T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Martin Pettai

Funktsionaalne programmeerimiskeel ja selle semantika

Magistritöö (20 ap)

Juhendaja: teadur H. Nestra

Autor: ..... "....." mai 2008
Juhendaja: ..... "....." mai 2008

Lubada kaitsmisele
Professor: ..... "....." mai 2008

TARTU 2008

Sisukord

1 Sissejuhatus

Selle töö eesmärgiks on luua uus funktsionaalne programmeerimiskeel Fumontrix, realiseerida sellele interpretaator ja uurida selle keele olulisemate konstruktsioonide semantikat.

See keel peaks olema laisk ja puhtalt funktsionaalne, staatilise ja võimalikult täpse tüübisüsteemiga, võimalikult polümorfne, võimaldama mingil määral ka imperatiivset stiili kasutada ning olema loetav ülevalt alla (seega programmis hiljem defineeritavaid identifikaatoreid ei tohiks enamasti saada kasutada). Samuti ei tohiks peataseme skoop olla väga erinev tavalisest skoobist (seega suvalises alamskoobis peaks saama defineerida uusi andmetüüpe jne). Samas peaks tüübikontroll jääma garanteeritult termineeruvaks. Töös uurime mõningaid erinevaid võimalusi nende eesmärkide saavutamiseks ning Fumontrixis tehtud valikuid. Fumontrixi üldiseid põhimõtteid vaatleme peatükis 2.

Fumontrixi eeskujuks on Haskell [1], mis on samuti laisk puhas funktsionaalne keel ning milles on realiseeritud Fumontrixi interpretaator. Interpretaatoris antakse enamikule süntaktilistele konstruktsioonidele semantika funktsionaalsel kujul, kasutades ainult puhtalt funktsionaalseid andmestruktuure (sh monaade, kuid mitte sisendit-väljundit), seega on sealt võimalik välja lugeda teatud liiki denotatsiooniline semantika. Sellest semantikast anname ülevaate peatükis 7. Ka Fumontrixi süntaks on sarnane Haskelli omaga ja seda vaatleme lähemalt peatükis 3.

Enamik olulisemaid Haskelli konstruktsioone või nende alternatiivid on ka Fumontrixis realiseeritud. Haskellil on mõningaid puudusi, mida on Fumontrixis üritatud kõrvaldada.

Haskellis on tüübitaseme programmeerimise võimalused piiratud. Ka GHC laiendustes ([2], peatükk 8) on selle kasutamine millegi muu kui tüübiklasside jaoks kohmakas. Fumontrixis saab tüübitasemel defineerida suvalisi lihtrekursiivseid funktsioone ning lisaks tüüpidele saab arvutada ka väärtustega, kuid ainult parameetriliselt polümorfselt. Ka tüübiklassid on realiseeritud tüübitaseme funktsioonide abil, mis seavad tüübile vastavusse klassi eksemplari väärtuse selle tüübi jaoks. Tüübitaseme funktsioone vaatleme lähemalt peatükis 5.

Samuti on Haskellis kõrvalefektide kasutamine ebamugav. Haskell on puhas funktsionaalne keel ja kõrvalefektide jaoks kasutatakse monaade. Kõrvalefekte sisaldavaid väärtusi ei saa puhast väärtust ootavale funktsioonile argumendiks anda sama lihtsalt kui mittepuhastes keeltes. Fumontrixis on funktsiooni rakendamise operaator üle laaditud ja seda saab kasutada erinevate monaadiliste väärtuste kombinatsioonide korral, kus Haskellis tuleks kasutada erinevaid funktsioone nagu `liftM`, `ap` jne. Seda võime vaadelda ühe

polümorfismi liigina. Polümorfismi vaatleme lähemalt peatükis 6, kus uurime erinevaid polümorfismi liike ja nende kasutamise võimalusi.

Keeles on olemas ka universaalselt ja eksistentsiaalselt kvantifitseeritud tüübid, kus kvantorid võivad esineda suvalise tüübikomponendi ümber, mitte ainult tipmisel tasemel. Rakendamise hetkel tipmise taseme üldisuskvantoritega tüüpide väärtusi on võimalik funktsiooni rakendamisel polümorfselt kasutada. Tüübisüsteemi olulisemaid iseärasusi vaatleme peatükis 4.

Tööle on CD-plaadil (lisa 1) lisatud realiseeritud Fumontrixi interpretaatori lähtekood koos mõnede näidetega. Selle kasutusjuhend on failis `README`.

2 Fumontrixi üldised põhimõtted

2.1 Ülalt alla loetavus

Haskellis (ning ka Javas) on ühes deklaratsioonide nimekirjas (peatasemel, `let`-avaldises, `where`-deklaratsioonis) olevad deklaratsioonid alati üksteise skoobis. Seega võivad ühes deklaratsioonis esineda viited nii ülal- kui allpool defineeritud muutujatele. See muudab koodi lugemise raskeks, kui ühes skoobis on palju deklaratsioone, nagu peataseme skoobis tavaliselt on. Koodi lugemisel on vaja pidevalt hüpata edasi-tagasi. Samuti võib programmeerija kogemata tekitada rekursiooni, mida ta ei kavatsenud tekitada.

Haskellis ei saa ka defineerida uut muutujat eelmise samanimelise muutuja kaudu. Sageli on vaja defineerida uus väärtus, mille tähendus on väga sarnane mingi varem defineeritud ja nimetatud väärtuse tähendusega, ning on kindel, et vanale väärtusele enam viidata vaja ei ole. Imperatiivsel programmeerimisel saab sel juhul lihtsalt omistada muutujale uue väärtuse. Funktsionaalsel programmeerimisel tuleb defineerida uus muutuja. Vaatleme järgmist pseudo-Haskelli avaldist:

```
let
   r = x `mod` m
   r = if r*2 < m then
          r
       else
          r - m
in
   f r
```

Haskellis nii ei lubata kirjutada, ühes `let`-avaldises ei lubata defineerida mitut samanimelist muutujat. Seega peame kasutama mitut üksteise sees olevat `let`-avaldist:

```
let
   r = x `mod` m
in let
   r = if r*2 < m then
          r
       else
          r - m
in
   f r
```

Nii lubatakse küll Haskellis kirjutada, aga see ei ole vajaliku semantikaga, kuna Haskellis on `let`-deklaratsioonid alati (potentsiaalselt) rekursiivsed ja teises `let`-is võrdusmärgist paremal olevad muutuja `r` kasutused viitavad rekursiivselt sellele samale teise `let`-i muutujale `r`, mitte esimese `let`-i omale.

Seega ei jää muud võimalust kui nimetada üks muutujatest ümber, näiteks esimese `r` asemel `r0` või teise `r` asemel `r'`. Programmeerija jaoks on muutujanimede väljamõtlemine ja meeldejätmine suur vaev ning selliste sarnaste tähenduste ja nimedega muutujate nimed võivad kergesti segamini minna. Seega programmeerija võib kogemata kasutada vale muutujat, kuigi ta teadis, et vaja on kasutada ainult viimasena defineeritud muutujat. Kui keel lubaks defineerida mitu sama nimega muutujat, siis seda probleemi ei esineks, kuna kättesaadav on ainult viimane definitsioon.

Seetõttu Fumontrixis on `let`-deklaratsioonid vaikimisi mitterekursiivsed ning ühes `let`-avaldises saab defineerida mitu samanimelist muutujat. Kättesaadav on kasutamise kohale eelnevatest samanimeliste muutujate definitsioonidest viimane. Fumontrixis võib eelneva näite kirjutada nii:

```
let
   r = x % m;
   r = if (r*2 < m)
          r
          (r - m);
in
   f r
```

Rekursiivse väärtuse defineerimiseks saab kasutada `rec`-deklaratsiooni:

```
let
   rec ones : List Int = 1 :: ones;
in
   f ones
```

Seega Fumontrixis saab koodi lugeda ülevalt alla, kuna viidata saab vaid eelnevatele definitsioonidele. Vaid `rec`-deklaratsiooni korral on võimalik viidata pooleliolevale definitsioonile, kuid sel juhul tuleb lisaks muutuja nimele definitsiooni alguses deklareerida ka muutuja tüüp. Seega kasutamise hetkeks on muutuja tüüp juba lugejale teada.

2.2 Tüübiinferents

Haskellis kasutatakse Hindley-Milneri tüübituletusalgoritmi ([1], jaotis 4.5). Et eristada sellist unifitseerimist ja võrrandite lahendamist kasutavat tüübituletust tavalisest (lihtsamast) tüübituletusest, nimetame selles töös esimest

tüübiinferentsiks. Tüübiinferentsiga sarnast meetodit liikide määramiseks nimetame liigiinferentsiks.

Hindley-Milneri tüübiinferentsi korral määrab translaator paljudele muutujatele ise tüübi, seega programmeerija ei pea alati tüübiannotatsioone lisama. Samas muudab see koodi lugemise raskemaks, kuna tüübiinferentsi algoritmi peast rakendamine võib keerulisematel juhtudel liiga raskeks osutuda. Samuti sõltub muutuja tüüp tema kasutustest, mis võivad esineda koodis selle definitsioonist palju hiljem. See läheb vastuollu ülalt alla loetavuse põhimõttega.

Haskell lubab lisada muutujadefinitsioonidele tüübisignatuure, kuid nende kasutamine ei ole kohustuslik. Funktsiooni tüübisignatuur peab sisaldama ka tagastusväärtuse tüüpi, mis mitterekursiivsete funktsioonide korral ei oleks tegelikult vajalik, kuna tagastusväärtuse tüübi saab lihtsalt argumentide tüüpide järgi leida.

Fumontrixis (nagu ka GHC-s) on võimalik kasutada kõrgemat järku universaalselt kvantifitseeritud tüüpe. Üldjuhul on aga kõrgemat järku tüüpide korral tüübiinferents mittelahenduv ([2], jaotis 8.7.4.2). Ka GHC-s on vaja teist ja kõrgemat järku tüüpide kasutamisel annotatsioonid lisada.

Seetõttu Fumontrixis Haskelli stiilis tüübiinferentsi ei kasutata. Selle asemel tuleb λ-seotud muutujate sissetoomisel tüübid juurde kirjutada. Näiteks:

```
succ = \ x : Int . x + 1
```

`let`- ja `case`-seotud muutujate korral tüüpi juurde pole vaja lisada (v.a `rec`-deklaratsioonide korral), kuna seal on muutuja väärtustamiseks kasutatav avaldis kohe juures ja selle tüüp on teada.

Samuti ei ole Fumontrixis liigiinferentsi. Seega tuleb kvantoriga tüübimuutuja sissetoomisel lisada liigiannotatsioon. Liigiannotatsiooni võib ära jätta, kui liigiks on `*`.

2.3 Skoopide võrdväärsus

Haskellis on peataseme skoop väga erinev tavalisest alamskoobist. Tüübiklasse, esindajaid, algebralisi andmetüüpe, tüübisünonüüme on võimalik defineerida ainult peatasemel. Seega need definitsioonid on kõik globaalselt kättesaadavad ja kogemata on võimalik mingis kohas kasutada definitsiooni, mida tegelikult ei tohiks seal kasutada. Mingil määral on moodulite abil võimalik definitsioone peita, kuid väga väikesteks juppideks ei ole mõtet koodi tükeldada, seega see on kasutatav põhiliselt suuremate alamskoopide jaoks. Samuti ei ole ühes Haskelli moodulis võimalik defineerida tüübiklassi, andmetüüpi jne, millega samanimeline olem (nimetame olemiks kõiki asju, millele on võimalik

koodis mingi identifikaatoriga viidata) on juba mõnes kasutatavas moodulis defineeritud (ja eksporditud).

Fumontrixis on tüübiklasse, tüübitaseme funktsioone, algebralisi andmetüüpe jne võimalik defineerida kõigis skoopides — nii peatasemel kui `let`-avaldistes. Samuti on võimalik defineerida uut olemit, mille nimi langeb kokku mõne olemasolevaga. Sellisel juhul alamskoobis enam vanale olemile selle nimega viidata ei saa, kuid see võib jääda kaudselt kättesaadavaks.

2.4 Kõrvalefektid, erindid ja laiendatud tüübid

Paljudes keeltes võivad avaldiste väärtused sisaldada lisaks tavalisele väärtusele kõrvalefekte. Näiteks Java funktsioon, mille tagastusväärtuse tüübiks on `boolean`, võib enne tõeväärtuse tagastamist teha mingeid kõrvalefekte (muuta muutujate väärtusi, teha sisendit-väljundit jne) või tagastada tõeväärtuse asemel midagi muud (nt erindi, mõnede teiste tüüpide korral ka nullviida). Seega Java tüüp `boolean` on väga erinev matemaatilisest (kaheelemendilisest) tõeväärtustüübist. Samas võib ka Java funktsioon tagastada tõeväärtuse ilma erindita ja kõrvalefekte tegemata, seega Java tüüp `boolean` sisaldab alamtüübina matemaatilist tõeväärtustüüpi. Võime öelda, et Javas on kasutusel laiendatud tõeväärtustüüp.

Kui tüübisüsteem toetab ainult laiendatud `boolean`-i, siis me ei saa lasta tüübisüsteemil staatiliselt kontrollida, et funktsiooni tagastusväärtus kuulub puhta `boolean`-i tüüpi. Et garanteerida, et tagastusväärtus ei sisalda erindeid, on vaja käsitsi kontroll koodi sisse kirjutada. Kuid sellisel juhul on võimalik tüübiveast teada saada alles pärast koodi käivitamist, seega tegemist on dünaamilise tüübikontrolliga.

Kui me soovime, et keel oleks staatilise tüübikontrolliga (nagu Haskell), siis peaks ka puhast `boolean`-i ja laiendatud `boolean`-i saama staatiliselt eristada, vastasel korral on tüübikontroll ainult pooleldi staatiline ja pooleldi dünaamiline. Kui on olemas puhas `boolean`, siis sellest saab vajaduse korral laiendatud `boolean`-i defineerida (näiteks monaadide abil), vastupidine ei ole aga võimalik nii, et staatiline tüübikontroll säiliks.

Vaatleme, milleks on kasulik, kui on olemas puhas `boolean` (mitte ainult laiendatud `boolean`). Näiteks olgu tegemist krüptoloogilise teegiga. Seal on mingi funktsioon, mis kasutab salajasi andmeid ja tagastab kasutajale ainult ühe biti informatsiooni. Kasutaja võib olla pahatahtlik ja üritab salajaste andmete kohta infot saada. Kui funktsiooni tagastustüübiks on puhas `boolean`, siis ta ei saa ühe päringuga kätte rohkem kui ühe biti salajast informatsiooni. Kui funktsiooni tagastustüübiks on laiendatud `boolean`, siis ta võib turvaauke ära kasutades panna funktsiooni tagastama erindit, mis sisaldab palju rohkem kui ühe biti salajast informatsiooni.

Vaatleme Java näidet:

```
class BlackBox {
   private BigInteger secret;

   BlackBox(BigInteger secret) {
      this.secret = secret;
   }

   public boolean f(BigInteger x) {
      if (x.equals(new BigInteger("1000666"))) {
         throw new RuntimeException(secret.toString());
      }
      return x.compareTo(secret) < 0;
   }
}

class Adversary {
   public static BigInteger attack(BlackBox box) {
      try {
         boolean dummy = box.f(new BigInteger("1000666"));
      } catch (RuntimeException e) {
         return new BigInteger(e.getMessage());
      }
      return null;
   }
}

class Test {
   public static void main(String[] args) {
      BlackBox box = new BlackBox(
         new BigInteger("123456789123456789123456789"));
      System.out.println("Adversary.attack(box) = "
         + Adversary.attack(box));
   }
}
```

Siin defineeritakse must kast `box`, mis sisaldab salajast informatsiooni (seda hoitakse piiramatu suurusega täisarvulises muutujas `secret`), kuid selle salajase väärtuse kohta on võimalik infot saada ainult funktsiooni `f` kaudu. See funktsioon saab sisendiks piiramatu suurusega täisarvu `x` ning tulemuse

tüübi `boolean` järgi peaks tagastama väärtuse, mis sisaldab maksimaalselt ühe biti infot kastis oleva salajase väärtuse kohta.

Antud näites on aga funktsiooni `f` koodi sokutatud tagauks, mis teatud argumendi `x` väärtuse korral viskab erindi, mis sisaldab kogu salajase info. Seda tagaust kasutab meetod `Adversary.attack`, et mustast kastist ainult ühe päringuga kogu salajane info kätte saada. See on võimalik seetõttu, et Javas on võimalik nn *run-time* erindeid visata suvalisest meetodist ilma seda meetodi tüübis (päises) deklareerimata. Kui funktsiooni tüübiga oleks võimalik erindite viskamine täiesti ära keelata, siis oleks võimalik välistada sellise tagaukse koodi sokutamine.

Seetõttu Fumontrixis ei kasutata implitsiitselt laiendatud (v.a $\bot$-ga) tüüpe ning võimalik kõrvalefektide või erindite sisaldumine on tüübist näha.

2.5 Imperatiivne stiil

Mõningaid algoritme on lihtsam kirja panna imperatiivses stiilis kui funktsionaalses. Seega võiks funktsionaalne keel sisaldada ka võimalusi imperatiivse stiili kasutamiseks, näiteks muudetavaid viitasid ja massiive. Haskellis on selleks olemas IO- ja ST-monaad (IO-monaadi kohta vt [3], peatükk 2). ST-monaadi korral kontrollitakse tüübisüsteemi abil, et viitasid ja massiive kasutatakse ainult selles lõimes, kus need on defineeritud ning alles pärast nende defineerimist. ST-monaadist on võimalik arvutuse tulemust ohutult kätte saada, kuna arvutus toimub eraldi lõimes, mis on võimalik lõpuni jooksutada ning kuhugi mujale viiteid selle lõime seisundile ei jää. IO-monaadist väärtusi ohutult kätte ei saa. Seega puhtas funktsionaalses keeles oleks imperatiivsete arvutuste jaoks parem kasutada ST-monaadi.

Ka Fumontrixis on ST-monaad realiseeritud ning selles saab kasutada (lugeda ja kirjutada) viitasid ja massiive. Selleks kasutatakse sisemiselt Haskelli IO-monaadi ja arvutuse väärtus saadakse kätte `unsafePerformIO` abil, kuid Fumontrixi tüübisüsteemi abil kontrollitakse, et kasutada saaks ainult ohutuid arvutusi, mis on jooksutatavad eraldi lõimes.

Jaotises 6.5 vaatleme monaadiliste väärtuste polümorfismi, mis muudab Fumontrixis imperatiivse stiili kasutamise mugavamaks.

2.6 Anonüümsed funktsioonid ka tüübitasemel

Fumontrixis on olemas tüübitaseme funktsioonid. Erinevalt GHC-st on Fumontrixis tüübitaseme funktsioonid *first-class*. Neid saab luua ka ilma nime andmata (anonüümsete funktsioonidena), isegi kui tegemist on (liht)rekursiivse funktsiooniga, ning neid saab anda argumendiks teistele tüübitaseme

funktsioonidele. See säästab teatud juhtudel programmeerijat nimede väljamõtlemise ja meeldejätmise vaevast ning koodi hakkimisest. Fumontrixi tüübitaseme funktsioone vaatleme lähemalt jaotises 5.1.

3 Süntaks

3.1 Võrdlus Haskelliga

Fumontrixi süntaks on mingil määral sarnane Haskelli omaga, kuid seal on mõned olulised erinevused.

Erinevalt Haskellist ei kasutata reavahetusi ja treppimist süntaksi osana. Reavahetuse asemel kasutatakse (deklaratsioonide, valikualternatiivide jne) nimekirjas eraldajana semikoolonit. Nimekirja lõpetamiseks kasutatakse negatiivse treppimise asemel võtmesõna `end` või, kui nimekiri ei asu konstruktsiooni lõpus, siis mõnda muud võtmesõna (`let`-konstruktsiooni korral näiteks `in`). Nimekirja viimase elemendi järel on semikoolon lubatud, kuid mitte kohustuslik.

Fumontrixis kasutatakse λ-avaldises noole asemel punkti nagu lambdaarvutuses. See vähendab vajadust sulgude järele, kuna näidise annotatsiooniks olevas tüübiavaldises võidakse kasutada noolt funktsioonitüübi tähistamiseks ja see võiks segi minna λ-avaldise noolega. Punkti kasutatakse ka kvantorite korral nagu GHC-ski.

Erinevalt Haskellist on Fumontrixis kvantoritega seotud muutujate nimed suure algustähega, kuna kvantoritega sissetoodud muutujad on Fumontrixis sarnased algebraliste andmetüüpide konstruktoritega (neid saab kasutada samades kontekstides, näiteks tüübiavaldistes või tüübinäidistes). Kvantoriga sissetoodud konstruktori abil konstrueeritavad tüübid on erinevad kõigist varem skoobis olnud tüüpidest.

Tüübisünonüümide nimed on Fumontrixis väikese algustähega nagu andmetaseme väärtuste sünonüümid (andmetaseme muutujad). Nende sissetoomine ei tekita uusi tüüpe või väärtusi, vaid annab lihtsalt olemasolevale tüübile või väärtusele (uue) nime.

Tüübiklasside nimed on Fumontrixis väikese algustähega, kuna Fumontrixis tähistab tüübiklassi nimi selle tüübifunktsiooni nime, mis seab igale klassi tüübile (esindajale) vastavusse klassi eksemplari (klassi väärtuse (mis tavaliselt on mingi funktsioonide kogum) realisatsiooni antud tüübi jaoks).

3.2 Süntaksi ülevaade

3.2.1 Süntaktilised kategooriad

Järgnevates alajaotistes on erinevate süntaktiliste kategooriate elemente tähistavad muutujad järgnevate nimedega (millele võivad lisanduda indeksid):

muutuja (sünonüüm)	x
konstruktor	C
täisarvkonstant	n
lihtrekursiooni identifikaator	f
avaldis	e
tüübiavaldis	t
liigiavaldis	k
näidis	p
tüübinäidis	tp
tüübiklassi nimi	c
deklaratsioon	d
andmekonstruktori kirjeldus	dc
valikualternatiiv	ca
tüübitaseme valikualternatiiv	tca

Selles peatükis vaatleme ainult süntaksit, semantikat vaatleme peatükis 7.

3.2.2 Kommentaarid

Fumontrixis on kahte liiki kommentaare. Sümbol `#` alustab kommentaari, mis jätkub jooksva rea lõpuni. Suvalise pikkusega kommentaarid paiknevad `/*` ja `*/` vahel.

3.2.3 Muutujad ja konstruktorid

Muutujateks on identifikaatorid, mis algavad väiketähe või alakriipsuga, ning konstruktoriteks on identifikaatorid, mis algavad suurtähega.

Lisaks saab süntaktilise suhkruna kasutada infiksoperaatoreid. Nii nagu GHC-s, loetakse kooloniga algavad infiksoperaatorid konstruktoriteks ja ülejäänud muutujateks. Kasutada saab ka infiksoperaatorite vasakut ja paremat lõiget, näiteks `(1 -)` või `(/ 2)`.

Infiksoperaatorite prioriteedid ja assotsiatiivsus on Fumontrixis sisse ehitatud ja seda programmeerija muuta ei saa. Kõik infiksoperaatorid peale aritmeetikatehete `+ - * /` ja `%` (jäägi leidmine) on paremassotsiatiivsed ning ühe ja sama madala prioriteediga. Tehted `+` ja `-` on vasakassotsiatiivsed ja keskmise prioriteediga ning tehted `* / %` on vasakassotsiatiivsed ja kõrge prioriteediga.

3.2.4 Avaldised

Fumontrixis on olemas järgmised avaldisekonstruktsioonid:

muutujaavaldis	x
konstruktoravaldis	C
täisarvkonstant	n
funktsioonirakendamine	$e_1\ e_2$
polümorfse väärtuse spetsialiseerimine	e `$:` t
avaldise tüübiannotatsioon	e `:` t
λ-abstraktsioon	`\` p `:` t `.` e
polümorfne avaldis (parameetriline)	`forall` C `:` k `.` e
polümorfne avaldis (tüübiklassiga)	`forall` $c\ C$ `:` k `.` e
eksistentsiaalseks pakkija (parameetriline)	`exists` C `:` k `.` t
eksistentsiaalseks pakkija (tüübiklassiga)	`exists` $c\ C$ `:` k `.` t
valikuavaldis	`case` e `of` ca_1`;` `...`; ca_n `end`
`let`-avaldis	`let` d_1`;` `...`; d_n `in` e
tüübitaseme avaldis	`type` t

Valikualternatiivid on kujul p `->` e. Süntaktilise suhkruna on lisaks olemas `do`-avaldised, mida vaatleme lähemalt jaotises 6.5.2.

Liigiannotatsiooni võib liigi `*` korral ära jätta (koos eelneva kooloniga).

3.2.5 Näidised

Fumontrixis on olemas järgmised näidisekonstruktsioonid:

muutujanäidis	x
konstruktornäidis	$C\ p_1\ \ldots\ p_n$
täisarvnäidis	n
eksistentsiaalne näidis	`exists` C `:` k`.` p
topeltnäidis	p_1 `@` p_2
ignoreerimisnäidis	`_`

Liigiannotatsiooni võib liigi `*` korral ära jätta (koos eelneva kooloniga).

3.2.6 Tüübiavaldised

Fumontrixis on olemas järgmised tüübiavaldisekonstruktsioonid:

muutujaavaldis	x
konstruktoravaldis	C
täisarvutüüp	`Int`
funktsioonirakendamine	$t_1\ t_2$
funktsioonitüüp	t_1 `->` t_2
universaalne tüüp (parameetriline)	`forall` C `:` k `.` t
universaalne tüüp (tüübiklassiga)	`forall` $c\ C$ `:` k `.` t
eksistentsiaalne tüüp (parameetriline)	`exists` C `:` k `.` t
eksistentsiaalne tüüp (tüübiklassiga)	`exists` $c\ C$ `:` k `.` t
lihtrekursiivne pöördumine	`rec:`f t
mitterekursiivne λ-abstraktsioon	`\` x `:` k `.` t
lihtrekursiivne λ-abstraktsioon	`\` x `:` k `rec:`f k`.` t
valikuavaldis	`case` t `of` tca_1`;` `...`; tca_n `end`
tüübitaseme väärtus	`value` e
avaldise tüüp	`typeof` e
avaldise tüüp ilma tipmiste monaadideta	`basetypeof` e

Tüübitaseme valikualternatiivid on kujul tp `->` t.

Liigiannotatsiooni võib liigi `*` korral ära jätta (koos eelneva kooloniga). Lihtrekursiivses λ-abstraktsioonis teise liigiannotatsiooni ees koolonit ei ole, kuid selle annotatsiooni võib ka ära jätta liigi `*` korral.

Samuti võib `rec:`f asemel kirjutada lihtsalt `rec`, kui soovitakse lihtrekursioonile nime mitte anda.

3.2.7 Tüübinäidised

Fumontrixis on olemas järgmised tüübinäidisekonstruktsioonid:

muutujanäidis	x
konstruktornäidis	$C\ tp_1\ \ldots\ tp_n$
funktsiooninäidis	`(`tp_1 `->` tp_2`)`
täisarvutüübinäidis	`Int`
ignoreerimisnäidis	`_`

3.2.8 Liigiavaldised

Fumontrixis on olemas järgmised liigiavaldisekonstruktsioonid:

tüüp	`*`
väärtus	`@`
funktsiooniliik	k_1 `->` k_2

Erinevalt Haskellist on olemas liik *väärtus*, mis võimaldab ka tüübitasemel andmetaseme väärtustega arvutada.

3.2.9 Deklaratsioonid

Fumontrixis on olemas järgmised deklaratsioonikonstruktsioonid:

sidumisdeklaratsioon	p `=` e
rekursiivne sidumisdeklaratsioon	`rec` p `:` t `=` e
algebralise andmetüübi definitsioon	`data` C C_1 `...` C_n `=` dc_1 `|` `...` `|` dc_n
tüübitaseme sidumisdeklaratsioon	`type` x `=` t
klassideklaratsioon	`class` c t
monaadi sissetoomise deklaratsioon	`newmonad` C
monaadi eemaldamise deklaratsioon	`unmonad` C
monaadide kaotamise deklaratsioon	`nomonads`

Andmekonstruktorite kirjeldused on kujul $C\ t_1\ \ldots\ t_n$.

4 Tüübisüsteemist

4.1 Fumontrixi tüübisüsteemist üldiselt

Fumontrixi tüübisüsteemis on võimalik tüüpe koostada baastüübi `Int`, tüübikonstruktorite, üldisuskvantorite ja olemasolukvantorite abil. Sarnaselt Haskelliga klassifitseeritakse tüübitaseme objektid liikidesse, kuid erinevalt Haskellist on Fumontrixis olemas ka liik `@`, mis tähistab tüübitasemel kasutatavat andmetaseme väärtust.

Erinevalt Haskellist ei kasutata Fumontrixis tüübiinferentsi ning tüüpide määramiseks vajalik info saadakse põhiliselt λ-abstraktsioonides näidisele lisatud tüübiannotatsioonidest. Üldisuskvantori avamiseks on Fumontrixis operaator `$:`, mis avab tipmise taseme üldisuskvantori ja väärtustab sellega seotud muutuja antud tüübiga. Kuna avatakse kõige tipmine üldisuskvantor, siis erinevalt GHC-st (kus kvantori avamiseks tuleb lisada annotatsioon, kus kogu tüübiavaldis on ümber kirjutatud ning kõik vaadeldava muutuja esinemised asendatud (ühe ja sama) soovitud tüübiga) on Fumontrixis kvantorite järjekord oluline. Seega tüübid

```
forall A. forall B. A -> B
forall B. forall A. A -> B
forall A. A -> (forall B. B)
```

on Fumontrixis kõik erinevad, erinevalt GHC-st, kus analoogilised tüübid (vaid kvantifitseeritud muutujad on väikese tähega) on kõik võrdsed ja GHC teisendab need automaatselt kujule

```
forall a b. a -> b
```

Fumontrixis ei oleks selliseid automaatseid teisendusi võimalik teha, kuna see muudab tüübiavaldise semantikat (operaator `$:` käitub nendel kolmel juhul erinevalt).

4.2 Baastüübid

Fumontrixis on sisseehitatud täisarvutüüp `Int` (piiramatu suurusega täisarvud, vastab Haskelli tüübile `Integer`). Funktsioonitüübi noolt ei vaadelda Fumontrixis eraldi konstruktorina, funktsioonitüübi konstruktsioon t_1 `->` t_2 on eraldi konstruktsioon, mitte infiksoperaatori kasutus. See väldib mõningaid probleeme. Vastasel korral oleks näiteks võimalik konstruktor (`(`A `->)` monaadiks defineerida ja selle juhu arvestamine teeks monaadiliste väärtuste polümorfismi semantika keeruliseks.

Lihtsuse huvides on esialgu keelest välja jäetud ujukomaarvud ning märgija stringitüübid. Vajadusel saab täisarvutüüpi kasutada märgitüübina, kujutades märki täisarvuna, mis vastab selle Unicode'i koodile. Stringid oleks siis lihtsalt täisarvude listid.

4.3 Tüübikonstruktorid

Fumontrixis on tüübikonstruktoreid võimalik sisse tuua `data`-deklaratsiooniga (algebralised andmetüübid), `exists`-näidisega (olemasolukvantoriga seotud konstruktor) või `forall`-avaldisega (üldisuskvantoriga seotud konstruktor).

Nende konstruktorite semantika on ühesugune. Konstruktorile seatakse vastavusse unikaalne identifikaator (UID), mis on täisarv alates 1-st. Kõigi skoobis eksisteerivate konstruktorite (sh nende, mis on teise samanimelise konstruktori tõttu varjatud) UID-d on erinevad.

Lisaks sellele on keelde sisse ehitatud tüübikonstruktor `ST` (ST-monaadi jaoks), mille UID on negatiivne (-2). Kui tulevikus lisanduvad keelde muud sisse ehitatud tüübikonstruktorid, siis need on samuti negatiivse UID-ga. UID -1 on reserveeritud `IO`-konstruktori jaoks, mida praegu keeles ei ole.

4.4 Universaalselt kvantifitseeritud tüübid

Fumontrixis on olemas universaalselt kvantifitseeritud tüübid (vt [4], peatükk 4) kujul `forall` C `:` k `.` $t[C]$ ja `forall` c C `:` k `.` $t[C]$. Üldisuskvantorid võivad esineda suvalise tüübikomponendi ümber, mitte ainult tipmisel tasemel.

Universaalset tüüpi väärtuse loomiseks tuleb kasutada `forall`-avaldist, mis toob sisse uue muutuja, mis on skoobis selle avaldise piires. Näiteks

```
f = forall A . \ x : A .
   let
      ys = (x :: x :: Nil) : List A
   in
      head ys;
```

Üldisuskvantoriga seotavat muutujat saab skoopi tuua ka GHC-s:

```
f :: forall a. a -> a
f x =
   let
      ys :: [a]
      ys = [x,x]
```

```
in
   head ys
```

Nendes näidetes muutujad `A` ja `a` on skoobis funktsiooni `f` definitsiooni kehas ning parameeter `x` on seal monomorfset tüüpi `A` või `a`, samuti on monomorfset tüüpi `ys`, millele annotatsioon on lisatud.

GHC-s on vaja tüübiparameetri nime skoopi toomiseks vaja tingimata kasutada tüübisignatuuri (ja seega tuleb deklareerida ka tagastusväärtuse tüüp). Kui viimases näites signatuur ära jätta, siis GHC seda enam ei aktsepteeri (isegi, kui näidisele `x` annotatsioon lisada (`x :: a`)), kuna muutuja `a` ei ole siis skoobis (ning samuti ei sobi signatuuri `[a]` polümorfne interpretatsioon `forall a. [a]`). Fumontrixis piisab ainult argumentide tüüpide deklareerimisest, tagastusväärtuse tüüpi pole vaja deklareerida.

Fumontrixis üldisuskvantori ilmutatult avamiseks kasutatavat operaatorit vaatlesime juba jaotises 4.1. Polümorfse funktsioonirakendamise käigus toimub ka automaatne üldisuskvantorite avamine, kuid ainult funktsiooni ja argumendi tipmisel tasemel. Seda vaatleme lähemalt jaotises 6.4.

4.5 Eksistentsiaalsed tüübid

Fumontrixis on olemas eksistentsiaalsed tüübid (vt [4], peatükk 5) kujul `exists` C `:` k `.` $t[C]$ ja `exists` c C `:` k `.` $t[C]$.

Eksistentsiaalset tüüpi väärtuse loomiseks tuleb kasutada pakkimisfunktsiooni, mis on süntaktiliselt samal kujul nagu tulemuseks olevat eksistentsiaalset tüüpi tähistav tüübiavaldis. Pakkimisfunktsiooni tüübiks (tüübiklassiga juhul) on
`forall` c C `:` k `.` $t[C]$ `->` `(exists` c C `:` k `.` $t[C]$`)`.

Eksistentsiaalset tüüpi väärtust võib vaadelda kui paari tüübiparameetrist C ja tüüpi $t[C]$ väärtusest. Fumontrixi semantikas on tüübiparameetri asemel tüübiklassi c eksemplari väärtus tüübi C jaoks. Seega eksistentsiaalne tüüp on paar kahest väärtusest. Parameetrilise (ilma tüübiklassita) kvantifitseerimise korral on eksemplari väärtuse asemel ühiktüübi element, mis infot ei sisalda.

Eksistentsiaalset tüüpi väärtusi automaatselt ei avata, seega ainuke võimalus sealt info kättesaamiseks on kasutada `exists`-näidist
`exists` c C `:` k `.` p, mis lisab konteksti uue tüübikonstruktori C ning muudab selle klassi c esindajaks, lisades C jaoks klassi c eksemplari, mis on võrdne eksistentsiaalse väärtuse sees olnud eksemplariga. Eksistentsiaalse tüübi sees olnud väärtuse jaoks kasutatakse näidist p.

Erinevalt GHC-st on võimalik olemasolukvantoreid kasutada suvalise tüübikomponendi ümber (GHC-s ainult seoses algebraliste andmetüüpidega, vt

[2], jaotis 8.4.4).

5 Tüübitaseme programmeerimine

5.1 Tüübitaseme funktsioonid Fumontrixis

5.1.1 Tüübitaseme funktsioonid

Vaatleme nüüd, millised on tüübitaseme funktsioonide kasutamise võimalused Fumontrixis. Erinevalt Haskellist, kus keerulisemaid tüübitaseme funktsioone tuleb defineerida tüübiklasside kaudu loogilise programmeerimise stiilis implikatsioonidena ([2], jaotis 8.6.3, käesolevas töös vaatleme neid lähemalt jaotises 5.3), on Fumontrixis tüübitaseme funktsioonide defineerimiseks eraldi konstruktsioonid ja neid saab defineerida funktsionaalse programmeerimise stiilis λ-abstraktsioonina nagu andmetaseme funktsioonegi.

Defineerime näiteks tüübitaseme funktsiooni, mis seab tõeväärtustüübile vastavusse täisarvutüübi, täisarvutüübile tõeväärtustüübi ning kõigile teistele tüüpidele ühiktüübi:

```
type tf11 = \ a .
   case a of
      Bool -> Int;
      Int  -> Bool;
      _    -> Unit;
   end;
```

Näeme, et tüübitaseme funktsioone defineeritakse λ-abstraktsiooniga nagu andmetaseme funktsioonegi ning tüübitasemel saab tüüpide sisse vaadata `case`-konstruktsiooniga sarnaselt andmetasemega. `case`-konstruktsioon ei pea katma kõiki võimalusi — kui eelmisest definitsioonist viimane valikualternatiiv ära jätta, siis tuleks funktsiooni `tf11` rakendamisel mingile muule tüübile kui `Bool` või `Int` tüübiviga, nagu andmetasemel tuleb analoogilisel juhul $\bot$.

Samuti näeme, et tüübitaseme funktsioonile nime andmiseks saab kasutada `type`-deklaratsiooni. Seda saab kasutada ka muude tüübitaseme objektide (tüüpide, väärtuste jne), mitte ainult funktsioonide jaoks. Antud juhul oli tegemist kõige lihtsama tüübitaseme funktsiooniga (liiki `* -> *`), mis teisendab tavalise tüübi (liiki `*`) tavaliseks tüübiks.

Fumontrixis saab defineerida ka keerulisemaid tüübitaseme funktsioone, näiteks liiki `* -> * -> *`, `(* -> *) -> *` või `((* -> *) -> *) -> * -> *`.

Erinevalt Haskelli tüübiklassidega defineeritud tüübitaseme funktsioonidest saab Fumontrixi tüübitaseme funktsioone kasutada ka tüübiannotatsioonides. Näiteks:

```
f13 = \ x : tf11 Bool . x + 3;
```

Siin on annotatsioonis kasutatud `Int` asemel `tf11 Bool`.

5.1.2 Väärtustega arvutamine tüübitasemel

Tüübitasemel saab arvutada lisaks tüüpidele ka väärtustega, tüübitaseme väärtuse liik on `@`. Seega on kasutatavad ka tüübitaseme funktsioonid liiki `@ -> @` (sarnaneb andmetaseme funktsioonidega), `* -> @` jne.

Defineerime näiteks täisarvu ruudu arvutamise funktsiooni:

```
type tf12 = \ a : @ .
   value (type a) * (type a);
```

Siin tuleb määrata argumendi `a` liik `@`, kuna vaikimisi on liigiks `*`. Argumendi tüüpi ei ole tüübitaseme funktsiooni korral vaja määrata, see funktsioon on kasutatav kõikide tüüpide väärtuste jaoks, mille korral ei tule funktsiooni kehas tüübiviga, ehk antud juhul ainult täisarvude jaoks, kuna korrutustehe on defineeritud ainult täisarvude jaoks (ujukomaarve Fumontrixis ei ole).

Siin kasutatakse ka võtmesõnu `type` ja `value`, mis on vajalikud tüübitaseme ja andmetaseme vahel liikumiseks. Need konstruktsioonid ulatuvad süntaktiliselt nii kaugele paremale kui võimalik (nagu `let`-konstruktsiooni võtmesõna `in` järel olev osa).

Võtmesõna `type` muudab tüübitaseme väärtuse (liiki `@`) andmetaseme väärtuseks, mida saab näiteks anda argumendiks andmetaseme funktsioonile (kui see väärtus on õiget tüüpi). Viimases näites `*` on andmetaseme funktsioon, kuid `a` on tüübitaseme väärtus, seega lihtsalt `a * a` ei saa kirjutada.

Võtmesõna `value` muudab andmetaseme väärtuse tüübitaseme väärtuseks (liiki `@`). Viimases näites on see vajalik, kuna tüübitaseme funktsioon saab tagastada ainult tüübitaseme väärtust (või muud tüübitaseme objekti), aga mitte andmetaseme väärtust.

Erinevalt Haskellist on Fumontrixis olemas ka võtmesõna `typeof`, mis võimaldab kasutada avaldise tüüpi ja sellega arvutada ning näiteks annotatsioonides kasutada. Näiteks

```
type argtype = \ a : @ .
   case typeof type a of (arg -> res) -> arg end;
f14 = \ x : argtype (value (+)) . x + x;
```

Siin defineerime kõigepealt funktsiooni `argtype`, mis leiab monomorfse andmetaseme funktsiooni argumendi tüübi. Kasutades seda funktsiooni, saame funktsiooni `f14` argumendi tüübi määrata selles kasutatava funktsiooni `(+)` argumendi tüübi põhjal nagu tüübiinferentsi korral. Praegusel juhul on küll lihtsam argumendi tüüp `Int` välja kirjutada, aga keerulisema tüübi korral võib `argtype` olla mugavam kasutada.

5.1.3 Tüübitaseme lihtrekursiivsed funktsioonid

Fumontrixis on võimalik tüübitasemel defineerida ka rekursiivseid funktsioone. Erinevalt GHC-st, kus vajaliku laienduse sisselülitamisel on võimalik kirjutada funktsioone, mis lähevad lõpmatusse rekursiooni (vt jaotist 5.3.3), on Fumontrixis tüübitasemel kasutatav ainult lihtrekursioon üle tüüpide (nii lihtsate kui ka keerulisemat liiki tüüpide). See garanteerib tüübikontrolli termineeruvuse, samas lihtrekursioonist peaks piisama enamiku kasulike tüübitaseme funktsioonide realiseerimiseks.

Lihtrekursioon tähendab, et funktsiooni väärtuse arvutamiseks mingil argumendil võib selle sama funktsiooni (rekursiivselt arvutatud) väärtusi kasutada ainult selle argumendi komponentide (vahetute või mitte) jaoks. Näiteks funktsiooni arvutamiseks argumendil

```
List (Tuple3 (Pair Int Bool) Unit Int)
```

võib kasutada selle sama funktsiooni väärtusi argumentidel

```
Tuple3 (Pair Int Bool) Unit Int

Pair Int Bool

Unit

Int

Bool
```

Defineerime näiteks tüübitaseme unaarsed naturaalarvud ning liitmise ja korrutamise nendel:

```
data Zero;
data Succ A;

type tfSum = \ a . \ b rec .
   case b of
      Zero    -> a;
      Succ b' -> Succ (rec b');
   end;

type tfProd = \ a . \ b rec .
   case b of
      Zero    -> Zero;
      Succ b' -> tfSum (rec b') a;
   end;
```

Nii `tfSum` kui `tfProd` on rekursiivsed oma teise argumendi suhtes. Lihtrekursiivse tüübitaseme funktsiooni defineerimiseks kasutatakse võtmesõna `rec` λ-abstraktsiooni päises. Sama võtmesõna kasutatakse ka rekursiivse funktsiooni kehas selle funktsiooni rekursiivselt arvutatud väärtuse kasutamiseks (toodud näites `rec b'` leiab funktsiooni väärtuse argumendil `b'`).

Viimases näites oli rekursiivse tüübitaseme funktsiooni tagastusväärtus liiki `*`. Kui tagastusväärtus on mingit teist liiki, siis tuleb see liik λ-abstraktsiooni päises märkida, et hiljem rekursiivselt arvutatud väärtuse kasutamisel (`rec`-konstruktsiooniga) oleks selle liik teada (kuna Fumontrixis liigiinferentsi ei toimu). Näiteks

```
type typeNatToValue = \ a rec @ .
   case a of
      Zero   -> value 0;
      Succ b -> value 1 + (type rec b);
   end;
```

See funktsioon teisendab tüübitaseme täisarvu andmetaseme täisarvu kujule. Kuna tulemus on liiki `@`, siis on see λ-abstraktsiooni päises märgitud.

Vaatleme veel, kuidas kahekordset lihtrekursiooni kasutada. Defineerime näiteks tüübitaseme funktsiooni, mis arvutab m-korda-n-elemendilise korrutustabeli elementide summa, simuleerides rekursioonidega tsükleid üle naturaalarvude:

```
type tfProdSum = \ m . \ n .
   (\ a rec:outer * -> * . \ b rec .
      case b of
         Zero    ->
            case a of
               Zero    ->
                  Zero;
               Succ a' ->
                  rec:outer a' n;
            end;
         Succ b' ->
            rec b' `tfSum` (a `tfProd` b);
      end
   ) m n;
```

Siin on välise rekursiooni jaoks lisatud võtmesõnale `rec` kooloniga identifikaator `outer`. Seda kooloniga notatsiooni kasutatakse nii rekursiivse λ-abstraktsiooni päises kui selle rekursiooni poolt eelnevalt arvutatud väärtuste

kasutamisel. Sisemise rekursiooni jaoks ei ole vaja identifikaatorit kasutada, kuna tavaline `rec` viitab kõige sisemisele ilma nimeta rekursioonile, mis on antud skoobis kättesaadav. (Seega võib ka sisemisele rekursioonile nime anda ja välimise nimeta jätta.)

Fumontrixis saab lihtrekursiooni kasutada ka üle keerulisemat liiki tüüpide. Defineerime näiteks unaarsed tüübitaseme naturaalarvud, millel on lisaks üks üleliigne argument. Neid saab kasutada näiteks ühikuga suuruste jaoks. See üleliigne argument on siis suuruse väärtus ning unaarne tüübitaseme naturaalarv väljendab suuruse ühikut: null — dimensioonita suurus, üks — meeter, kaks — ruutmeeter jne. Suuruse ühik on siis tüübitasemel staatiliselt teada, erinevalt selle väärtusest, mis leitakse dünaamiliselt.

```
data FZero B = FZero B;
data FSucc (A : * -> *) B = FSucc B;

type tfFSum = \ a : * -> * . \ b : * -> * rec * -> * .
   case b of
      FZero    -> a;
      FSucc b' -> FSucc (rec b');
   end;
```

Siis saab kasutada näiteks avaldist

```
FSucc 3 $: (tfFSum (FSucc FZero) (FSucc (FSucc (FSucc FZero)))),
```

mis defineerib viienda astme (siin on viis `FSucc` konstruktorit) ühikuga suuruse, mille väärtus on 3.

Fumontrixis funktsionaalset liiki tüübinäidiste kasutamisel tuleb arvestada, et siin ei kehti ekstensionaalsuse printsiip. Ekstensionaalsuse korral kehtiks samaväärsus $\lambda x.f\ x \equiv f$, st funktsiooni semantika oleks määratud tema väärtustega kõigil võimalikel argumentidel. See muudab ebapraktiliseks (lõpliku, kuid suure määramispiirkonna korral) või mittelahenduvaks (lõpmatu määramispiirkonna korral) funktsioonide võrdsuse kontrollimise. Seetõttu on tüübitasemel funktsionaalsete näidiste kasutamiseks Fumontrixis tüübitasemel ekstensionaalsusest loobutud. Seega funktsioonid `FZero` ja
`\ x . FZero x` on erinevad (kuigi ekstensionaalselt võrdsed) ning esimene neist sobitub näidisega `FZero`, kuid teine mitte. Fumontrixi andmetasemel siiski ekstensionaalsus kehtib.

Ka GHC-s ei kehti tüübitasemel ekstensionaalsus. Nimelt tüübisünonüüme ei saa defineerida tüübiklassi elementideks. Näiteks

```
class C (a :: * -> *)
```

```
type F = Either Int
type G a = Either Int a
--instance C G  -- ei ole lubatud
instance C F    -- on lubatud
```

Siin tüübifunktsioonid `F` ja `G` on ekstensionaalselt võrdsed, kuna `F a` ja `G a` on iga `a` korral võrdsed (mõlemad on `Either Int a`). Samas esindajadeklaratsioonis on `F` lubatud, aga `G` keelatud.

5.2 Mõningaid tüübitaseme funktsioonide rakendusi

5.2.1 Tüübitaseme funktsioonid ja *ad-hoc*-polümorfism

Üks tüübitaseme funktsioonide rakendus on *ad-hoc*-polümorfsete funktsioonide realiseerimine. *Ad-hoc*-polümorfset funktsiooni saab rakendada erinevat tüüpi argumentidele ja erinevate tüüpide korral võib funktsioon käituda erinevalt. Sellisel juhul võib kirjutada tüübitaseme funktsiooni, mis seab vaadeldava funktsiooni igale lubatud argumendi tüübile vastavusse monomorfse funktsiooni, mis ootab argumendiks seda tüüpi väärtust. Näiteks Javas võime defineerida *ad-hoc*-polümorfse funktsiooni `f`:

```
boolean f(int x) {
   return x != 0;
}

int f(boolean x) {
   return x ? 1 : 0;
}
```

Siis saame kasutada avaldisi nagu `f(2)` ja `f(true)`. Siin on kummagi lubatud argumenditüübi (`int` ja `boolean`) jaoks defineeritud eraldi monomorfne funktsioon ning need definitsioonid kokku annavad *ad-hoc*-polümorfse funktsiooni. Siin on iga argumendi tüübi jaoks vaja anda eraldi definitsioon ja koodi lugeja peab kõik skoobis olevad definitsioonid läbi vaatama, et kogu polümorfse funktsiooni definitsioon kätte saada.

Fumontrixis defineeritakse tüübitaseme funktsioon ühe definitsiooniga:

```
type f = \ x : @ .
   case typeof type x of
      Int ->
         value (type x) != 0;
      Bool ->
         value if (type x) 1 0;
   end;
```

Siis saab kasutada avaldisi nagu `type f (value 2)` ja `type f (value True)`. See süntaks on natuke kohmakas, kuna tüübitaseme ja andmetaseme vahel liikumiseks tuleb kasutada võtmesõnu `type` ja `value`. Soovi korral võib lisada süntaktilist suhkrut, mis teeb selle tasemete vahel liikumise mugavamaks, siis võiks näiteks kirjutada eelnevalt vaadeldud avaldised kujul `'f 2` ja `'f True` ning tüübitaseme funktsiooni definitsioonis `type x` asemel `@x`. Praegu ei ole lihtsuse huvides seda keelde lisatud.

Liiki `@ -> @` tüübitaseme funktsioonid ehk need, mis seavad väärtusele vastavusse väärtuse, sarnanevad väärtustaseme funktsioonidega. Kõiki väärtustaseme funktsioone saab ka tüübitaseme funktsioonidena defineerida. Samas väärtustasemel saab defineerida ainult neid funktsioone, millel on tüübisüsteemis olemas tüüp.

Tüübitasemel on võimalik defineerida ka selliseid funktsioone, millel tüübisüsteemis tüüpi ei ole (selline on ka funktsioon `f` viimases näites). See ei tähenda staatilise tüübikontrolli puudumist. Sellisel funktsioonil on olemas implitsiitne tüüp. Iga tüübi jaoks saab staatiliselt kontrollida, kas see tüüp sobib antud funktsiooni argumendiks ja, kui sobib, siis saab staatiliselt määrata rakendamise tulemuse tüübi.

Kuna tüübitaseme funktsioonil tüübisüsteemis tüüpi ei ole, siis ei saa seda anda argumendiks andmetaseme funktsioonile. Küll aga saab seda anda argumendiks teisele tüübitaseme funktsioonile (näiteks liiki `(@ -> @) -> @`). Samuti saab (näiteks liiki `@ -> @ -> @`) tüübitaseme funktsiooni tulemuseks olla teine tüübitaseme funktsioon, seega saab kasutada *curried*-kujul tüübitaseme funktsioone. Seega tüübitaseme funktsioonid on tüübitasemel *first-class*.

5.2.2 Tüübitaseme funktsioonid ja tüübiinferents

Kuna tüübitaseme funktsiooni korral ei ole vaja määrata argumendi tüüpi (liik tuleb siiski määrata), siis saab tüübitaseme funktsioone kasutada tüübiinferentsi asemel. Näiteks võime andmetaseme funktsiooni

```
double = \ x : Int . x + x;
```

asemel defineerida tüübitaseme funktsiooni:

```
type double = \ x : @ . value (type x) + (type x);
```

Erinevalt andmetasemest ei ole tüübitasemel funktsiooni argumendi tüüpi määratud. Samas selle funktsiooni rakendamisel mingit muud tüüpi kui `Int` argumendile tuleb staatiliselt tüübiviga (kuna muud tüüpi väärtust ei saa funktsiooni `+` argumendiks anda).

Samas võib juhtuda, et programmeerija kirjutab kogemata `+` asemel `||`. Esimeses definitsioonis tuleb see viga kohe välja, kuna `x` tüübiks on määratud `Int`, kuid `||` ootab argumendiks `Bool`-tüüpi väärtust. Tüübitaseme funktsiooni korral ei tule see viga definitsiooni kontrollimisel välja. Definitsioon

```
type double = \ x : @ . value (type x) || (type x);
```

on täiesti tüübikorrektne (ja rakendatav Bool-tüüpi väärtustele), kuid see erineb sellest, mida programmeerija plaanis kirjutada. Viga tuleb välja alles siis, kui funktsiooni üritatakse rakendada `Int`-tüüpi väärtusele, kuid näiteks teegi kirjutamisel võib juhtuda, et mõnda defineeritud funktsiooni ei rakendata kordagi (selle teegi koodis).

Sama probleem esineb ka tüübiinferentsi korral. Kui Haskellis kirjutada definitsioon

```
double x = x || x
```

siis translaator ei tea, et programmeerija plaanis argumendi tüübiks `Int`, ning tüübiviga jääb avastamata.

See on ka üks põhjus, miks Fumontrixis tüübiinferentsi ei kasutata. Vajaduse korral saab kasutada tüübitaseme funktsioone, kuid siis peab programmeerija lihtsalt ettevaatlikum olema. Haskellis kasutatakse tüübiinferentsi alati ja programmeerija ei saa seda keelata. GHC-s saab küll funktsiooni argumendile tüübiannotatsiooni lisada, kuid programmeerija võib selle mõnes kohas ära unustada. Translaator sellisel juhul ei hoiata ja tüübivead võivad jääda avastamata.

Tüübitaseme funktsioonide korral esineb veel üks probleem. Näiteks definitsioon

```
type tb = \ x : @ . value 3 + True;
```

tüübiviga ei anna, kuigi selles sisalduv avaldis `3 + True` ei ole tüübikorrektne. Siin on lihtsalt tegemist tüübitaseme funktsiooniga, mis ei ole ühegi argumendi korral defineeritud, samas kui funktsioon ise on defineeritud. Iga katse seda funktsiooni rakendada annab tüübivea. Tüübiviga on tüübitaseme analoog andmetaseme $\bot$-le. Vaadeldaval juhul alamavaldis ei sõltu funktsiooni argumendist `x`, kuid üldjuhul võib sõltuda ning sel juhul on juba keeruline kindlaks teha, kas funktsioon annab iga argumendi korral tüübivea. Fumontrixis sellist analüüsi ei üritata teha. Selliste vigade vältimiseks võib tüübitaseme funktsiooni argumendist mittesõltuvad alamavaldised tüübilambda alt välja tuua eraldi definitsioonidesse. See muudab küll koodi hakitumaks.

Liigivead ja tundmatute tüübiidentifikaatorite kasutused tulevad siiski kohe tüübitaseme funktsiooni defineerimisel välja, kuna need ei sõltu kunagi argumendist (argumendi liik on fikseeritud).

5.3 Tüübitaseme funktsioonid Haskellis

5.3.1 *Ad-hoc*-polümorfsed funktsioonid

Haskell 98-s on olemas tüübisünonüümid, mille abil on võimalik defineerida ainult parameetriliselt polümorfseid tüübitaseme funktsioone. Samuti on võimalik kasutada tüübiklasse, mille abil saab defineerida polümorfseid funktsioone. GHC-s on selliselt defineeritavate polümorfsete funktsioonide hulk palju suurem kui Haskell 98-s. Kuna tüübisünonüümid on Haskellis väga piiratud võimalustega, siis neid me selles töös lähemalt ei vaatle ning Haskelli või GHC tüübitaseme funktsioonide [5] all peame silmas tüübiklasside abil defineeritavaid polümorfseid funktsioone.

Haskell 98-s olemas ainult ühe parameetriga tüübiklassid, mille abil on võimalik defineerida *ad-hoc*-polümorfseid funktsioone (erinevat tüüpi argumentide korral võib tulemuse väärtuse leidmiseks argumendi väärtuse põhjal kasutada erinevat koodi), mille tulemuse tüüp sõltub argumendi tüübist ainult parameetriliselt polümorfselt (tulemuse tüüp tuleb avaldada ühe tüübiavaldisena, mis võib argumendi tüüpi sisaldada ainult parameetrina ehk tüübimuutujana). Näiteks võime kirjutada funktsiooni, mis nii tõeväärtustele kui täisarvudele seab vastavusse täisarvu, kuid erineva koodiga:

```
class C1 a where
   f1 :: a -> Integer
instance C1 Bool where
   f1 b = bool2int b
instance C1 Integer where
   f1 x = x
```

Siis näiteks `f1 (3::Integer)` annab tulemuseks `3` ja `f1 True` annab tulemuseks `1`. Siin eeldame, et eelnevalt on olemas definitsioonid

```
int2bool x = x /= 0
bool2int b = if b then 1 else 0
```

Samuti võime Haskell 98-s kirjutada funktsiooni, mis tõeväärtustele seab vastavusse tõeväärtuste listi ning täisarvudele täisarvude listi, kuid erineva koodiga:

```
class C2 a where
   f2 :: a -> [a]
instance C2 Bool where
   f2 b = [b,b]
instance C2 Integer where
   f2 x = [x,x,x]
```

Samuti võime keerulisema argumendi tüübi jaoks tulemuse väärtuse defineerimisel kasutada rekursiivselt lihtsama argumendi tüübi jaoks tulemuse arvutamist. Näiteks võime eelnevale lisada

```
instance C2 a => C2 [a] where
   f2 xs = map f2 xs
```

Siin defineerime funktsiooni listide jaoks, kasutades rekursiivselt selle listi elementide jaoks defineeritud funktsiooni.

5.3.2 *Ad-hoc*-polümorfism tulemuse tüübi jaoks

Kui me tahame, et ka funktsiooni tulemuse tüüp sõltuks argumendi tüübist *ad-hoc*-polümorfselt, siis Haskell 98-st ei piisa. Sellisel juhul on võimalik kasutada GHC-d, kus on olemas mitme parameetriga tüübiklassid ja funktsionaalsed sõltuvused (selleks tuleb GHC-s sisse lülitada võti `-fglasgow-exts`). Sellisel juhul saame kirjutada näiteks funktsiooni, mis tõeväärtustele seab vastavusse täisarvu ja täisarvudele tõeväärtuse:

```
class C3 a b | a -> b where
   f3 :: a -> b
instance C3 Bool Integer where
   f3 b = bool2int b
instance C3 Integer Bool where
   f3 x = int2bool x
```

5.3.3 Rekursioon tulemuse tüübi leidmiseks

Proovime nüüd sarnaselt eespool olnud näitele laiendada selle funktsiooni ka listide jaoks:

```
instance C3 a b => C3 [a] [b] where
   f3 xs = map f3 xs
```

Seda GHC vaikimisi ei aktsepteeri, kuna nn *Coverage Condition* (vt [2], jaotis 8.6.3.1) ei kehti. See nõuab, et esindajadeklaratsiooni peas (antud juhul `C3 [a] [b]`) funktsionaalse sõltuvuse tulemustüüp (antud juhul `[b]`) ei sisaldaks selliseid tüübimuutujaid, mis ei esine selle funktsionaalse sõltuvuse argumenttüüpides (antud juhul `[a]`). Kuna tulemuses esineb muutuja `b`, mida argumendis ei esine (seal on ainus muutuja `a`), siis tingimus ei kehti. See tingimus tähendab, et ei ole võimalik tulemustüübi leidmisel kasutada rekursiivselt leitud tüüpi (antud juhul `b`, mis on `C3` poolt rekursiivselt tüübile `a` vastavusse seatud tüüp), kuna see tüüp on vaja tähistada uue muutujaga, mida argumenttüüpides ei esine.

Et sellest tingimusest vabaneda, on GHC-s võimalik sisse lülitada võti `-fallow-undecidable-instances`. Sellisel juhul GHC aktsepteerib selle viimati toodud deklaratsiooni. See võti aga muudab võimalikuks ka mittetermineeruvate tüübifunktsioonide kirjutamise. Näiteks aktsepteerib GHC siis järgmised deklaratsioonid:

```
class C5 a where
   f5 :: a -> Integer
instance C5 [a] => C5 a where
   f5 = const 3
```

Kui nüüd üritada `f5` kasutada, siis läheb tüübikontroll lõpmatusse rekursiooni (reaalselt katkestatakse see teatud sügavusele jõudmisel ära).

Eelnevas `C3` näites oli tegelikult ainult lihtrekursiooni vaja, mitte üldist rekursiooni, kuid GHC seda ei võimalda. Fumontrixis kasutatakse tüübitasemel lihtrekursiooni, mis garanteerib tüübikontrolli termineeruvuse. Lähemalt vaatlesime seda jaotises 5.1.3.

5.3.4 Hargnemine tüübi struktuuri järgi

Vaatleme nüüd, kuidas kasutada hargnemist tüübi struktuuri järgi, nn tüübitaseme `case`-konstruktsiooni ekvivalenti. Haskellis (GHC-s) tuleb selleks kirjutada eraldi esindajadeklaratsioon iga argumenttüübi variandi jaoks (eelnevas `C3` näites `Bool`, `Integer`, `[a]`). See on kohmakas ja ebamugav, kuna klassi nimi tuleb korduvalt välja kirjutada. Fumontrixis on selle asemel olemas tüübitaseme `case`-konstruktsioon (vt jaotist 5.1.1).

Oletame nüüd, et me soovime kasutada hargnemist mitte argumenttüübi, vaid rekursiivselt arvutatud tüübi struktuuri järgi. Proovime näiteks kirjutada

```
class C4 a b | a -> b where
   f4 :: a -> b
instance C4 Integer Bool where
   f4 x = True
instance C4 a Integer => C4 [a] Bool where
   f4 x = int2bool (f4 (head x))
instance C4 a Bool => C4 [a] Integer where
   f4 x = bool2int (f4 (head x))
```

Siin me soovime, et kui tüüpi `a` väärtusele seatakse vastavusse täisarv, siis tüüpi `[a]` väärtusele seataks vastavusse tõeväärtus ning vastupidi, kui rekursiivselt arvutati välja tõeväärtus, siis tulemuseks oleks täisarv. Seda aga

GHC ei aktsepteeri, kuna `C4 [a]` jaoks on mitu deklaratsiooni ning GHC seda ei luba, isegi kui kontekstid on mittelõikuvad, nagu on praegusel juhul.

Selle asemel tuleb hargnemise jaoks defineerida eraldi tüübiklass:

```
class C4a a b | a -> b where
   f4a :: a -> b
instance C4a Bool Integer where
   f4a b = bool2int b
instance C4a Integer Bool where
   f4a x = int2bool x
```

Siis saab `C4` defineerida järgmiselt:

```
class C4 a b | a -> b where
   f4 :: a -> b
instance C4 Integer Bool where
   f4 x = True
instance (C4 a b, C4a b c) => C4 [a] c where
   f4 x = f4a (f4 (head x))
```

Näeme, et GHC-s on tüübitaseme funktsioonid ühetasemelised, iga alamploki jaoks tuleb defineerida eraldi tüübitaseme funktsioon ning need funktsioonid peavad kõik olema peataseme skoobis. See on kohmakas ja muudab koodi väga hakituks, samuti peab programmeerija kõigile neile funktsioonidele nimed välja mõtlema ja meelde jätma. Fumontrixis on tüübitaseme funktsioone võimalik defineerida suvalises alamskoobis, üksteise sees saab kasutada kui tahes palju tüübitaseme `case`-konstruktsioone ja rekursiivseid väljakutseid ning samuti saab kasutada anonüümseid tüübitaseme funktsioone (ka rekursiivseid), et programmeerija ei peaks igale funktsioonile nime välja mõtlema.

Fumontrixis saab selle `C4` näite kirjutada järgmiselt:

```
type tf4 = \ v : @ .
   value
      (type
         (\ t rec @ .
            case t of
               Int    ->
                  value const True $: t;
               List a ->
                  case typeof type rec a of
                     (_ -> Bool) ->
                        value const 3 $: t;
```

```
                    (_ -> Int) ->
                       value const True $: t;
                end;
          end
       ) (typeof type v)
    ) (type v);
```

Siin kasutatakse anonüümset tüübitaseme funktsiooni, mis seab tüübile (kuna lihtrekursiooni saab teha ainult üle tüüpide, mitte üle väärtuste) vastavusse monomorfse funktsiooni. Sellele tüübitaseme funktsioonile antakse seejärel argumendiks esialgse argumentväärtuse `v` tüüp ning saadud monomorfsele funktsioonile antakse andmetasemel argumendiks väärtus `v`.

5.3.5 Vaikevariandid hargnemisel

Mõnikord oleks kasulik tüübi järgi hargnemisel kasutada vaikevariante, et ei peaks kõiki variante ükshaaval välja kirjutama. Kirjutame näiteks

```
class C6 a where
   f6 :: a -> Integer
instance C6 Bool where
   f6 = bool2int
instance C6 a where
   f6 = const 3
```

Siin peaks `f6` tõeväärtuste korral käituma ühtmoodi, aga ülejäänud tüüpide väärtuste korral teistmoodi. Neid deklaratsioone GHC vaikimisi ei aktsepteeri, kuna tüübi `a` kohta käiv deklaratsioon oleks rakendatav ka tüübile `Bool`. GHC-s on võimalik sisse lülitada lipp `-fallow-incoherent-instances`, sellisel juhul valib GHC kõige täpsema (kõige väiksema muutujate arvuga) esindajadeklaratsiooni, mis vaadeldava tüübiga sobib. Seega tüübi `Bool` jaoks valitakse esimene esindajadeklaratsioon, ülejäänud tüüpide jaoks teine.

Haskellis on võimalik kasutada ka vaikedefinitsioone. Näiteks

```
class C6 a where
   f6 :: a -> Integer
   f6 = const 3
instance C6 Bool where
   f6 = bool2int
instance C6 Char
instance C6 [a]
```

See võimaldab `f6` definitsiooni mõnedest esindajadeklaratsioonidest ära jätta, kuid esindajadeklaratsioone siiski ära jätta ei saa ning seega on tegemist vaid süntaktilise suhkruga ja seda konstruktsiooni me siin lähemalt ei vaatle.

Proovime vaikevariante kasutada ka tulemuse tüübi defineerimisel:

```
class C7 a b | a -> b where
   f7 :: a -> b
instance C7 Bool Int where
   f7 = bool2int
instance C7 a Bool where
   f7 = const True
```

Seda aga GHC ei luba, isegi kui võti `-fallow-incoherent-instances` on sisse lülitatud. Seega siin tuleb kõigi vajalike tüüpide jaoks ükshaaval esindajadeklaratsioonid kirjutada, isegi kui kõigi jaoks peale ühe on need sama sisuga. Kui vajalikku tüüpi ei ole veel defineeritud (see defineeritakse teises moodulis), siis ei saagi kohe seda esindajaks defineerida.

Samuti ei pruugi alati kõige täpsemat esindajadeklaratsiooni olla. Näiteks

```
class C8 a where
   f8 :: a -> Integer
instance C8 (Integer,b,c,d) where
   f8 (x,_,_,_) = x
instance C8 (a,Integer,c,d) where
   f8 (_,y,_,_) = 10*y
instance C8 (a,b,Integer,d) where
   f8 (_,_,z,_) = 100*z
instance C8 (a,b,c,Integer) where
   f8 (_,_,_,w) = 1000*w
instance C8 (a,b,c,d) where
   f8 (_,_,_,_) = 0
```

Siin leidub kõige täpsem deklaratsioon ainult siis, kui nelikus maksimaalselt üks tüüp on `Integer`. Kui neid on mitu, siis sobib mitu deklaratsiooni, millest ükski pole kõige täpsem. Antud juhul lubab GHC kasutada funktsiooni `f8` ainult selliste väärtuse jaoks, mille tüübi nelikus on maksimaalselt üks `Integer`. Muude variantide kasutamisel tuleb tüübiviga. Selleks, et need muud variandid oleks kasutatavad, tuleb lisada täpsemaid esindajadeklaratsioone. Antud juhul oleks vaja kokku $2^4 = 16$ esindajadeklaratsiooni, et kõik nelikutüübid oleks kasutatavad. See muutuks väga kohmakaks, kuna paljud nendest variantidest oleks sama sisuga.

Vaja oleks, et mitme sobiva esindajadeklaratsiooni korral valitaks nendest esimene. Siis piisaks nendest viiest esindajadeklaratsioonist. GHC-s seda ei

võimaldata. Siin on esindajadeklaratsioonid nagu loogilise programmeerimise implikatsioonid. Nende implikatsioonide järjekorda ei loeta oluliseks. Loogilises programmeerimises saaks implikatsioonide eeldustesse kirjutada eelmiste implikatsioonide paremate poolte eitused, aga GHC-s sellist eituse konstruktsiooni ei ole ning seda ei saa ka ise defineerida nagu `C7` näitest näha (kui see näide töötaks, siis saaks predikaadiga `C7 a Bool` nõuda, et `a` ei oleks tüüpi `Bool`).

Fumontrixis on tüübitaseme `case`-konstruktsioonis võimalik kasutada vaikevariante (näidise `_` või tüübimuutujate abil) ning variante vaadeldakse nende esinemise järjekorras, esimese sobiva leidmisel enam edasi ei vaadata. Vaadeldud `C7` näite saab Fumontrixis kirjutada näiteks nii:

```
type tf7 = \ v : @ .
   case typeof type v of
      Bool -> value bool2int (type v);
      _    -> value True;
   end;
```

`C8` näite saab kirjutada nii:

```
type tf8 = \ v : @ .
   value let
      Tuple4 x y z w = type v
   in type
      case typeof type v of
         Tuple4 Int b c d -> value x;
         Tuple4 a Int c d -> value 10*y;
         Tuple4 a b Int d -> value 100*z;
         Tuple4 a b c Int -> value 1000*w;
         Tuple4 a b c d   -> value 0;
      end;
```

See töötab kõigi nelikutüüpide korral.

5.3.6 GHC tüübitaseme funktsioonide puudusi

GHC-s on tüübiklasside abil võimalik defineerida polümorfseid andmetaseme funktsioone. Funktsionaalsete sõltuvuste abil on võimalik defineerida ka funktsioone, mis seavad tüübile vastavusse tüübi, kuid erinevalt Fumontrixist ei ole vaikevariantide kasutamine sel juhul võimalik, kõik variandid peavad olema lõikumatud.

Keerulisemat liiki tüübitaseme funktsioone ei saa GHC tüübiklasside abil üldse defineerida, kuna tüübitaseme funktsioone ei saa tüübiklasside argumendina kasutada. Erinevalt Fumontrixist ei ole GHC tüübitaseme funktsioonid tüübitasemel *first-class*. Seega ei saa defineerida selliseid kasulikke operaatoreid nagu jaotises 5.1.2 defineeritud `argtype` ning erinevad funktsiooni rakendamise operaatorid nagu jaotises 6.4.2 defineeritav `applyImpred` impredikatiivseks funktsiooni rakendamiseks.

Samuti ei ole GHC-s `typeof`-operaatorit ning tüübiklasside abil defineeritud tüübitaseme funktsioonidega arvutatud tüüpe ei saa annotatsioonides kasutada.

GHC-s on olemas ka tüübisünonüümid, mis võivad ka keerulisemat liiki olla, kuid nende abil on võimalik ainult oma argumendi suhtes parameetriliselt polümorfseid funktsioone kirjutada, argumendi struktuuri järgi hargnemist kasutada ei saa.

Nii tüübisünonüüme kui tüübiklasse ja esindajaid saab GHC-s defineerida ainult peataseme skoobis, mis suurendab koodi hakitust. Fumontrixis saab tüübitaseme funktsioone defineerida suvalises skoobis.

6 Polümorfism

6.1 Üldiselt polümorfismist

Polümorfism võimaldab kasutada sama nime mitme sarnase tähendusega (kuid erinevat tüüpi) väärtuse jaoks. See vähendab programmeerija jaoks nimede väljamõtlemise ja pikkade nimede kirjutamise vaeva ning muudab koodi lühemaks.

Polümorfismi on mitut liiki. Funktsionaalsetes keeltes laialt levinud parameetriline polümorfism võimaldab kasutada sama koodi (funktsiooni definitsiooni) erinevat tüüpi argumentide jaoks. See vähendab vajadust sama koodi mitmesse kohta kopeerimiseks.

Ad-hoc-polümorfism võimaldab defineerida funktsioone, mis on kasutatavad erinevat tüüpi argumentide jaoks ning nende erinevate tüüpide jaoks rakendatakse erinevat koodi. Sisuliselt on tegemist mitme sama nimega funktsiooniga, millest translaator valib tüübikontrolli käigus argumendi tüübi põhjal õige.

6.2 Polümorfism ja tüübitaseme funktsioonid

Fumontrixis on polümorfismi võimalik realiseerida ka tüübitaseme funktsioonide abil. Liiki `@ -> @` tüübitaseme funktsiooni saab kasutada sarnaselt andmetaseme funktsiooniga (kuigi süntaks on natuke kohmakam). Tüübitaseme funktsiooni korral ei ole argumendi tüüpi ilmutatult deklareeritud, see on määratud ilmutamata kujul sellega, millistes kontekstides seda argumenti funktsiooni kehas kasutatakse.

Tüübitaseme funktsiooni tulemuse tüüp võib sõltuda argumendi tüübist suvalise lihtrekursiivse funktsioonina kvantoriteta tüüpidel (kuna kvantoritega tüüpide sisse ei võimalda Fumontrixi tüübitaseme `case`-konstruktsioon vaadata). Seega tüübitaseme funktsioon väärtustel võib olla suvaline osaliselt rekursiivne funktsioon väärtustel (mille tüüp ei sisalda kvantoreid), mis sama tüüpi väärtustele seab alati vastavusse sama tüüpi tulemuse ning mille poolt määratud funktsioon tüüpidel on lihtrekursiivne.

Siin tuleb tüüpide hulka lugeda ka nn $\bot$-tüüp ehk tüübiviga. Tüübitaseme funktsioonid on tüüpide suhtes agara semantikaga, seega tüübiviga ei saa esineda tüübikonstruktori argumendina ning andes tüübivea argumendiks tüübitaseme funktsioonile, on tulemuseks ikka sama tüübiviga.

Tüübitaseme funktsioonid liiki `@ -> @` võtavad argumendiks ja annavad tulemuseks andmetaseme väärtusi. Kui me soovime tüübitaseme funktsiooni argumendi või tulemusena kasutada teisi tüübitaseme funktsioone, siis tuleb kasutada kõrgemat liiki (näiteks `(@ -> @ -> @) -> @ -> @`) tüübitaseme

funktsioone. Ka nendel on olemas ilmutamata tüüp, mis sõltub sellest, millistes kontekstides argumenti kasutatakse.

Kui tüübitaseme funktsioonina saab defineerida praktiliselt kõiki polümorfseid funktsioone, siis milleks on andmetaseme polümorfseid funktsioone vaja?

Tüübitaseme funktsioone ei saa hoida andmetaseme andmestruktuuride sees, selleks tuleks kasutada tüübitaseme andmestruktuure. Fumontrixis saab algebraliste andmetüüpide konstruktoritele argumendiks anda ainult tüüpe ja tüübikonstruktoreid, mitte suvalisi tüübitaseme funktsioone. See on vajalik tüübikontrolli lahenduvuse säilitamiseks. Vastasel korral saaks kirjutada näiteks järgmised definitsioonid:

```
data Dt21 (A : * -> *);
type tf21 = \ pf .
   case pf of
      Dt21 tf21' -> tf21' pf;
   end;
```

Siin defineerime uue tüübikonstruktori, mis saades argumendi liiki `* -> *`, konstrueerib uue tüübi liiki `*`. Seejärel defineerime funktsiooni `tf21`, mis on liiki `* -> *`. Konstruktori `Dt21` abil saame selle funktsiooni pakkida tavaliseks liiki `*` tüübiks, mille saame anda argumendiks sellele samale funktsioonile, mis pakib selle lahti, saades kätte iseenda koopia, mille kutsub rekursiivselt välja. Tekib lõpmatu rekursioon. Kuna Fumontrixis oli eesmärk säilitada tüübikontrolli lahenduvus, siis tuli vaadeldav kitsendus sisse viia.

Üks olukord, kus veel andmetaseme polümorfseid funktsioone vaja läheb, on siis, kui me soovime kirjutada polümorfset funktsiooni, mis töötaks ka selliste tüüpide korral, mida selle funktsiooni defineerimise hetkel veel ei eksisteeri ning mille käitumine nende tüüpide jaoks ei ole samuti veel täpselt teada. Jaotises 6.3 vaatleme, kuidas sellisel juhul polümorfismi kasutada.

Samuti läheb andmetaseme polümorfseid funktsioone vaja juhul, kui me soovime anda polümorfset väärtust argumendiks (polümorfsele või monomorfsele) funktsioonile, mis ootab monomorfset väärtust. Seda olukorda vaatleme jaotises 6.4.

6.3 Polümorfism ja dünaamilise skoopimise elemendid

6.3.1 Veel üks polümorfismi liik

Nii parameetrilise kui *ad-hoc*-polümorfismi korral on argumendiks sobivate argumentide hulk funktsiooni definitsiooniga fikseeritud (parameetrilise polümorfismi korral kuuluvad sinna ka need tüübid, kus parameetriks on mingi

veel defineerimata tüüp). Sageli oleks vaja, et funktsiooni argumendiks sobiksid kõik teatud tingimustele vastavad tüübid, ka need, mida ei ole funktsiooni defineerimise skoobis veel olemas. Selliseks tingimuseks võib olla näiteks mõne teise funktsiooni rakendatavus antud tüüpi väärtustele.

Näiteks võime defineerida polümorfse võrduse kontrollimise funktsiooni `(==)` ning seejärel seda kasutava mittevõrduse kontrollimise funktsiooni `(!=)`, mis on rakendatav sama tüüpi väärtusele nagu funktsioon `(==)`. Kui me nüüd mingis alamskoobis funktsiooni `(==)` definitsiooni laiendame, nii et see on rakendatav mingit uut tüüpi väärtustele, siis tavalise *ad-hoc*-polümorfismi korral funktsioon `(!=)` jääb kasutama vana `(==)` definitsiooni ning ei ole rakendatav uut tüüpi väärtustele. Tegelikult me sooviks, et `(==)` definitsiooni uuendamisel uueneks automaatselt ka `(!=)` väärtus, st ühe funktsiooni definitsioon kasutaks parameetrina teise funktsiooni väärtust selle esimese funktsiooni väljakutsumise hetkel ehk dünaamilises skoobis.

Seega tekib vajadus teatud dünaamilise skoopimise elementide järele. Funktsiooni definitsioonis oleks vaja kasutada lisaks leksilise skoobi identifikaatoritele ka mõnda dünaamilise skoobi identifikaatorit. Üks võimalus selle realiseerimiseks oleks panna funktsiooni semantika sõltuma lisaks leksilise skoobi keskkonnale ka dünaamilise skoobi keskkonnast ning eristada kuidagi süntaktiliselt, kas identifikaator viitab leksilise või dünaamilise keskkonna muutujale. Samas ei ole funktsiooni defineerimise hetkel teada, millised on selles definitsioonis kasutatavate dünaamilise keskkonna muutujate tüübid funktsiooni väljakutsumise skoobis. Seega tuleks kasutatavate dünaamilise skoopimisega muutujate tüübid kuidagi deklareerida.

6.3.2 Dünaamilise skoopimise elemendid Haskellis

GHC-s on dünaamilise skoopimise jaoks võimalik kasutada implitsiitseid parameetreid ([2], jaotis 8.7.2):

```
double :: (?single :: Integer) => Integer
double = 2 * ?single
xs = (let ?single = 3 in double, let ?single = 4 in double)
```

Siin `xs` saab väärtuseks `(6,8)`. Näeme, et implitsiitse parameetri `?single` tüüp `Integer` on fikseeritud.

Selles näites funktsioonil `double` ainult implitsiitne parameeter ongi ja tavalist argumenti ei ole. Sageli on aga funktsioonil olemas ka tavaline argument ja implitsiitne parameeter sõltub selle tavalise argumendi tüübist. Eespool vaadeldud funktsioon `(!=)` korral oleks implitsiitseks parameetriks funktsiooni `(==)` (monomorfne) realisatsioon funktsioonile `(!=)` argumendiks antud väärtuse tüübi jaoks.

Haskellis on selleks olemas tüübiklassid. Näiteks võime defineerida

```
class Equal a where
   (===) :: a -> a -> Bool

(!=) :: Equal a => a -> a -> Bool
x != y = not (x === y)
```

Siin on `(==)` asemel kasutatud `(===)`, kuna esimene on prelüüdis defineeritud ja Haskell ei luba peatasemel muutujaid ümber defineerida. Siin funktsiooni `(!=)` implitsiitseks parameetriks on funktsiooni `(===)` monomorfne realisatsioon tüübi `a` jaoks. Tüüp `a` selgub funktsioonile `(!=)` esimese argumendi andmisel, sel hetkel kehtivast skoobist võetakse ka `(===)` realisatsioon, mis funktsioonile `(!=)` antakse. Funktsiooni `(!=)` tüüp sisaldab tingimust `Equal a`, mis määrab, et see funktsioon on rakendatav kõikidele tüüpidele `a`, mis rahuldavad predikaati `Equal` (ehk millele on rakendatav funktsioon `(===)`.

6.3.3 Dünaamilise skoopimise elemendid Fumontrixis

Fumontrixis on samuti olemas tüübiklassid. Defineerime klassi `Equal` ja selle eksemplari täisarvutüübi jaoks:

```
class equal forall A. A -> A -> Bool;

type equal = \ a .
   case a of
      Int -> value intEq;
   end;
```

Fumontrixis on tüübiklasside esindajad määratud tüübitaseme funktsiooniga, mille nimi ühtib klassi nimega ja mis seab tüübile vastavusse monomorfse väärtuse (klassi eksemplari), mis on polümorfse väärtuse realisatsioon antud tüübiparameetri väärtuse jaoks. Kui mingis alamskoobis on vaja klassi esindajaid lisada või muuta, siis tuleb lihtsalt vastav tüübitaseme funktsioon ümber defineerida. Lisame näiteks eksemplari tõeväärtustüübi jaoks:

```
type equal = \ a .
   case a of
      Bool -> value boolEq;
      _    -> equal a;
   end;
```

Näidise _ abil säilitame siin ka varem defineeritud eksemplarid.

Tüübiklassile vastava polümorfse andmetaseme funktsiooni defineerime järgmiselt:

```
(==) = forall equal A. type equal A;
```

Nüüd saame defineerida ka mittevõrduse kontrollimise funktsiooni:

```
(!=) = forall equal A.
   \ x : A . \ y : A .
      not (x == y);
```

Siin ei ole vaja enam tüübitaseme funktsiooni `equal` välja kutsuda, vaid võime kasutada eelnevalt defineeritud andmetaseme funktsiooni `(==)`. Kvantoriga muutuja sissetoomisel tuleb vaid lisada kitsendus `equal`, mis nõuab, et muutuja kuuluks sellesse klassi.

Fumontrixis on olemas ainult ühe argumendiga tüübiklassid, kitsendada saab ainult kvantifitseeritavat muutujat (mitte seda parameetrina sisaldavat tüüpi) ning ühele muutujale saab lisada ainult ühe kitsenduse. Seega võrreldes Haskelli ja eriti GHC-ga on siin tegemist üsna lihtsa tüübiklasside realisatsiooniga. Fumontrixis on põhirõhk asetatud tüübitaseme funktsioonidele, mis sellistel juhtudel, kus dünaamilist skoopimist vaja ei lähe, on palju suuremate võimalustega kui GHC analoogilised konstruktsioonid. Seda vaatlesime lähemalt peatükis 5.

6.3.4 Dünaamilise skoopimise elemendid ja objektorienteeritus

Ka objektorienteeritud keeltes esineb selline dünaamilise skoopimisega seotud polümorfism. Näiteks Javas võime eelmistes jaotistes vaadeldud näite analoogina defineerida abstraktse klassi ja seda realiseeriva mitteabstraktse klassi:

```
abstract class Equal {
   abstract boolean equal(Equal other);

   boolean notequal(Equal other) {
      return !equal(other);
   }
}

class MyInt extends Equal {
   int x;
```

```
   public boolean equal(Equal other) {
      return x == ((MyInt) other).x;
   }
}
```

Siin abstraktse funktsiooni `equal` definitsioon sõltub konkreetsest klassi `Equal` alamklassist, kuid funktsioon `notequal` defineeritakse selle kaudu juba ülemklassis. Selle alamklassi funktsiooni `equal` realisatsioon antakse implitsiitse parameetrina (nn `this`-argumendi koosseisus) funktsioonile `notequal` selle väljakutsumisel. Alamklassi definitsioon ei pea olema ülemklassi defineerimisel kättesaadav, implitsiitse parameetri väärtus võetakse dünaamilisest skoobist.

6.4 Polümorfsed väärtused funktsiooni argumendina

6.4.1 Polümorfsed väärtused

Seni vaatlesime polümorfismi funktsioonide jaoks, mis võivad argumendiks võtta erinevat tüüpi väärtusi ning käituda nende korral kas ühtemoodi või erinevalt. Polümorfsed võivad aga lisaks funktsioonidele olla ka mittefunktsionaalsed väärtused, mis samuti võivad olla erinevate tüüpide korral kas samal või erineval sisemisel kujul. Neid väärtusi on vastavalt vajadusele võimalik kasutada erinevat tüüpi väärtustena.

Polümorfsete funktsioonide korral saab funktsiooni käitumise ja tulemuse tüübi määrata (monomorfse) argumendi tüübi põhjal. Polümorfsete väärtuste korral saab nende tüübi määrata selle (monomorfse) funktsiooni deklareeritud argumendi tüübi järgi, millele see väärtus argumendiks antakse.

Seega kui polümorfne on ainult funktsioon või ainult argument, kuid mitte mõlemad, siis ei teki probleeme argumendile monomorfse tüübi määramisega ja seejärel tulemuse tüübi määramisega, kuna see monomorfne tüüp on üheselt määratud (eeldusel, et vaadeldav polümorfne tüüp vaadeldava monomorfse tüübiga üldse kooskõlas on).

Probleem tekib siis, kui polümorfsed on nii funktsioon kui sellele argumendiks antud väärtus. Siis ei pruugi tulemuse tüüp enam üheselt määratud olla. Kui funktsioonile Fumontrixi tüüpi `forall A. Pair A Int -> A` anda argument tüüpi `forall B. Pair Bool B`, siis on tulemuse tüüp `Bool` veel üheselt määratud. Kui aga funktsioonile tüüpi `forall A. A -> List A` anda argument tüüpi `forall B. List B`, siis võib tulemus olla nii tüüpi `forall B. List (List B)` (kui väärtustatakse `A` = `List B`) kui ka `List (forall B. List B)` (kui väärtustatakse `A` = `forall B. List B`).

See probleem tekib sellisel juhul, kui funktsiooni tüübis on argumendi koha peal (kontravariantses positsioonis) ainult tüübimuutuja, mis on selle

funktsiooni tüübi tipmisel tasemel kvantoriga seotud. Seda tüübimuutujat on võimalik unifitseerida iga tüübiga ja seega juhul, kui argumendiks antud väärtuse tüüp sisaldab tipmisel tasemel kvantoreid, siis on võimalik üks kõik mitu nendest kvantoritest avada (kas kõik, mõned või mitte ühtegi) enne unifitseerimist. Avamata kvantorid lähevad tulemuse tüübi ette.

Kui funktsiooni tüübis argumendi koha peal on midagi muud kui ainult tüübimuutuja, siis seda probleemi ei teki. Kui seal on tipmisel tasemel konstruktor (või olemasolukvantor), siis tuleb igal juhul argumendiks antava väärtuse kõik tipmised üldisuskvantorid avada, et unifitseerimine võimalik oleks. Kui seal on tipmisel tasemel üldisuskvantor, siis tuleb argumendis antavas väärtuses avada nii palju kvantoreid, et avamata jäänud tipmiste kvantorite arv saaks võrdseks funktsiooni tüübis argumendi koha peal olevate tipmiste kvantorite arvuga.

6.4.2 Impredikatiivne polümorfism

Eelmises alajaotises vaadeldud probleemi vältimiseks on Haskell 98-s keelatud impredikatiivne polümorfism. Impredikatiivne polümorfism võimaldab kvantifitseeritud tüübimuutuja väärtustada polümorfse tüübiga. Seega Haskell 98-s väärtustatakse eelnevas näites `A = List B` ja kvantorid jäävad alati tipmisele tasemele.

Ka GHC-s ja Fumontrixis valitakse selles näites sama väärtustus, kuid siin on võimalik teatud juhtudel ka impredikatiivsust kasutada (GHC impredikatiivsuse kohta vt [2], jaotis 8.7.5). Defineerime näiteks GHC-s

```
f1 :: forall a. a -> [a]
f1 = undefined
v2 :: forall b. [b]
v2 = undefined
```

Siis `f1 v2` on tüüpi `forall b. [[b]]`, aga

```
(f1 :: (forall b. [b]) -> [forall b. [b]]) v2
```

annab tulemuse tüübiks `[forall b. [b]]`, kuna siin väärtustatakse `f1` tüübis muutuja `a` ilmutatult tüübiga `forall b. [b]`. Polümorfse väärtuse tüübis tüübimuutuja ilmutatud väärtustamine on GHC-s kohmakas, kuna selleks tuleb kogu väärtuse tüüp ümber kirjutada ja asendada seal muutuja kõik esinemised selle soovitava tüübiga. Selleks, et seda soovitavat tüüpi ei peaks mitu korda välja kirjutama, võiks defineerida tüübisünonüümi:

```
type T1 = forall b. [b]
```

ja siis kasutada

```
(f1 :: T1 -> [T1]) f2
```

Samas saab GHC-s tüübisünonüümi defineerida ainult peataseme skoobis ning, kui seda tüübimuutuja väärtustamist on vaja teha kuskil sügavas alamskoobis, siis jääb kood hakituks. Samuti on vaja selle polümorfse väärtuse tüübi struktuur ümber kirjutada ning see struktuur võib mõnikord olla väga keeruline.

Fumontrixis võime defineerida

```
f1 = bottom $: (forall A. A -> List A);
v2 = bottom $: (forall B. List B);
```

Siis `f1 v2` on tüüpi `forall B. List (List B)`, aga

```
f1 $: (forall B. List B) v2
```

on tüüpi `List (forall B. List B)`. Operaator `$:` avab tipmise taseme üldisuskvantori ja väärtustab sellega seotud muutuja antud tüübiga, seega erinevalt GHC-st ei pea Fumontrixis kogu väärtuse tüüpi ümber kirjutama, vaja on kirjutatada vaid see tüüp, millega muutuja väärtustatakse. Fumontrixis võime kasutada ka `typeof`-operaatorit ja kirjutada viimase näitega samaväärse

```
f1 $: (typeof v2) v2
```

Kui me ei soovi argumenti kaks korda välja kirjutada (see võib olla näiteks mingi keeruline avaldis), siis võime defineerida eraldi funktsiooni rakendamise operaatori tüübitaseme funktsioonina:

```
type applyImpred = \ f : @ . \ x : @ .
   value (type f) $: (typeof type x) (type x);
```

ja siis kasutada

```
type applyImpred (value f1) (value v2)
```

Operaator `applyImpred` on kasutatav sel juhul, kui funktsiooni tüübis argumendi kohal on tüübimuutuja, mis on seotud selle funktsiooni tüübis kõige tipmisel tasemel asuva üldisuskvantoriga.

Impredikatiivne polümorfism võib esineda ka sel juhul, kui tüübimuutuja ei esine funktsiooni tüübis otse argumendi kohal, vaid on seal näiteks mingi konstruktori parameetriks. Näiteks GHC-s

```
f3 :: forall a. [a] -> [a]
f3 = undefined
v4 :: [forall b. [b]]
v4 = undefined
```

Siis `f3 v4` on tüüpi `[forall b. [b]]`. Siin ei ole vaja ilmutatult tüübimuutujat väärtustada, kuna siin on unifitseerimine võimalik ainult ühel viisil. Analoogiliselt Fumontrixis

```
f3 = bottom $: (forall A. List A -> List A);
v4 = bottom $: (List (forall B. List B));
```

Siin `f3 v4` on tüüpi `List (forall B. List B)`.

Praegu vaatlesime parameetrilist polümorfismi. Olukord on analoogiline juhul, kui kvantifitseeritud muutujatel on tüübiklassikitsendused. Keerulisema polümorfismi korral (nagu saab Fumontrixi tüübitasemel realiseerida) tuleb juhul, kus polümorfsed on nii funktsioon kui argument, kasutatav argumendi tüüp (või tüübid, kui soovime rohkem polümorfismi säilitada) ilmutatult määrata.

6.4.3 Polümorfsed väärtused Fumontrixi tüübitasemel

Polümorfseid mittefunktsionaalseid väärtusi võib Fumontrixi tüübitasemel kujutada kui liiki `* -> @` funktsioone, mis seavad tüübile vastavusse seda tüüpi väärtuse. Siin saame kasutada kõiki Fumontrixi tüübitaseme programmeerimise võimalusi, sh lihtrekursiooni, seega on võimalik kujutada palju keerulisemaid polümorfseid väärtusi kui andmetasemel.

Samas nende tüübitaseme väärtuste kasutamine on ebamugavam kui andmetaseme polümorfsete väärtuste kasutamine. Kui me soovime neid anda argumendiks (potentsiaalselt polümorfsele) tüübitaseme funktsioonile liiki `@ -> @`, siis peame selle liiki `* -> @` väärtuse teisendama monomorfseks liiki `@` väärtuseks, määrates liiki `*` tüübiparameetri. Saadud monomorfse väärtuse saame anda argumendiks tüübitaseme funktsioonile. Rakendame näiteks tüübitaseme polümorfset funktsiooni `tf` tüübitaseme polümorfsele väärtusele `pv`, määrates viimase tüübiks `Int`:

```
type tf (pv Int)
```

Kuna siin polümorfne argument muudetakse enne (polümorfse) funktsiooni rakendamist ilmutatult monomorfseks, siis ei teki siin neid eelmises jaotises vaadelnud probleeme, mis tekivad, kui nii funktsioon kui argument on polümorfsed. Kui see tüübitaseme funktsioon on monomorfne, siis tundub see argumendi monomorfseks muutmine ülearune, aga monomorfne tüübitaseme funktsioon on alati võimalik andmetaseme funktsioonina realiseerida. Olgu näiteks tüübitaseme funktsiooni `tf` oodatava argumendi tüüp `Int`. Siis vastav andmetaseme funktsioon oleks

```
\ x : Int . type tf (value x)
```

Kui soovime tüübitaseme polümorfse funktsiooni rakendamisel tüübitaseme polümorfsele väärtusele saada tulemuseks polümorfset väärtust, siis võime rakendamisel määrata polümorfsele väärtusele ka rohkem kui ühe monomorfse tüübi ning jätta konkreetse monomorfse tüübi sõltuma polümorfse tulemuse tüübiparameetrist. Näiteks

```
\ t .
   case t of
      Bool   -> tf (pv Int);
      List a -> tf (pv (Maybe a));
   end;
```

Siin peaks siis `tf` täisarvulise argumendi korral tulemuseks andma tõeväärtuse ning `Maybe`-tüüpi argumendi korral listi.

Tüübitaseme polümorfsele väärtusele võime soovida rakendada ka andmetaseme funktsiooni. Kui see funktsioon on monomorfne, siis tõenäoliselt see ongi andmetasemel realiseeritud. Sellisel juhul saame funktsiooni argumendi tüübi leidmiseks kasutada jaotises 5.1.2 defineeritud tüübitaseme funktsiooni `argtype` ja ei pea seda tüüpi ilmutatult välja kirjutama. Andmetaseme monomorfse funktsiooni `f` rakendamine tüübitaseme polümorfsele väärtusele `pv` oleks siis

```
f (type pv (argtype (value f)))
```

Selle asemel võime defineerida ka tüübitaseme funktsiooni

```
type applyMtp = \ f : @ . \ pv : * -> @ .
   value f (type pv (argtype (value f)));
```

Seda saab siis kasutada funktsiooni rakendamise operaatorina juhul, kui funktsioon on andmetaseme monomorfne funktsioon ja argument on tüübitaseme polümorfne väärtus. Eelmine näide oleks siis

```
type (value f) `applyMtp` pv
```

Kui me soovime tüübitaseme polümorfsele väärtusele rakendada andmetaseme polümorfset funktsiooni, siis selleks on kaks võimalust. Esimene võimalus on muuta see funktsioon monomorfseks, määrates tüübiparameetri operaatoriga `$:` ja kasutades `applyMtp` operaatorit:

```
type applyMtp (value f $: t1) pv
```

Teine võimalus on muuta tüübitaseme polümorfne väärtus andmetaseme monomorfseks väärtuseks, määrates vastava monomorfse tüübi, ja seejärel anda funktsioonile argumendiks:

```
f (type pv t2)
```

See teine võimalus on lühem, v.a juhul, kui `t2` on oluliselt keerulisem tüüp kui `t1`. Need tüübid ei pruugi võrdsed olla, kuna `t2` argumendi monomorfne tüüp, kuid `t1` on tüübiparameeter, millest argumendi tüüp sõltub (neid tüübiparameetreid võib ka mitu olla, siis tuleb kasutada näiteks `f $: t1a $: t1b` jne).

6.5 Monaadilised väärtused ja polümorfism

6.5.1 Monaadilised väärtused ja funktsiooni rakendamine

Nagu Haskell, on ka Fumontrix puhas funktsionaalne keel ning kõrvalefektide ja erindite jaoks kasutatakse monaade. Monaadid võimaldavad kasutada imperatiivset programmeerimisstiili puhtalt funktsionaalses keeles.

GHC-s on olemas ST-monaad, milles saab programmeerida imperatiivses stiilis, kasutades näiteks viitasid ja massiive. Funktsiooni `runST` abil saab ST-monaadist ohutult väljuda.

Samas on GHC-s üsna ebamugav kasutada korraga monaadilisi ja tavalisi väärtusi. Näiteks SML-is saab kõrvalefekte sisaldavaid täisarvulisi väärtusi ja puhtaid täisarvulisi väärtusi anda argumendiks ühele ja samale (täisarvu ootavale) funktsioonile, kuna need väärtused on sama tüüpi. GHC-s on need väärtused erinevat tüüpi (`Integer` ja `M Integer`, kus `M` on mingi monaad) ja nende argumendiks andmiseks tüüpi `Integer` väärtust ootavale funktsioonile tuleb kasutada erinevaid funktsiooni rakendamise operaatoreid, vastavalt tavalist funktsiooni rakendamise operaatorit ning funktsiooni `liftM`.

Samuti tuleb kasutada erinevaid operaatoreid, kui funktsiooni tulemus või kogu funktsioon ise on monaadiline väärtus. Kui kasutada korraga maksimaalselt ühte monaadi, siis on kokku 16 erinevat kombinatsiooni funktsiooni ja argumendi tüüpide jaoks:

```
f               x       f x     GHC standardfunktsioonidega
-------------------------------------------------------------
a -> b          a       b       f x
a -> M b        a       M b     f x
M (a -> b)      a       M b     f `ap` return x
M (a -> M b)    a       M b     join (f `ap` return x)
a -> b          M a     M b     f `liftM` x
a -> M b        M a     M b     f =<< x
M (a -> b)      M a     M b     f `ap` x
M (a -> M b)    M a     M b     join (f `ap` x)
M a -> b        a       b       f (return x)
```

```
M a -> M b      a       M b     f (return x)
M (M a -> b)    a       M b     f `ap` return (return x)
M (M a -> M b)  a       M b     join (f `ap` return (return x))
M a -> b        M a     b       f x
M a -> M b      M a     M b     f x
M (M a -> b)    M a     M b     f `ap` return x
M (M a -> M b)  M a     M b     join (f `ap` return x)
```

Selles tabelis on kirjas, kuidas nendel juhtudel GHC standardfunktsioonide abil funktsiooni rakendada. Nende 16 juhu jaoks tuleb kokku kasutada 10 erinevat funktsiooni rakendamise operaatorit. Et nende kasutamine oleks vähem kohmakas, võime defineerida need 10 infiksoperaatorit (mõned on juba standardfunktsioonidena olemas), näiteks:

```
f               x       f x     infiksoperaator
-----------------------------------------------
a -> b          a       b       $
a -> M b        a       M b     $
M (a -> b)      a       M b     %=
M (a -> M b)    a       M b     =%<<=
a -> b          M a     M b     =%
a -> M b        M a     M b     =<<
M (a -> b)      M a     M b     %%
M (a -> M b)    M a     M b     =%<<
M a -> b        a       b       $=
M a -> M b      a       M b     $=
M (M a -> b)    a       M b     %==
M (M a -> M b)  a       M b     =%<<==
M a -> b        M a     b       $
M a -> M b      M a     M b     $
M (M a -> b)    M a     M b     %=
M (M a -> M b)  M a     M b     =%<<=
```

Lisaks tavalisele funktsiooni rakendamisele on vaja ka infiksoperaatoreid kasutada. Neid võib muidugi tavaliste funktsioonidena (prefikskujul) kasutada, kuid see on ebamugav. Seega on vaja veel vähemalt ühte operaatorit (kui infiksoperaatori tüüp võib ka monaadi sisaldada, siis on rohkem operaatoreid vaja):

```
($%) :: forall a b (m :: * -> *).
        (Monad m) => m a -> (a -> b) -> m b
($%) = flip liftM
```

Vaatleme nüüd järgmist imperatiivset algoritmi (järjestamise kiirmeetod ehk Quicksort):

```
void qs(int[] a, int l, int r) {
   if r-l <= 1 then
      return;
   else {
      void swap(int u, int v) {
         int t = a[u];
         a[u] = a[v];
         a[v] = t;
      }
      int x = a[l];
      int j = l+1;
      int i = j;
      while i < r {
         if a[i] < x {
            swap(i, j);
            j = j+1;
         }
         i += 1;
      }
      swap(j-1, l);
      qs(a, l, j-1);
      qs(a, j, r);
   }
}
```

Imperatiivne kiirmeetod sorteerib massiivi *in-place*, s.t kasutades ainult konstantset hulka lisamälu lisaks sorteeritavale massiivile. See on vajalik näiteks sellisel juhul, kui sorteeritav massiiv asub välisel andmekandjal (nt kõvakettal või võrgus) ning on nii suur, et mällu ei mahu. Funktsionaalselt realiseeritud kiirmeetodit ei saaks sellisel juhul kasutada, kuna seal on vaja pidevalt uusi liste moodustada, mille maht rekursiooni ülemisel tasemel on sorteeritava massiiviga samas suurusjärgus.

Haskellis võib selle algoritmi eespool defineeritud infiksoperaatorite abil kirjutada nii:

```
qs :: STArray s Int Int -> Int -> Int -> ST s ()
qs a l r =
   if r-l <= 1 then
      return ()
```

```
   else do
      let al = writeArray a
      let ar = readArray a
      let
         swap u v = do
            t <- ar u
            al u =<< ar v
            al v $ t
      x <- ar l
      (jl,jr) <- makeSTRef (l+1)
      (il,ir) <- makeSTRef =<< jr
      while (ir $% (<) %= r) $ do
         when =% ((ar =<< ir) $% (<) %= x) =%<<= do
            swap =% ir =%<< jr
            jl =<< jr $% (+) %= 1
         il =<< (+1) =% ir
      swap =% (jr $% (-) %= 1) =%<<= l
      qs a l =<< (jr $% (-) %= 1)
      qs a =% jr =%<<= r
```

Siin `makeSTRef` tekitavad uue viida ning tagastab selle viida vasaku ja parema semantika. Analoogiliselt saaks massiivi defineerimiseks kasutada funktsiooni `makeSTArray`, mis tagastab selle massiivi vasaku ja parema semantika. Siin on massiiv `a` juba eelnevalt konstrueeritud, funktsioonis defineeritakse vaid selle vasak ja parem semantika (`al` ja `ar`). Funktsioon `while` on intuitiivse semantikaga. Näeme, et antud koodis on vaja funktsiooni rakendamise jaoks kasutada 7 erinevat operaatorit (`$% %= $ =% =%<< =<< =%<<=`). See on väga ebamugav, programmeerija peab pidevalt mõtlema, millist operaatorit kasutada, ning infiksoperaatorite prioriteedid lähevad kaduma. Samuti on kood halvasti loetav (kuigi seda annab parandada, kui lasta tekstiredaktoril süntaks värvida nii, et need operaatorid oleks raskesti nähtavad, näiteks tumehall mustal taustal).

Fumontrixis on tavaline funktsiooni rakendamise operaator (`$` või järjest kirjutamine) üle laaditud ja kasutatav kõigil 16 juhul (eeldusel, et `ST`-konstruktori monaadina käsitlemine on sisse lülitatud). Seega viimase näite võime kirjutada järgmiselt:

```
rec qs : forall S. stArray S Int -> Int -> Int -> ST S Unit =
   forall S. \ a @ Pair al ar : stArray S Int .
             \ l : Int . \ r : Int .
      mif $: (ST S) (r-l <= 1)
         Unit
```

```
do
   . swap = \ u : Int . \ v : Int . do
      . t <- ar u;
      al u $ ar v;
      al v $ t;
   end;
   . x <- ar l;
   . Pair jl jr <- makeSTRef $: S (l+1);
   . Pair il ir <- makeSTRef $: S jr;
   while (ir < r) do
      when (ar ir < x) do
         swap ir jr;
         jl $ jr+1;
      end;
      il $ (+1) ir;
   end;
   swap (jr-1) l;
   qs a l (jr-1);
   qs a jr r;
end;
```

Siin on funktsiooni rakendamise operaatori korral sisuliselt tegemist *ad-hoc*-polümorfismiga ja seda saaks ka tüübitaseme funktsioonidega realiseerida, kuid siis tuleks iga monaadikonstruktori jaoks operaatori käitumine eraldi defineerida, kuna Fumontrix ei toeta geneerilist programmeerimist.

6.5.2 `do`-notatsioon

Haskellis on imperatiivse programmeerimisstiili mugavamaks kasutamiseks olemas `do`-notatsioon ([1], jaotis 3.14), mis on defineeritud monaadioperatsioonide `>>=` ja `fail` kaudu. Kuna Fumontrixis on funktsiooni rakendamise operaator eelmises jaotises kirjeldatud viisil üle laaditud, siis on võimalik `do`-avaldised defineerida funktsiooni rakendamise operaatori, mitte otse monaadioperatsioonide kaudu.

Fumontrixis on `do`-avaldis defineeritud süntaktilise suhkruna järgmiselt:

```
do in e                ≡ e
do e1; st in e         ≡ let _e = e1 in
                             (\ Unit : Unit . do st in e) _e
do . p = e1; st in e   ≡ let _e = e1 in
                             (\ p : typeof _e . do st in e) _e
do . p <- e1; st in e  ≡ let _e = e1 in
                             (\ p : basetypeof _e . do st in e) _e
do st end              ≡ do st in Unit
```

Siin *st* tähendab suvalist (tühja või mittetühja) `do`-notatsiooni lausete jada. Operaator `basetypeof` sarnaneb operaatoriga `typeof`, kuid avaldise tüübist eemaldatakse kõik tipmisel tasemel olevad monaadid. Seega kui avaldis on monaadilist tüüpi (täpselt ühe monaadiga), siis kasutatakse funktsiooni rakendamise operaatori `$` asemel operaatorit `=<<`.

Näeme, et erinevalt Haskellist on `do`-avaldisel kaks erinevat kuju — `do` *st* `in` *e* ja `do` *st* `end`. Esimese variandi korral tagastatakse `do`-avaldise tulemusena avaldise *e* väärtus. Siin on tagastatav väärtus ülejäänud lausetest eraldatud (ning ülelaadimise tõttu ei ole seal `return` vaja kasutada), erinevalt Haskellist, kus tagastatav väärtus määratakse `do`-konstruktsiooni viimase lausega, mille ette tuleb enamasti `return` kirjutada. Teise variandi korral on `do`-avaldise tulemuseks ühiktüübi väärtus, mitte viimase lause väärtus. Seega Fumontrixis on tagastusväärtus selgelt ülejäänud lausetest eraldatud, mis muudab koodi lugemise lihtsamaks.

Erinevalt Haskellist ei ole Fumontrixis monaadidel funktsiooni `fail`. Kui `do`-avaldises näidisesobitamine ebaõnnestub, siis on tulemuseks $\bot$. Samas ei ole see automaatne funktsiooni `fail` väljakutsumine hädavajalik, kuna selle sobitumise kontrolli ja ebaõnnestumise korral mingi funktsiooni väljakutsumise saab ka käsitsi teha, nagu järgnevas näites (vt ka [1], jaotis 3.14):

```
do
   st1;
   . e <- f 10;
in
   case e of
      Just x -> do st2 in expr;
      _      -> fail;
   end;
```

See vastaks siis järgnevale koodile, kui automaatne `fail` väljakutsumine oleks olemas:

```
do
   st1;
```

```
   . Just x <- f 10;
   st2;
in
   expr
```

6.5.3 Mitme monaadi korraga kasutamine

Seni vaatlesime põhiliselt juhtu, kus korraga on kasutusel maksimaalselt üks monaad. Mõnikord on vaja korraga mitut monaadi kasutada. Olgu
`f :` M_1 `(`M_2 `a ->` M_3 `b)` ja
`x :` M_4 `(`M_2 `a)`
Siis peaks funktsiooni rakendamisel olema
`f x :` M_1 `(`M_4 `(`M_3 `b))` Siin M_1, M_2, M_3 ja M_4 on 0 või enama monaadi kompositsioonid (need kompositsioonid ei pruugi ise olla monaadid).

Fumontrixis ongi võimalik selliselt kasutada korraga mitut monaadi. Iga funktsiooni rakendamise korral koostatakse kasutatavate monaadide monaadifunktsioonide põhjal sobiv funktsiooni rakendamise operaator ja rakendatakse seda. Siin on erinevaid funktsiooni rakendamise operaatoreid palju rohkem kui 10, mis oli ühe monaadi korral, aga programmeerija ei pea nende pärast muretsema, kuna tavaline funktsiooni rakendamise operaator töötab kõigil neil juhtudel.

Kui $M_1 = M_4 = M_3 =$ `M` ehk kasutusel on ainult üks monaad, siis oleks `f x : M (M (M b))`. Eelnevalt vaadeldud ühe monaadiga juhu korral oli tulemuseks `f x : M b`. Fumontrixis koondatakse sellisel juhul (monaadifunktsiooni `join` abil) kõrvutisattuvad võrdsed monaadikonstruktorid. Esialgsete kompositsioonide M_1, M_4, M_3 seest kõrvuti olnud võrdseid monaadikonstruktoreid ei koondata.

Kui funktsiooni rakendamise operaator on selliselt erinevate monaadide kasutamiseks üle laetud, siis saab `do`-avaldises vabalt kasutada vaheldumisi erinevaid monaade. Sellisel juhul võib lõpuks `do`-avaldise tüüp olla midagi sellist:
`M1 (M2 (M3 (M1 (M2 (M3 Int))))).`
Siin `M1`, `M2`, `M3` on üksikud monaadid, mitte kompositsioonid.

Sellist tüüpi väärtusest täisarvu kättesaamiseks võiks kirjutada `run` funktsiooni, mis töötab kõigi väärtuste jaoks, mille tüüp on kujul
m_1 `(`m_2 `...` `(`m_n `a)...)`,
ning annab välja tüüpi `M a` väärtuse, kus `M` on mingi monaad. Siin monaadid $m_1, m_2, ..., m_n$ on mingi tüübiklassi esindajad.

Selle jaoks on vaja, et iga monaadi m_i jaoks oleks olemas teisendaja tüüpi m_i `a -> M a`. Muidugi oleks võimalik see teisendaja igal monaadi kasutamisel ilmutatult välja kutsuda, kuid see läheks kohmakaks. Mugavam oleks

välja kutsuda `run` funktsioon ühe korra terve `do`-avaldise jaoks.

GHC-s saab teha midagi sellist:

```
class ListRunnable a b | a -> b where
   runList :: a -> [b]

newtype Stop a = Stop a

maybeToList :: Maybe a -> [a]
maybeToList (Just x) = [x]
maybeToList Nothing  = []

instance ListRunnable (Stop a) a where
   runList (Stop x) = [x]
instance ListRunnable a b => ListRunnable [a] b where
   runList xs = concat (map runList xs)
instance ListRunnable a b => ListRunnable (Maybe a) b where
   runList xs = concat (map runList (maybeToList xs))

(>=>) :: forall a1 r (m :: * -> *).
         (Monad m) => m a1 -> (a1 -> r) -> m r
(>=>) = flip liftM

lr =
   [1,2,3]  >=> \ x ->
   Just 700 >=> \ y ->
   [10,20]  >=> \ z ->
   Stop (x + y + z)
```

Siin kasutame korraga kahte monaadi: `[]` ja `Maybe`. `lr` on tüüpi

```
forall t. (Num t) => [Maybe [Stop t]]
```

ehk

```
[Maybe [Stop Integer]]
```

kui arvutüübid monomorfsed oleks. Siis

```
runList lr :: [Integer]
```

Kuna GHC ei luba tüübitaseme funktsioonides tulemuse tüübi defineerimisel vaikevariante kasutada (vt jaotist 5.3.5), siis on siin vaja konstruktor `Stop` sisse tuua.

Fumontrixis saame sama näite realiseerida tüübitaseme funktsioonidega:

```
type runList = \ v : @ .
   value
      (type
         (\ t rec @ .
            case t of
               List a  ->
                  value \ xs : t .
                     concat (map (type rec a) xs);
               Maybe a ->
                  value \ mx : t .
                     concat (map (type rec a) (maybeToList mx));
               _ ->
                  value \ x : t .
                     x :: Nil;
            end
         ) (typeof type v)
      ) (type v);

newmonad List;
newmonad Maybe;
lr = do
   . x <- 1 :: 2 :: 3 :: Nil;
   . y <- Just 700;
   . z <- 10 :: 20 :: Nil;
in
   x + y + z;

main = type runList (value lr)
```

Siin ei ole erinevalt GHC-st vaja `Stop`-konstruktorit sisse tuua ning kasutada saab `do`-notatsiooni. `lr` on tüüpi

```
List (Maybe (List Int))
```

ning `type runList (value lr)` on tüüpi `List Int`.

6.5.4 Monaadiliste väärtuste polümorfismi piiramine

Mõnikord võib selles peatükis vaadeldav monaadiliste väärtuste polümorfism ootamatuid tulemusi anda, kui me tegelikult ei plaaninud seda kasutada. Näiteks avaldise

```
Just (3 :: 4 :: Nil)
```

väärtuse tüüp ilma monaadiliste väärtuste polümorfismita on `Maybe (List Int)`, aga polümorfismiga on hoopis `List (Maybe Int)`. See tuleb sellest, et `Just` on tüüpi `forall A. A -> Maybe A`, kuid talle antakse argumendiks väärtus tüüpi `List Int`. Nagu kvantorite avamisel, avatakse argumendiks antud väärtuse tüübis enne unifitseerimist võimalikult palju tipmisel tasemel olevaid monaade. Seega ei väärtustata kvantori avamisel `A = List Int`, vaid `A = Int` ning funktsiooni rakendamisel kasutatakse operaatorit `fmap` (listimonaadi oma), mitte tavalist funktsiooni rakendamist.

Et selliseid üllatusi vältida, on Fumontrixis võimalik määrata, milliseid konstruktoreid käsitletakse mingis skoobis monaadidena (monaadiliste väärtuste polümorfismi mõttes). Alguses ei käsitleta ühtegi konstruktorit monaadina, isegi kui see on `monad` klassi esindajaks defineeritud. Deklaratsioon

```
newmonad M;
```

muudab monaadikonstruktori `M` antud alamskoobis monaadina käsitletavaks. Kui mingis alamskoobis ei soovita seda enam monaadina käsitleda, siis võib anda deklaratsiooni

```
unmonad M;
```

Kui mingis alamskoobis ei soovita ühtegi konstruktorit monaadina käsitleda, siis võib anda deklaratsiooni

```
nomonads;
```

selle asemel, et iga monaadi jaoks `unmonad`-deklaratsioon anda. Alamskoopides on siis võimalik vajaduse korral jälle mõni `newmonad`-deklaratsioon anda.

`newmonad`- ja `unmonad`-deklaratsioone võib kasutada ka keerulisemat liiki kui `* -> *` monaadikonstruktorite korral, näiteks

```
newmonad ST;
```

(`ST` on liiki `* -> * -> *`). Sellisel juhul käib deklaratsioon kõigi monaadide kohta, mis on antud konstruktoriga konstrueeritavad. Praegusel juhul siis `ST a` kohta iga tüübi `a` jaoks.

7 Semantika kirjeldamisest

7.1 Fumontrixi interpretaatori realiseerimisest

Fumontrixi interpretaator on realiseeritud Haskellis. Süntaksi parseri genereerimiseks kasutatakse Happy't [6], mis on Yacc'i analoog, kuid genereerib C koodi asemel Haskelli koodi. Lekser on realiseeritud käsitsi Haskellis.

Interpretaatoris on süntaktilistele konstruktsioonidele üritatud anda semantika funktsionaalsel kujul, kasutades ainult puhtalt funktsionaalseid andmestruktuure (sh monaade, kuid mitte sisendit-väljundit). Seega on interpretaatori koodist võimalik välja lugeda teatud liiki denotatsiooniline semantika. Et semantika jääks võimalikult lihtsaks, ei ole optimiseerimisele eriti rõhku pandud ning see interpretaator on seetõttu aeglasem ja mälunõudlikum (ebaefektiivsem) kui traditsioonilised interpretaatorid.

Samuti ei ole selles interpretaatoris rõhku pandud veateadete (nt tüübivigade) informatiivsusele, kuna see jääb käesoleva töö skoobist välja. Seetõttu on koodi mittetundval kasutajal veateatest aru saamine raskendatud.

7.2 Interpretaatori koodi struktuur

Lekser on realiseeritud failis `Lexer.hs`. Lekser teisendab stringikujul oleva lähtekoodi lekseemide listiks. Lekseemid on defineeritud failis `Syntax.hs`.

Parseri kirjeldus on antud failis `Parser.y`, millest genereeritakse Happy abil fail `Parser.hs`. See viimane on ka interpretaatori lähtekoodi juurde alles jäetud, et interpretaatori kompileerimiseks ei oleks vaja Happy't installida. Parser teisendab lekseemide listi abstraktse süntaksi kujule. Abstraktne süntaks on defineeritud failis `AbsSyntax.hs`.

Failis `Types.hs` on defineeritud mõned tüübisünonüümid, mida mujal kasutatakse.

Failis `SemCats.hs` on defineeritud semantilised kategooriad (tüübid, väärtused jne) ning kontekst ja keskkond. Failis `SemHelpers.hs` on mõningad abifunktsioonid semantika andmiseks (kvantorite lisamine ja avamine, muutujate asendamine tüübiavaldises jne). Failis `SemUnify.hs` on defineeritud unifitseerimise ja monaadiliste väärtuste polümorfismiga seotud funktsioonid, mida kasutatakse andmetaseme funktsiooni rakendamise operaatori semantika andmisel.

Failis `Denot.hs` on põhiline denotatsioonilise semantika kirjeldus. Seal antakse nii staatiline kui dünaamiline semantika erinevate süntaktiliste kategooriate jaoks, samuti lihtrekursiooni ja tüübiklasside semantika.

Failis `SemInitCtx.hs` defineeritakse Fumontrixi programmi algkontekst ja -keskkond (sisseehitatud tüübid ja operatsioonid) ning kogu programmi

semantika. Failis `Interpret.hs` defineeritakse interpretaator ise IO-monaadilise tegevusena. IO-monaadi kasutataksegi ainult siin ning Fumontrixi ST-monaadi realiseerimiseks.

7.3 Fumontrixi semantika kirjeldamisest

7.3.1 Semantika tasemed

Fumontrixis antakse semantika kolmel erineval tasemel: liigitase, tüübitase ja andmetase. Neist esimesed kaks moodustavad staatilise semantika, kuna need arvutatakse välja tüübikontrolli käigus, see arvutuste käik on igal programmi käivitamisel sama (kui kood ei ole muutunud) ning nende arvutuste termineerumine on garanteeritud.

Andmetaseme semantika moodustab dünaamilise semantika, mille arvutamine ei pruugi termineeruda. Siin võiks ka arvutuse käik erinevatel programmi käivitamistel olla erinev, kui programm saaks teha sisendi-väljundi operatsioone. Fumontrixis siiski sisendit-väljundit realiseeritud ei ole (v.a ST-monaadi viidad ja massiivid, kuid neid saab kasutada ainult muust maailmast eraldatud lõimedes, mille algseisund on fikseeritud, seega on arvutused seal deterministlikud).

Failis `Denot.hs` antakse liigitaseme semantika funktsioonidega, mille nimi lõpeb sufiksiga `Kem`, tüübitaseme semantika sufiksiga `Zem` ning andmetaseme semantika sufiksiga `Sem`. Näiteks avaldiste semantika antakse funktsioonidega `exprKem`, `exprZem` ja `exprSem`.

7.3.2 Seosed semantika tasemete vahel

Kuigi Fumontrixi semantika antakse kolmel erinevale tasemel, on need tasemed mingil määral omavahel seotud. Näiteks tüübi(konstruktori) rakendamisel universaalselt kvantifitseeritud tüüpi väärtusele (polümorfse väärtuse spetsialiseerimisel operaatori `$:` abil) määratakse rakendatava tüübikonstruktori liik liigitaseme semantikaga. See liik peab ühtima polümorfse tüübi kvantoriga seotud muutuja liigiga, kuid viimane selgub alles tüübitaseme semantikas, kvantifitseeritud tüüp ise on liiki `*` ja liigitasemel selle kohta rohkem infot kätte ei saa. Näiteks

```
(type tf Int) $: List
```

kus `tf` on tüübitaseme funktsioon liiki `* -> @`. Siis `type tf Int` on väärtus ning rakendatav tüüp `List` on liiki `* -> *`. Liigitasemel väärtuse tüübi kohta mingit infot teada ei ole. Tüübitaseme funktsioon `tf` võib tagastada nii `forall A. A` kui `forall M : * -> * . M Int` tüüpi väärtuse, olenevalt

argumendist. Kuna liigitasemel tüübitaseme funktsioone ei väärtustata (seda tehakse tüübitasemel), siis ei ole liigitasemel võimalik neil juhtudel vahet teha.

Seega toimub liikide ühtimise kontroll antud juhul tüübitasemel ja selleks on vaja edastada liigitaseme semantikas välja arvutatud rakendatava tüübikonstruktori liik (antud juhul `List : * -> *`) tüübitasemele. Alternatiiv oleks see info tüübitasemel uuesti välja arvutada. See tähendaks korduvat sama asja uuesti väljaarvutamist. Näiteks kui see `$:` rakendus asub kuskil tüübitaseme funktsiooni sees ning seda tüübitaseme funktsiooni kutsutakse korduvalt välja, siis oleks (selle `$:` rakenduse) liigitaseme semantika igal väljakutsel sama, kuid tüübitaseme semantika võib olla iga kord erinev, sõltudes selle tüübitaseme funktsiooni argumendist. Sellepärast ongi liigitase tüübitasemest eraldatud.

Seega on vaja edastada infot liigitasemelt tüübitasemele ja Fumontrixi interpretaatoris kasutatakse selleks `KemInfo` tüüpi väärtusi. Mingile süntaktilisele objektile liigitaseme semantika arvutamisel leitakse lisaks `KemInfo` väärtus, mis sisaldab tüübitasemele edastatavat infot nii selle süntaktilise objekti kui selle alamobjektide jaoks. See võimaldab arvutada välja liigitaseme semantika kogu programmi jaoks ning saada kohe kätte tüübitasemele edastatav info kõigi selle alamobjektide jaoks. See `KemInfo` väärtus antakse siis tüübitaseme semantika arvutamisel parameetriks. Süntaktiliste objektide tüübitaseme semantika võib sõltuda alamobjektide semantikast, mille arvutamiseks vajalik `KemInfo` väärtus sisaldub parameetrina kõrgema taseme objekti `KemInfo` väärtuses.

Seega info liigub lihtsalt liigitasemelt tüübitasemele. Natuke ebamugav on küll see, et `KemInfo` väärtusi tuleb töödelda ka nende süntaktiliste objektide juures, mis ise seda infot ei kasuta, kuid mille alamobjektid kasutavad seda. Näiteks tavalise tüübitaseme funktsiooni rakendamise jaoks tuleb liigitasemel funktsiooni ja argumendi `KemInfo` ühendada (konstruktori `KIList` abil) ning tüübitasemel jälle eraldada ja anda alamobjekti semantikate arvutamisel parameetriks.

Lisaks liigitaseme ja tüübitaseme seotusele on seotud ka tüübitase ja andmetase (näiteks andmetaseme funktsiooni rakendamise korral määratakse andmetasemel rakendamiseks kasutatav monomorfne operaator tüübitasemel, kuna see sõltub (ainult) funktsiooni ja argumendi tüüpidest, andmetasemel on vaja ainult funktsiooni ja argumendi väärtused sellele operaatorile argumendiks anda). Seal kasutatakse analoogilist meetodit info edastamiseks ühelt tasemelt teisele. `KemInfo` asemel kasutatakse siin `StatSemInfo` tüüpi väärtusi.

7.3.3 Monaadid semantika esitamisel

Antav semantika vastab enam-vähem denotatsioonilisele semantikale, kuna konstruktsioonide semantika antakse puhtalt funktsionaalsel kujul alamkonstruktsioonide (komponentide) kaudu (kompositsiooniliselt). Tüübikontrolli ja liigikontrolli jaoks kasutatakse siiski (puhtalt funktsionaalselt defineeritud) monaade. Tüübikontrolli monaad `TC` võimaldab hoida kas tavalist väärtust või erindit (tüübiviga). Sellisel juhul ei ole vaja igas kohas, kus võib tüübiviga tekkida, seda kontrollida — kui kuskil tekib tüübiviga, siis on see lõpptulemuses näha.

Liigikontrolli monaad `KC` võimaldab lisaks erindile (liigiveale) kasutada ka lihtsat seisundit. Seda kasutatakse unikaalsete identifikaatorite genereerimiseks, et nummerdada tüübitaseme väärtuste kontekste. See on vajalik leksilise skoopimise säilitamiseks, kuna tüübitaseme väärtust võidakse kasutada ka mingis muus skoobis kui see, milles see defineeriti.

7.3.4 Andmestruktuurid semantika esitamisel

Semantika andmisel kasutatakse konteksti (staatilises semantikas) ja keskkonda (dünaamilises semantikas). Kumbki neist koosneb Haskelli `Map`-tüüpi andmestruktuuridest. `Map` on andmestruktuur, mis seab mõnedele teatud tüüpi väärtustele (see tüüp peab olema `Ord`-klassi esindaja) vastavusse maksimaalselt ühe mingit teist tüüpi väärtuse.

See andmestruktuur on Haskellis realiseeritud funktsionaalselt, seega olemasolevat andmestruktuuri ei saa muuta. Kui on vaja seal andmestruktuuris muudatusi teha (kirjeid lisada, kustutada, muuta), siis luuakse uus andmestruktuur, mis erineb vanast nende muudatuste võrra, kuid vana andmestruktuur jääb ka alles ning viidad sellele jäävad kehtima (kui ükski viit enam sinna ei viita, siis võib prügikoristus selle minema visata). Kuna vanal ja uuel andmestruktuuril on siiski üsna palju ühist, siis ei pea iga muudatuse jaoks täielikku koopiat tegema, vaid andmeid võib jagada erinevate versioonide vahel. Seetõttu see funktsionaalne andmestruktuur ei ole väga palju aeglasem ja mälunõudlikum kui vastavad imperatiivsed andmestruktuurid.

7.4 Lihtsamate konstruktsioonide semantikast

7.4.1 Avaldised

Enamiku avaldisekonstruktsioonide semantika on üsna lihtne ja standardne. Avaldise semantika määrab selle tüübi ja väärtuse, mis sõltuvad kontekstist ja keskkonnast.

Muutujate jaoks on kontekstis kujutus `zVar`, mis seab muutuja nimele vastavusse selle tüübi, ning keskkonnas kujutus `eVar`, mis seab muutuja nimele vastavusse selle väärtuse.

Tüübikonstruktorite jaoks on kontekstis kujutus `zCon`, mis seab konstruktori nimele vastavusse selle tüübi avaldisena kasutamisel (väli `zConExpr` kujutuse tulemuseks olevas kirjes), selle semantika näidisena kasutamisel (väli `zConPatt`) ning selle aarsuse (`zConArity`). Konstruktori kasutamisel avaldisena läheb ainult esimest komponenti vaja, ülejäänud on vajalikud konstruktornäidise semantika jaoks. Keskkonnas on kujutus `eCon`, mis seab konstruktori nimele vastavusse selle väärtuse avaldisena kasutamisel (`eConExpr`) ning selle semantika näidisena kasutamisel (`eConPatt`).

`exists`-avaldise `exists` C `:` k `.` $t[C]$ andmetaseme semantikaks on nn pakkimisfunktsioon:

```
Exists _pi _astr _ke _ua ->
   VForall $ \ vpi -> VFun $ \ va -> VExists vpi va
```

See on polümorfne funktsioon, mis võtab argumendiks monomorfset tüüpi $t[C]$ väärtuse `va` ning pakib selle koos tüübiparameetrile C vastava tüübiklassi c eksemplariga (või ühiktüübi väärtusega parameetrilise polümorfismi korral) `vpi` eksistentsiaalseks väärtuseks `VExists vpi va`.

`forall`-avaldise tüübitaseme semantikas

```
Forall pi astr ke ea -> do
   let KIType cid kie = ki
   zpi <- polymorphismInfoZem pi ctx
   let k = kindExprKem ke
   let (aid, ctx') = addTypeVar astr k ctx
   ctx2 <- addInstance zpi aid cid $ ctx'
   (te,ssie) <- exprZem ea kie ctx2
   return (makeTForall zpi aid k te, SSIForall cid ssie)
```

luuakse funktsiooniga `addTypeVar` uus tüübikonstruktor, millele eraldatakse uus unikaalne identifikaator `aid`, mida veel kontekstis kasutusel ei ole. Funktsiooniga `addInstance` muudetakse see uus tüüp (konstruktor) tüübiklassi esindajaks (kui kvantifitseeritud muutuja oli seotud tüübiklassiga). Eksemplari väärtus lisatakse keskkonda andmetaseme semantikas funktsiooniga `addVInstance`. Lähemalt vaatleme seda jaotises 7.6.4.

Funktsiooni rakendamise semantikat vaatleme lähemalt jaotises 7.5. `type`-avaldise semantikat vaatleme jaotises 7.6.2.

7.4.2 Tüübiavaldised

Enamiku tüübiavaldisekonstruktsioonide semantika on samuti üsna lihtne, v.a tüübitaseme programmeerimisega seotud konstruktsioonid, mille semantikat vaatleme jaotises 7.6. Tüübiavaldise semantika määrab selle tüübitaseme objekti, mida see avaldis kujutab.

Tüübimuutujate (sünonüümide) ja tüübikonstruktorite liigitaseme semantika jaoks on kontekstis kujutus `zTVarKind`, mis seab tüübimuutuja või -konstruktori nimele vastavusse selle liigi. See on ka ainus osa kontekstist, mida liigitaseme semantika andmisel kasutatakse.

Tüübimuutujate tüübitaseme semantika jaoks on kontekstis kujutus `zTVar`, mis seab muutuja nimele vastavusse tüübitaseme objekti (tüübi, tüübikonstruktori, tüübitaseme funktsiooni või tüübitaseme väärtuse).

Tüübikonstruktorite tüübitaseme semantika jaoks on kontekstis kujutus `zTCon`, mis seab konstruktori nimele vastavusse tüübitaseme objekti (antud juhul tüübikonstruktori).

7.4.3 Näidised

Näidise semantika määrab, kuidas väärtuses sisalduvat infot antud näidise poolt keskkonda (ja selle staatilist infot keskkonda) salvestatakse, s.t näidise semantika seab väärtusele ja selle tüübile vastavusse vastavalt keskkonna- ja kontekstiteisenduse. Dünaamilist semantikat määrava funktsiooni tüüp on

```
pattSem :: Patt -> Value -> StatSemInfo -> Environment
           -> Maybe Environment
```

Siin kasutatakse `Maybe`-tüüpi, kuna näidisesobitamine võib (mõnes keskkonnas) ebaõnnestuda ning sellisel juhul on tulemuseks `Nothing`.

Muutujanäidise korral lihtsalt seotakse väärtus vastava muutujanimega. Konstruktornäidise korral kontrollitakse, et väärtus oleks moodustatud õige konstruktoriga ning rakendatakse konstruktori argumentväärtustele alamnäidiste semantikat, mille ühendamiseks kasutatakse `Maybe`-monaadis `foldM`-funktsiooni, seega kui mõne alamnäidise sobitamine ebaõnnestus, siis on kogu tulemuseks `Nothing`.

`exists`-näidise korral tuuakse sisse uus tüübikonstruktor, mis vajadusel muudetakse tüübiklassi esindajaks (kasutades eksistentsiaalse väärtuse sisse pakitud eksemplari), sarnaselt `forall`-avaldise semantikaga. Eksistentsiaalse väärtuse sees olevale tavalisele väärtusele rakendatakse alamnäidist.

Ülejäänud andmetaseme näidisekonstruktsioonid on lihtsa semantikaga.

Analoogiliselt andmetaseme näidistega määrab tüübinäidise semantika, kuidas tüübitaseme objektis sisalduvat infot konteksti salvestada. Eraldi tüü-

binäidisekonstruktsioonid on olemas ainult tüübikonstruktorite, funktsioonitüüpide ja täisarvutüübi jaoks, tipmiste kvantoritega tüübid sobituvad ainult muutujanäidise ja ignoreerimisnäidisega. Tüübitaseme funktsioone ja väärtusi ei saa üldse tüübinäidistega kasutada. Tüübinäidiseid saab kasutada ainult tüübitaseme `case`-konstruktsioonis, mitte tüübitaseme λ-abstraktsioonis ega `let`-avaldises.

7.4.4 Algebralised andmetüübid

Algebraliste andmetüüpide semantika antakse kahes osas. Funktsioonidega `dataConKem`, `dataConZem`, `dataConSem` antakse andmekonstruktorite (algebralise andmetüübi variantide) semantika. Funktsioonidega `dataTypeKem`, `dataTypeZem`, `dataTypeSem` antakse tüübikonstruktori semantika.

Algebralise andmetüübi tüübikonstruktori staatiline semantika lisab selle uue tüübikonstruktori konteksti. Lisaks defineeritakse kolm funktsiooni: `constr`, mis on selle konstruktori semantika tüübiavaldises kasutamisel; `destr`, mis kontrollib, et tüüp oleks konstrueeritud just vaadeldava konstruktoriga ja tagastab selle konstruktori argumendid; `addVarsToContext`, mis lisab algebralise andmetüübi definitsioonis määratud tüübikonstruktori parameetrid uute konstruktoritena konteksti.

Neid kolme funktsiooni kasutatakse selle algebralise andmetüübi andmekonstruktorite staatilise semantika defineerimisel. Iga andmekonstruktori semantikas lisatakse `addVarsToContext` abil tüübiparameetrid konteksti ning leitakse andmekonstruktori argumendiks antud tüübiavaldisele vastavad tüübid antud kontekstis. Need võivad sisaldada tüübiparameetreid. Et leida konstruktori tüüpi avaldises kasutamisel (`constrType`), moodustatakse nende argumenditüüpide ja oodatava tulemustüübi põhjal *curried*-kujul funktsioonitüüp ning seotakse (kõik algebralise andmetüübi deklaratsioonis esinenud) üldisuskvantoritega. Samuti leitakse konstruktori näidisena kasutamise semantika (`destr`) ja aarsus (`arity`).

Andmekonstruktori dünaamilises semantikas määratakse analoogilise avaldisena ja näidisena kasutamise semantikad (vastavalt `sv` ja `sp`).

7.4.5 Deklaratsioonid

Deklaratsiooni semantikaks on konteksti- ja keskkonnateisendus.

Mitterekursiivse `let`-deklaratsiooni semantika moodustatakse standardselt näidise ja avaldise semantika põhjal, lisades avaldise poolt määratud väärtuse (või selle osad) näidise abil keskkonda ning selle väärtuse tüübi kohta käiva info (eksistentsiaalsete näidiste korral) konteksti.

Rekursiivse `rec`-deklaratsiooni korral on annotatsioonina antud ka rekursiivse funktsiooni tüüp, mis lisatakse enne seda funktsiooni kirjeldava avaldise tüübi leidmist konteksti. Dünaamilises semantikas kasutatakse püsipunktioperaatorit `fix`, et lisada defineeritava funktsiooni väärtuse sellesse samasse keskkonda, milles see funktsioon väärtustatakse.

`data`-deklaratsiooni korral leitakse algebralise andmetüübi tüübikonstruktori semantika ning selle abil ka andmekonstruktorite semantikad ning lisatakse need konteksti ja keskkonda.

`newmonad-`, `unmonad-` ja `nomonads`-deklaratsioonid muudavad kontekstis olevat kujutust `zMonad`, mis seab tüübikonstruktori unikaalsele identifikaatorile vastavusse ühiktüübi elemendi. Sisuliselt on tegemist tüübikonstruktorite hulgaga. See on nende tüübikonstruktoride hulk, mille abil defineeritud monaade käsitletakse monaadidena monaadiliste väärtuste polümorfismi mõttes (vt ka jaotist 7.5.3).

`type`-deklaratsioon lisab konteksti tüübitaseme objekti ning seob selle muutujanimega. Seda vaatleme ka jaotises 7.6. `class`-deklaratsiooni vaatleme jaotises 7.6.4.

7.5 Funktsiooni rakendamise semantikast

7.5.1 Funktsiooni rakendamine

Fumontrixi funktsiooni rakendamise semantika on oluliselt keerulisem kui matemaatiline funktsiooni rakendamine. Funktsiooni rakendamiseks kasutatav operaator sõltub funktsiooni ja argumendi tüüpidest, mis võivad sisaldada tipmisel tasemel üldisuskvantoreid ja monaadikonstruktoreid.

Funktsiooni rakendamise semantika on kirjeldatud põhiliselt moodulis `SemUnify`. Funktsioon `unifyPolymorphic` määrab funktsiooni ja argumendi tüübi ning konteksti põhjal rakendamise tulemuse tüübi ning andmetasemel rakendamiseks kasutatava operaatori tüüpi

```
Value -> Value -> Environment -> Value
```

Lisaks kasutatakse parameetreid `generalOpenVForall` ja `getMonadInstanceForType`, et saaks kasutada neid moodulis `Denot` defineeritud funktsioone, kuna moodulit `Denot` moodulisse `SemUnify` importida ei saa, sest vastupidine sõltuvus on juba olemas.

7.5.2 Üldisuskvantorite avamine

Fumontrixis toimub universaalsete tüüpide (parameetrilise ja tüübiklassi-) polümorfismi realiseerimiseks funktsiooni rakendamise korral funktsiooni ja

argumendi tüübi tipmiste üldisuskvantorite avamine. Olgu funktsiooni tüüp kujul

$$\texttt{forall } C_1^f\texttt{. } \ldots \texttt{ forall } C_{n_f}^f\texttt{. (forall } C_1^d\texttt{. } \ldots \texttt{ forall } C_{n_d}^d\texttt{. } t_d\texttt{) -> } t_c$$

ning argumendi tüüp kujul

$$\texttt{forall } C_1^a\texttt{. } \ldots \texttt{ forall } C_{n_a}^a\texttt{. } t_a$$

Siis kõigepealt avatakse kvantorid funktsiooni tüübi eest (mis seovad muutujaid $C_1^f, \ldots, C_{n_f}^f$). Seejärel vaadatakse, kas kvantoreid on rohkem argumendi (n_a kvantorit) või määramispiirkonna tüübi (n_d kvantorit) ees. Esimesel juhul avatakse argumendi eest tipmised $n_a - n_d$ kvantorit (muutujad $C_1^a \ldots C_{n_a-n_d}^a$). Teisel juhul on tegemist tüübiveaga, kuna argument ei ole piisavalt polümorfne.

Avatud kvantifitseeritud muutujaid väärtustatakse unifitseerimise käigus (vt jaotist 7.5.4). Need muutujad, mis jäävad väärtustamata (kitsendamata), seotakse uuesti kvantoritega, mis lähevad tulemuse tüübi ette selles järjekorras nagu nad avati (kõige ette funktsiooni ees olnud ja seejärel argumendi ees olnud).

Lihtsuse mõttes vaatlesime selles näites parameetriliselt seotud kvantoreid. Tegelikult võib (mõne) kvantori juures olla ka tüübiklassikitsendus. Sellisel juhul tuleb see info ka avamisel säilitada (koodis väärtused `openPIKinds'` ja `openPIKinds2`), et määrata polümorfse funktsiooni ja argumendi spetsialiseerimisel kasutatavate eksemplaride väärtusi (`generalOpenVForall` abil), ning väärtustamata jäänud muutujate taaskvantifitseerimiseks.

7.5.3 Monaadiliste väärtuste polümorfism

Fumontrixi monaadiliste väärtuste polümorfismi realiseerimiseks funktsiooni rakendamisel toimub funktsiooni ja argumendi tüüpides tipmiste (`newmonad`-deklaratsioonidega määratud konstruktorite abil defineeritud) monaadide avamine sarnaselt üldisuskvantorite avamisega. Olgu funktsiooni tüüp kujul

$$M_1^f\ \texttt{(}\ldots\texttt{(}M_{n_f}^f\ \texttt{(}M_1^d\ \texttt{(}\ldots\texttt{(}M_{n_d}^d\ t_d\texttt{)}\ldots\texttt{))\ -> } t_c\texttt{)}\ldots\texttt{)}$$

ning argumendi tüüp kujul

$$M_1^a\ \texttt{(}\ldots\texttt{(}M_{n_a}^a\ t_a\texttt{)}\ldots\texttt{)}$$

Siis kõigepealt avatakse monaadid funktsiooni eest. Seejärel vaadatakse, kas monaade on rohkem argumendi või määramispiirkonna tüübi ees ning avatakse nii palju monaade, et argumendi ja määramispiirkonna tipmiste monaadide arv saaks võrdseks. Erinevalt üldisuskvantoritest võib määramispiirkonna

tüübi ees olla rohkem monaade kui argumendi ees, kuna monaadide jaoks on olemas `return`-funktsioon, mis võimaldab väärtuse monaadi sisse viia. Lisaks avatakse siin muutumispiirkonna tüübi t_c tipmisel tasemel olevad monaadid, kuigi siin on oluline ainult kõige tipmine monaad, mida võidakse koondada (monaadi `join`-funktsiooni abil), kui see satub tulemuse tüübis kõrvuti võrdse monaadiga.

Funktsiooniga `constructMonadicAp` konstrueeritakse avatud monaadide `return`-, `fmap`- ja `join`-funktsioonide abil monaadiline funktsiooni rakendamise operaator (andmetasemel rakendamise läbiviimiseks). Funktsiooniga `monadicApResMonads` määratakse need monaadid, mis lisatakse tagasi tulemuse tüübi ette.

7.5.4 Unifitseerimine

Pärast üldisuskvantorite ja monaadide avamist on vaja määramispiirkonna tüüp ja argumendiks antud väärtuse tüüp unifitseerida, et kontrollida, kas funktsiooni rakendamine on üldse võimalik, ning määrata avatud kvantoritega seotud muutujate väärtustused. Selleks kasutatakse funktsioone `unifyMonomorphic` ja `unifySolve`.

`unifyMonomorphic` võtab kaks tüüpi ja võrdleb rekursiivselt nende struktuure. Kui ühes struktuuris on mingi koha peal unifitseeritav muutuja (selleks kasutatakse predikaati `isVar`, mis eristab unifitseeritavaid muutujaid tavalistest konstruktoritest), siis see on unifitseeritav suvalise tüübiga teises struktuuris ning siit saadakse võrrand kujul *muutuja = tüübiavaldis*. Kui kummaski struktuuris ei ole antud koha peal unifitseeritav muutuja, siis peavad struktuurid kokku langema ning nende alamstruktuurid unifitseeruma.

Sellise rekursiivse võrdlemise teel saadakse lõpuks mingi hulk võrrandeid eespool toodud kujul. See võrrandisüsteem ei anna veel lõplikku muutujate väärtustust, kuna üks muutuja võib olla mitme erineva võrrandi vasakuks pooleks ning paremad pooled võivad samuti sisaldada unifitseeritavaid muutujaid. Seega see võrrandisüsteem on vaja lahendada.

Selleks kasutatakse funktsiooni `unifySolve`, mis lahendab võrrandisüsteemi asendusmeetodi abil. Võrrandite nimekirjast võetakse üks võrrand ning kontrollitakse, et see ei oleks rekursiivne (kui parem pool sisaldab vasakuks pooleks olevat muutujat, siis unifitseerimine ebaõnnestub, v.a triviaalse võrrandi korral, mis nõuab muutuja võrdumist iseendaga, mille võib lihtsalt minema visata). Seejärel asendatakse see võrrand ülejäänud võrranditesse, kaotades selle võrrandi vasakuks pooleks oleva muutuja ülejäänud võrranditest (kui see muutuja esineb veel mõne teise võrrandi vasaku poolena, siis tuleb nende võrrandite paremad pooled unifitseerida, saades mingi hulga (mis võib olla ka tühi) uusi võrrandeid). Kui nii on tehtud kõigi võrranditega, siis

on kõigi võrrandite vasakud pooled erinevad ning vasakutes pooltes esinevaid muutujaid paremates pooltes ei esine (muid muutujaid võib esineda).

Nii saamegi kätte muutujate väärtustuse. Asendades selle väärtustuse muutumispiirkonna tüüpi, saame kätte funktsiooni rakendamise tulemuse tüübi. Mõned muutujad võivad jääda väärtustamata, need seotakse uuesti kvantoritega, mis lähevad tulemuse tüübi ette.

Fumontrixis kasutatakse unifitseerimist ainult funktsiooni rakendamise semantika andmisel, mitte näidiste sobitamisel, valikuavaldises ega muutuja tüübi määramise (tüübiinferentsi) jaoks nagu Haskellis. Seega käituvad polümorfsed väärtused polümorfselt ainult funktsiooni rakendamisel, mitte aga näiteks valikualternatiivi paremas pooles. Seega peavad valikuavaldises alternatiivide paremad pooled olema kõik sama tüüpi, vajadusel tuleb polümorfsete väärtuste kvantoreid käsitsi avada.

Valikualternatiivis saaks polümorfismi võimaldada, kui defineerida selle semantika funktsiooni rakendamise semantika kaudu (valikualternatiivist moodustataks siis λ-abstraktsioon, kus näidiseks oleks valikualternatiivi vasakuks pooleks olev näidis), nagu on tehtud `do`-avaldise korral. Fumontrixis seda siiski tehtud ei ole, kuna erinevalt `do`-avaldisest ei ole valikuavaldis realiseeritud süntaktilise suhkruna.

7.6 Tüübitaseme programmeerimise semantikast

7.6.1 Tüübitaseme funktsioonid

Tüübitaseme funktsioonid konstrueeritakse Fumontrixis λ-abstraktsioonidena. Selleks on kaks eraldi konstruktsiooni — mitterekursiivne λ ja lihtrekursiivne λ.

Mitterekursiivse tüübitaseme λ-abstraktsiooni semantikaks on funktsioon, mis seab (fikseeritud liiki) tüübitaseme objektidele vastavusse (samuti mingit fikseeritud liiki, mis võib olla erinev argumentide liigist) tüübitaseme objektid. Sellele funktsioonile argumendi andmisel lisatakse argument konteksti λ-abstraktsiooni peas määratud muutujanime alla ning selles täiendatud kontekstis väärtustatakse λ-abstraktsiooni keha.

Lihtrekursiivse tüübitaseme λ-abstraktsiooni semantikaks on samuti funktsioon, mis seab tüübitaseme objektidele vastavusse tüübitaseme objektid. See funktsioon moodustatakse teistmoodi kui mitterekursiivsel juhul. Seda vaatleme lähemalt jaotises 7.6.3.

λ-abstraktsiooniga defineeritud tüübitaseme funktsioonide jaoks kasutatakse eraldi semantilist kategooriat `GenType`, mis sisaldab ka tavaliste tüüpide kategooriat `Type`. See on vajalik selleks, et λ-abstraktsiooni ei saaks tüübikonstruktori parameetriks anda. Nagu nägime jaotises 6.2, oleks vastasel

korral võimalik defineerida mittetermineeruv tüübitaseme funktsioon.

7.6.2 Väärtused tüübitasemel

Fumontrixis on võimalik andmetaseme väärtustega arvutada ka tüübitasemel. Kuna tüübitase ja andmetase on teineteisest eraldatud ja automaatset väärtuste liikumist nende vahel ei toimu, siis tuleb see liikumine ilmutatult teha. Selleks on Fumontrixis olemas `value`- ja `type`-konstruktsioonid.

`value`-konstruktsioon muudab andmetaseme väärtuse tüübitaseme objektiks liiki `@`. Süntaktiliselt on `value`-konstruktsiooni argumendiks andmetaseme avaldis, mille väärtus võib sõltuda keskkonnast. Tüübitasemel keskkonda ei ole, seega tüübitasemel väärtustega arvutamisel välja arvutatud väärtuse sisse vaadata ei saa ning väärtust saab kasutada musta kastina, mille kohta on teada ainult selle tüüp.

Et pärast tüübitasemel töötlemist tulemuseks saadud väärtust jälle andmetasemel kasutada, on vaja teada keskkonda, milles see väärtus välja arvutada. Kui võtta selleks see keskkond, milles toimub väärtuse kasutamine andmetasemel, siis oleks `value`-konstruktsiooni sees olevas avaldises esinevad identifikaatorid, mis ei ole defineeritud selle sama konstruktsiooni sees, dünaamilise skoopimisega.

Selline piiramatu dünaamiline skoopimine, kus kasutatav on kogu dünaamiline keskkond, tekitab probleeme. Näiteks on võimalik tekitada rekursioon ilma rekursioonioperaatorit kasutamata. Saame kirjutada isegi tüübitaseme funktsiooni, mis ei termineeru:

```
type rf = \ x : @ . value (type rf x);
v = type rf (value 3);
```

Kuna `rf` kutsutakse välja `v` defineerimisel, siis `rf` sees olevale avaldisele `type rf x` rakendatakse dünaamilist skoopi, milles on see sama `rf` juba olemas, seega `rf` saab kasutada rekursiivselt iseennast ning praegusel juhul tuleb lõpmatu rekursioon.

Kuna Fumontrixis soovitakse garanteerida tüübikontrolli termineerumine, siis siin dünaamilist skoopimist ei kasutata ning ka `value`-konstruktsiooni sees rakendatakse leksilist skoopimist. Selleks on `value`-konstruktsiooni rakendamise ajal vaja kätte saada see keskkond, milles see konstruktsioon tekitati. Kuna väärtustega arvutatakse ka tüübitasemel, siis keskkonda ennast ei saa `value`-konstruktsiooni juurde lisada (seda ei ole tüübitasemel veel olemas). Selle asemel määratakse skoobile unikaalne identifikaator ning lisatakse see `value`-konstruktsioonile selle moodustamisel. Samuti arvutatakse kohe moodustamisel välja `value`-konstruktsioonis oleva avaldise tüüp (seda saab tüübitasemel teha). `value`-konstruktsiooni semantikaks on siis nelik, mis

koosneb avaldisest, keskkonna (selle, milles avaldis tuleb väärtustada) identifikaatorist, selle avaldise tüübist ning lisainfost (`StatSemInfo`), mis tuleb tüübitasemelt andmetasemele edastada selle avaldise väärtustamiseks.

Selleks, et `value`-konstruktsiooni sees oleva avaldise väärtust kasutada on vaja see tüübitasemelt tagasi andmetasemele viia. Selleks on olemas `type`-konstruktsioon, mis muudab liiki `@` tüübitaseme objekti andmetaseme väärtuseks. Liiki `@` tüübitaseme objekte saab tekitada ainult `value`-konstruktsiooniga ning tüübitasemel nende sisu muuta ei saa, seega kui liiki `@` tüübitaseme avaldis väärtustada, siis tulemuseks on mingi `value`-konstruktsiooni semantika.

`value`-konstruktsioon võib esineda ainult mingi tüübitaseme avaldise sees. Selleks võib olla kas `type`-konstruktsioon või `type`-deklaratsiooni parem pool. Põhimõtteliselt võiks seda kasutada ka tüübiannotatsioonis (näiteks argumendina liiki `@ -> *` tüübikonstruktorile), aga sellel pole mõtet, kuna selliste tüüpide võrdsust ei saa tüübitasemel kontrollida (`value`-konstruktsiooni sees oleva avaldise väärtus selgub alles andmetasemel) ja seetõttu loetakse iga `value`-tüüp kõigi muude tüüpidega (sh iseendaga) mittevõrdseks ja mitteunifitseeruvaks. Seega selline annotatsioon põhjustab alati tüübivea.

Seetõttu määratakse liigitasemel (skoobi piires) unikaalsed identifikaatorid kõigi `type`-deklaratsioonide ja `type`-avaldiste skoopide jaoks. Andmetasemel salvestatakse `type`-deklaratsiooni korral selle keskkond vastava unikaalse identifikaatori alla jooksvasse keskkonda. Selleks on keskkonnas kujutus `eSavedEnv`, mis seab skoobi identifikaatorile vastavusse salvestatud keskkonna. Samuti salvestatakse keskkond `type`-avaldise väärtustamise ajaks vastava unikaalse identifikaatori alla.

Igale `value`-konstruktsioonile seatakse liigitasemel vastavusse kõige sisemise seda sisaldava `type`-deklaratsiooni või `type`-avaldise skoobi identifikaator, mille järgi saab hiljem andmetasemel jooksvast keskkonnast salvestatud (leksilise skoobi) keskkonna kätte.

Andmetasemel `value`-konstruktsiooni sees oleva avaldise kasutamiseks tuleb see `type`-konstruktsiooniga tagasi andmetasemele tuua. Seega `type`-konstruktsiooni sees olev tüübitaseme objekt väärtustatakse, saades `value`-konstruktsiooni, mille juures on skoobi identifikaator. Selle järgi võetakse jooksvast keskkonnast salvestatud leksilise skoobi keskkond ja väärtustatakse avaldis selles salvestatud keskkonnas. Saadud tulemus ongi `type`-avaldise väärtuseks.

Põhimõtteliselt võiks ka otse `value`-konstruktsioonidele skoobiidentifikaatorid juurde lisada, kuid sellisel juhul tuleks ka tüübitaseme avaldistele andmetaseme semantika anda. Samuti tuleks kõik `type`-deklaratsiooni sees olevate `value`-avaldiste salvestatud leksilised keskkonnad lisada samasse skoopi kui selle `type`-deklaratsiooniga defineeritav tüübitaseme muutuja. See skoop

on kõrgemal tasemel kui nende `value`-avaldiste tegelik skoop ning info madalamalt tasemelt kõrgemale liigutamine muudaks semantika üsna keeruliseks. Seetõttu on Fumontrixis valitud eespool kirjeldatud variant, mis on lihtsama semantikaga.

7.6.3 Lihtrekursioon

Fumontrixis on tüübitaseme programmeerimisel võimalik kasutada lihtrekursiooni. Erinevalt üldisest rekursioonist, on lihtrekursiooni korral garanteeritud termineerumine. Kuna Fumontrixis soovitakse garanteerida tüübikontrolli termineerumist, siis üldist rekursiooni siin tüübitasemel ei võimaldata.

Lihtrekursiooni moodustamiseks on Fumontrixis lihtrekursiivne tüübitaseme λ-avaldis (mis on mitterekursiivsest λ-avaldisest eraldi). Selle λ-avaldise sees lihtrekursiivse pöördumise jaoks saab kasutada `rec`-avaldist.

Lihtrekursioonikonstruktsiooni semantika andmiseks defineeritakse kõigepealt funktsioon g, mis saab parameetriks arvutatava funktsiooni väärtused mingi tüübi (või tüübikonstruktori) t kõigi komponentide (vahetute või mitte) jaoks ning tagastab selle funktsiooni väärtused tüübi t jaoks (vt ka näidet jaotises 5.1.3). Funktsiooni g defineerimisel saab neid parameetreid (arvutatava funktsiooni väärtusi komponentidel) kasutada `rec`-avaldise abil. Funktsiooni väärtus komponendil t_1 on näiteks `rec` t_1.

Selline funktsioon määrab üheselt mingi lihtrekursiivse funktsiooni f. Et arvutada selle funktsiooni väärtust mingi argumendi jaoks, arvutatakse kõigepealt rekursiivselt väärtused selle komponentide jaoks ning seejärel kasutatakse eespool mainitud funktsiooni, et leida funktsiooni väärtus vajaliku argumendi jaoks. Et tüüpide ja tüübikonstruktorite struktuur on Fumontrixis lõplik, siis selle rekursiivne läbimine ja komponentidel funktsiooni väärtuste väljaarvutamine alati termineerub (eeldusel, et ka funktsioon g termineerub, aga Fumontrixis ei saa kirjutada mittetermineeruvat tüübitaseme funktsiooni). Funktsioon f ongi lihtrekursioonikonstruktsiooni semantikaks.

`rec`-avaldise semantika andmiseks on kontekstis kujutus `zSiRec`, mis seab lihtrekursiooni identifikaatorile (vt jaotist 5.1.3) vastavusse teise kujutuse. See kujutus seab nendele argumendi komponentidele, millele vastav funktsiooni väärtus on vaadeldava identifikaatoriga lihtrekursiooni käigus juba välja arvutatud, vastavusse need funktsiooni väärtused. `rec`-avaldisele semantika andmiseks võetakse lihtsalt sellest kujutusest vastav väärtus.

7.6.4 Tüübiklassid

Tüübiklassi defineerimiseks on Fumontrixis `class`-deklaratsioon kujul `class` c t. See toob sisse nimega c tüübiklassi, mille eksemplaride tüüp

on kirjeldatud tüübiavaldisega t. See tüübiavaldis peab olema universaalselt kvantifitseeritud kujul `forall` A `:` k `.` $t'[A]$. Tüübiklassi esindajad on sellisel juhul liiki k ning esindajale A vastav eksemplar peab olema tüüpi $t'[A]$. Tüübiklasside jaoks on kontekstis kujutus `zClass`, mis seab tüübiklassi nimele vastavusse selle eksemplaride tüüpe kirjeldava tüübi.

Eksemplarid määratakse tüübitaseme funktsiooniga, mille nimi ühtib vastava tüübiklassi nimega. See funktsioon seab tüübile (või tüübikonstruktorile) vastavusse eksemplari. See eksemplar peab olema eelmises lõigus kirjeldatud tüüpi, kuid seda kontrollitakse alles eksemplari kasutamisel (tüübiklassiga kitsendatud üldisuskvantori avamisel), mitte eksemplari defineerimisel.

Polümorfsed väärtused, mille tüüp sisaldab tipmisel tasemel tüübiklassiga kitsendatud üldisuskvantorit, on semantiliselt esitatud funktsioonidena, mis seavad tüübiklassi eksemplari väärtusele vastavusse (monomorfsema) väärtuse. Kui selle väärtuse tüübi üldisuskvantor avatakse (kas automaatselt funktsiooni rakendamise käigus või käsitsi), siis võetakse jooksvast keskkonnast tüübiklassi eksemplari väärtus selle tüübi jaoks ning antakse polümorfsele väärtusele vastavale funktsioonile argumendiks. Selle polümorfse avaldise sees seotakse see argumendiks antud eksemplar selle kvantoriga seotud tüübikonstruktoriga, mis muutub seega selle tüübiklassi esindajaks.

Selle sidumise realiseerimiseks salvestatakse see eksemplar tühja nimega muutujasse, mis lisatakse keskkonda, saadud keskkond salvestatakse sellele eraldatava unikaalse skoobiidentifikaatori abil jaotises 7.6.2 vaadeldud viisil. Sisuliselt moodustatakse `value`-konstruktsioon, mille sisuks on see tühja nimega muutuja. Tüübiklassi eksemplare määravat tüübitaseme funktsiooni täiendatakse nii, et vaadeldavale (uuele) tüübile määratakse eksemplariks see `value`-konstruktsioon. Kui nüüd millalgi on vaja selle uue tüübi eksemplari kasutada, siis väärtustatakse see `value`-konstruktsioon, mille käigus võetakse salvestatud keskkonnast eksemplari väärtus.

Tüübiklasse saab kasutada ka eksistentsiaalsete tüüpide korral. Tüübiklassiga kitsendatud olemasolukvantori avamisel (`exists`-näidise abil) tuuakse analoogiliselt üldisuskvantoriga sisse uus konstruktor, mis muudetakse tüübiklassi esindajaks, määrates vastavaks eksemplariks selle eksistentsiaalse väärtuse sees olnud eksemplari.

Kokkuvõte

Selles töös lõime uue funktsionaalse programmeerimiskeele Fumontrix ning realiseerisime sellele interpretaatori. Töös võrdlesime muuhulgas selle keele võimalusi Haskelli (GHC) omadega.

Selles keeles realiseerisime enamiku olulisemaid Haskelli konstruktsioone, sealhulgas algebralised andmetüübid, parameetriliselt polümorfsed funktsioonid ja tüübiklassid. Sarnaselt GHC-ga realiseerisime impredikatiivse polümorfismi. Impredikatiivse polümorfismi võimaluse korral ei pruugi funktsiooni rakendamise tulemus enam üheselt määratud olla. Seetõttu on teatud juhtudel vaja üldisuskvantoreid käsitsi avada, mida Fumontrixis on oluliselt mugavam teha kui GHC-s, kuna erinevalt GHC-st ei ole vaja kogu väärtuse tüüpi ümber kirjutada, piisab anda vaid avatava kvantoriga seotud tüübimuutuja väärtustus.

Tüübiklassidest realiseerisime küll lihtsama variandi kui Haskellis, kuid nagu nägime, ei ole see väga suur puudus, kuna Fumontrixis on olemas tüübitaseme funktsioonid, mis on palju suuremate võimalustega kui tüübiklassid ja enamasti kasutatavad ka tüübiklasside asemel. Tüübiklassid osutusid vajalikuks ainult selleks, et teatud piiratud dünaamilise skoopimise elemente sisse tuua. Mujal kasutatakse Fumontrixis leksilist skoopimist, kuna, nagu nägime, võimaldaks piiramatu dünaamiline skoopimine tüübitasemel lõpmatu rekursiooni tekitada.

Samuti nägime, et Fumontrixi tüübitaseme funktsioonid pakuvad palju suuremaid võimalusi kui GHC tüübiklassid ning on defineeritavad loomulikul viisil λ-abstraktsioonidena, samas kui GHC-s on nende defineerimine väga kohmakas, kuna GHC tüübiklassid pole tüübitaseme funktsioonide defineerimiseks mõeldud. Fumontrixis saame tüübitasemel arvutada ka väärtustega, seda küll musta kastina, mille kohta on teada ainult selle väärtuse tüüp. Väärtustega arvutamist saab kasutada ka tüübiklasside eksemplaride defineerimisel, kuna eksemplaride väärtused seatakse tüüpidele vastavusse tüübitaseme funktsiooni abil.

Tüübitasemel rekursiooni kasutamiseks otsustasime Fumontrixis realiseerida üldise rekursiooni asemel lihtrekursiooni, mis on enamiku kasulike tüübitaseme funktsioonide realiseerimiseks piisav, kuid samas garanteerib tüübikontrolli termineeruvuse, mis oli üks Fumontrixi eesmärke. Töös vaatlesime ka mõningaid tüübitaseme funktsioonide rakendusi, näiteks polümorfsete funktsioonide realiseerimist ning erinevate kasulike operaatorite defineerimist, mida GHC-s ei saa defineerida.

Fumontrixi üheks põhimõtteks oli skoopide võrdväärsus, seega tüübiklasse, tüübitaseme funktsioone, algebralisi andmetüüpe jne võimaldatakse defineerida suvalises skoobis (nii peatasemel kui `let`-avaldistes), erinevalt Has-

kellist, kus neid saab defineerida vaid peataseme skoobis.

Fumontrixis realiseerisime ka mõned võimalused imperatiivse programmeerimisstiili kasutamiseks, näiteks ST-monaadi, milles saab kasutada muudetavad viitasid ja massiive, kuid need arvutused saab läbi viia muust maailmast eraldatud lõimes, mistõttu arvutused on seal deterministlikud ja keel jääb puhtalt funktsionaalseks. Monaadiliste väärtuste mugavamaks kasutamiseks realiseerisime monaadiliste väärtuste polümorfismi, mis võimaldab tavalisel funktsiooni rakendamisel kasutada erinevaid monaadilisi ja mittemonaadilisi väärtusi, erinevalt Haskellist, kus selleks tuleks defineerida palju erinevaid funktsiooni rakendamise operaatoreid. Samuti realiseerisime süntaktilise suhkruna `do`-notatsiooni, mida erinevalt Haskellist saab kasutada korraga ka mitme erineva monaadi jaoks.

Töös uurisime ka erinevaid polümorfismi liike ning vaatlesime võimalusi nende kasutamiseks Fumontrixis — parameetrilist polümorfismi, tüübiklasse, tüübitaseme funktsioone ning monaadiliste väärtuste polümorfismi.

Töös realiseerisime ka eksistentsiaalsed tüübid, mida saab kasutada nii parameetriliselt kui tüübiklassiga kitsendatult, kuid need eksistentsiaalsed tüübid ei ole polümorfsed. Erinevalt Haskellist saab olemasolukvantorit kasutada suvalise tüübikomponendi ümber (nagu ka üldisuskvantorit), mitte ainult seoses algebraliste andmetüüpidega.

Interpretaatori realiseerimisel andsime süntaktilistele konstruktsioonidele semantika funktsionaalsel kujul, kasutades ainult puhtalt funktsionaalseid andmestruktuure. Seega määrab selle interpretaatori kood teatud liiki denotatsioonilise semantika. Sellest semantikast andsime ka töös ülevaate.

Viited

[1] Haskell 98 Language and Libraries. The Revised Report, toimetanud Simon Peyton Jones. 2002. `http://www.haskell.org/onlinereport/` (viimati väisatud 15.05.2008)

[2] The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.8.2. The GHC Team. 2007. `http://haskell.org/ghc/docs/6.8.2/html/users_guide/` (viimati väisatud 15.05.2008)

[3] Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction, Marktoberdorf Summer School,* 2002.

[4] Luca Cardelli ja Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, Vol.17, No.4, 1985.

[5] Thomas Hallgren. Fun with Functional Dependencies. *Proceedings of the Joint CS/CE Winter Meeting,* 2001.

[6] Happy User Guide, Version 1.17. Simon Marlow ja Andy Gill. 2007. `http://www.haskell.org/happy/doc/html/` (viimati väisatud 26.05.2008)

A Functional Programming Language and its Semantics

Master Thesis

Martin Pettai

Resume

We have created a new functional programming language Fumontrix and implemented an interpreter for it. In this thesis, we give an overview of its features and their semantics and also compare the features to those of Haskell (GHC).

We have implemented most of the standard Haskell constructions, including algebraic data types, parametrically polymorphic functions, and type classes. Similarly to GHC, we have implemented impredicative polymorphism. When impredicative polymorphism is allowed, the result of function application is no longer guaranteed to be unique, thus it is often necessary to explicitly specialize universally quantified types. Whereas GHC requires to annotate the whole result type after specialization, Fumontrix allows to only specify the type to be substituted for the quantified variable, which requires less work from the programmer.

While we have implemented type classes, our version is more primitive than that of Haskell (and especially GHC). As we see in the thesis, this is not very restrictive, as we have implemented type-level functions, which are much more powerful than type classes and in most cases can be used as an alternative to type classes. Type classes were only introduced in Fumontrix to have some limited elements of dynamic scoping. Elsewhere we use lexical scoping, as unlimited dynamic scoping would allow infinite recursion to occur at the type level.

Our type-level functions can be defined naturally as type-level λ-abstractions, whereas GHC type classes are not intended for general type-level programming and, while usable for defining some type-level functions, are unnatural to use and require ugly code. In Fumontrix, we can also use values (in addition to types) in type-level computations, although only as a black box for which only type is known. The black box value obtained as a result of type-level computation can only be opened at the data level.

To allow recursion at the type level, we decided to implement only primitive recursion, not general recursion. This is enough for most useful type-level functions and also guarantees termination of type checking. In the thesis, we

also see some applications of type-level functions, such as polymorphic functions or defining some useful operators that are not definable in GHC.

One of the principles of Fumontrix is the equivalence of scopes, therefore type classes, type functions, algebraic data types, etc. can be defined in any scope (including `let`-expressions), unlike in Haskell where these declarations are restricted to the top level.

We have also implemented some features that allow using imperative programming style, e.g. ST monad which allows to use mutable references and arrays but in a thread separate from the outside world, thus retaining the purely functional property. To make the use of monadic values less ugly, we have implemented a polymorphism of monadic values, which allows us to use the ordinary function application for different monadic and non-monadic values, unlike in Haskell where we would have to define many different function-application operators for these cases, such as `liftM` and `ap`. We have also implemented the `do`-notation (as syntactic sugar), which can also be used for several different monads at a time, unlike in Haskell.

In this thesis we also investigate different kinds of polymorphism and the possibilities of using them in Fumontrix.

We have also implemented existential types, which can be used both parametrically and restricted by a type class, although the existential types are not polymorphic. The existential quantification can be used around any type component, not only at the top level.

Our implementation of the interpreter gives a semantics to syntactic constructions in a functional style, using only purely functional data structures. This defines a certain kind of denotational semantics. In the thesis we also give an overview of this semantics.