Erindid Haskellis

Jaak Randmets
jaak.ra@gmail.com

Programmeerimiskeelte semantika uurimisseminar
Arvutiteaduse instituut, Tartu Ülikool

28. november 2009. a.

Kokkuvõte

Erinditeta programmeerimine on sarnane ilma turvavööta auto juhtimisele. Kõik töötab suurepäraselt, kuni midagi valesti ei lähe. Paratamatult on aegajalt vaja eriliste olukordadega mingil viisil hakkama saada. Just sel põhjusel pakuvad kõik tõsiselt võetavad programmeerimiskeeled, kaasa arvatud Haskell, vahendeid nende käsitlemiseks.

Imperatiivsetes programmeerimiskeeltes on erindid klassikaliselt seotud kontrollvooga ning erindi viskamine on lihtsalt hüpe kontrollvoo graafis. Haskell aga on omaette erind programmeerimiskeelte seas. Tegemist on laisa funktsionaalse keelega ja kontrollvoog ei ole hästi defineeritud ning seepärast seda lähenemist kasutades me ei saa erinditele semantikat anda. Käesolevas seminaritöös anname ülevaate ebatäpsete erindite semantikast Haskellile. Toome denotatsioonilise semantika lihtsalt erinditega täiendatud laisale keelele ning tutvume ka toodud semantika probleemidega.

1 Sissejuhatus

Erindid Haskellis on problemaatilised. Ühelt poolt piinavad meid keerukused nende implementeerimisega ning raskused nende mõistmisega laisa programmeerimiskeele kontekstis. Teiselt poolt ei ole erindite kasutamine kaugeltki nii mugav, kui näiteks mõnes objektorienteeritud keeles nagu Java. Käesolevas töös keskendume esimesele probleemile.

Agarates imperatiivsetes keeltes vaadeldakse erindeid, kui kontrollvoo modifikatsioone. Haskell on laisk keel ning seega ei ole kontrollvoog hästi defineeritud ja sellel põhjusel me ei saa seda lähenemist kasutada. Näiteks, kui vaatleme täisarvude järjendit siis on sisuliselt kolm erinevat viisi erindi tekkimiseks. Esiteks võib järjend ise visata erindi. Teiseks võib järjendi saba olla erind ning kolmandaks võib mõni järjendis salvestatud väärtus erindi visata. Tuleb tähele panna, et mitte igal juhul ei propageeru erind edasi. Näiteks, kui arvutame ainult järjendi pikkust siis järjendi elemente ei ole vaja väärtustada.

Programmi optimeerimiseks võib kas programmeerija või kompilaator avaldiste väärtustamise järjekorda muuta. Näiteks agarusanalüüsi tulemuste põhjal võime mõned funktsiooni argumendid agaralt väärtustada. Paraku aga võib

avaldiste väärtustamise järjekorra muutmine mõjutada millist erindit esimesena visatakse. Üheks lahenduseks on piirata sellised kooditeisendused alamavaldistele, mis erindeid kindlasti ei viska. Alternatiivseks lahenduseks on vahetada täpsus efektiivsuse vastu ning lubada enam kooditeisendusi olles ebatäpsem visatavate erindite suhtes. Seda lähenemist tuntakse ebatäpsete erindite [2,4] nime all.

Käesolevas seminaritöös anname ülevaate erinditest Haskellis. Esmalt vaatleme erindeid kui väärtusi ja uurime, kuidas on võimalik erindeid modelleerida funktsionaalses laisas keeles ilma, et keel neid otseselt toetatakse. Tutvume ka selle lähenemise nõrkustega, mis on motivatsiooniks keele semantika erinditega täiendamiseks. Saades selgeks ilmutatud erindite nõrkused jõuame loomulikult ebatäpsete erindite disainini. Toome süntaksi ja semantika lihtsale erinditega täiendatud laisale funktsionaalsele keelele. Lõpetuseks tutvume ebatäpsete erindite nõrkustega. Käesolevas töös eeldame, et lugeja on tutvunud programmeerimiskeelega Haskell ja on kokku puutunud monaadide mõistega selles keeles. Kasuks tuleb ka baasteadmiste omamine denotatsioonilise semantika vallast.

Seminaritöö toetub peamiselt S. P. Jones, A. Reid, *et al.* artiklile "*A semantics for imprecise exceptions*" [2] kus pakutakse välja ebatäpsete erindite semantika Haskellile. Edasist infot ebatäpsetest erinditest Haskellile võib lugeja leida artiklitest [4] ja [5]. Monaadilise semantikaga võib lugeja tutvuda artiklis [6].

2 Erindid kui väärtused

Sissejuhatuses nägime, et laisas keeles peame erindeid vaatlema kui väärtusi, sest iga väärtuse kasutamine võib tekitada erindi. On hästi teada, et erindeid on võimalik ilmutatult väärtustes kodeerida, kui programmeerimiskeel toetab algebralisi andmetüüpe. Piisab kui defineerima kahe konstruktoriga andemtüübi, millest üks esitab "normaalseid" väärtusi ning teine "erilisi" väärtusi:

```
data ExVal a = OK a
             | BAD Exception
```

Konstruktor *OK* esitab siin tavalisi väärtusi ning konstruktor *BAD* erindeid. Seda andmetüüpi saame kasutada erindeid viskavate väärtuste esitamiseks puhtas Haskellis. Näiteks funktsiooni täisarvudest täisarvudesse, mis teatavatel argumentidel tagastab erindi, saame esitada järgnevat tüübisignatuuri omava funktsiooniga:

```
f :: Int → ExVal Int
f x = ...   -- funktsiooni definitsioon
```

Kahjuks peame nüüd iga funktsiooni f kasutuse korral kontrollima, et ega erindeid ei visata:

```
case (f 3) of
  OK val  → ...   -- normaalne juht
  BAD ex → ...   -- eriline juht
```

Muidugi on võimalik erindi kontrollimist edasi lükata, kui kasutame funktsioone, mis võtavad erilisi väärtusi argumentidena.

Sellel lähenemisel on mitmeid positiivseid külgi. Esiteks me ei pea keelt erinditega laiendama ning teiseks näitab tüübisüsteem selgelt millised funktsioonid erindeid võivad visata ja millised mitte. Peale selle ei ole võimalik erindite käsitlemist kunagi unustada.

2.1 Puudused

Kahjuks on ilmutatud erinditel hulgaliselt puudusi, mis kaaluvad selle lähenemise tugevused suurelt üle:

- Väga lihtne on teha funktsioone agaramaks, kui need tegelikult on, testides argumente liiga vara. Selle probleemi vältimiseks peame väärtusi testima täpselt enne nende kasutamiseks ning mitte varem, see võib aga osutuda väga ebamugavaks.

- Oluline erindite omadus on, et need propageeruvad automaatselt ilma, et programmeerija peaks koodi risustama erindi viskamise ning püüdmise kohtade vahel. Selle vastandina peame ilmutatud erindeid kasutades erindi viskamise ning püüdmise kohtade vahel oma koodi alati modifitseerima nende käsitlemiseks. Näiteks, kui soovime täiendada lihtsat koodi

  $$(f\ x) + (g\ y)$$

  käsitlemaks erindeid, siis nüüd peame selle asemel kasutama märgatavamalt pikemat koodi:

  $$\begin{array}{l}\textbf{case}\ (f\ x)\ \textbf{of}\\ \quad BAD\ ex \rightarrow BAD\ ex\\ \quad OK\ xv \quad \rightarrow \textbf{case}\ (g\ y)\ \textbf{of}\\ \qquad BAD\ ex \rightarrow BAD\ ex\\ \qquad OK\ xy \quad \rightarrow OK\ (xv + yv)\end{array}$$

  Kusjuures me ei tegele siin otseselt ei erindite püüdmise ega viskamiseks.

  Monaadiline [6] lähenemine muudab koodi küll hallatavamaks:

  $$\begin{array}{l}\textbf{do}\\ \quad xv \leftarrow f\ x\\ \quad yy \leftarrow g\ y\\ \quad return\ (xv + yv)\end{array}$$

  või isegi $liftM2\ (+)\ (f\ x)\ (g\ y)$, kuid paratamatult peame mingil määral olemasolevat koodi muutma.

- Meil ei ole võimalik käsitleda keelde sisse ehitatud erindeid. Näiteks nulliga jagamisi või mustri sobitamise vigasid. See on väga tõsine probleem kui kirjutame suuri programme, mis kasutavad teiste poolt kirjutatud komponente. Meil ei ole võimalik sellistest vigadest kuidagi taastuda ega neid vältida, kui need juhtuvad meie kontrolli alt väljas olevates komponentides.

- Erindid peaksid maksma koodi efektiivsuse seisukohalt väga vähe kui neid tegelikult ei visata. Ilmutatud erindid aga vastupidiselt maksavad väga

palju, kuna peame igal sammul **case** analüüse läbi viima isegi juhul, kui tahame erindit ainult edasi propageerida. See mõjub programmi efektiivsusele laastavalt.

- Efektiivsus kannatab lisaks, kuna monaadilises stiilis kirjutatud koodil on palju vähem korrektseid teisendusi, kui puhtal koodil.
- Asünkroonseid erindeid ei ole võimalik nii modelleerida.

Kõik need puudused ei tähenda, et ilmutatud erindid kasutud on. Sageli võib see lähenemine muuta koodi arusaadavamaks, eriti juhul, kui ei kasuta seda "erilistest" olukordadest taastumiseks.

3 Ebatäpsed erindid

Käesolevas sektsioonis anname intuitiivse ülevaate ebatäpsetest erinditest. Toome välja omadused, mida nõuame semantikalt ning põhjendame miks täpsed erindid meie jaoks ei kõlba.

Haskell on laisk keel ja seega on erindid seotud väärtustega, mitte kontroll vooga. See erineb fundamentaalselt klassikalistest imperatiivsete ja agarate keelte lähenemisest, kus erindid on seotud kontrollvooga. Üheks näiteks erinditest, kui väärtustest on `IEEE` ujuvkoma arvud, kus näiteks `NaN` (*not a number*) esitatakse väärtuses endis ning operatsioonid arvudel propageerivad seda edasi.

3.1 Nõuded

Soovime erindite semantikalt järgmisi omadusi:

- Kood, mis ei viska ega püüa erindeid peaks jääma semantiliselt samaks ja jooksma sama efektiivselt. Kusjuures tähendab see, et me ei pea koodi liigselt risustama nagu juhtus ilmutatud erinditega.
- Kõik teisendused, mis on lubatud puhtal Haskelli koodil peaksid olema võimalikud ka erinditega täiendatud Haskelli koodil. Tuleb välja, et seda eesmärki me päris ei saavuta ning hiljem näeme täpselt, kus tulevad välja probleemid.
- Peame saama formaalselt arutleda selle üle milliseid erindeid programm võib või ei või visata. Näiteks, soovime tõestada, et programm ei viska erindeid või et programm ei termineeru. See ei ole loomulikult üldisel juhul võimalik.

3.2 Idee

Toetume ilmutatud erindite ideedle. Nimelt igat tüüpi väärtused on kas "normaalsed" või "erilised". Näiteks tüüp *Int* ei sisalda mitte ainult täisarve vaid lisaks ka kõike võimalikke erilisi väärtusi. Erilisi väärtusi esitame *Exception* andmetüübis:

```
data Exception
  = DivideByZero
  | Overflow
```

| *UserError String*
...

Defineeritud andmetüüp võib olla keerulisem, kuid lihtsuse huvides piirdume siin sellega. Paneme tähele, et see lähenemine ei ole kõige kasutajasõbralikum, kuna nii ei ole programmeerijal võimalik erindeid ei mugavalt laiendada ega neid hierarhiasse struktureerida. Programmeerijasõbralikuma erindite teegiga võib lugeja tutvuda artiklis [3].

Iga tüübi a jaoks sisestab **throw** argumendina saadud erindi sellesse andmetüüpi:

throw :: $Exception \rightarrow a$

Näeme kuidas käesolev lähenemine erineb ilmutatud erinditest. Iga tüüp võib sisaldada erilisi väärtusi – ennem olid sellised ainult $ExVal\ t$ tüüpi väärtused.

Oleks loomulik anda erindite püüdmise funktsioonile *getException* tüüp $a \rightarrow ExVal\ a$. Sellega signatuuriga on aga tõsine probleem. Vaatame avaldist:

getException ((1 / 0) + *error* `"Urk"`)

Tekib küsimus, et kas tagastatakse erind *DivideByZero* või *UserError* `"Urk"`. Sellele probleemile on kolm lahendust:

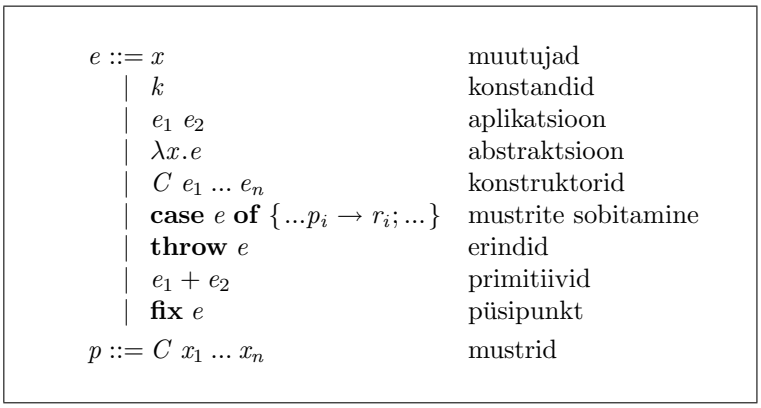
- Fikseerime liitmise argumentide väärtustamise järjekorra. Näiteks võime nõuda, et esimene argument väärtustatakse esimesena. Sellisel juhul on ka hästi defineeritud milline erind visatakse. See on klassikaline lähenemine, mis on võetud mitmete funktsionaalsete keelte poolt nagu näiteks `ML`. Selle lähenemise tugevuseks on semantika lihtsus, aga suureks nõrkuseks tõsiasi, et kaotame hulgaliselt kooditeisendusi. Näiteks kaotame liitmise kommutatiivsuse.

- Anname liitmisele semantika nii, et argumentide väärtustamise järjekord valitakse mittedeterministlikult. Sellisel juhul võib implementatsioon valida argumentide väärtustamise järjekorra nii, nagu parajasti vaja. Kahjuks on selle lähenemisega üks väga tõsine probleem, nimelt kaotame β-reduktsiooni. Näiteks, vaatame avaldist:

  let $x = (1 / 0) +$ *error* `"Urk"`
  in *getException* $x \equiv$ *getException* x

  Toodud avaldise väärtus peaks olema *True*, aga kui asendame muutuja x esinemised tema definitsiooniga, siis mittedeterminismi tõttu võime teha erinevad valikud liitmiste esinemistes ja seega võib avaldise väärtuseks osutuda *False*. Oleme sunnitud selle lähenemise kõrvale lükkama, kuna me ei ole nõus β-reduktsioonist loobuma.

- Kavalam on anda liitmisele tähendus nii, et tagastatakse mõlemad erindid. Nimelt, kui mõlemad väärtused ei ole "tavalised" tagastab liitmise operaator hulga erindeid, mis saadakse harude poolt tekitatud erindite hulkade ühendamisel. Selle lähenemise tugevus on, et paljud kooditeisendused lähevad nii läbi ja ei teki probleeme, mis tekiksid mittedeterministliku semantikaga. Oleme valinud just selle lähenemise.



Joonis 1: Keele süntaks

Meie valitud lahendus parandab küll liitmise kommutatiivsuse, aga nüüd võivad avaldised tagastada erindite hulga, seega peame *getException* funktsiooni meie valikuga kooskõlasse viima. Selleks on kaks võimalust.

Esimeseks võimaluseks on lasta *getException* funktsioonil tagastada kõikide erindite hulk. Kahjuks on see lähenemine on täielik katastroof implementatsiooni seisukohalt, kuna peame alati meeles pidama võimalike erindite hulka. Peale selle, kui esimene argument viskaks erindi siis peaksime tegelikult teise argumendi väärtustama, et kõik võimalikud erindid kokku koguda.

Teine lahendus on lasta *getException* funktsioonil valida erind mittedeterministlikult. Loomulikult paljastab see mittedeterminismi uuest, aga me kasutame siin lihtsat trikki. Laseme *getException* funktsiooni kutsuda ainult *IO* monaadis:

$$getException :: a \rightarrow IO\ (ExVal\ a)$$

Lihtsuse huvides me ei kasuta erindite püüdmiseks *catch* funktsiooni:

$$catch :: IO\ a \rightarrow (Exception \rightarrow IO\ a) \rightarrow IO\ a$$

Selle semantika ja implementatsiooniga võib lugeja tutvuda artiklis [1]. Idee on väärtustada esimene argument ning tagastada resultaat, kui see osutub tavaliseks väärtuseks ning kui visatakse erind, siis rakendada sellele teine argument.

4 Erindite semantika

Lihtsale erinditega laiendatud keelele, mille süntaks on toodud joonisel 1, anname semantika kahes astmes. Esiteks toome denotatsioonilise semantika keele puhtale osale ning seejärel anname operatsioonilise semantika kõrvalefekte tekitavale *IO* tasemele.

Kogu programmiks on funktsiooni *main* keha, mis on tüüpi $IO\ ()$. Näiteks võib üks programm välja näha kui:

$$\begin{aligned} &main :: IO\ () \\ &main = getChar \quad \gg\!= (\lambda ch \rightarrow \\ &\qquad putChat\ ch \gg\!= (\lambda() \rightarrow \end{aligned}$$

$return\ ()$
))

Kasutatud funktsioonid on tüüpi:

$$
\begin{array}{ll}
(\gg\!=) & :: IO\ a \to (a \to IO\ b) \to IO\ b \\
return & :: a \to IO\ a \\
getChar & :: IO\ Char \\
putChat & :: Char \to IO\ ()
\end{array}
$$

Esimesed kaks nendest on monaadi kombinaatorid ja kaks viimast on vastavalt tähemärgi konsoolilt lugemiseks ja kirjutamiseks. Eeltoodud koodinäide seega loeb konsoolilt ühe tähemärgi ja kirjutab selle konsoolile tagasi ning seejärel lõpetab töö. Lihtsuse huvides vaatame IO taseme semantika juures ainult neid funktsioone.

4.1 Domeenid

Alustame semantilise domeeni kirjeldamisega. Esmalt defineerime domeeni $[\![\tau]\!]$, mis on seotud iga Haskelli tüübiga τ. Selleks defineerime kõikide võimalike erindite hulga:

$$\mathcal{E} = \{\, DivideByZero, Overflow, UserError, ... \}$$

Kasutame monaadilist notatsiooni:

$$
\begin{array}{rcl}
\mathcal{M}\ t & = & \{\, Ok\ v \,|\, v \in t \,\} \cup \\
 & & \{\, Bad\ s \,|\, s \subseteq \mathcal{E} \,\} \cup \\
 & & \{\, Bad\ (\mathcal{E} \cup \{\, NonTermination \,\})\}
\end{array}
$$

Tavalised väärtused oleme märgendanud konstruktoriga Ok ning eriliste väärtuste hulgad konstruktoriga Bad. Intuitiivselt esitab Bad kõiki erindeid, mida avaldis võib visata. Võib tekkida küsimus, et mida tähendab eriline väärtus $Bad\ \{\,\}$. Hiljem näeme, et see osutub kasulikuks **case** avaldistele tähenduse andmisel.

Järjestuse defineerime hulkade sisalduvuse kaudu informatsiooni kasvamise suunas:

$$s_1 \sqsubseteq s_2 \quad \Leftrightarrow \quad s_1 \supseteq s_2$$

Mida väiksem on erindite hulk seda kindlamalt teame, millist erindit võidakse visata. Vähima erindite hulga $\bot$ defineerime järgmiselt:

$$\bot = \mathcal{E} \cup \{\, NonTermination \,\}$$

Olles defineerinud erindite monaadi saame teisendada Haskelli tüübid erinditega laiendatud domeeni loomulikul viisil:

$$
\begin{array}{rcl}
[\![Int]\!] & = & \mathcal{M}\ \mathcal{Z} \\
[\![\tau_1 \to \tau_2]\!] & = & \mathcal{M}\ ([\![\tau_1]\!] \to [\![\tau_2]\!]) \\
[\![(\tau_1, \tau_2)]\!] & = & \mathcal{M}\ ([\![\tau_1]\!] \times [\![\tau_2]\!])
\end{array}
$$

Piirdume siin ainult nende teisendustega ning ei anna suvaliste algebraliste andmetüüpide teisendamise eeskirja. Idee on asendada Haskelli tavaline monaad uue monaadiga $\mathcal{M}$.

4.2 Kombinaatorid

Alustame primitiivsete operatsioonidele semantika andmisega:

$$[\![e_1 + e_2]\!]\ \rho = \begin{cases} v_1 \oplus v_2 & \text{kui} \quad Ok\ v_1 = [\![e_1]\!]\ \rho \\ & \qquad\ Ok\ v_2 = [\![e_2]\!]\ \rho \\ Bad\ (\mathcal{S}\ ([\![e_1]\!]\ \rho) \cup \mathcal{S}\ ([\![e_2]\!]\ \rho)) & \text{vastasel juhul} \end{cases}$$

Esimene variant on juhuks, kus alamavaldised ei viska erindeid. Sellisel juhul kasutame väärtuste peal operaatorit $\oplus$, mis võib omakorda tagastada erindi. Teist varianti kasutame juhul, kui vähemalt üks argumentidest on eriline. Siis ühendame erindid, mida võidakse visata, abifunktsiooniga $\mathcal{S}$ ja tagastame need. Abifunktsioon $\mathcal{S}$ projekteerib iga $\mathcal{M}\ t$ erindite hulgaks loomulikul viisil:

$$\begin{aligned} \mathcal{S}\ (Ok\ v)\ &= \emptyset \\ \mathcal{S}\ (Bad\ s) &= s \end{aligned}$$

Normaalsetele väärtustele seatakse vastavusse tühi hulk ning erindite hulgale see sama hulk.

Edasi toome semantika erindite viskamise funktsioonile:

$$[\![\mathbf{throw}\ e]\!]\ \rho = \begin{cases} Bad\ s & \text{kui } Bad\ s = [\![e]\!]\ \rho \\ Bad\ \{\, C \,\} & \text{kui } Ok\ C = [\![e]\!]\ \rho \end{cases}$$

Juhul kui argument ei viska erindeid tehakse sellest ühe elemendiline erindite hulk. See on võimalik, kuna **throw** on tüüpi $Exception \to a$ ning seega on C kindlasti erind. Vastasel juhul, kui argument viskab erindi, see lihtsalt propageeritakse edasi.

Nüüd on meil juba võimalik aru saada avaldisest:

$$loop + \mathbf{throw}\ (UserError\ \texttt{"urp"})$$

Väärtusega *loop* esitame lõpmatut tsüklit s.t. $[\![loop]\!]\ \rho = \bot$. Liidetava vasaku poole tähendus on $\bot$, mis on võrdne kõikide erindite hulgaga, seega on ka kogu avaldise tähendus $\bot$. Paneme tähele, et ka *UserError* kuulub hulka $\bot$.

Aplikatsioon ja abstraktsioon tähendused on samuti üpris loomulikud:

$$\begin{aligned} [\![\lambda x.e]\!]\ \rho &= Ok\ (\lambda y.[\![e]\!]\ \rho[y/x]) \\ [\![e_1\ e_2]\!]\ \rho &= \begin{cases} f\ ([\![e_2]\!]\ \rho) & \text{kui } Ok\ f = [\![e_1]\!]\ \rho \\ Bad\ (s \cup \mathcal{S}\ ([\![e_2]\!]\ \rho)) & \text{kui } Bad\ s = [\![e_1]\!]\ \rho \end{cases} \end{aligned}$$

Lambda abstraktsioon on alati tavaline väärtust. Kusjuures sellest tuleneb, et $\lambda x.\bot \neq \bot$ mis on Haskellis tõepoolest nii. Tavalise funktsiooni rakendamine väärtusele käib loomulikul viisil aga huvitavam on olukord, kui funktsioon on eriline väärtus. Sellisel juhul peame funktsiooni väärtustamisel tekkinud erindid ühendama argumendi poolt visatavate erinditega. See on oluline kuna me võime funktsiooni argumente varem väärtustada näiteks agarusanalüüsi tulemusena.

Reeglid konstantide ja konstruktori rakendamiseks on lihtsad, me tagastame mõlemal juhul tavalise väärtuse. Me ei propageeri konstruktori argumentide poolt tekkinud erindeid kuna konstruktori rakendamine on laisk. Muutujad ning püsipunkt saavad tähendused tavalisel viisil:

$$\begin{aligned} [\![k]\!]\ \rho &= Ok\ k \\ [\![C\ e_1\ ...\ e_n]\!]\ \rho &= Ok\ (C\ ([\![e_1]\!]\ \rho)\ ...\ ([\![e_n]\!]\ \rho)) \\ [\![x]\!]\ \rho &= \rho(x) \\ [\![\mathbf{fix}\ e]\!]\ \rho &= \bigsqcup_{k=0}^{\infty} ([\![e]\!]\ \rho)^k(\bot) \end{aligned}$$

Jääb järele vaadata **case** avaldisi, millega on lugu veidike keerulisem. Lihtsam juht on kui argument ei viska erindit ning muster sobib:

$$\begin{array}{rcl} [\![\textbf{case}\ e\ \textbf{of}\ \{p_i \to r_i\}]\!]\ \rho & = & [\![r_i]\!]\ \rho[v/p_i] \\ \text{kus } Ok\ v & = & [\![e]\!]\ \rho \end{array}$$

Avaldis $\rho[v/p_i]$ tähendab, et seame keskkonnas mustri p_i muutujad vastavusse vastavate väärtuse v komponentidega.

Huvitav ja veidi üllatav on "eriline" juht:

$$\begin{array}{rcl} [\![\textbf{case}\ e\ \textbf{of}\ \{p_i \to r_i\}]\!]\ \rho & = & Bad\ (s \cup \bigcup_i (\mathcal{S}\ ([\![r_i]\!]\ \rho[Bad\ \{\}/p_i]))) \\ \text{kus } Bad\ s & = & [\![e]\!]\ \rho \end{array}$$

Paneme tähele, et me ühendame kõik erindid, mida **case** avaldise harud tekitada võivad. Loomulik viis oleks tagastada ainult argumendi poolt tekitatud erindite hulk ning ignoreerida avaldise harusid aga tuleb välja, et sellisel juhul ei läheks mõned **case** avaldiste teisendused läbi. Mustrite p_i muutujatele seame vastavusse erindite hulga $Bad\ \{\}$. Me oleme sunnitud sellist kummalist väärtust kasutama kuna meil lihtsalt ei ole muid võimalusi.

4.3 Operatsiooniline semantika

Sisend väljund kihile anname semantika operatsiooniliselt. Käsitleme primitiive algebraliste andmetüüpidena, mille peal anname teisendusreeglid. Kogu programm saab tähenduse, kui operatsioonilise semantika teisendustee sammude jada.

Monaadi struktuuri teisenduse reeglid anname järgnevalt:

$$\frac{v \to u}{v \gg\!= k \to u \gg\!= k}$$

$$((return\ v) \gg\!= k) \to (k\ v)$$

Esimene reegel võimaldab teisendada operaatori $\gg\!=$ esimest argumenti ning teine lubab funktsiooni rakendada, kui oleme esimese argumendis jõudnud *return* konstruktorini.

Sisend ning väljund operaatorite reeglid näevad välja järgmiselt:

$$\begin{array}{rcl} getChar & \xrightarrow{?c} & return\ c \\ putChar\ c & \xrightarrow{!c} & return\ () \end{array}$$

Tähis "?c" tähendab, et teisendus viiakse läbi, kui konsoolilt on loetud sümbol c. Vastupidiselt tähistab "!c", et teisendus viiakse läbi siis, kui konsoolile on kirjutatud tähemärk c.

Jääb järele anda tähendus *getException* funktsioonile:

$$\begin{array}{rcll} getException\ (Ok\ v) & \to & return\ (OK\ v) & \\ getException\ (Bad\ s) & \to & return\ (BAD\ x) & \\ & \text{kui} & x \in s & \\ getException\ (Bad\ s) & \to & getException\ (Bad\ s) & \\ & \text{kui} & NonTermination \in s & \end{array}$$

Kui argumendina oleme saanud tavalise väärtuse siis lihtsalt tagastame selle *OK* konstruktoris. Eriliste väärtuste korral on meil kaks võimalust. Esiteks võime

kas tagastada suvalise erindi kõikide võimalike erindite hulgast. Teisel juhul, kui *NonTermination* kuulub erindite hulka, teisendame tagasi samasse seisundisse. Teisendusreeglid on meelega mittedeterministlikud. Nimelt, kui argumendina saadakse $\bot$ siis võidakse kas minna lõpmatusse tsüklisse või tagastada mingil suvalisel sammul suvaline erind.

4.4 Kooditeisendused

Paljud programmiteisendused meie semantikaga töötavad korrektselt. Näiteks vaatame kahte avaldist:

$$\begin{array}{rcl} lhs & = & (\textbf{case}\ e\ \textbf{of}\ \{\ True \to f; False \to g\ \})\ x \\ rhs & = & \textbf{case}\ e\ \textbf{of}\ \{\ True \to f\ x; False \to g\ x\ \} \end{array}$$

Võttes $e = \textbf{throw}\ A$, $x = \textbf{throw}\ B$ ja $f = g = \lambda v.1$ saame, et $[\![lhs]\!]\ \rho = Bad\ \{A, B\}$ aga $[\![rhs]\!]\ \rho = Bad\ \{A\}$. Tõepoolest ei ole nende avaldiste semantikad samaväärsed, kuid siiski kehtib, et $lhs \sqsubseteq rhs$. Artiklis [2] on vaieldud, et teisendused, mis suurendavad informatsiooni, see tähendab resultaat programm võib visata vähem erindeid kui originaal, on lubatud. Seega võime avaldise lhs teisendada avaldiseks rhs.

Paraku ei lähe kõik kooditeisendused mida tahaksime korrektselt läbi. Üheks standard [1] teisenduseks on:

$$takeWhile\ p\ (map\ h\ l) \equiv map\ h\ (takeWhile\ (p \circ h)\ l)$$

Võttes $p = null$, $h = tail$ ja $l = \textbf{throw}\ A$, kus l on tüüpi $[[Int]]$ viskab vasakpoolne avaldis erindi A aga parempoolne võib lisaks visata ka "tail: empty list" vea funktsioonist *tail*. Oleme vältinud antud näite põhjalikku seletust kuna see vajab kasutatud abifunktsioonide defineerimist. Sellega võib lugeja tutvuda artiklis [5].

5 Kokkuvõte

Oleme andnud ebatäpsete erindite semantika lihtsale laisale funktsionaalsele programmeerimiskeelele. Toodud semantika on piisavalt lihtne, efektiivselt realiseeritav ja kergesti täiendatav näiteks asünkroonsete erinditega [1]. Vaatamata semantika nõrkustele [5] on see leidnud rakendust populaarsetes Haskelli realisatsioonides nagu näiteks `GHC` ja `Hugs`. Nendes implementatsioonides on kahjuks ebatäpsete erindite tõttu võimalik kirjutada koodi, mis töötab ühte moodi kompileerides optimisatsioonideta aga teisiti, kui kompileerida optimisatsioonidega. Praktikas ei ole need probleemid osutunud oluliseks. Me ei ole siin keskendunud semantika implementatsiooni detailidega. Baasideed efektiivseks realiseerimiseks tuuakse artiklis [4].

[1] Näiteks mõne "deforestation" algoritmi erijuhuna

Viited

[1] S.P. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, pages 47–96, 2001.

[2] S.P. Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 25–36. ACM New York, NY, USA, 1999.

[3] S. Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 96–106. ACM New York, NY, USA, 2006.

[4] Alastair Reid. Handling exceptions in haskell. Technical report, In submitted to Practial Applications of Declarative Languages (PADL'99), 1999.

[5] F.Štenger and J. Voigtländer. *Parametricity for Haskell with imprecise error semantics*. Springer, 2008.

[6] P. Wadler. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.