analogiju sa Haskell-om. Umesto kategorije u kojoj su objekti skupovi, možemo posmatrati kategoriju u kojoj su objekti tipovi. *Lista* s bi u ovom slućaju bio funktor *List a* odnosno tipski konstruktor u Haskell-u. Takođe, fanktori koji preslikavaju kategoriju u samu sebe, kao što je to slučaj u primeru gore, odnosno $F : C \to C$ se nazivaju *endofunktori*. Svi fanktori u Haskell-u su endofunktori pošto posmatramo samo kategoriju tipova.

Da bismo dalje istražili funktore u Haskell-u daćemo još jedan primer u Haskell-ovoj sintaksi.

Primer 7. Maybe je unarni tipski konstruktor u Haskell-u. Može se posmatrati kao dekoracija za određen tip koji je označen tipskom promenljivom a. Konkretno, **Maybe** se koristi kao dekoracija za rezultat parcijalnih. Ukoliko za neki original funkcija ima sliku tipa $x : \tau$ onda će rezultat biti $Just\ x$, a ukoliko nema onda će rezultat biti $Nothing$. Data je deklaracija:

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

Maybe je tipski konstruktor tj. funkcija koja preslikava tipove u tipove, odnosno tek za dati konkretni tip τ, $Maybe\ \tau$ predstavlja tip. Možemo posmatrati sada kategoriju gde su objekti tipovi T, i $Maybe$ kao funkciju koja preslikava objekte $Maybe : Obj(T) \to Obj(T)$. *Da li Maybe može biti definisan kao fanktor?*

Potrebno je prvo definisati fukciju nad morfizmima tj. $Maybe : Hom_T(b) \to Hom_T(c)$ za sve $b, c \in Obj(T)$. Ako je $f : b \to c$ gde su $b, c \in Obj(T)$ funkciju koja f preslikava u $Maybe\ f$ obeležićemo sa $f' : Maybe\ b \to Maybe\ c$. Data je funkcija:

```
fmap :: (b -> c) -> (Maybe b -> Maybe c)
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

Ukoliko $Maybe\ b$ označava term tipa b onda će se funkcija primeniti "ispod " dekoracije. A ukoliko označava $Nothing$ onda će se samo vratiti $Nothing$. Kao i u prethodnom primeru pokazaćemo da za ovako definisanu funkciju nad morfizmima važe funktorski uslovi. Da bismo ovo pokazali koristićemo tehniku rezonovanja prema jednakosti - pošto su u Haskellu funkcije definisane kao jednakost, leva i desna strana se mogu zameniti u bilo kom kontekstu, uz moguće preimenovanje promenljivih da bi se izbegli konflikti. (1) uslov:

```
fmap id (Just x) = {definicija fmap} Just (id x)
= {definicija id} Just x = {definicija id} id (Just x)
```

Što se tiče prvog uslova, **id** je polimorfna funkcija pa se u prvom pojavljivanju može posmatrati kao morfizam id_b, a u drugom kao $id_{Maybe\ c}$.

Za drugi uslov razmotrićemo dva slučaja - **Nothing** i **Just** x:

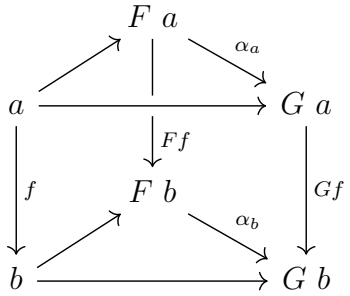
```
fmap (g. f) Nothing =
= {definicija fmap} Nothing
= fmap g Nothing
= fmap g (fmap f Nothing)
= (fmap g. fmap f) Nothing

fmap (g. f) (Just x)
= Just ((g. f) x)
= Just (g (f x))
= fmap g (Just (f x))
= fmap g (fmap f (Just x))
= (fmap g. fmap f) (Just x)
```

4.4 Prirodne transformacije

Funktori su preslikavanja između kategorija koja održavaju strukturu. Oni opisuju način na koji je jedna kategorija ugrađena u drugu. Drugim rečima, kako da se prepozna jedna kategorija u okviru druge. Međutim, ništa ne sprečava da postoji više različitih funktora između dve kategorije. Ako posmatramo dve kategorije C i D, na primer, jedan funktor može da preslikava svaki objekat iz C u zaseban objekat u D, dok drugi može da preslikava sve objekte C u išti objekat iz D. Pitanje je kako možemo porediti različite funktore. Odgovor na ovo pitanje su prirodne transformacije.

Prirodne transformacije su preslikavanja funktora sa određenim osobinama. Kao što su funktori preslikavanja kategorija koja održavaju njihovu strukturu, tako su prirodne transformacije preslikavanja funktora koja održavaju funktorijalnu strukturu.



Definicija 4.4.1. *Neka su C i D kategorije, a F i G funktori iz C u D.* ***Prirodna transformacija*** *α je funkcija koja svaki objekat a iz C preslikava u strelicu iz D $\alpha_a : F\ a \to G\ a$. Neka su $a, b \in C$ i $f \in Hom_c(a, b)$ morfizam, funktori F i G preslikavaju objekte i morfizme kao na dijagramu. Zakon komutativnosti treba da važi za prirodnu transformaciju:*

$$G\ f \circ \alpha_a = \alpha_b \circ F\ f$$

Primer 8. Posmatramo kategoriju u kojoj su objekti tipovi T. Neka je funktor G $List \circ List$, a funktor F $List$. Prirodna transformacija između ova dva funktora je konkatenacija liste koja u Haskell-u ima potpis:

```
concat :: forall a. [[a]] -> [a]
```

Ovo je polimorfna funkcija koja predstavlja primer tzv. parametarskog polimorfizma. Zbog polimorfizma važe "besplatne" teoreme o ponašanju polimorfnih funkcija. Konkretno, ako je $f : a- > b$ neka funkcija u Haskell-u, znamo da ukoliko je $l : [[a]]$ vazi:

```
fmap f (concat l) = concat (fmap f l)
```

Ovo važi pošto je **concat** parameterska funkcija i ne može da modifikuje sam objekat, može samo da manipuliše njima. Na primer, kao posledica ove teoreme znamo da funkcija koja ima ima polimorfan tip $\forall\alpha.\alpha \to \alpha$ može biti samo funkcija identiteta. Funkcija sa istim tipom ne mora biti konkatenacija (može npr. i da briše elemente) ali će i dalje važiti data jednakost. Na ovaj način zakon prirodnih transformacija se dobija automatski u Haskell-u uz pomoć parametarskog polimorfizma.

4.5 Monade

U funkcionalnom jeziku kao što je Haskell sve funkcije su "čiste". Međutim, u praksi često se javljaju problemi koji se ne mogu izraziti ovakvim funkcijama kao što su:

- parcijalnost
- nedeterminizam: funkcije koje mogu dati različite rezultate
- propratni efekti: računanja koja modifikuju određena stanja
- izuzeci (eng. exceptions): parcijalne funkcije
- kontinuacije: mogućnost da se sačuva stanje programa i da se kasnije obnovi po potrebi
- interaktivni ulaz/izlaz

Monade predstavljaju okvir za rešenje svih nabrojanih problema. Ako želimo da modelujemo propratni efekat u čistu funkciju $f : a \to b$ on se može modelovati u rezultatu tako da umesto b vraća "dekorisano" $M\ b$, $f : a \to M\ b$. Pitanje je kako definisati kompoziciju ovako "dekorisanih" funkcija. Neformalno govoreći, monade predstavljaju način kompozicije.

Prvo je dato neformalno objašnjenje i način definisanja monada u Haskell-u. Zatim je u nastavku dat formalan prikaz monada i Kliesli kategorije koja je teorijska osnova.

Kao što je već rečeno, svi nabrojani efekti se mogu modelovati uz pomoć nekog tipskog konstruktora koji ćemo obeležiti sa M. M predstavlja kontekst ili dekoraciju rezultata funkcije. Na primer, za parcijalne funkcije M je tipski konstruktor $Maybe$; $f : a \to Maybe\ b$ je parcijalna funkcija. Drugi primer efekata je modifikovanje stanja programa. Ovaj efekat se može modelovati tako što ćemo za M uzeti tzv. S binarni tipski konstruktor gde je prvi argument tip stanja, a drugi tip rezultata. Tako da funkcija koja modifikuje stanje ima tip $f : a \to S\ s\ b$.

Neka je $f : a \to M\ b$ i $g : b \to M\ c$. Za ove dve funkcije bez dekoracije rezultata možemo pronaći kompoziciju. Pitanje je šta je kompozicija za funkcije sa dekorisanim rezultatima. Ovakva kompozicija ne može definisati uniformno za sve M, već da zavisi od tipskog konstruktora odnosno konteksta.

U Haskell-u na sledeći način možemo da definišemo tipski konstruktor m koji je instanca monade:

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

Slika 4.2: Deklaracija monade u Haskell-u

Ovo nije jedini način definicije monade u Haskell okruženja. Međutim, ova deklaracija je verovatno najbolja za ilustraciju i intuiciju. Struktura monade koja se najčešće koristi biće data u nastavku. U datoj deklaraciji, (>=>) se naziva Kliesli strelica. To je zapravo operator kompozicije za monade, kao što je $(\circ)$ operator kompozicije za čiste funkcije. *return* je analogan identičkoj funkciji *id*, drugim rečima to je neutralan element za kompoziciju. Videćemo da ova deklaracija zapravo predstavlja Kliesli kategoriju u kojoj su *return* identički morfizmi, a (>=>) kompozicija morfizama. Na slici 4.2 data je samo deklaracija, da bi date funkcije zapravo bile kompozicija i identički morfizam potrebno je da važe asocijativnost i levi i desni neutralni element:

```
(f >=> g) >=> h = f >= (g >=> h)
return >=> f = f
f >=> return = f
```

Sad ćemo definisati (>=>) funkciju. Ova funkcija za argumente f i g treba da da funkciju njihove kompozicije.

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> let mb = f a
                in . ..
```

Za dato a treba da proizvedemo rezultat tipa $m\ c$ koristeći $f : a \to m\ b$ i $g : b \to m\ c$. Možemo primeniti funkciju f na argument a i dobiti međuvrednost $mb : m\ b$. Međutim, sada ne možemo primeniti g na mb. Potrebna nam je funkcija tipa $bind : m\ b \to (b \to m\ c) \to m\ c$ pošto na raspolaganju imamo $g : b \to m\ c$ i $mb : m\ b$, a potrebno je rezultat tipa $m\ c$. To nam govori da je za definisanje kompozicije dovoljno definisati samo *bind* funkciju, pošto (>=>) ima išti oblik za sve m:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> let mb = f a
                in bind mb g
```

Ovo je i najčešći način definisanja monada u Haskell-u (gde je bind (>>=))

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

Međutim, možemo iskoristiti to što je m funktor i dalje pojednostaviti definiciju monade. Uz pomoć odgovarajuće $fmap$ možemo primeniti funkciju $a \to m\ b$ na $m\ a$. Rezultat je sada $m\ (m\ b)$. Sad je potreban način da se od $m\ (m\ b)$ dobije $m\ b$. Intuitivno govoreći, potrebno je poravnati dva ugnežđena konteksta m. Ova funkcija u Haskell-u se naziva *join*:

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

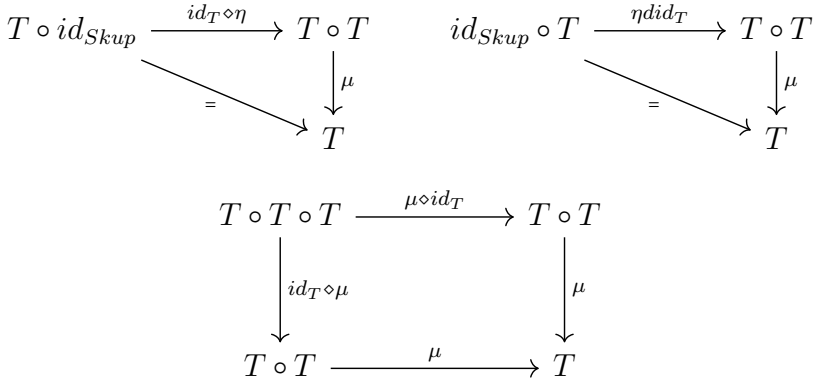
Ova definicija monade u Haskell-u je najslicnija teorijskim rezultatima koji su dati u nastavku.

Definicija 4.5.1 ($\diamond$)**.** *Neka su B,C,D i E kategorije, a $G_1, G_2 : C \to D$ funktori, i neka je $\alpha : G_1 \to G_2$ prirodna transformacija. Pretpostavimo da je $F : B \to C$ (respektivno $H : D \to E$) funktor. Operacija $\diamond$ je "pre" $\alpha \diamond F : G_1 \circ F \to G_2 \circ F$ (ili respektivno "post" $H \diamond \alpha : H \circ G_1 \to H \circ G_2$) i definiše se na sledeći način. Za svaki objekat $b \in Ob(B)$ komponenta $(\alpha \diamond F)_b : (G_1 \circ F)\ b \to (G_2 \circ F)\ b$ je definisana kao morfizam $\alpha_{F\ b}$.*

Definicija 4.5.2 (Monada)**.** *Monada kategorije **Skup** se sastoji iz:*

- *A. funktor $T :$ **Skup** $\to$ **Skup***
- *B. prirodna transformacija $\eta : \mathbf{id}_{Skup} \to T$*
- *C. prirodna transformacija $\mu : T \circ T \to T$*

Tako da su sledeći dijagrami komutativni:



Definicija 4.5.3 (Kleisli kategorija)**.** *Neka je $\mathbf{M} = (M, \eta, \mu)$ monada kategorije **Skup**. Na osnovu ovog tripleta Kliesli kategorija, označena sa $\mathbf{Kls(M)}$, se može definisati na sledeći nacin:*

$$Ob(Kls(\mathbf{M})) := Ob(\mathbf{Skup})$$
$$Hom_{Kls(T)}(a, b) := Hom_{Skup}(a, T\ b)$$

za objekte $a, b \in \mathbf{Skup}$. *Identički morfizam* $id_a : a \to a$ *u* $Kls(M)$ *dat je sa* $\eta : a \to T\ a$. *Da bismo definisali kategoriju* $Kls(\mathbf{M})$ *potrebno je još definisati kompoziciju. Ako su* $f : a \to b$ *i* $g : b \to c$ *morfizmi u* $Kls(\mathbf{M})$ *kako možemo naći kompoziciju* $g \circ f$*? Možemo naći odgovarajuću kompoziciju u kategoriji* **Skup**. *U ovoj kategoriji* f *je* $f : a \to T\ b$ *i* $g : b \to T\ c$. *Primenimo funktor* M *da dobijemo* $M\ g : M\ b \to M\ M\ c$. *Da bismo dobili morfizam* $a \to M\ c$ *potrebno je još da komponujemo sa* $\mu_c : M\ M\ c \to M\ c$. *Imamo* $\mu_c \circ g \circ f : a \to M\ c$. *Znači, kompozicija* $g \circ f : a \to c$ *u kategorji* $Kls(\mathbf{M})$ *data je kompozicijom* $\mu_c \circ g \circ f : a \to M\ c$ *u kategoriji* **Skup**.

Glava 5

Implementacija u Haskell-u

U ovom delu biće dat pregled implementacije rešenja u Haskell-u. Rešenje prati teorijske osnove koje su date u prvom poglavlju. Implementira datu sintaksu restrikcija koja omogućava razdvajanje faze generisanja i rešavanja. Takođe, kao što je prikazano u prvom poglavlju, sintaksa je modelovana tako da omogućava i optimizaciju. Implementacija je inspirisana i neki delovi su preuzeti iz [15] i [16].

Sintaksa U osnovi se nalazi podskup ML programskog jezika. Odnosno, proširenje čistog λ-računa. Pored promenljive, aplikacije i apstrakcije, izrazi obuhvataju konstante, **let** izraz, operator za fiksnu tačku **fix**, **if** konstrukciju i binarni operater **op**.

Data je sintaksa izraza u BN formi:

$$
\begin{aligned}
t &::= x \mid c \mid bool \mid \lambda x.t \mid t\ t \mid let\ x = t\ in\ t \mid fix\ t \mid if\ t\ t\ t \mid op\ binop\ t\ t \\
c &::= 0 \mid 1 \mid 2 \mid ... \\
bool &::= true \mid false \\
binop &::= add \mid sub \mid mul \mid eql
\end{aligned}
$$

U nastavku je data odgovarajuća sintaksa u Haskell-u. Definisana je kao novi tip Expr koristeći algebarske tipove Haskell-a.

```
type Name = String

data Expr
  = Var Name
    | App Expr Expr
    | Lam Name Expr
    | Let Name Expr Expr
    | Lit Lit
    | Fix Expr
    | If Expr Expr Expr
    | Op Binop Expr Expr
    deriving (Show, Eq, Ord)
```

```
data Lit
  = LInt Integer
  | LBool Bool
  deriving (Show, Eq, Ord)

data Binop = Add | Sub | Mul | Eql
  deriving (Show, Eq, Ord)
```

Data je implementacija ML tipova - monotipova:

```
data Type
  = TVar TVar
  | TCon String
  | TArr Type Type
  deriving (Eq, Ord)
```

I tipskih shema:

```
data Scheme = Forall [TVar] Type
  deriving (Show, Eq, Ord)
```

Potrebni su nam i tipovi konstanti koji predstavljaju konkretne tipove u Herbrandovom univerzumu (tipske konstruktore 0-arnosti):

```
typeInt, typeBool :: Type
typeInt  = TCon "Int"
typeBool = TCon "Bool"
```

Sintaksa restrikcija Sintaksa restrikcija blisko prati sintaksu koja je data u prvom delu uz minimalne modifikacije koje su praktične prirode. CScheme i Constr su uzajamno definisani. CScheme je tip sa jednim tipskim konstruktorom CForall koji predstavlja restriktovano tipsku shemu i kao jedan od argumenata ima listu restrikcija.

```
data CScheme = CForall [TVar] [Constr] Type
deriving (Eq, Ord)
```

Sintaksa restrikcija Constr prati BN sintaksu datu u prvom poglavlju sa razlikom da ne postoje konstruktori za konjunkciju i egzistenciju restrikcija. U implementaciji konjukcija restrikcija je predstavljena listom. Uloga $\exists$ kvantifikatora u originalnoj postavci je ograničavanje dometa promenljive i izbegavanje problema sa konfliktom imena. Međutim, u implementaciji koristimo drugačije rešenje - generisanje uvek svežih imena promenljivih, tako da se na ovaj način izbegava problem sa konfliktom i eksplicitno korišćenje $\exists$ kvantifikatora nije neophodno. Iz istog razloga nije neophodno korišćenje ni $\forall$ kvantifikatora, međutim radi preglednosti sintaksa restrikcija i sheme ipak sadrži ovaj kvantifikator. Data je deklaracije restrikcija:

```
data Constr
  = CEq Type Type
  | CInstN Name Type
  | CInst CScheme Type
  | CDef Name CScheme [Constr]
```

```
  | CLet Name CScheme [Constr]
  deriving (Eq, Ord)
```

Što se tiče modelovanja novih tipova u Haskell-u u ovom delu potreban je još pojam tipske supstitucije. Znamo da je rešenje skupa restrikcija tispka supstitucija. Supstituciju je preslikavanje tipskih promenljivih na tipove i možemo je modelovati koristeći Haskell mapu iz standardne biblioteke:

```
newtype Subst = Subst (Map.Map TVar Type)
 deriving (Eq,Ord,Show,Monoid)
```

Do sada smo deklarisali sintaksu tipova i restrikcija za podskup ML jezika. Potrebne su i određene operacija kao što je supstitucije nad tipovima, ali i nad restrikcijama. Uniforman tretman jedne operacije nad više različitih tipa, kao što je ovaj slučaj, možemo postići *tzv. ad-hok* polimorfizmom koji je u Haskell-u implementiran pomoću tipskih klasa. Deklarisaćemo novu klasu Substitutable sa dve polimorfne funkcije apply za primenjivanje supstitucije i ftv koja vraća skup slobodnih promenljivih:

```
class Substitutable a where
  apply :: Subst -> a -> a
  ftv   :: a -> Set.Set TVar
```

Sad ćemo definisati instance ove klase, kao što je rečeno, to su tipovi (sa tipksim shemama) i restrikcije (sa restriktovanim tispkih shemama). Ovde ćemo dati implementaciju samo za instancu tipova, a celokupna implementacija za ostale instance se nalazi u dodatku.

```
instance Substitutable Type where
  apply _ (TCon a) = TCon a
  apply (Subst s) t@(TVar a) = Map.findWithDefault t a s
  apply s (t1 `TArr` t2) = apply s t1 `TArr` apply s t2

  ftv TCon{} = Set.empty
  ftv (TVar a) = Set.singleton a
  ftv (t1 `TArr` t2) = ftv t1 `Set.union` ftv t2
```

Što se tiče apply funkcije, rekurzivno se pretražuje stablo tipa - samo u slučaju kad je čvor stabla neka tipska promenljiva TVar a i kad promenljiva sa imenom postoji u mapi zamenićemo promenljivu za odgovarajući tip.

5.1 Generisanje restrikcija

U ovom delu prikazan je algoritam za generisanje restrikcija.

Kao što smo rekli, implementacija koristi generisanje uvek novih imena promenljivih da bi se izbegli konflikti sa imenima. Da bismo ovo postigli, definisacemo beskonačnu različitih listu *string*-ova, koristeći Haskell-ovo lenjo izvršavanje:

```
letters :: [String]
letters = [1..] >>= flip replicateM ['a'..'z']
```

Za definisanje letters koristi se lista kao monada, odnosno koristi se odgovarajući *bind* operater. Za slućaj liste, *join* funcija je konkatenacija. Ukratko, funkcija koja je drugi argument operatora (>>=) konzumira elemente liste koja je prvi argument i za svaki mora da proizvodi listu kao rezultat. Rezultat (>>=) je konkatencija rezultata. Pogledajmo konkretni slućaj, prvi argument je beskonačna lista [1..]. Drugi argument je funkcija koja koristi funkciju:

```
replicateM :: Monad m => Int -> m a -> m [a]
```

koja proizvodi listu replikovanih argumenata a. **flip** je pomoćna funkcija koja obrće redosled argumenata. Znači, beskonačna lista se konstruiše tako što za svaki $n :: Int$ iz beskonačne liste $replicateM$ replikuje svaki karakter iz liste $['a'..'z']$ n-puta gradeći listu liste karaktera tj. *string*-va i na kraju *bind* operater vraća konkateniranu listu. U nastavku ćemo videti kako se ova lista koristi prilikom generisanja restrikcija.

Infer Algoritam za ulaz koji je izraz tipa $Expr$, odnosno program u podskupu ML-a koji smo definisali, treba da proizvede skup tipskih restrikcija i tip. Principijelni tip za izraz se dobija kad na dobijeni tip primenimo susptituciju koja je rešenje restrikcija. Pored toga, potrebni su i propratni efekti: stanje i mehanizam za upravljanje greškama. Stanje se koristi kao evidencija poslednjeg uzetog imena iz beskonačne liste imena. Da bismo ovo modelovali, definisaćemo monadu $Infer$:

```
type Infer a = StateT InferState (Except TypeError) a
```

gde je:

```
data InferState = InferState { count :: Int }
```

Stanje je predstavljeno Int tipom podatka koji je samo upakovan u *count* funkciju koja ima tip $count : InferState \to Int$.

Ova definicija koristi transformer monadu StateT. Transfomeri monada omogućavaju ugnježđavanje monada. State je monada koja ima dva parametra: tip stanja i tip rezultata. Transfomer verzija State monade omogućava da tip rezultata takođe bude monada. U ovom slučaju to je Except e a. Na ovaj način je tip a dekorisan sa dve monade.

State monada Kako je definisana *State* monada u Haskell-u? Kao što je rečeno, cilj *State* monada je modelovanja stanja programa. Funkcije mogu da pristupaju i da menjaju stanje. Neka je f npr. neka funkcija u imperativnom jeziku koja pored "čistog" dela koji ćemo obeležiti sa $f' : a \to b$ takođe pristupa i menja neke globalne promenljive tj. stanje. Da bismo ovo obuhvatili u funkcionalnom jeziku, stanje možemo predstaviti tipom s i modelovati funkciju kao $f : (a, s) \to (b, s)$. Međutim, ako želimo da ovaj efekat modelujemo kao monadu, data funkcija ne uklapa sa definicijom *Kliesli* kategorije pošto su dekorisani i argument i rezultat funkcije. Za monadu je između ostalog potrebna Haskell funkcija tj. prirodna transformacija

oblika $f : a \rightarrow m\ b$. Datu funkciju možemo jednostavno transformisati koristeći *currying* u ovaj oblik, $g : a \rightarrow (s \rightarrow (b, s))$. Funkcija sa ovakvim potpisom intuitivno govori šta zapravo g radi. Za argument a funkcija proizvodi funkciju $s \rightarrow (b, s)$.

Sada vidimo kakva dekoracija je potrebna za modelovanje stanja i pomoću nje se definiše tip *State*.

```
newtype State s a = State (s -> (a,s))
```

gde je s tip stanja, a a tip rezultata. U Haskell-u je zapravo tip *State* dat na malo drugačiji nacin:

```
newtype State s a = State  {runState :: s -> (a,s)}
```

Ovde je funkcija stanja samo upakovana sa *runState*-om koji predstavlja način da se pristupi ovoj funkciji tj.:

```
runState :: State s a -> (s -> (a,s))
```

Da se ne bi morao koristiti *pattern matching* da bi se pristupilo funkciji ispod tipskog konstruktora. U nastavku je data implementacija **return** i (>>=) operatora.

```
instance Monad (State s) where
return x = State $ \s -> (x,s)
(State g) >>= f = State $ \s -> let (a, s') = g s
                                    (State h) = f a
                                in h s'
```

return za dato x vraća funkciju koja ne menja stanje s već ga samo prosleđuje. Kompozicija koja je data u vidu *bind* (>>=) operatora, slaže efekte nad stanjem s. *State* g predstavlja rezultat tipa a i modifikaciju na stanjem s. Funkcija f je funkcija $f : a \rightarrow (s \rightarrow (b, s))$ - za argument a vraća b uz modifikaciju stanja. (>>=) slaže modifikacije koje nad stanjem pravi g i rezultat f-a tako što stanje dobijeno nakon proizvodnje a argumenta šalje kao ulazno stanje za proizvodnju b argumenta.

Glavna funkcija za generisanje restrikcija obeležena je sa $infer$. Kao što već napomenuto, ova funkcija za ulazni tip ima $Expr$, a izlaz treba bude tip i skup restrikcija koji u okviru sintakse koju smo razvili možemo predstaviti sa $(Type, [Constr])$. Međutim, da bi funkcija podržavala navedene propratne efekte dekorisacemo izlaz sa $Infer$ monadom. Na osnovu ovoga, funkcija je tipa $infer : Expr \rightarrow Infer\ (Type, [Constr])$

Jedina pomoćna funkcija koja je preostala da se definiše za glavnu funkciju algoritma je funkcija koja generiše nove tipske promenljive. Da bi ova funkcija mogla da se koristi jednostavno u **do** notaciji $infer$ funkcije definisana je kao $Infer$ monada takodje:

```
fresh :: Infer Type
fresh = do
  s <- get
  put s{count = count s + 1}
  return $ TVar $ TV ((letters !! count s) ++ "'")
```

Ova funkcije je i jedino mesto gde se pristupa stanju iz $Infer$ monade i razlog zašto je uopšte potrebno stanje za implementaciju algoritma. Rečeno je da stanje služi kao evidencija prilikom generisanja novih imena i da je tipa Int, pošto se nova imena nalaze u beskonačnoj listi, stanje predstavlja indeks koji se uveceva nakon jednog "uzimanje" imena iz liste. $fresh$ prvo preuzima trenutno stanje, upisuje stanje povećano za jedan i vraća tipsku promenljivu sa imenom iz liste sa indeksom $count\ s$.

Infer funkcija Sada je definisano sve što je potrebno za implementaciju $infer$ funkcije za generisanje restrikcija. $infer$ je rekurzivna funkcija koja prebrisava stablo $Expr$ i na svakom čvoru uvodi nove tipske promenljive i restrikcije. Funkcija koristi *pattern matching* po $Expr$. Implementacija prati translaciju izraza u restrikcije uz određene razlike. Naime, razlika potiče od toga što je translacija, kako je definisana, binarna operacija odnosno preslikavanje para (a, τ), gde je a izraz i τ tip, u restrikciju c. Dok je $infer$ preslikavanje izraza a u par $(\tau, [Constr])$. To će rezultovati u tome da samo neke restrikcije iz restrikcija u translaciji budu ugrađena u tip u slučaju $infer$ funkcije. Razlika se vidi u slučajevima za *App* i *Lam*.

```
infer :: Expr -> Infer (Type,[Constr])
infer expr = case expr of
      ...
```

U nastavku će biti prikaži svi slučajevi u *pattern matching*-u:

Konstante Prvi slučajevi su za konstante koje smo definisali. Za ove slučajeve vraćaju se predefinisani tipovi i prazna lista restrikcija.

```
Lit (LInt _) -> return $ (typeInt,[])
Lit (LBool _) -> return $ (typeBool,[])
```

Slučaj za promenljivu je zapravo isti kao translacija 2.5.

```
Var x -> do
  tv <- fresh
  return (tv, [CInstN x tv])
```

Generiše se nova tipska promenljiva i uvodi se restrikcija $x \leq tv$. Kao što je navedeno u prvom delu ovde se koristi relacija instance umesto jednakost, zato što ukoliko je x uvedeno pomoću *let* konstrukcije, može da označava tipsku shemu.

```
Lam x e -> do
  (t, c) <- infer e
  tv <- fresh
  return (tv `TArr` t, [CDef x (CForall [] [] tv) c])
```

U translaciji se da to da tip izraza bude strelica tip dodaje kao restrikcija, dok je ovde ta restrikcija dato u povratnom tipu. Ukratko, rekurzivno se poziva $infer$ da

bi se dobio tip i restrikcije (t, c) za podizraz e, generiše se nova tipska promenljiva za x, vraća se povratni tip strelica i restrikcija eksplicitne supstitucije x za tip tv u restrikcijama c. Takođe, još jedna razlika jeste da se x u okviru $CDef$-a vezuje za shemu tipa, umesto za monotip kako je dato u translaciji. Poznato da je monotip podskup tipskih shema koje su podskup restriktovanih tipskih shema, ali ovde to moramo eksplicitno navesti.

```
App e1 e2 -> do
  tv <- fresh
  (t1,c1) <- infer e1
  (t2,c2) <- infer e2
  return (tv, c1 ++ c2 ++ [CEq t1 (t2 `TArr` tv)])
```

Iz *App* i *Lam* slučaja vidi se razlika u odnosu na translaciju. Razlika je u smeru propagiranja tipa, u translaciji se tip propagira od gore kad dole, dok se u implementaciji propagira od dole ka gore. Za *App e1 e2* ukoliko je izraz pravilno tipiziran, znamo da $e1$ mora imati strelica tip. U translaciji se to osigurava tako što se to propagirati na sledeći način $[\![e1 : \alpha \to \tau]\!]$. Dok se ovde poziva rekurzivno $infer$ na argumentu *e1* i onda se nad dobijenim tipom postavlja restrikcija da mora biti strelica tip, `CEq t1 (t2 `TArr` tv)`. Takođe, pravi se kolekcija svih ostalih restrikcija koja su prikupljena.

```
Let x e1 e2 -> do
  (t1,c1) <- infer e1
  let sc = genCScheme t1 c1
  (t2,c2) <- infer e2
  return (t2, [CLet x sc c2])
```

Slučaj za deklaraciju **let** prati blisko translaciju. Ovaj slučaj koristi pomoćnu funkciju `genCScheme` za generisanje tipske sheme sa restrikcijama.

```
Fix e1 -> do
   (t1,c1) <- infer e1
   tv <- fresh
   return (tv, c1 ++ [CEq (tv `TArr` tv) t1])
```

Ovo je slučaj za operator za fiksnu tačku. Znamo da podizraz $e1$ mora imati tip strelica tip $\tau \to \tau$ za neko τ i to se samo forsira u restrikcijama sa `CEq (tv `TArr` tv) t1`.

Za slučaj binarnih operatora predefinisani su tipovi za operatare:

```
ops :: Binop -> Type
ops Add = typeInt `TArr` (typeInt `TArr` typeInt)
ops Mul = typeInt `TArr` (typeInt `TArr` typeInt)
ops Sub = typeInt `TArr` (typeInt `TArr` typeInt)
ops Eql = typeInt `TArr` (typeInt `TArr` typeBool)
```

Slučaj za binarne operatore:

```
Op op e1 e2 -> do
    (t1,c1) <- infer e1
    (t2,c2) <- infer e2
    tv <- fresh
    return (tv, c1 ++ c2 ++ [CEq (ops op) (t1 `TArr` (t2 `TArr` tv))])
```

Ovaj slučaj koristi predefinisane tipove za operatore. U restrikcijama je restriktovano da se u zavisnosti od binarne operacije tipovi argumenata slažu, a povratni tip je izlazni tip za binarnu operaciju.

```
If cond tr fl -> do
      (t1,c1) <- infer cond
      (t2,c2) <- infer tr
      (t3,c3) <- infer fl
      return (t2, c1 ++ c2 ++ c3 ++ [CEq t1 typeBool, CEq t2 t3])
```

U restrikcijama se forsira da *cond* izraz ima tip *bool* i da podizrazi u slučaju *true* i $false$ imaju išti tip, što je takođe i povratni tip.

Ovo zaključuje sve slučajeve izraza *Expr* za generisanje restrikcija. Kao što je i praksa za monade u Haskell-u, definisana je i pomoćna funkcija za pokretanje generisanja runInfer. Pošto je $infer$ stek monada *State* i *Except* sve što je potrebno jeste proslediti inicijalno stanje:

```
runInfer :: Infer (Type, [Constr]) -> Either TypeError (Type, [Constr])
runInfer m = runExcept $ evalStateT m initInfer
```

Inicijalno stanje initInfer je:

```
initInfer :: InferState
initInfer = InferState {count = 0}
```

5.2 Rešavanje restrikcija

U nastavku dat je pregled implementacije rešavanja restrikcija koja su generisana pomoću funkcije date u prethodnom poglavlju. Ono što je predstavljeno u teorijskom prvom delu rada, u ovom poglavlju ćemo translirati u Haskell program.

5.2.1 Transformacija restrikcija

solver funkcija je unifikator prvog reda koji radi po pravilima unifikacije datih u slici 5.1. Sintaksa restrikcija data je sa:

```
data Constr
  = CEq Type Type
  | CInstN Name Type
  | CInst CScheme Type
  | CDef Name CScheme [Constr]
  | CLet Name CScheme [Constr]
  deriving (Eq, Ord)
```

Unifikator prihvata samo restrikcije sa sintaksom CEq Type Type, koja označava jednakost, odnosno listu ovakvih restrikcija. Kao što je rečeno, konjunkcija je predstavljana listom, a egzistencijalni kvanitifikator nije neophodan pošto je garantovano da su sve imena promenljivih jedinstvena. Neophodna je translacija restrikcija Constr tako da ovo bude zadovoljeno.

U teorijskom delu date su ekvivalencije koje omogućavaju ovavku translaciju. Koristićemo pomenute ekvivalencije da definišemo translaciju. Takođe, rečeno je da ovako definisane restrikcije sa formom eksplicitne supstitucije omogućavaju određenu optimizaciju. Naime, pre supstitucije moguće je prvo pronaći rešenu formu za restrikcije iz tipske shemu, tako da se izbegne rešavanje istog skupa restrikcija više puta. Naša implementacija podržava ovaj vid optimizacije i u nastavku je dat prikaz.

Translacija koju ćemo nazvati *flatten* se sastoji iz dva dela: supstitucije i prezapisivanje za instance. Odnosno, sledeća pravila prezapisivanja:

$$\textbf{def}\ x : \sigma\ inC \longrightarrow [x \mapsto \sigma^s]C$$

$$\textbf{let}\ x : \sigma\ inC \longrightarrow [x \mapsto \sigma^s]C$$

$$(\forall \bar{\alpha}[C].\tau) \preceq \tau' \longrightarrow \exists \bar{\alpha}.(C \wedge \tau = \tau')$$

gde je σ^s rešena forma (sa praznim skupom restrikcija) za restriktovanu tipsku shemu σ. Slućaj **let** je išti kao **def** zato što se rešavanjem restrikcija u σ implicitno proverava da li postoje instance (sto je bio dodatni uslov za **let**).

Pošto će se koristiti unifikator prilikom translacije restrikcija, potrebno je i ovde uključiti mehanizam za greške. Pa ćemo definisati tip koji uključuje samo monadu za greške:

```
type Excpt a = ExceptT TypeError Identity a
```

Rešavanje restriktovanih tipskih shema treba da se vrši na svakom nivou u restrikcijama. Transformaciju je između ostalog potrebno izvršavati na listi restrikcija. Iz ovog razloga potreban je način da se mapira funkcija koja može biti terminisana greškom $f : a \to Excpt\ b$ na listu, ali tako da konačni rezultat bude lista $[b]$ ili greška. Odnosno, ukoliko bi se primenila odgovarajuća $fmap$ funkcija za listu $fmap :: (a \to b) -> [a] -> [b]$, rezultat bi bio lista tipa $[Excpt\ b]$. Što se svakako ne slaže sa namenom monada za greške - greška treba da se vrati čim se prvi put pojavi. Za ovu svrhu definisacemo sledeću funkciju:

```
mapExcpt :: (a -> Excpt b) -> [a] -> Excpt [b]
mapExcpt f (hd:tl) = do
  rhd <- f hd
  rtl <- mapExcpt f tl
  return (rhd:rtl)
mapExcpt f _ = return []
```

Transliranje restrikcija se sastoji iz dve funkcije: flattenConstrSupst i flattenConstrInst sa tipom [Constr] -> Excpt [Constr]. Prva je supstitucija, a druga je prezapisivanje instance. Ove funkcije mapiraju funkcije rewriteSupst i rewriteInst respektivno na liste restrikcija.

rewriteSupst i solveFlatten su dve uzajamno rekurzivne funkcije. Funkcija solveFlatten koristi za rešavanja liste restrikcija u restriktovanoj tipskoj shemi:

```
solveFlatten :: [Constr] -> Excpt Subst
solveFlatten cons = do
  cons' <- fmap concat $ mapExcpt rewriteSupst cons
  su <- let fcons = flattenConstrInst cons' in
        evalStateT solver (emptySubst, fcons)
  return su
```

Funkcija solveFlatten prvo primenjuje funkciju solveFlatten na restrikcije cons. Rezultujuće restrikcije cons' su supstituisana forma restrikcija tj. ne sadrže forme eksplicitne supstitucije **def** ili **let**. To znači da se sad može bezbedno izvršiti prezapisivanje instanci pomoću funkcije flattenConstrInst. Nakon prezapisivanja dobija transformisana forma restrikcija za koju se pomoću rešavača može pronaći rešenje u obliku supstitucije.

rewriteSupst funkcija je implementirana na sledeći nacin:

```
rewriteSupst :: Constr -> Excpt [Constr]
rewriteSupst (CLet n cs cons) = case cs of
  CForall as' [] t' ->
      fmap (map (supstName n cs))
           (fmap concat $ mapExcpt rewriteSupst cons)
  CForall as' cons' t' -> do
    su <- solveFlatten cons'
    fmap (map $supstName n (CForall as' [] $apply su t'))
         (fmap concat $ mapExcpt rewriteSupst cons)
rewriteSupst (CDef n cs cons) =   ae   cs of
  CForall as' [] t' ->
      fmap (map (supstName n cs))
           (fmap concat $ mapExcpt rewriteSupst cons)
  CForall as' cons' t' -> throwError $ DefaultFail
rewriteSupst x = return [x]
```

Prvo primetimo da se nakon supstitucije za jednu restrikciju dobija lista restrikcija. Iz tog razloga se restrikcije koje nisu CLet ili CDef vraćaju upakovana u listu. Kao što vidimo, ukoliko tipska shema sadrži praznu listu restrikcija, susptitucija može odmah da se izvrši. Za ovu svrhu se koristi:

```
supstName :: Name -> CScheme -> Constr -> Constr
```

zamena imena za tipsku shemu u restrikciji. Supstitucija se vrši tako što se mapira parcijalno primenjena funkcija supstName ncs na listu restrikcija $[Constr]$. Da bi se forsirala dalja supstitucija, rewriteSupst se rekurzivno poziva tj. mapira na listu restrikcija i supstitucija za tipsku shemu se vrši nad ovakvim restrikcijama. Ovde

se koristi preslikavanje koje smo definisali u funkciji mapExcpt koje je opisano. U slučaju kada lista restrikcija u tipskoj shemi nije prazna, prvo se rešavaju restrikcije primenjujući funkciju solveFlatten. Implementacija je dalje slična kao za prvi slučaj, osim što se ime suptituiše sa rešenom formom koja se dobija primenjujući dobijenu supstituciju. (CForall as' [] $apply su t'). Pošto je rezultat funkcije dekorisan kao Excpt[Constr] moramo koristi *fmap*-e kako bi se funkcije primenile ispod dekoricacije tj. na [Constr].

Pogledajmo još implementaciju funkcije suspt:

```
supstName :: Name -> CScheme -> Constr -> Constr
supstName n cs constr =   ae    constr of
                          CEq t1 t2 -> CEq t1 t2
                          CInstN n1 t | n == n1 -> CInst cs t
                                      | otherwise -> CInstN n1 t
                          CDef n1 cs1 cons | n == n1 -> constr
                                           | otherwise ->
                            CDef n1 cs1 (map (supstName n cs) cons)
                          CLet n1 cs1 cons | n == n1 -> constr
                                           | otherwise ->
                            CLet n1 cs1 (map (supstName n cs) cons)
                          _ -> constr
```

Ovde je treba napomenuti da se vodi računa o dometu imena. Npr. ako se u CDef ukoliko se vezuje promenljiva istog imena, onda se neće vršiti dalja zamena.

Kada su definisane gradivne funkcije za transliranje restrikcije, sad možemo definisati funkciju koja će ih objediniti tako da za listu restrikcija dobije lista transliranih restrikcija (uz moguću gresku):

```
flattenConstr :: [Constr] -> Excpt [Constr]
flattenConstr cons = do
  fcons <- flattenConstrSupst cons
  return $flattenConstrInst fcons
```

Takođe, možemo definisati solver koji će raditi direktno sa restrikcijama koje daje faza za generisanje.

```
runSolver' :: [Constr] -> Excpt Subst
runSolver' cons =  do
  fcons <- flattenConstr cons
  su <- evalStateT solver (emptySubst,fcons)
  return su
```

U ovom je sličan onom runSolver sa tim što pre rešavanja transliraju restrikcije koristeći flattenConstr.

Za transliranje restrikcija se koristi unifikator tj. solver, kao što smo videli. Pa može delovati da dve faze više nisu nezavisne. Treba napomenuti da su faze generisanja i rešavanja su i dalje razdvojene, pošto umesto solver možemo koristi bilo koju drugu implementaciju sve dok je specifikacija ista.

5.2.2 Unifikacija

Svakako najznačajniji deo rešavanja restrikcija je **unifikacija**. Generalna definicija jeste da unifikacija je algoritam za rešavanje jednačina između simboličkih izraza. U našem slućaju izrazi su tipovi. Ukoliko su τ i τ' dva tipa, unifikator je supstitucija θ takva da važi:

$$\theta := [\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, ..., \alpha_k \mapsto \tau_k]$$
$$[\theta]\tau = [\theta]\tau'$$

Data su pravila za unifikaciju za podskup ML jezika koji koristimo:

$$\textsc{uni-const}\quad c \sim c : []$$

$$\textsc{uni-var}\quad \alpha \sim \alpha : []$$

$$\textsc{uni-VarLeft}\quad \frac{\alpha \notin ftv(\tau)}{\alpha \sim \tau : [\alpha \mapsto \tau]}$$

$$\textsc{uni-varRight}\quad \frac{\alpha \notin ftv(\tau)}{\tau \sim \alpha : [\alpha \mapsto \tau]}$$

$$\textsc{uni-con}\quad \frac{\tau_1 \sim \tau_1' : \sigma_1 \qquad [\sigma_1]\tau_2 \sim [\sigma_1]\tau_2 : \sigma_2}{\tau_1\ \tau_2 \sim \tau_1'\ \tau_2' : \sigma_2 \circ \sigma_1}$$

$$\textsc{uni-arrow}\quad \frac{\tau_1 \sim \tau_1' : \sigma_1 \qquad [\sigma_1]\tau_2 \sim [\sigma_1]\tau_2 : \sigma_2}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' : \sigma_2 \circ \sigma_1}$$

Slika 5.1: Pravila unifikacije

Kao što vidimo, unifikacija je jednostavna. U osnovi, pretražuje se po strukturi dva tipa sve dok se ne dođe do forme $\alpha \sim \tau$ koja direktno daje supstituciju $[\alpha \mapsto \tau]$ kao što se vidi u pravilima uni-VarLeft i uni-VarRight. I takođe prave se kompozicije ovih supstitucija tokom pretrage po strukturi kao što vidi u pravilima uni-Con i uni-Arrow. Jedini problem za unifikaciju može biti za slučaj $\alpha \to \tau$ kad se α pojavljuje slobodno u τ. Za ovakav slućaj ne postoji unifikator pošto npr. $\alpha = \alpha \to \beta$ susptitucija bi bila $\theta = [\alpha \mapsto \alpha \to \beta]$. Međutim nakon supsitucije $\theta\alpha = \theta(\alpha \to \beta)$ uvek će drugi tip biti veći. Jedino bi bilo moguće kad bi postojao tip beskonačne dužine, međutim u jeziku koji koristimo ne postoje takvi tipovi. Da bi se ovo sprečilo u premisama pravila se nalaži uslov $\alpha \notin ftv(\tau)$.

U implementaciji za ovu svrhu koristi se funkcija occursCheck.

```
occursCheck :: Substitutable a => TVar -> a -> Bool
occursCheck a t = a `Set.member` ftv t
```

Pošto je definisana tipska klasa Substitutable implementacija ove funkcije je jednostavna pošto koristi odgovarajuću ftv funkciju.

Podsetimo se supstitucija je definisana kao **newtype**Subst = Subst (Map.Map TVar Type).

Definisana je funkcija za kompoziciju supstitucija na sledeći nacin:

```
compose :: Subst -> Subst -> Subst
(Subst s1) `compose` (Subst s2) =
        Subst $ Map.map (apply (Subst s1)) s2 `Map.union` s1
```

Implementacija unifikatora se sastoji iz dve uzajmno rekurzivne funkcije unifies i unifyMany. Prva funkcija rešava pojedinačno restrikcije, a druga listu restrikcija. Rezultat obe funkcije treba da bude supstitucija. I ovde je potreban mehanizam za greške i stanje. Ove dve funkcije ne koriste stanje - stanje se koristi u glavnoj funkciji za unifikaciji, gde je potrebno da se čuva privremeni rezultat rešavanja, što se može videti u nastavku. Stanje je Unifier, a monada je Solve i definisani su na sledeći nacin:

```
type Unifier = (Subst, [Constr])
type Solve a = StateT Unifier (ExceptT TypeError Identity) a
```

Prikazana je implementacija funkcija:

```
unifies :: Constr -> Solve Subst
unifies (CEq (TVar v) t) = v `bind` t
unifies (CEq t (TVar v)) = v `bind` t
unifies (CEq (TArr t1 t2) (TArr t3 t4)) = unifyMany [CEq t1 t3, CEq t2 t4]
unifies (CEq t1 t2) | t1 == t2 = return emptySubst

unifyMany :: [Constr] -> Solve Subst
unifyMany [] = return emptySubst
unifyMany (cs:cs1) =
  do su1 <- unifies cs
     su2 <- unifyMany (apply su1 cs1)
     return (su2 `compose` su1)
```

Kao što vidimo, prva dva slučaja odgovaraju pravilima za promenljive. Treći slučaj za upoređivanje strelica tipova konstruiše listu restrikcija i poziva unifyMany funkciju. unifyMany rešava listu restrikcija. Za rešenu restrikciju dobija se supstitucija su1 koja se onda primenljuje na ostatak restrikcija iz liste. Zatim se ostatak rešava rekurzivno. Provera occursCheck se dešava u bind funkciji.

```
bind :: TVar -> Type -> Solve Subst
bind a t | t == TVar a     = return emptySubst
         | occursCheck a t = throwError $ InfiniteType a t
         | otherwise       = return (Subst $ Map.singleton a t)
```

Ova funkcija vraća supsituciju ili grešku koja je opisana gore.

Kada su definisane ove dve funkcije, možemo definisati i glavnu funkciju za rešavanje restrikcija solver:

```
solver :: Solve Subst
solver = do
  (su, cs) <- get
    ae    cs of
    [] -> return su
    (cs:cs1) -> do
      su1 <- unifies cs
      put (su1 `compose` su, apply su1 cs1)
      solver
```

Za razliku od unifiesMany funkcije koja za argument ima listu restrikcija, ova funkcija taj argument dobija kao inicijalno stanje dok je sam mehanizam isti kao

i kod unifiesMany. Rešava se jedna po jedna restrikcija iz liste i privremen rezultat tj. supsitucija i preostala ogranicenaj se čuvaju u stanju.

Implementirana je i pomoćna funkcija za pokretanje rešavanja koja listu restrikcija šalje kao inicijalno stanje i otpakuje transformatore monada tako da je finalni rezultat grešku ili supstitucija:

```
runSolver :: [Constr] -> Either TypeError Subst
runSolver cs = runIdentity $ runExceptT $ evalStateT solver st
  where st = (emptySubst, cs)
```

Na kraju cemo dati nekoliko primera izraza.

Primer 9. Izraz:

```
let x = (let x = \x.x in x) in x
```

Restrikcije:

```
(d',[let x : All[c'][let x : All[a',b'][def x :
    All[][].b' in [x <= a']].b' --> a' in [x <= c']].c' in [x <= d']])
```

Transformisane restrikcije:

```
[a' --> a' = d']
```

Resena forma:

```
Subst (fromList [(d',a' --> a')]) \\
```

Principelni tip: a' -> a'

Primer 10. Izraz:

```
let y = (let z = \x.x in z) in y 2}
```

Restrikcije:

```
(d',[let y : All[c'][let z : All[a',b'][def x :
     All[][].b' in [x <= a']].b' --> a' in [z <= c']].c' in [y <= e',e' = Int
     --> d']])
```

Transformisane restrikcije:

```
[a' --> a' = e',e' = Int --> d']
```

Resena forma:

```
Subst (fromList [(a',Int),(d',Int),(e',Int --> Int)])
```

Principijelni tip: Int

Zaključak

Postojanje principijelnog tipa i algoritam za određivanje tipova je ključna karakteristika ML jezika. ML je poslužio kao osnova za čitavu familiju funkcionalnih programskih jezika. ML je inspirisao jezike kao što su Haskell, Miranda, OCaml, Rust, Scala itd. Ovi jezici se razvijaju na osnovi ML jezika i često se implementiraju nadogradnje i nove mogućnosti kao što je egzistencijalni tipovi, ad-hok polimorfizam itd. Treba napomenuti da neke nadogradnje kao što su generalizovani algebarski tipovi (skraćeno GADTs) predstavljaju izuzetno težak problem za određivanje tipova. Sva do sad poznata rešenja uvode neke vidove ograničenja ekspresivnosti GADTs-a. Jedno od predloženih rešenja predstavljeno je u [17].

Restrikcije kao među-jezik i modularna implementacija omogućavaju jednostavnije proširenje algoritma tako da se podrže nove konstrukcije programskih jezika. U idealnom slučaju, prevodilac za programski jezik bi trebalo samo da proizvede skup restrikcija kao među-prezentaciju, a da se za njihovo rešavanje koristi nezavisan zajednički rešavač. Iako su teorijske osnove restrikcija za određivanje tipova dobro poznate, primeri implementacija nisu toliko česti. Iz tog razloga fokus ovog rada bio je i na teorijskim osnovama, ali i na imlementaciji na čisto funkcionalnom jeziku.

Treba napomenuti da srodan problem poznat kao tipska rekonstrukcija nije isto što i određivanje tipova koje je prezentovano u ovom radu. Tipska rekonstrukcija podrazumeva elobaraciju implicitno tipiziranog terma u eksplicitno tipizirani. Iako naizgled veoma slični, ovaj problem uvodi nove probleme drugačije prirode i zahteva značajnu nadogradnju algoritma za određivanje tipova. Za više detalja o tipskoj rekonstrukciji pogledati [18].

Reference

[1] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, Camridge, Massauchesetts, London, England, 2005.

[2] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[3] Xavier Leroy. *The Objective Caml system: Documentation and user's manual*. 2000. http://caml.inria.fr.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. 1982.

[6] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[7] C.B. Yelles. *Type Assignment in the Lambda-Calculus: Syntax and Semantics*. PhD thesis, University of Wales, 1979.

[8] R.Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 1969.

[9] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[10] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/milner style type systems in constraint form. Technical report, 1999.

[11] François Pottier. A Semi-Syntactic Soundness Proof for HM(X). Research Report RR-4150, INRIA, 2001.

[12] Benjamin C. Pierce. *Basic Category Theory for Computer Science*. MIT Press, Cambridge, MA, USA, 1991.

[13] David I. Spivak. *Category Theory for the Sciences*. MIT Press, 2014.

[14] Bartosz Milewski. Category theory for programmers. https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface.

[15] Mark P. Jones. Typing haskell in haskell. In *Haskell Workshop*, 1999.

[16] Stephen Diehl. Write you a haskell, 2016. http://dev.stephendiehl.com/fun/.

[17] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, September 2006.

[18] Sunil Kothari. Type reconstruction algorithms - a survey. 2007.

[19] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Camridge, Massauchesetts, London, England, 2002.

[20] Luca Cardelli. *Type Systems*. CRC Press, 1997.

[21] Miran Lipovača. *Learn You a Haskell for Great Good!*

[22] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[23] O'Sullivan. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[24] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[25] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag New York, 1978.

[26] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[27] Michael Barr and Charles Wells, editors. *Category Theory for Computing Science, 2Nd Ed.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

Dodatak

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module Infer where

-- imports
import Control.Monad.State
import Control.Monad.Except
import Control.Monad.Identity
import Data.Monoid

import Data.List (nub)
import qualified Data.Map as Map
import qualified Data.Set as Set

-- ------------------------------------------------------------------------------
-- Syntax: expressions, types and constraints
-- ------------------------------------------------------------------------------

-- | Expressions
type Name = String

data Expr
  = Var Name
    | App Expr Expr
    | Lam Name Expr
    | Let Name Expr Expr
    | Lit Lit
    | Fix Expr
    | If Expr Expr Expr
    | Op Binop Expr Expr
    deriving (Show, Eq, Ord)

data Lit
  = LInt Integer
  | LBool Bool
  deriving (Show, Eq, Ord)

-- | Binary operations
```

```
data Binop = Add | Sub | Mul | Eql
  deriving (Eq, Ord, Show)

-- | Types
newtype TVar = TV String
  deriving (Eq, Ord)

data Type
  = TVar TVar
  | TCon String
  | TArr Type Type
  deriving (Eq, Ord)

-- | Type scheme
data Scheme = Forall [TVar] Type
  deriving (Show, Eq, Ord)

-- | Supstitution
newtype Subst = Subst (Map.Map TVar Type)
  deriving (Eq,Ord,Show,Monoid)
-- ovde izbrisi show posle
class Substitutable a where
  apply :: Subst -> a -> a
  ftv   :: a -> Set.Set TVar

instance Substitutable Type where
  apply _ (TCon a) = TCon a
  apply (Subst s) t@(TVar a) = Map.findWithDefault t a s
  apply s (t1 `TArr` t2) = apply s t1 `TArr` apply s t2

  ftv TCon{} = Set.empty
  ftv (TVar a) = Set.singleton a
  ftv (t1 `TArr` t2) = ftv t1 `Set.union` ftv t2

instance Substitutable Scheme where
  apply (Subst s) (Forall as t) = Forall as $ apply s' t
                                  where s' = Subst $ foldr Map.delete s as
  ftv (Forall as t) = ftv t `Set.difference` Set.fromList as

-- | Concrete types
typeInt, typeBool :: Type
typeInt  = TCon "Int"
typeBool = TCon "Bool"

data TypeError
  = UnificationFail Type Type
  | InfiniteType TVar Type
  | UnboundVariable String
  | DefaultFail
  | UnificationMismatch [Type] [Type]
  | GenerationFail
```

```
-- | Constrained type scheme
data CScheme = CForall [TVar] [Constr] Type
  deriving (Eq, Ord)

instance Substitutable CScheme where
  apply (Subst s) (CForall as cns t) =
    CForall as (apply s' cns) (apply s' t)
    where s' = Subst $ foldr Map.delete s as

  ftv (CForall as cns t) =
      (ftv cns `Set.union` ftv t) `Set.difference` Set.fromList as

-- | Constraints
data Constr
  = CEq Type Type
  | CInstN Name Type
  | CInst CScheme Type
  | CDef Name CScheme [Constr]
  | CLet Name CScheme [Constr]
  deriving (Eq, Ord)

instance Substitutable Constr where
  apply s (CEq t1 t2) = CEq (apply s t1) (apply s t2)
  apply s (CInstN n t1) = CInstN n (apply s t1)
  apply s (CInst cs t1) = CInst (apply s cs) (apply s t1)
  apply s (CDef n cs cns) = CDef n (apply s cs) (apply s cns)
  apply s (CLet n cs cns) = CLet n (apply s cs) (apply s cns)

  ftv (CEq t1 t2) = ftv t1 `Set.union` ftv t2
  ftv (CInstN n t1) = ftv t1
  ftv (CInst cs t1) = ftv cs `Set.union` ftv t1
  ftv (CDef n cs cns) = ftv cs `Set.union` ftv cns
  ftv (CLet n cs cns) = ftv cs `Set.union` ftv cns

instance Substitutable a => Substitutable [a] where
  apply = map . apply
  ftv = foldr (Set.union . ftv) Set.empty


instance Show TVar where
  show (TV s) = s

instance Show Type where
  show (TVar x) = show x
  show (TCon s) = s
  show (TArr t1 t2) = (show t1) ++ " --> " ++ (show t2)

instance Show CScheme where
  show (CForall vars constrs t) =
    "All" ++ (show vars) ++ (show constrs) ++ "." ++ (show t)

instance Show Constr where
```

```
  show (CEq t1 t2) = (show t1) ++ " = " ++ (show t2)
  show (CInstN x t) = (x) ++ " <= " ++ (show t)
  show (CInst cs t) = (show cs) ++ " <= " ++ (show t)
  show (CDef x cs constrs) =
    "def " ++ (x) ++ " : " ++ (show cs) ++ " in " ++ (show constrs)
  show (CLet x cs constrs) =
    "let " ++ (x) ++ " : " ++ (show cs) ++ " in " ++ (show constrs)

-- ------------------------------------------------------------------------
-- Generate constrains
-- ------------------------------------------------------------------------

data InferState = InferState { count :: Int }

initInfer :: InferState
initInfer = InferState { count = 0}

-- | Environment
data Env = TypeEnv {types :: Map.Map Name CScheme}

-- | Fresh names
letters :: [String]
letters = [1..] >>= flip replicateM ['a'..'z']

type Infer a = StateT InferState (ExceptT TypeError Identity) a

-- | Generate fresh names
fresh :: Infer Type
fresh = do
  s <- get
  put s{count = count s + 1}
  return $ TVar $ TV ((letters !! count s) ++ "'")

-- | Generate constrained type scheme
genCScheme :: Type -> [Constr] -> CScheme
genCScheme t cons = CForall (Set.toList $ftv t) cons t

-- | Predefined types for binary operations
ops :: Binop -> Type
ops Add = typeInt `TArr` (typeInt `TArr` typeInt)
ops Mul = typeInt `TArr` (typeInt `TArr` typeInt)
ops Sub = typeInt `TArr` (typeInt `TArr` typeInt)
ops Eql = typeInt `TArr` (typeInt `TArr` typeBool)

-- Infer - generate constraints
infer :: Expr -> Infer (Type,[Constr])
infer expr = case expr of

  Lit (LInt _) -> return $ (typeInt,[])
  Lit (LBool _) -> return $ (typeBool,[])

  Var x -> do
```

```
    tv <- fresh
    return (tv, [CInstN x tv])

  Lam x e -> do
    (t, c) <- infer e
    tv <- fresh
    return (tv `TArr` t,[CDef x (CForall [] [] tv) c])

  App e1 e2 -> do
    tv <- fresh
    (t1,c1) <- infer e1
    (t2,c2) <- infer e2
    return (tv, c1 ++ c2 ++ [CEq t1 (t2 `TArr` tv)])

  Let x e1 e2 -> do
    (t1,c1) <- infer e1
    let sc = genCScheme t1 c1
    (t2,c2) <- infer e2
    return (t2, [CLet x sc c2])

  Fix e1 -> do
     (t1,c1) <- infer e1
     tv <- fresh
     return (tv, c1 ++ [CEq (tv `TArr` tv) t1])

  Op op e1 e2 -> do
      (t1,c1) <- infer e1
      (t2,c2) <- infer e2
      tv <- fresh
      return (tv, c1 ++ c2 ++ [CEq (ops op) (t1 `TArr` (t2 `TArr` tv))])

  If cond tr fl -> do
        (t1,c1) <- infer cond
        (t2,c2) <- infer tr
        (t3,c3) <- infer fl
        return (t2, c1 ++ c2 ++ c3 ++ [CEq t1 typeBool, CEq t2 t3])

runInfer :: Infer (Type, [Constr]) -> Either TypeError (Type, [Constr])
runInfer m = runExcept $ evalStateT m initInfer


-- ------------------------------------------------------------------------------
-- Transform constraints
-- ------------------------------------------------------------------------------

type Excpt a = ExceptT TypeError Identity a

-- \ Map exception monad over list
mapExcpt :: (a -> Excpt b) -> [a] -> Excpt [b]
mapExcpt f (hd:tl) = do
  rhd <- f hd
  rtl <- mapExcpt f tl
```

```
  return (rhd:rtl)
mapExcpt f _ = return []

-- | Name susptitution
supstName :: Name -> CScheme -> Constr -> Constr
supstName n cs constr = case constr of
                          CEq t1 t2 -> CEq t1 t2
                          CInstN n1 t | n == n1 -> CInst cs t
                                      | otherwise -> CInstN n1 t
                          CDef n1 cs1 cons | n == n1 -> constr
                                           | otherwise ->
                            CDef n1 cs1 (map (supstName n cs) cons)
                          CLet n1 cs1 cons | n == n1 -> constr
                                           | otherwise ->
                            CLet n1 cs1 (map (supstName n cs) cons)
                          _ -> constr

-- | Supstition
rewriteSupst :: Constr -> Excpt [Constr]
rewriteSupst (CLet n cs cons) = case cs of
  CForall as' [] t' ->
    fmap (map (supstName n cs)) (fmap concat $ mapExcpt rewriteSupst cons)
  CForall as' cons' t' -> do
    su <- solveFlatten cons'
    fmap (map $supstName n (CForall as' [] $apply su t'))
         (fmap concat $ mapExcpt rewriteSupst cons)
rewriteSupst (CDef n cs cons) = case cs of
    CForall as' [] t' ->
      fmap (map (supstName n cs)) (fmap concat $ mapExcpt rewriteSupst cons)
    CForall as' cons' t' -> throwError GenerationFail
rewriteSupst x = return [x]

-- | Solve type scheme constraints
solveFlatten :: [Constr] -> Excpt Subst
solveFlatten cons = do
  cons' <- fmap concat $ mapExcpt rewriteSupst cons
  su <- let fcons = flattenConstrInst cons' in
        evalStateT solver (emptySubst,fcons)
  return su

-- | Rewrite instance
rewriteInst :: Constr -> [Constr]
rewriteInst (CInst (CForall as cons t) t1) =
  (concat $map rewriteInst cons) ++ [CEq t t1]
rewriteInst constr = [constr]

-- | Map supstition over constraint's list
flattenConstrSupst :: [Constr] -> Excpt [Constr]
flattenConstrSupst cons = fmap concat $mapExcpt rewriteSupst cons

-- | Map rewrite instance over constraint's list
flattenConstrInst :: [Constr] -> [Constr]
```

```
flattenConstrInst cons = concat cons' where
                         cons' = map rewriteInst cons

-- | Compose supstition and rewrite instance
flattenConstr :: [Constr] -> Excpt [Constr]
flattenConstr cons = do
  fcons <- flattenConstrSupst cons
  return $flattenConstrInst fcons


-- ------------------------------------------------------------------------
-- Solve constraints
-- ------------------------------------------------------------------------

type Unifier = (Subst, [Constr])
type Solve a = StateT Unifier (ExceptT TypeError Identity) a

-- | The empty substitution
emptySubst :: Subst
emptySubst = mempty

-- | Compose substitution
compose :: Subst -> Subst -> Subst
(Subst s1) `compose` (Subst s2) =
    Subst $ Map.map (apply (Subst s1)) s2 `Map.union` s1

-- | Unify single constraint
unifies :: Constr -> Solve Subst
unifies (CEq (TVar v) t) = v `bind` t
unifies (CEq t (TVar v)) = v `bind` t
unifies (CEq (TArr t1 t2) (TArr t3 t4)) = unifyMany [CEq t1 t3, CEq t2 t4]
unifies (CEq t1 t2) | t1 == t2 = return emptySubst

-- | Unify multiple constraints
unifyMany :: [Constr] -> Solve Subst
unifyMany [] = return emptySubst
unifyMany (cs:cs1) =
  do su1 <- unifies cs
     su2 <- unifyMany (apply su1 cs1)
     return (su2 `compose` su1)

-- | Bind variable in supsitution
bind :: TVar -> Type -> Solve Subst
bind a t | t == TVar a     = return emptySubst
         | occursCheck a t = throwError $ InfiniteType a t
         | otherwise       = return (Subst $ Map.singleton a t)


occursCheck :: Substitutable a => TVar -> a -> Bool
occursCheck a t = a `Set.member` ftv t

-- | Unification
```

```
solver :: Solve Subst
solver = do
  (su, cs) <- get
  case cs of
    [] -> return su
    (cs:cs1) -> do
      su1 <- unifies cs
      put (su1 `compose` su, apply su1 cs1)
      solver

-- | Run solver on source constraints
runSolver' :: [Constr] -> Excpt Subst
runSolver' cons =  do
  fcons <- flattenConstr cons
  su <- evalStateT solver (emptySubst, fcons)
  return su

-- | Run solver on flatten constraints
runSolver :: [Constr] -> Either TypeError Subst
runSolver cs = runIdentity $ runExceptT $ evalStateT solver st
  where st = (emptySubst, cs)

-- ------------------------------------------------------------------------
-- Main infer function - generate and solve constraints
-- ------------------------------------------------------------------------
mainInfer :: Expr -> Excpt Type
mainInfer expr = do
  (t,cons) <- evalStateT (infer expr) initInfer
  fcons <- flattenConstr cons
  su <- evalStateT solver (emptySubst, fcons)
  return $ apply su t
```