TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Liisi Kerik

Funktsionaalse programmeerimiskeele liigisüsteem

Magistritöö (30 EAP)

Juhendaja: Härmel Nestra, PhD

Tartu 2018

Funktsionaalse programmeerimiskeele liigisüsteem

Lühikokkuvõte:
Staatilised tüübisüsteemid võimaldavad leida programmidest teatud vigu juba enne koodi käivitamist ja aitavad seega kaasa töökindlama koodi kirjutamisele. Paljud funktsionaalsed keeled, näiteks Haskell ja Idris, on staatiliselt tüübitud.

Mida väljendusrikkam on keele tüübisüsteem, seda rohkem vigu on võimalik juba tüübikontrolli ajal leida. Osad keeled, näiteks Idris, kasutavad sõltuvaid tüüpe, mis muudavad tüübisüsteemi võimsaks aga ka keeruliseks. Hiljuti Haskellis sisse toodud edutamine tõstab tüübisüsteemi väljendusrikkust ilma sõltuvate tüüpideta, võimaldades tüübitaseme andmeid, näiteks tüübitaseme naturaalarvusid ja liste. Koos üldistatud algebraliste andmetüüpidega laseb edutamine programeerijal konstrueerida kasulikke andmestruktuure, näiteks staatilise pikkusega vektoreid ning üldistatud ennikuid.

Käesoleva töö eesmärgiks oli arendada staatiliselt tüübitud funktsionaalne programmeerimiskeel, mille tüübisüsteemi on rikastatud edutamise abil. Tulemuseks olev keel Awful näitab, et edutamise teel saadud liikide ja tüüpidele on palju kasulikke rakendusi ka keeles, milles puuduvad üldistatud algebralised andmetüübid.

Awful kasutab üldistatud algebraliste andmetüüpide abil piiravamat aga oluliselt lihtsamat uut andmetüüpide konstrueerimise viisi. *Hargnevad andmetüübid* võimaldavad konstrueerida paljusid kasulikke struktuure, näiteks staatilise pikkusega vektoreid, mille jaoks Haskellis või Idrises kasutataks üldistatud algebralisi andmetüüpe.

Võtmesõnad: andmetüübid ja struktuurid, deklaratiivne programmeerimine, funktsionaalprogrammeerimine, liigid, polümorfism, edutamine, tüübisüsteemid

CERCS: P175 Informaatika, süsteemiteooria

A Kind System for a Functional Programming Language

Abstract:
Static type systems find some classes of bugs before the program is run, thereby assisting in writing safer code. Many functional languages, for example, Haskell and Idris, are statically typed.

The more expressive a language's type system is, the more bugs can be found during type checking. Some languages, for example, Idris, use dependent types, which result in an expressive albeit complex type system. Promotion, which has been recently introduced in Haskell, improves the expressiveness of the type system without resorting to dependent types by endowing the language with type level data, like type level natural numbers and lists. Together with generalised algebraic data types it allows the programmer to construct, for example, statically sized vectors and tuples of arbitrary length.

The aim of this work was to develop a statically typed functional language with a type system that has been enriched via promotion. The language which is the result of this work, Awful, shows that promoted kinds and types have many useful applications even in a

language that does not have generalised algebraic data types.

Instead of generalised algebraic data types Awful employs a more restrictive but also considerably simpler new way of constructing new data types. *Branching data types* enable construction of many useful data structures, like statically sized vectors, for which we would use generalised algebraic data types in Haskell or Idris.

Keywords: data types and structures, declarative programming, functional programming, kinds, polymorphism, promotion, type systems

CERCS: P175 Informatics, systems theory

Sisukord

Tänusõnad

Autor soovib tänada käesoleva töö valmimisele kaasa aidanud inimesi.

Minu juhendaja Härmel Nestra on juhendamisprotsessi käigus ilmutanud otsatut kannatlikkust, ka siis kui asjad ei ole läinud plaanipäraselt. Tänan teda huvitavate ideede ja nõuannete ning põhjaliku tagasiside eest, ja selle eest, et mul on magistritöö vallas olnud palju loomevabadust ning huvitav ja inspireeriv teema.

Jaak Randmets on see, tänu kellele mul tekkis huvi programmeerimiskeelte vastu ja idee üritada programmeerimiskeelt luua. Ilma temata ei oleks ma enda jaoks seda huvitavat valdkonda avastanud. Tema jagatud teadmised ning nõuanded keele disaini ja teostuse vallas, aga eelkõige nakkav entusiasm programmeerimiskeelte teemal ja toetav suhtumine, on interpretaatori kirjutamisel olnud hindamatuks abiks.

1 Sissejuhatus

Käesoleva töö eesmärgiks oli arendada staatiliselt tüübitud puhas funktsionaalne programmeerimiskeel. Dünaamiliselt tüübitud keeltes, nagu näiteks Python [8] ja Wolfram Language [10], selguvad kõik vead programmi töö käigus. Klassikalisteks näideteks vigadest, mis võivad programmi töö käigus juhtuda, on funktsiooni kutsumine vale tüüpi argumendi peal, massiivi indekseerimine liiga suure arvuga, katse liita arvu ja sõnet, või nulliga jagamine. Staatiliselt tüübitud keeltes, nagu Haskell [5] ja Idris [6], aitab tüübikontroll ennetada osasid programmi jooksutamise käigus tekkivaid vigu ja soovimatut käitumist. Staatiline tüübikontroll muudab koodi tüübiturvalisemaks.

Turingi-täieliku keele puhul ei saa tüübikontroll, mille termineerumine on alati tagatud, ennetada kõiki vigu, ilma samas osasid korrektseid programme kõrvale heitmata. See tähendab, et iga tüübikontrolli-algoritm, mis alati termineerub ja on täielik (kiidab heaks kõik korrektsed programmid), peab lubama osasid ebakorrektseid programme [19]. Mida väljendusrikkam on tüübisüsteem, seda rohkem vigu on võimalik ennetada. Näiteks on osades programmeerimiskeeltes kõik massiivide pikkuse ja indekseerimisega seotud vead dünaamilised, aga osades keeltes on olemas staatilise pikkusega vektorid, mille puhul vektori pikkus on tema tüübi osa ja seega näiteks programm, kus üritatakse leida skalaarkorrutist kahest erineva pikkusega vektorist, tunnistatakse ebakorrektseks juba enne käivitamist.

Kuna staatiline tüübikontroll ennetab osasid programmi töö käigus tekkivaid vigu, vähendab see soovimatu käitumise, programmi kokku jooksmise ja turvaaukude ohtu. Seega on tüübisüsteemid suure praktilise tähtsusega.

Käesoleva töö tulemuseks on staatiliselt tüübitud puhas funktsionaalne programmeerimiskeel nimega Awful. Awfuli tüübisüsteem sisaldab parameetrilist polümorfismi [20] ja ühe muutuja tüübiklasse [22] ning tüübisüsteemi on rikastatud edutamise abil, võttes eeskuju Haskellist [24]. Üks oluline erinevus teistest keeltest on andmetüübid. Haskellis on lisaks tavalistele algebralistele andmetüüpidele olemas üldistatud algebralised andmetüübid. Awfulis puuduvad üldistatud algebralised andmetüübid ja nende asemel tuuakse sisse *hargnevad andmetüübid*, mis on piiravamad, aga samas ka oluliselt lihtsamad.

Käesolevas töös alustame sellest, et kirjutame tüübiklassidest ning võrdleme, millised omadused on Haskelli ja Awfuli tüübiklassidel. Meenutame lühidalt üldistatud algebralisi andmetüüpe, tutvustame tüübitaseme andmeid ja edutamist ning toome näiteid sellest, kuidas edutamine muudab tüübisüsteemi väljendusrikkamaks. Meenutame lühidalt ka sõltuvaid tüüpe ja vaatleme, kuidas need võrdlevad edutamisega.

Töö teises pooles tutvustame käesoleva töö teemaks olevat keelt Awfulit. Spetsifitseerime keele süntaksi. Kirjeldame keelt, põhjendame disainivalikuid ja toome koodinäiteid. Viimases peatükis räägime täiendustest ja parandustest, mida on tulevikus kavas teostada. Peamised plaanitavad muudatused on liikide käsitlemine kategooriatena ning klasside käsitlemine alamliikidena, aga lisaks on kavas ka väiksemad keele kasutajasõbralikkuse ja väljendusrikkuse seisukohast olulised täiendused.

2 *Ad hoc* polümorfism ja tüübiklassid

Parameetriline polümorfism võimaldab käsitleda koos andmeid, millel on sama struktuur aga erinevat tüüpi sisu. Näiteks ei ole parameetrilise polümorfismiga keeles tarvis luua kahte erinevat andmetüüpi tähtede listide ja täisarvude listide jaoks. Selle asemel võib luua parametriseeritud andmetüübi, mille abil saab konstrueerida ükskõik millise sisuga liste. Sellise andmetüübi jaoks saab kirjutada ka parameetriliselt polümorfseid funktsioone. Näiteks käitub listide konkateneerimine samamoodi olenemata sellest, kas konkateneerime tähtede või täisarvude liste.

Parameetriline polümorfism ei ole ainus polümorfismi liik, mida programmeerimiskeeled kasutavad. Teine oluline abstraktsioon on *ad hoc* polümorfism [20].

2.1 *Ad hoc* polümorfism

Kui parameetrilise polümorfismi korral tegeleme andmete ja funktsioonidega, millel on samasugune struktuur ja käitumine olenemata parameetrite väärtusest, siis *ad hoc* polümorfism võimaldab kokku võtta funktsioone, millel on sarnane otstarve, aga erinev käitumine, mis oleneb andmete tüübist.

Üks lihtne näide on aritmeetikatehted. Aritmeetikatehteid on tarvis paljude erinevate struktuuride jaoks. Me tahame liita ja korrutada täisarvusid, jäägiklassiringide elemente, loogikaväärtusi, vektoreid, kompleksarvusid, maatrikseid, harilikke murdusid, polünoome, kvaternione ja pajusid muid tüüpe. Kõigi puhul on liitmise ja korrutamise teostused erinevad, aga on ka olulisi ühiseid jooni (näiteks see, et liitmise ja korrutamise vahel kehtib distributiivsus).

On mugav, kui erinevate tüüpide peal töötavate aritmeetikatehete jaoks on võimalik kasutada samu funktsiooninimesid ja operaatoreid. Kontekstist on enamasti selge, mida liidetakse või korrutatakse, ning seega arvukalt erinevaid funktsiooninimesid või operaatoreid ainult muudab koodi pikemaks ja vähem loetavaks.

Eriti kasulik on *ad hoc* polümorfism siis, kui on tarvis parameetriliselt polümorfse funktsiooni sees erineva teostusega aga sarnase otstarbega funktsioone kasutada. Toome näiteks vektorite liitmise. Olgu meil staatilise pikkusega vektorite tüüp `Array`, mille esimene argument näitab suurust ja teine elementide tüüpi. Olgu tühja vektori konstruktor `Empty_Array` ja mittetühja vektori konstruktor `Construct_Array`. Juhul, kui Haskellis ei oleks *ad hoc* polümorfismi, peaks vektorite liitmise funktsioon, mis töötab erineva sisuga vektorite peal, võtma lisaargumendiks ka vastava tüübi elementide liitmise.

```
add_array :: (t -> t -> t) -> Array n t -> Array n t -> Array n t
add_array _ Empty_Array Empty_Array = Empty_Array
add_array add_t (Construct_Array x a) (Construct_Array y b) =
  Construct_Array (add_t x y) (add_array add_t a b)
```

Lisaargument `add_t`, mis ütleb, kuidas täpselt vektorite elemente liita, tuleks anda igal `add_array` kutsel. Selline lähenemine resulteerub pikas ja halvastiloetavas koodis.

Õnneks on Haskellis ja ka paljudes teistes keeltes olemas *ad hoc* polümorfism, mis võimaldab sellised lisaargumendid likvideerida – erinevaid tüüpe saab liita sama funktsiooninime või operaatoriga ja keel suudab ise tuvastada, millist liitmist rakendada. Haskelli *ad hoc* polümorfism on teostatud tüübiklasside kaudu [22].

2.2 Tüübiklassid Haskellis

Toome alustuseks lihtsa näite tüübiklassist, selgitades, kuidas eelmise jaotise näide vektorite liitmisest tüübiklasside olemasolul muutub. Seejärel loetleme, millised piirangud ja nõuded on seatud Haskelli tüübiklassidele.

Kui tahame luua funktsiooni `add`, mis töötaks erinevate andmetüüpide peal erinevalt, peab ta olema tüübiklassi meetod. Loomegi selle jaoks lihtsa ühe meetodiga tüübiklassi `Additive`. Tüübiklassid võivad sisaldada ka mitut meetodit, aga selles näites piirdume lühiduse huvides ainult ühega.

```
class Additive t where
  add :: t -> t -> t
```

See klassideklaratsioon ütleb, et kui tüüp `t` kuulub klassi `Additive`, on tema jaoks defineeritud meetod `add` tüüpi `t -> t -> t`. Nüüd ütleme, et vektor on selle klassi *esindaja* (ing. k. *instance*).

```
instance Additive t => Additive (Array n t) where
  add Empty_Array Empty_Array = Empty_Array
  add (Construct_Array x a) (Construct_Array y b) =
    Construct_Array (add x y) (add a b)
```

Esimene rida ütleb, et vektor üle tüübi `t` on liidetav parajasti siis, kui tüüp `t` on liidetav. Järgmised kolm rida defineerivad meetodi `add` teostuse vektorite jaoks. Erinevus võrreldes eelmise versiooniga on see, et nüüd ei ole enam vaja pikka tüübispetsiifilist funktsiooninime `add_array` ega lisaargumenti `add_t`. Kui varem oleksime pidanud täisarvuliste vektorite liitmiseks kutsuma funktsiooni `add_array (+)`, siis nüüd piisab sellest, et kutsume lihtsalt funktsiooni `add` ja Haskelli kompilaator tuletab ise lähtuvalt kontekstist, millist `add` meetodi teostust on mõeldud ja milliseid lisaargumente ta vajab.

Tasub tähele panna, et mittetühjade vektorite liitmise avaldises kasutatud meetodi `add` kutsed tähistavad tegelikult kahte erinevat väärtust: esimene tähistab vektori elementide liitmist (näite eelmises versioonis argument `add_t`), teine tähistab vektorite liitmist (näite eelmises versioonis `add_array add_t`).

2.2.1 Pärilus

Tüübiklassid, nagu ka klassid objektorienteeritud keeltes, võivad üksteist pärida. Toome näiteks Haskelli klassi `Applicative`, mille esindajad peavad alati kuuluma ka klassi `Functor`.

```
class Functor f where
  fmap :: (t -> u) -> f t -> f u
class Functor f => Applicative f where
  (<*>) :: f (t -> u) -> f t -> f u
  pure  :: t -> f t
```

Pärilus ei tähenda, et klassi `Applicative` esindajad defineeritakse kompilaatori poolt automaatselt ka klassi `Functor` esindajateks. See tähendab, et kui tüüp defineeritakse klassi `Applicative` esindajaks, peab ta olema defineeritud ka klassi `Functor` esindajaks – programmeerija peab seda ise tegema (või kasutama võtmesõna `deriving` vastava esinaja tuletamiseks).

```
instance Functor [] where
  fmap f x =
    case x of
      [] -> []
      y : z -> f y : fmap f z
instance Applicative [] where
  x <*> y =
    case x of
      [] -> []
      f : z -> fmap f y ++ z <*> y
  pure x = [x]
```

Rohkem näiteid tüübiklassidest, kasutades juba Awfuli süntaksit, on peatükis 5. Järgnevates alamjaotistes kirjeldame detailsemalt, milliseid võimalusi Haskelli tüübiklassid pakuvad ja millised piirangud on neile seatud.

2.2.2 Tüübiklassid Haskell 2010 standardis

Haskell 2010 standard spetsifitseerib tüübiklassid järgnevalt [5].

- Klassi deklaratsioon on kujul `class` cx `=>` C u `where` $cdecls$, kus C on uue klassi nimi, u on tüübimuutuja, cx spetsifitseerib, mis klassid klass C pärib (teisisõnu spetsifitseerib C ülemklassid) ja $cdecls$ sisaldab klassi meetodeid. Iga meetodi deklaratsioon on kujul v_i `::` cx_i `=>` t_i, kus v_i on meetodi nimi, t_i on meetodi tüüp ja cx_i on tüübimuutujatele seatavate kitsenduste nimekiri.

- Ülemklasse spetsifitseerides tohib kasutada ainult tüübimuutujat u, mis on klassi deklareerides sisse toodud, ja mitte ühtegi teist tüüpi ega ka mitte mõnda uut tüübimuutujat. Toome kaks näidet klassidest, mis ei vasta antud nõudele ja on seega ebakorrektsed.

```
class Bug Int => Bug' t
class Bug u => Bug' t
```

- Pärilussuhetes ei tohi tekkida tsüklit. Klasside pärilussuhted moodustavad suunatud atsüklilise graafi.

- Haskell võimaldab mitmest pärilust.

```
class (A t, B t) => C t
```

- Tüüp t_i peab sisaldama klassi päises sisse toodud tüübimuutujat u. Põhjuseks on see, et vastasel juhul ei ole võimalik klassi meetodi kutsel tuletada, millise tüübi jaoks mõeldud teostust kasutada. Toome näite, mis juhtuks, kui lubada meetodite tüüpe, mis ei vasta sellele kitsendusele.

```
class Bug t where
  bug :: Int
instance Bug Char where
  bug = 0
instance Bug Int where
  bug = 1
```

  Kui kasutatakse meetodit `bug`, ei ole tüüpimisalgoritmil võimalik teada, kas tegu on meetodi `bug` teostusega `Char`, `Int` või mõne muu tüübi jaoks. Ka eksplitsiitne tüübiannotatsioon ei aita, sest meetodi tüüp on sama kõigi teostuste korral. Seetõttu ei oleks sellist avaldist standartses Haskellis võimalik väärtustada.

- Kitsenduste nimekiri cx_i tohib kitsendada ainult tüübimuutujaid, mis on sisse toodud tüübis t_i. Ei tohi kitsendada tüüpe, mis ei ole tüübimuutujad.

```
class Bug t where
  bug :: Bug' Int => t
```

  Ei tohi tuua kitsendustes sisse tüübimuutujaid, mida tüübis t ei esine.

```
class Bug t where
  bug :: Bug' u => t
```

Ei tohi kitsendada ka klassi päises deklareeritud tüübimuutujat.

```
class Bug t where
  bug :: Bug' t => t
```

- *cdecls* võib sisaldada ka operaatorite assotsieerumissuuna ja prioriteedi deklaratsioone (ing. k. *fixity declaration*) ning vaikedefinitsioone klassi meetodite jaoks.

Klasside esindajad on spetsifitseeritud järgnevalt [5].

- Kui meil on klass nimega C, siis tema esindaja on kujul `instance` cx' `=>` C (T $u_1 \ldots u_k$) `where {`d`}`, kus $k \in \mathbb{N}_0$. Kitsenduste nimekiri cx' võimaldab kitsendada tüübikonstruktorile T argumentiks antud tüübimuutujaid $u_1 \ldots u_k$. Plokk d sisaldab klassi meetodite definitsioone tüübi T jaoks.
- Programmis ei tohi olla kahte esindajat, kus on sama klass C ja tüüp T.
- Tüübikonstruktori T argumendid $u_1 \ldots u_k$ peavad olema tüübimuutujad. Argumendiks ei tohi olla midagi muud, näiteks on keelatud selline esindaja:

  ```
  instance Bug [Char]
  ```

  Nimetatud kitsendusega on seotud ka reegel, et klassi esindajaks ei tohi defineerida tüübisünonüümi. T peab olema tüübikonstruktor. Näiteks ei tohi klassi esindaja olla `String`, sest `String` on sünonüüm tüübile `[Char]` ning programmis võib olla ka klassi esindaja tüübi `[t]` jaoks, mis juhul tekiks kahe esindaja vahel kattuvus.
- Tüübimuutujad $u_1 \ldots u_k$ peavad olema erinevad. Näiteks on keelatud selline esindaja:

  ```
  instance Bug (t, t)
  ```

- Tüübi T $u_1 \ldots u_k$ liik peab olema sama, mis klassi C tüübimuutuja u liik. Liikidest räägime täpsemalt jaotises 3.3. Seni aga toome ühe lihtsa näite liikide sobimatusest. Klass `Functor` on mõeldud ühe argumendiga tüübikonstruktorite jaoks.

  ```
  class Functor f where
    fmap :: (t -> u) -> f t -> f u
  ```

Seda on näha meetodi `fmap` signatuurist, kus tüübimuutuja `f` võtab ühe argumendi. Seega peavad klassi `Functor` esindajad olema tüübikonstruktorid, mis võtavad täpselt ühe argumendi (näiteks `Maybe`, listi tüübikonstruktor, aga ka tüübikonstruktor `Either t`). Järgmised kaks esindajat on keelatud, sest tüübid `Int` ja `[t]` ei võta piisavalt argumente:

```
instance Functor Int
instance Functor [t]
```

Järgmine esindaja on aga keelatud, sest `Either`, millele ei ole argumendiks antud ühtegi tüübimuutujat, võtab mitte üks vaid kaks argumenti, seega liiga palju.

```
instance Functor Either
```

Jaotises 3.3 selgitame, kuidas liik võib olla ka palju keerulisem kui tüübikonstruktori argumentide arv, aga praeguseks piirdume ülalpool toodud lihtsate näitega.

- Tüüp $T\ u_1 \ldots u_k$ kitsendustega cx' peab olema ka kõigi C ülemklasside esindaja. Näiteks kuna klass `Functor` on klassi `Applicative` ülemklass, ei tohi kirjutada klassi `Applicative` esindajat ilma vastava `Functor` esindajata. Lisaks tähendab see reegel, et ülemklassi esindajale ei tohi seada rangemaid kitsendusi kui alamklassi (päriva klassi) esindajale. Näiteks on keelatud järgmine kood:

  ```
  class Bug t => Bug' t
  instance Ord t => Bug [t]
  instance Bug' [t]
  ```

  Selline kood ei ole lubatud, sest klass `Bug` on klassi `Bug'` ülemklass ning alati, kui tüübi peal kehtib kitsendus `Bug'`, peaks järelikult kehtima ka `Bug`.

- Kitsenduste nimekiri cx' tohib kitsendada ainult tüübimuutujaid $u_1 \ldots u_k$. Keelatud on kitsendada tüüpe, mis ei ole tüübimuutujad.

  ```
  instance Bug Int => Bug' [t]
  ```

  Samuti on keelatud kitsendustes uusi tüübimuutujaid sisse tuua.

  ```
  instance Bug u => Bug' [t]
  ```

- Plokk d tohib sisaldada ainult klassi C meetodite definitsioone ning muud definitsioonid on keelatud. Haskell lubab osade meetodite definitsioonid kirjutamata jätta. Juhul, kui defineerimata meetodi jaoks on olemas vaiketeostus, kasutatakse seda. Kui ei ole, on meetodi väärtus antud tüübi jaoks `undefined`.

- Ploki d definitsioonid peavad vastama klassi deklaratsioonis toodud tüübisignatuuridele. Sealhulgas tähendab see, et meetodi v_i definitsioon tüübi $T\ u_1 \ldots u_k$ jaoks ei tohi nõuda rangemaid kitsendusi t_i kui on määratud klassi deklaratsioonis tüübisignatuuris, kitsenduste nimekirjas cx_i. Samuti ei tohi definitsioon nõuda rangemaid kitsendusi tüübimuutujatele $u_1 \ldots u_k$ kui on sisse toodud kitsenduste nimekirjas cx' esindaja deklaratsiooni päises. Toome kaks näidet esindajatest, mis ei läbi tüübikontrolli, kuna definitsiooni sisust järelduksid liiga ranged kitsendused.

```
class Bug t where
  bug :: t -> f t
instance Bug [t] where
  bug = pure
```

Loome klassi `Bug`, millel on meetod `bug`. Meetodil `bug` on kitsendamata tüübimuutuja `f`. Esindaja `Bug [t]` ei läbi tüübikontrolli, sest funktsioon `pure` eeldab klassi `Applicative`, aga klassi deklaratsioonis ei ole tüübimuutuja `f` kitsendamist ette nähtud.

```
class Bug t where
  bug :: t -> t
instance Bug [t] where
  bug = sort
```

Ka see näide ei läbi tüübikontrolli, sest meetodi teostus eeldab mitte-ettenähtud kitsendusi esindaja deklaratsiooni päises sisse toodud tüübimuutujale. Funktsioon `sort` eeldab, et listi elemendid on võrreldavad, aga samas ei ole seda eeldust sisse toodud. Viga saab parandada, muutes eksplitsiitseks eelduse, et tüübimuutuja `t` kuulub klassi `Ord`.

```
instance Ord t => Bug [t] where
  bug = sort
```

- Ploki d definitsioonidele on keelatud kirjutada tüübisignatuure, sel lihtsal põhjusel, et klassi meetodite tüübisignatuurid on toodud juba klassi deklaratsioonis ning esindajat defineerides ei ole need seega vajalikud.

2.2.3 Tüübiklassidega seotud laiendused

Haskellil on ka palju laiendusi, mis võimaldavad tüübiklasside vallas paindlikkust lisada. Loetleme siinkohal mõned tüübiklassidega seotud laiendused [4]. Me ei hakka detailselt käsitlema nimetatud laienduste kasutusvõimalusi, vaid lihtsalt mainime lühidalt, milliseid piiranguid need eemaldavad.

- `ConstrainedClassMethods` võimaldab klassi meetodite tüübisignatuurides kitsendada ka klassi päises sisse toodud tüübimuutujat.

  ```
  class C t where
    f :: Ord t => t -> t -> t
  ```

- `DefaultSignatures` võimaldab kirjutada erineva signatuuri klassi meetodile ja vaiketeostusele. See on kasulik näiteks juhuks, kui tahta kirjutada vaiketeostus, mis töötab ainult teatud kitsendatud juhtudel.

- `FlexibleContexts` võtab ära piirangu, et klassi kontekst cx on lihtsalt ülemklasside nimekiri. See võimaldab kitsenduste nimekirjas kasutada ka keerulisemaid avaldisi, mis ei pea koosnema tingimata ainult tüübimuutujatest vaid võivad sisaldada ka muid tüüpe.

- `FlexibleInstances` võimaldab luua esindajaid, kus tüüp ei koosne ainult konstruktorist ja tüübimuutujatest. Näiteks saab sellisel juhul klassi esindajaks defineerida tüübi `[Char]`.

  ```
  instance C [Char]
  ```

- `IncoherentInstances` ja `OverlappingInstances` on kaks aegunud laiendust, mis lubavad kattuvaid esindajaid.

- Standartses Haskellis on tüübiklassid üheparameetrilised. `MultiParamTypeClasses` lubab suvalise arvu parameetritega – sealhulgas ka ilma parameetrita – tüübiklasse.

  ```
  class C
  class B t u
  ```

- Laiendus `FunctionalDependencies` on mõeldud loomaks mitmeparameetrilisi tüübiklasse, kus osade tüübiparameetrite väärtused on üheselt määratud teiste tüübiparameetrite poolt. Näiteks oletame, et kirjutame algebrateeki ja soovime defineerida üldistatud korrutamisfunktsiooni, mis võib korrutada erinevat tüüpi argumente, kusjuures tulemuse tüüp ei pea tingimata kummagi argumendi omaga ühtima. Antud näites on korrutise tulemuse tüüp `v` üheselt määratud argumentide tüüpide `t` ja `u` poolt.

```
class Mult t u v | t u -> v where
  mult :: t -> u -> v
instance Num t => Mult (Array m t) (Array n t) (Matrix m n t)
instance Num t => Mult (Array n t) (Matrix n m t) (Array m t)
instance Num t => Mult (Matrix m n t) (Array n t) (Array m t)
instance Num t => Mult (Matrix l m t) (Matrix m n t) (Matrix l n t)
```

Kuna laiendus `FunctionalDependencies` eeldab mitmeparameetrilisi tüübiklasse, võetakse sellega koos automaatselt kasutusse ka `MultiParamTypeClasses`.

- `InstanceSigs` võimaldab tüübisignatuure esindaja sees meetodite teostuste juures. Üks võimalik kasutus on näiteks täiendav dokumentatsioon esindajate sees.
- `NullaryTypeClasses` on aegunud laiendus, mis võimaldab parameetriteta tüübiklasse. Selle asemel saab kasutada üldisemat laiendust `MultiParamTypeClasses`.
- `TypeSynonymInstances` võimaldab tüübisünonüüme, näiteks `String`, klasside esindajateks defineerida.
- `UndecidableInstances` ja `UndecidableSuperClasses` lubavad vastavalt esindajaid ja klasse, mille olemasolu võib resulteeruda mittetermineeruvas tüübikontrollis.

2.3 Tüübiklassid Awfulis

Ka käesoleva töö teemaks olev keel Awful võimaldab *ad hoc* polümorfismi tüübiklasside kaudu. Awful on tüübiklasside ja esindajate osas erinev Haskell 2010 standardist järgmistes aspektides:

- Lubatud on ainult ühene pärilus. Mitmene pärilus on plaanis lisada edasise töö käigus.
- Klassi meetodid ei pea sisaldama klassi päises sisse toodud tüübimuutujat. Toome näite olukorrast, kus selline meetod on praktikas kasulik: täisarvuline jagamine, mille puhul on garanteeritud, et jagaja ei ole null, sest jagaja on tüübitaseme naturaalarv. (Tüübitaseme naturaalarvudest räägime peatükis 3.) Haskellis on selline meetod keelatud.

```
class Nonzero (n :: Nat) where
  div' :: Integer -> Integer
```

Awfulis on analoogne meetod lubatud.

```
Class Nonzero{N : !Nat}(Div' : Int -> Int)
```

Erinevalt Haskellist, kus tüübiargumentide eksplitsiitne edastamine ei ole standardi osa ja vajab keelelaiendust, on Awfuli puhul algusest peale arvestatud tüübiargumentide eksplitsiitse edastamise vajadusega. Seega ei ole Awfulis piirangut, et meetodi teostust peaks saama tema tüübist tuletada. Tüübiargumentide eksplitsiitsest edastamisest räägime täpsemalt alamjaotises 5.7.5.

- Haskell võimaldab osade tüübiklasside esindajaid (näiteks `Eq`, `Ord` ja `Show`) automaatselt tuletada. Awfulis selline võimalus puudub.
- Ei ole võimalust lisada vaikedefinitsioone klassi meetodite jaoks. Kuna tegu on keele väljendusrikkuse seisukohast mittekriitilise võimalusega, ei ole seda hetkel kavas. Awfulis sisaldab klass ainult neid meetodeid, mis on minimaalselt vajalikud, ning kõik ülejäänud funktsioonid, mis saab nende kaudu defineerida, tuleb kirjutada väljaspool klassi. Selle tõttu ei luba Awful, erinevalt Haskellist, esindaja defineerimisel ühtegi meetodit kirjutamata jätta. Kuna puuduvad vaikedefinitsioonid, on defineerimata meetod alati viga.
- Awful nõuab, et esindajat teostades oleks meetodid samas järjekorras, mis klassi deklaratsiooni sees. Tegu oli teostuse lihtsusest lähtuva otsusega.

3 Tüübitaseme andmed ja edutamine

Selles peatükis refereerime, mis on edutamine ning kuidas see muudab tüübisüsteemi väljendusrikkamaks ja aitab kirjutada tüübiturvalisemat koodi. Kuna käesoleva töö autori loodud keelt tutvustatakse alles peatükkides 4 ja 5, ning kuna Haskell on esimene keel, mis võimaldab edutamist [24], on antud peatükis edutamise põhimõtete selgitamiseks kasutatud Haskelli süntaksit.

Sageli on tüübiturvalisema koodi kirjutamiseks kasulik, kui saab tüüpide tasemel väärtusi kasutada. Näiteks võimaldavad tüübitaseme naturaalarvud luua vektoreid, mille pikkus, erinevalt listide omast, on juba tüübikontrolli ajal teada. Vektorite pikkusega seotud vead tulevad sellisel juhul ilmsiks juba tüübikontrolli käigus, enne väärtustamist või kompileerimist.

See võimaldab kirjutada näiteks vektorite liitmise või vektori viimase elemendi leidmise funktsioone tüübiturvaliselt. Juhul, kui kirjutada neid funktsioone listide abil, ei ole kuidagi tagatud, et programmeerija ei kutsu vektorite liitmise funktsiooni välja kahe erineva pikkusega listi peal või ei kutsu viimase elemendi leidmise funktsiooni tühja listi peal. Staatilise pikkusega vektorite puhul on sellised vead tänu tüübisüsteemile välistatud.

Lisaks kannavad tüübid kasulikku teavet funktsioonide sisendite ja väljundite kohta, ning mida väljendusrikkam tüübisüsteem on seda detailsemalt saavad funktsioonide tüübid funktsioonide sisu kirjeldada. Näiteks juhul kui keeles on võimalik kasutada tüübitaseme naturaalarve, saab vektorite liitmise funktsiooni tüüpi vaadates kohe teada, et kaks vektorit peavad olema sama pikad. Vaadates vektori viimase elemendi leidmise funktsiooni tüüpi, on kohe aru saada, et vektor ei tohi olla pikkusega null, kuna esimene tüübiargument on `Next n`.

```
add_array :: Num t => Array n t -> Array n t -> Array n t
last_elem :: Array (Next n) t => t
```

Tüübitaseme andmed parandavad tüübiturvalisust, vähendavad korduva koodi hulka ning lisaks aitavad kaasa sellele, et tüübisüsteem oleks väljendusrikkam ja tüübid dokumenteeriks paremini funktsioonide sisu.

3.1 Üldistatud algebralised andmetüübid

Käesoleva peatüki näidetes kasutame üldistatud algebralisi andmetüüpe. Seega enne tüübitaseme andmete juurde liikumist selgitame lühidalt üldistatud algebralisi andmetüüpe ja toome mõned näited.

Tavalise algebralise andmetüübi puhul järgneb tüübi nimele tüübimuutujate loetelu ja seejärel konstruktorid.

```
data Maybe t = Nothing | Just t
```

Konstruktorid ei võimalda resulteeruvat tüüpi kuidagi kitsendada. Kui tüübil D, millel on n tüübiargumenti, on konstruktor C argumentidega tüüpi $T_1 \ldots T_m$, siis C tüüp on T_1 `->` $\ldots$ `->` T_m `->` $D\ t_1 \ldots t_n$. Tüübid $T_1 \ldots T_m$ võivad sisaldada ainult tüübimuutujaid $t_1 \ldots t_n$ ning tüübimuutujad $t_1 \ldots t_n$ peavad kõik olema erinevad. Näiteks `Maybe` puhul on andmekonstruktori `Nothing` tüüp `Maybe t` ja andmekonstruktori `Just` tüüp `t -> Maybe t`. Kummagi andmekonstruktori tulemuse tüüp on `Maybe t`, mitte näiteks `Maybe Int` või `Maybe [t]`.

Üldistatud algebralise andmetüüpide puhul [11] see kitsendus enam ei kehti. Näitame alustuseks, kuidas tüüpi `Maybe` üldistatud algebraliste andmetüüpide süntaksiga kirja panna:

```
data Maybe :: * -> * where
  Nothing :: Maybe t
  Just :: t -> Maybe t
```

Esimene rida näitab, et `Maybe` on tüübikonstruktor, mis võtab ühe argumendi. Seejärel kirjutame andmekonstruktorid. Erinevalt tavalisest algebralisest andmetüübist on üldistatud algebralise andmetüübi puhul vajalik märkida ka tulemuse tüüp, antud juhul `Maybe t`. Andmekonstruktori tulemuse tüübis ei pea aga tüübikonstruktor olema rakendatud ainult hulgale erinevatele tüübimuutujatele – ta võib olla rakendatud ka teistele tüüpidele, näiteks `Int` või `[t]`, ning antud tüüpides esinevad tüübimuutujad ei pea omavahel erinema.

```
data Example :: * -> * -> * where
  Constr_0 :: Example Int [t]
  Constr_1 :: Example t t
```

Üldistatud algebraliste andmetüüpide ja nende rakenduste põhjalik käsitlus jääb väljapoole antud töö skoopi. Seetõttu piirdume siinkohal ainult selle lühikese selgitusega ning jätkame tüübitaseme andmete ja nende võimalike kasutusjuhtude tutvustamisega.

3.2 Tüübitaseme andmed

Mõnedes keeltes, näiteks Idris [6], saab tüübitaseme andmete saavutamiseks kasutada sõltuvaid tüüpe. Sõltuvatest tüüpidest anname lühikese ülevaate jaotises 3.6. Keeles, kus sõltuvaid tüüpe ei ole, on tüübitaseme andmeid võimalik kirjutada algebraliste andmetüüpide abil [24]. Tüübitaseme naturaalarvud näeksid Haskelli algebralisi andmetüüpe kasutades välja sellised:

```
data Zr
data Next n
```

Sarnasel viisil saab teostada ka näiteks tüübitaseme liste:

```
data Empty_List
data Construct_List t l
```

Tegu on tühjade tüüpidega, millel puuduvad andmekonstruktorid ja mis on mõeldud spetsiaalselt tüübitaseme andmete rolli täitmiseks.

Toome mõned lihtsad näited andmetüüpidest, mille konstrueerimiseks on vaja tüübitaseme andmeid. Üldistatud algebraliste andmetüüpide abil saab tüübitaseme andmeid tarvitada loomaks näiteks vektoreid, mille pikkus on tüübi tasemel teada. Kirjutame kaks konstruktorit: üks tühja vektori jaoks, ja teine vektori jaoks pikkusega `n + 1`, mis võtab argumendiks pea ja `n`-elemendilise saba.

```
data Array :: * -> * -> * where
  Empty_Array :: Array Zr t
  Construct_Array :: t -> Array n t -> Array (Next n) t
```

Tüübitaseme naturaalarvusid saab kasutada ka selleks, et luua suvalise arvu muutujatega Boole'i funktsiooni tüüp. Null muutuja Boole'i funktsioon on üks konstant. `n + 1` muutuja Boole'i funktsiooni saab esitada kahe `n` muutuja Boole'i funktsiooni abil (jääkfunktsioonid, mille saame, asendades esimese muutuja vastavalt konstandiga `False` või `True`).

```
data Fun :: * -> * where
  Constant :: Bool -> Fun Zr
  Branch :: Fun n -> Fun n -> Fun (Next n)
```

Tüüpide listi abil saame luua üldistatud ennikuid, mis sarnanevad struktuuri poolest üleeelmises näites toodud vektoritele, selle vahega, et nad võivad sisaldada erinevat tüüpi andmeid.

```
data Tuple :: * -> * where
  Empty_Tuple :: Tuple Empty_List
  Construct_Tuple :: t -> Tuple l -> Tuple (Construct_List t l)
```

Kui saame luua tüübitaseme listi naturaalarvudest, on seda võimalik kasutada loomaks mitmemõõtmelist tabelit, mille kõik mõõtmed on tüübi tasemel teada.

```
data Array' :: * -> * -> * where
  Empty_Array' :: t -> Array' Empty_List t
  Construct_Array' ::
    Array n (Array' l t) -> Array' (Construct_List n l) t
```

Tüübitaseme andmed on kasulikud selleks, et saaks kirjutada tüübiturvalisemaid funktsioone. Näiteks tahame kirjutada vektorite liitmise funktsiooni. Kui kasutada vektorite esitamiseks liste, siis on tarvis arvestada ohuga, et programmeerija annab funktsiooni sisendiks erineva pikkusega listid. Erineva pikkusega listid resulteeruvad defineerimata käitumises või erindis programmi töö käigus ning tüüpimise ajal ei ole seda võimalik ennetavalt tuvastada.

Tüübitaseme andmete olemasolul saab vektorite liitmise funktsiooni kirjutada eelpool näiteks toodud `Array` tüübi peal. See tagab, et iga vektorite liitmise funktsiooni kutse puhul kontrollitakse juba tüüpimise ajal, et mõlemad argumendid oleks sama pikad.

3.3 Tüübid, liigid ja sordid

Eelmises jaotises kirjeldatud viisil teostatud tüübitaseme andmetel on oluline puudus. Tüübisüsteem ei ole piisavalt väljendusrikas kirjeldamaks programmeerija tegelikke kavatsusi. Ei ole võimalust seada piirangut, et kohas kus võib esineda näiteks tüüp `Char` või `Int`, ei tohi esineda naturaalarvuline tüüp, ja vastupidi – ehk tüübid ei ole piisavalt tugevalt tüübitud [24]. Tüübikontrollist lähevad läbi koodinäited, mis ei ole programmeerija tegelike kavatsustega kooskõlas, näiteks:

```
data Array :: * -> * -> * where
  Empty_Array :: Array Zr t
  Construct_Array :: t -> Array n t -> Array (Next t) n
```

Siin on konstruktori `Construct_Array` tulemuse tüübis vahetusse läinud tüübimuutujad `n` ja `t`. Antud koodinäide ei tohiks tegelikult tüübikontrolli läbida, sest need tüübimuutujad on täiesti erineva otstarbega: `n` kirjeldab vektori pikkust ja `t` vektori elementide sisu. Vältimaks sedalaadi vigu, oleks vaja tüübisüsteemi, mis on piisavalt võimas kirjeldamaks, et tüübikonstruktori `Array` esimene argument peab olema naturaalarv ja teine tavaline tüüp, ja et tüübikonstruktorit `Next` saab rakendada ainult naturaalarvule.

Selleks, et tüübitaseme naturaalarvusid, tüübitaseme liste ja teisi tüübitaseme andmeid saaks kasutada turvalisemalt, on vaja võimsamat liigisüsteemi. Liigid on tüüpide jaoks sama, mis tüübid väärtuste jaoks – tegu on tüüpide tüüpidega. Ka liikidel võivad omakorda olla tüübid ja neid nimetatakse sortideks [19].

Tuntud liigikonstruktorid Haskellis on näiteks liik `*` ja liigikonstruktor `->`, mille kaudu konstrueeritakse tüübikonstruktorite liike. Need tagavad, et tüübikonstruktoreid rakendataks õigele arvule õigetele argumentidele, näiteks et programmeerija ei üritaks kirjutada tüüpi `Int List`.

Võimsam liigisüsteem, mis sisaldaks lisaks eelmainitud liikidele ka kõigi tüübitaseme andmete liike (näiteks naturaalarvude liiki ja listide liiki), aitaks vältida tüüpide ja tüübimuutujate ebasobivat kasutust, mida nägime ülalpool toodud näites.

See, et jaotises 3.2 kirjeldatud viis tüübitaseme andmeid luua ei ole piisavalt tüübiturvaline, ei ole selle ainus nõrk külg. Lisaks eelmainitud peamisele puudusele on oluline miinus ka see, et programmeerija peab tüüpide tasemel duplitseerima andmeid, mis on väärtuste tasemel tegelikult juba olemas. Näiteks on tüübitaseme naturaalarvud samasuguse struktuuriga mis väärtuste taseme naturaalarvud ja tüübitaseme naturaalarvude eraldi kirjutamine on korduv kood.

Edutamine lahendab mõlemad probleemid, muutes tüübi- ja liigisüsteemi võimsamaks ning tagades, et programmeerija ei peaks ise käsitsi tüübitaseme andmeid kirjutama. Järgmistes jaotistes räägimegi sellest, kuidas edutamine töötab, kuidas see muudab tüübisüsteemi ja millised andmetüübid on edutatavad.

3.4 Edutamine

Edutamine tõstab sobivad andmetüübid automaatselt „üks tase ülespoole“. Andmetüübi edutamisel tehakse tüübikonstruktorist liigikonstruktor, kusjuures tüübiparameetritest saavad liigiparameetrid. Andmekonstruktoritest saavad vastava liigi tüübikonstruktorid, kusjuures argumentide tüüpidest saavad tüübikonstruktorite argumentide liigid [24].

3.4.1 Edutamise näiteid

Võtame esimeseks näiteks lihtsa andmetüübi, kus puudub polümorfism: naturaalarvud.

```
data Nat = Zr | Next Nat
```

Ilma edutamiseta saab siit tüübi `Nat :: *` ning kaks andmekonstruktorit: `Zr` tüüpi `Nat` ja `Next` tüüpi `Nat -> Nat`. Edutamise olemasolul saab lisaks ka liigi `Nat` ning kaks tüübikonstruktorit: `Zr` liiki `Nat` ja `Next` liiki `Nat -> Nat`.

Vaatleme keerulisemat näidet: liste.

```
data List (t :: *) = Empty_List | Construct_List t (List t)
```

Ilma edutamiseta saab sellest andmetüübist tüübikonstruktori `List :: * -> *` ning kaks andmekonstruktorit: `Empty_List` tüüpi `List (t :: *)` ja `Construct_List` tüüpi `(t :: *) -> List t -> List t`. Edutamise olemasolul saab sellest andmetüübist lisaks ka ühe argumendiga liigikonstruktori `List` ning kaks tüübikonstruktorit: `Empty_List` liiki `List k` ja `Construct_List` liiki `k -> List k -> List k`.

Listide puhul tuleb mängu polümorfism. Parametriseeritud tüüp muutub parametriseeritud liigiks. Tüübipolümorfsetest andmekonstruktoritest saavad liigipolümorfsed tüübikonstruktorid.

Standartne Haskell edutamist ei sisalda ning edutamise jaoks on vaja kasutada keele laiendusi. Vajalikud laiendused on `DataKinds` (edutamine) ja `KindSignatures` (liigisignatuurid) ning, kuna paljud edutamise rakendused nõuavad üldistatud algebralisi andmetüüpe, ka `GADTs`. Liigipolümorfismi jaoks on lisaks vajalik laiendus `PolyKinds`.

3.4.2 Andmetüübid, mida saab edutada

Edutamisele on mõistlik seada teatud piirangud ning edutada ainult osasid andmetüüpe, mitte kõiki. Piirangud on seotud sooviga kasutada edutamise võimalusi (näiteks tüübitaseme naturaalarvusid ja liste) ilma muutmata tüübisüsteemi keerulisemaks kui hädapärast vajalik. Loetleme siinkohal piirangud, mida on kasutatud Haskellis [24].

- Ei edutata üldistatud algebraliseid andmetüüpe, kuna see muudaks liigisüsteemi keerulisemaks tuues sisse liikide võrdsuskitsendused (ing. k. *equality constraints*).

- Ei edutata primitiive, näiteks tähti ja täisarvusid. Primitiivide edutamisega ei kaasne tegelikult põhimõttelisi raskusi, küll aga võib neile praktiliste rakenduste leidmine nõuda teisi keele täiendusi, mis ei ole sama lihtsad. Haskellis on primitiivide edutamata jätmise põhjuseks see, et keele laienduse autorite arvates ei ole primitiivide edutamine kasulik, kuni ei ole võimalik edutada tüüpie tasemele ka tehteid primitiividega.

- Ei edutata andmetüüpe, millel on tüübiparameetreid muud liiki kui liik `*`. Juhul kui edutada ainult sellele kitsendusele vastavaid andmetüüpe, on sortide süsteem äärmiselt lihtne. Kõik liigimuutujad on ühte sorti. See tähendab, et liigipolümorfismi korral ei pea liigimuutujate sorte märkima. Liigikonstruktori sort on lihtsalt naturaalarv, mis näitab, mitut argumenti liigikonstruktor vajab.

  Juhul, kui soovida edutada andmetüüpe, mis sisaldaks tüübiparameetrites mitte ainult liiki `*` vaid ka liigikonstruktorit `->`, ei ole sellele põhimõttelisi takistusi, aga selline täiendus nõuaks keerulisemat sortide süsteemi. Liigimuutujad võiksid sellisel juhul olla erinevat sorti. Seega oleks vaja liigipolümorfismi korral liigimuutujate sorte märkida. Edutamise teostamine muutuks keerulisemaks, samas kui vajadus sellise võimsusega liigisüsteemi järele on küsitav.

  Kui tahta minna veel kaugemale ja edutada andmetüüpe, mis on parametriseeritud üle omakorda edutamise teel saadud liikide, oleks see veel keerulisem. Sellisel juhul oleks vaja *topeltedutamist* – tüübid, mis ei ole parametriseeritud üle edutamise teel saadud liikide, tuleks edutada kaks taset ülespoole – või peaks tüübid ja liigid sõltuvad olema.

- Ei edutata liigipolümorfseid tüüpe, kuna sellisel juhul oleks vajalik sordipolümorfism.

- Loomulikult ei tohi edutada andmekonstruktoreid, mis võtavad mitte-edutatavat tüüpi argumente. Andmekonstruktorite edutamisel muutub tüüp liigiks, aga kuna mitte-edutatavat tüüpi liigiks muuta ei saa, ei ole see võimalik. Tüübi mitte-edutatavus levib kõigile andmekonstruktoritele, mis teda kasutavad.

Sarnastest piirangutest on lähtutud ka Awfuli loomisel, teatud muudatustega.

- Keeles ei ole üldistatud algebralisi andmetüüpe, aga on hargnevad andmetüübid, mida kirjeldame alamjaotises 5.4.5. Reegli asemel, et ei edutata üldistatud algebralisi andmetüüpe, kehtib Awfuli tüübisüsteemis reegel, et ei edutata hargnevaid andmetüüpe.

- Isegi kui jätta kõrvale asjaolu, et tüübitaseme tehteid ei ole Awfulisse plaanis lisada, siis võimalused andmetüüpide konstrueerimiseks on palju piiravamad kui üldistatud algebralised andmetüübid. Selle tõttu ei näe käesoleva töö autor hetkel Awfulis primitiivide edutamisele mõttekaid rakendusi. Tähtede (`Char`) ja täisarvude (`Int`) edutamine on siiski teostatud, juhuks kui andmetüüpide konstrueerimise viisid tulevikus võimsamaks muutuvad.

Jäägiklassiringide primitiivi (`Modular`) ei edutata, sest tegu on tüübiga, mis on parametriseeritud üle naturaalarvude ja seega ei vasta kitsendusele, et kõik tüübiargumendid peavad olema liiki `Star` (Haskelli `*`).

- Lihtsuse huvides edutab Awful ainult neid andmetüüpe, mille kõik konstruktorid on edutatavad. Andmetüübi edutamist koos ainult osade konstruktorite edutamisega ei tehta.
- Oleks võimalik edutada funktsiooni tüüpi sisaldavaid andmekonstruktoreid, näiteks

  ```
  data Fun t u = Fun (t -> u)
  ```

  Siit oleks võimalik edutamise teel saada kahe argumendiga liigikonstruktor `Fun` ning liigipolümorfne tüübikonstruktor `Fun` liiki `Fun(t -> u) -> Fun t u`, kus sümbol `->` tähistab tüübikonstruktori liiki. Selline edutamine võimaldaks edutada andmekonstruktoreid, mis võtavad argumendiks funktsioone, ja neist tuleks tüübikonstruktorid, mis võtavad argumendiks tüübikonstruktoreid.

  Awfulis, erinevalt Haskellist, sellist tüüpi hetkel ei edutata, sest selleks ei olnud otsest vajadust, aga antud täiendusele ei ole põhimõttelisi takistusi ja ei ole välistatud tulevikus selle võimaluse lisamine.

3.4.3 Tüübisüsteemi täiendused edutamise lisamisel

Selleks, et saaks toimuda edutamine (alamjaotises 3.4.2 kirjeldatud kitsendustega), on tarvis tüübisüsteemi täiendada. Selles alamjaotises kirjeldame lühidalt, mis muudatused olid Awfulis edutamise teostamiseks vajalikud.

- Kui edutame tüübipolümorfseid andmekonstruktoreid, näiteks listide konstruktorid, muutub tüübipolümorfism liigipolümorfismiks. Seega tekib vajadus liigipolümorfsete tüüpide järele.

  Täpselt nagu tüübipolümorfism ilma kohustusliku tüübiargumentide märkimiseta toob kaasa vajaduse tüübituletuse järele, toob liigipolümorfism ilma kohustusliku liigiargumentide märkimiseta kaasa vajaduse liigituletuse järele. Awful nõuab teostuse lihtsuse huvides liigiargumentide märkimist kõigi liigipolümorfsete tüüpide kasutamisel, et vätida liigituletust. Kuna liigipolümorfsed tüübid ei ole nii laias kasutuses kui tüübipolümorfsed andmekonstruktorid ja funktsioonid, ei muuda liigituletuse puudumine keelt nii kohmakaks kui tüübituletuse puudumine.
- Liigisüsteem peab olema piisavalt võimas ja paindlik, et võimaldada tüüpidest uusi liike teha. Enne edutamise lisamist oli Awfulis ainult äärmiselt primitiivne liigisüsteem, mis sisaldas liiki `Star` (analoogne Haskelli liigiga `*`) ja liiki `Arrow` (tüübikonstruktorite liik, analoogne Haskelli liigiga `->`). Interpretaatori sees olid liigid esitatud järgneva andmetüübi abil:

```
data Kind = Star_kind | Arrow_kind Kind Kind
```

Selgelt ei ole sellisele fikseeritud liigisüsteemile võimalik edutamist üles ehitada ja on tarvis olulisi muudatusi. Awfuli interpretaator kasutab liigi esitamiseks andmetüüpi

```
data Kind = Name_kind Name | Application_kind Kind Kind
```

Liike `Star` ja `Arrow` käsitletakse muudatuse järel lihtsalt kui nimede erijuhtumeid.

- Enne edutamise teostamist ei olnud Awfulis sorte. Sortide järele ei olnud vajadust, sest oli ainult kaks fikseeritud liigikonstruktorit ning programmeerija kirjutatud liigi korrektsust oli võimalik kontrollida otse parseri tasandil.

  Liigisüsteem, mida saab lõpmatult uute liikidega laiendada, nõuab sorte. Liike ei saa üksteisele suvaliselt rakendada. Näiteks `Star Arrow` ei ole korrektne liik ega isegi liigikonstruktor, sest `Star` ei võta argumente ja seega ei tohi teda millelegi rakendada. Õnneks on edutamisele seatud piisavalt ranged piirangud ja sortide süsteem ei ole seega keeruline: liigikonstruktori sordiks on naturaalarv, mis näitab, mitut argumenti antud liigikonstruktor nõuab.

  ```
  data Sort = Star_sort | Arrow_sort Sort
  ```

- Primitiivide edutamisel on vaja keelde sisse ehitada vastavad liigid ja tüübid. Tegu on võrdlemisi triviaalse täiendusega.

3.4.4 Edutamise kasutusjuhtusid

Selles alamjaotises külastame uuesti näiteid, mida tõime alamjaotises 3.2 rääkides tüübitaseme andmete rakendustest. Selgitame, kuidas edutamine muudab need näited kasutajasõbralikumaks ja tüübiturvalisemaks.

Esiteks ei ole edutamise olemasolu korral enam tarvis käsitsi eksplitsiitselt tüübitaseme andmeid kirjeldada. Näiteks kirjutame naturaalarvude andmetüübi

```
data Nat = Zr | Next Nat
```

ja sellest tuletatakse automaatselt liik `Nat` ning vastava liigi tüübikonstruktorid `Zr :: Nat` ja `Next :: Nat -> Nat`. Mis kõige tähtsam, tänu sellele, et nüüd on olemas liik `Nat`, on tüübitaseme naturaalarvud teostatud tüübiturvaliselt: tüübikonstruktori `Next` puhul on teada, et tema argumendiks kõlbab ainult naturaalarv ja mitte näiteks täht või list. Viga, mille tõime näiteks jaotises 3.3, ja ka teised analoogsed vead, saavad tüübikontrollija poolt tuvastatud.

Kirjutame uuesti alamjaotises 3.2 toodud andmetüübid, kasutades nüüd mitte algebraliste andmetüüpide kaudu käsitsi kirjutatud tüübitaseme andmeid, vaid edutamise teel saadud liike ja tüüpe.

Vektorite ja Boole'i funktsioonide puhul on esimese tüübiargumendi liik `Nat`. Seda on nüüd näha ka tüübikonstruktorite liigisignatuuridest.

```
data Array :: Nat -> * -> * where
  Empty_Array :: Array Zr t
  Construct_Array :: t -> Array n t -> Array (Next n) t
data Fun :: Nat -> * where
  Constant :: Bool -> Fun Zr
  Branch :: Fun n -> Fun n -> Fun (Next n)
```

Üldistatud ennikute tüübikonstruktori liigisignatuur kajastab nüüd seda, et esimene tüübiargument on tüüpide list.

```
data Tuple :: [*] -> * where
  Empty_Tuple :: Tuple []
  Construct_Tuple :: t -> Tuple l -> Tuple (t : l)
```

Mitmemõõtmeliste massiivide korral on tüübikonstruktori esimene argument tüübitaseme naturaalarvude list.

```
data Array' :: [Nat] -> * -> * where
  Empty_Array' :: t -> Array' [] t
  Construct_Array' :: Array n (Array' l t) -> Array' (n : l) t
```

Siinkohal võib tekkida küsimus, kas edutamisest on praktilist kasu keeles, kus ei ole üldistatud algebralisi andmetüüpe. Need lihtsad ja kasulikud rakendused, mida oleme selles alamjaotises vaadelnud, ei ole teostatavad ainult algebraliste andmetüüpide abil, kuna andmekonstruktorite resultaattüübid peavad erinema. Näiteks vektorite puhul annab üks andmekonstruktor tulemuseks vektori, mille pikkus on `Zr`, aga teine annab tulemuseks vektori pikkusega `Next n`.

Õnneks on paljud edutatud andmetüüpide rakendused siiski võimalikud ilma keelde üldistatud algebralisi andmetüüpe lisamata. On võimalik kasutada piiravamat ja lihtsamat viisi andmetüüpide konstrueerimiseks. Awful toob sisse *hargnevad andmetüübid*, mida tutvustame alamjaotises 5.4.5.

3.5 Edutamisega kaasnev nimekonflikt

Haskellis on sageli tavaks, et andmekonstruktori nimi ühtib tüübi nimega.

```
data Pair t u = Pair t u
```

Awful lausa nõuab ühe andmekonstruktoriga tüüpide (struktuuride) korral, et andmekonstruktori nimi ühtiks tüübi nimega. Ilma edutamiseta ei põhjusta selline kattuvus nimekonflikti. Kuna andmekonstruktor esineb ainult avaldiste tasandil ja sama nimega tüübikonstruktor esineb ainult tüüpide tasandil, ei teki vajadust neil kahel vahet teha. Edutamine muudab olukorda.

Edutades ülalpool mainitud andmetüüpi, saame lisaks tüübikonstruktorile `Pair` liiki `* -> * -> *` samanimelise tüübikonstruktori liiki `t -> u -> Pair t u`, kus `t` ja `u` on liigimuutujad ning `Pair` on edutamise teel saadud liigikonstruktor. Seega on meil nüüd kaks erineva liigi ja otstarbega, aga samanimelist tüübikonstruktorit. Nendel vahet tegemiseks on mitmeid erinevaid võimalusi.

3.5.1 Eksplitsiitne edutamine vajaduse korral

Haskellis kasutatakse juhtudel, kus tekib nimede konflikt, edutamise teel saadud tüübikonstruktori ees ülakoma [24]. Antud näite korral tähistab tüübikonstruktor `Pair` tavalist tüübikonstruktorit liiki `* -> * -> *` ning `'Pair` tähistab edutamise teel saadud tüübikonstruktorit liiki `t -> u -> Pair t u`.

Tasub mainida ka, et juhul kui kasutaja otsustab Haskellis kompileerimisel `-Wall` võtit kasutada, antakse hoiatused kõigi edutamise teel saadud tüübikonstruktorite kohta, mille ette ei ole ülakoma pandud – ka nende kohta, mille puhul ei ole nimekonflikti võimalust [4].

3.5.2 Nimekonflikti vältimine

Üks võimalus nimekonflikti lahendada on keelata tüübikonstruktori ja andmekonstruktori nimede ühtimine.

```
data Pair t u = MkPair t u
```

Antud lahendus on lihtne, aga tal on puudusi. Nii on võimalik vältida edutamise eksplitsiitset märkimist, aga samas muutuvad pikemaks tavalised andmekonstruktorid. Programmeerijatele, kes on harjunud tüübikonstruktorit ja andmekonstruktorit sama nimega nimetama, võib olla kergem kohaneda edutamise eksplitsiitse märkimisega kui andmekonstruktorite teisiti nimetamisega.

3.5.3 Eksplitsiitne edutamine alati

Nimekonflikti saab lahendada ka märkides kõik edutamise teel saadud liigid ja tüübid mõne spetsiaalse sümboliga, näiteks ülalpool mainitud andmetüübi `Pair` deklaratsioon annaks lisaks tavalisele andmekonstruktorile ja tüübikonstruktorile ka 2 argumendi liigikonstruktori `!Pair` ja tüübikonstruktori `!Pair` liiki `t -> u -> !Pair t u`.

Käesoleva töö autori arvates on tegu hea lahendusega, sest erinevalt eksplitsiitsest edutamisest ainult vajaduse korral näevad edutamise teel saadud tüübikonstruktorid ühtlasemad välja. Lisaks võib edutamise teel saadud tüübikonstruktorite eksplitsiitne eristamine tavalistest tüüpidest aidata programmeerijat, kes ei ole veel edutamisega harjunud ja ei tunne end kindlalt seda abstraktsiooni kasutades.

Selle tõttu ongi tegu lahendusega, mis sai valitud käesoleva töö raames loodud programmeerimiskeele jaoks. Awful kasutab edutamise tähistamiseks hüüumärki ning nõuab seda nii edutamise teel saadud liikide kui ka tüüpide ees.

3.6 Edutamise võrdlus sõltuvate tüüpidega

Paljude keelte tüübisüsteemides, sealhulgas Haskelli omas, kehtib *faasierisus* (ing. k. *phase distinction*). Väärtused võivad sõltuda tüüpidest aga mitte vastupidi. Ka edutamise olemasolul jääb faasierisus tegelikult kehtima. Kuigi näiteks vektorite puhul võib tunduda, et vektori tüüp sõltub naturaalarvulisest väärtusest, aga tegelikult on tüübid ja väärtused siiski rangelt eristatud [24].

Mõnedes teistes programmeerimiskeeltes, näiteks Agda [1], Coq [3] ja Idris [6], on olemas sõltuvad tüübid. See tähendab, et tüübid võivad sõltuda väärtustest. Sõltuvate tüüpidega keeltes puudub faasierisus.

Sõltuvad tüübid, nagu ka edutamine, võimaldavad luua tüübitaseme naturaalarvusid ja liste ning muid tüübitaseme andmeid. Tegu on aga võimsama tööriistaga kui edutamine. Toome näiteks mõned lihtsad funktsioonid, mida saab kirjutada sõltuvate tüüpidega keeles, aga mis ei ole võimalikud ainult andmetüüpide edutamise abil. Tarvitame näidetes Idrise süntaksit, sest see on väga sarnane Haskelli omaga. Siinkohal toodud näidetes on ainsateks erinevusteks see, et tüübi märkimiseks kasutatakse ühte ja listi konstrueerimiseks kahte semikoolonit ning liigi `*` asemel kasutatakse nime `Type`.

Üks lihtne ja kasulik funktsioon, mida edutamine, erinevalt sõltuvatest tüüpidest, ei võimalda kirjutada, on kahe staatilise vektori konkateneerimine. Kui me konkateneerime vektoreid pikkusega `m` ja `n`, siis tulemuse pikkus on `m + n`. Tegu on tüübitaseme naturaalarvude liitmisega. Kui edutatakse ainult tüüpe ja andmekonstruktoreid, ei ole tüübitaseme andmetele võimalik rakendada funktsioone.

```
cat : Array m t -> Array n t -> Array (m + n) t
cat Empty_Array b = b
cat (Construct_Array x a) b = Construct_Array x (cat a b)
```

Toome siinkohal veel ühe näite sõltuvate tüüpide kasutusest [6], mida ei ole võimalik teostada edutamise abil. Üldistatud algebraline andmetüüp `InElement` võimaldab staatiliselt tõestada, et element esineb listis. Funktsiooni `inList` tüüp `InList (5 : Int) [1, 2, 5]` on teoreem, mis ütleb, et `5` esineb listis `[1, 2, 5]`. See, et meil õnnestub vastavat tüüpi funktsioon kirjutada, tõestab tüübis toodud väite.

```
data InList : t -> List t -> Type where
  Here : InList x (x :: l)
  There : InList x l -> InList x (y :: l)
inList : InList (5 : Integer) [1, 2, 5]
inList = There (There Here)
```

Haskellis `InList` tüüp küll toimib, sest liste on võimalik tüübi tasemele edutada, aga kuna `Int` ei ole edutatav tüüp, ei ole Haskellis võimalik elemendi leidumise tõestust `Int` tüüpi täisarvude listi jaoks läbi viia. Veel halvem on olukord juhul, kui tahaksime analoogseid tõestusi läbi viia vektorite peal. Sõltuvate tüüpidega on see võimalik, aga tavaline ühekordne edutamine ei võimalda saavutada tüübitaseme vektoreid. Haskellis ei läbiks järgnev andmetüüp tüübikontrolli.

```
data InArray : t -> Array n t -> Type where
  Here : InArray x (Construct_Array x a)
  There : InArray x l -> InArray x (Construct_Array y a)
```

Näeme, et paraku on kasulikke ning üldsegi mitte keerulisi andmetüüpe ja funktsioone, mida edutamine, erinevalt sõltuvatest tüüpidest, ei võimalda kirjutada. Edutamisel on võrreldes sõltuvate tüüpidega siiski ka eeliseid mitte ainult puudusi [24]:

- Programmeerijatel, kes on harjunud faasierisusega, võib olla lihtsam mõista edutamist kui sõltuvaid tüüpe.
- Sõltuvad tüübid muudavad keerulisemaks tüübikontrolli ja tüübituletuse.
- Sellistes keeltes nagu Haskell on võimalik tüübikontrolli järel kõik tüübid kustutada (ing. k. *type erasure*). See, et programmi töö käigus ei ole enam tüüpe tarvis, aitab kaasa jõudlusele. Sõltuvate tüüpide korral on tüüpide kustutamine oluliselt keerulisem ülesanne.

Paljude kasutusjuhtude jaoks on edutamine piisav. Edutamine muudab tüübisüsteemi oluliselt võimsamaks ja väljendusrikkamaks, ohverdamata samas lihtsust ja nõudmata programmeerijalt oluliselt teistmoodi tüübisüsteemiga kohanemist.

4 Süntaks

Selles peatükis kirjeldame Awfuli süntaksit.

4.1 Süntaksi formaalne spetsifikatsioon

Toome konkreetse süntaksi formaalse spetsifikatsiooni. Lisaks püstkriipsule, mis tähistab valikut kahe variandi vahel, ja tähele ϵ, mis tähistab tühja sõnet, kasutame järgimisi sümboleid:

$$\begin{aligned} A^* &= AA^* \mid \epsilon \\ A? &= A \mid \epsilon \end{aligned}$$

Käesolev süntaksi spetsifikatsioon ei tegele leksiliste detailidega, nagu tühikud, reavahetused, kommentaarid ning muutujanimede, tähtede ja täisarvude leksiline struktuur. Neid detaile selgitame peatüki lõpus jaotistes 4.2 ja 4.3.

Loetavuse huvides on keele elementide tähistused ning süntaksi kirjeldamiseks kasutatav notatsioon tavalises kirjatüübis. Tavalised ümarsulud on kasutusel grammatiliste konstruktsioonide grupeerimiseks. Keele võtmesõnad ja sümbolid on helehallis fikseeritud laiusega kirjatüübis. Halli värvi ümar-, kant-, look- ja nurksulud on keele süntaksi osa.

$$\text{Fail} \quad P \quad ::= \quad I^* D^* C^* (F \mid I)^*$$

Awfuli fail koosneb imporditavate failide nimekirjast ning andmetüüpidest, klassidest, definitsioonidest ja esindajatest.

$$\text{Import} \quad I \quad ::= \quad \texttt{Load}\, x\texttt{.awf}$$

Tähega x tähistame süntaksi spetsifikatsioonis ükskõik millist nime: kas siis faili, tüübi, tüübimuutuja, konstruktori, struktuuri välja, definitsiooni või lokaalse muutuja oma. Nimede lekstilist struktuuri kirjeldame jaotises 4.3.

$$\text{Andmetüüp} \quad D \quad ::= \quad S \mid A \mid B$$

Andmetüüp võib olla struktuur, algebraline andmetüüp või hargnev andmetüüp. Hargnevate andmetüüpe sisulist poolt tutvustame alamjaotises 5.4.5. Käsolevas peatükis toome ainult süntaksi.

$$\text{Struktuur} \quad S \quad ::= \quad \texttt{Struct}\, x\, t\, \phi$$

Struktuuri deklaratsioon sisaldab nime, tüübimuutujaid ja väljasid.

$$\text{Tüübimuutujad} \quad t \quad ::= \quad (\texttt{[}\tau(\texttt{,}\ \tau)^*\texttt{]})?$$

Iga tüübimuutuja korral on kohustuslik märkida tema liik. Kui tüübimuutujate nimekiri on tühi, tuleb kandilised sulud ära jätta.

Tüübimuutuja koos liigiga	τ	$::=$	$x : L$

Nagu ka Idrises märgitakse tüüpe ja liike ühe mitte kahe kooloniga.

Liik	L	$::=$	$L_f \mid L_a \mid L_n$
Tüübikonstruktorite liik	L_f	$::=$	$L_1 \texttt{->} L$
Liigi rakendamine	L_a	$::=$	$L_0\ L_0\ L_0^*$
Liigi nimi	L_n	$::=$	$\texttt{!?}x$
Sulgudes liik	L_0	$::=$	$\texttt{(}(L_f \mid L_a)\texttt{)} \mid L_n$
	L_1	$::=$	$\texttt{(}L_f\texttt{)} \mid L_a \mid L_n$

Liikide süntaks on lihtne, koosnedes ainult nimedest ja rakendamisest ning binaarsest operaatorist `->`, millega saab asendada tüübikonstruktori liigi `Arrow` nime. Edutamise teel saadud liikide ette käib hüüumärk.

Väljad või argumendid	ϕ	$::=$	$(\texttt{(}v\texttt{(}\ \texttt{,}\ v)^*\texttt{)})?$
Muutuja koos tüübiga	v	$::=$	$x : T$

Struktuuri väljade ja funktsiooni argumentide süntaks on identne. Juhul, kui struktuuril ei ole väljasid või funktsioonil ei ole argumente, tuleb ümarsulud ära jätta.

Tüüp	T	$::=$	$T_f \mid T_t \mid T_a \mid T_0$
Funktsiooni tüüp	T_f	$::=$	$T_2 \texttt{->} T$
Paari tüüp	T_t	$::=$	$T_1 \texttt{*} T_2$
Tüübi rakendamine	T_a	$::=$	$T_p\ T_p\ T_p^*$
Atomaarne tüüp	T_0	$::=$	$T_n \mid T_c \mid T_i \mid n$
	T_2	$::=$	$\texttt{(}T_f\texttt{)} \mid T_t \mid T_a \mid T_0$
	T_1	$::=$	$\texttt{(}(T_f \mid T_t)\texttt{)} \mid T_a \mid T_0$
Sulgudes tüüp	T_p	$::=$	$\texttt{(}(T_f \mid T_t \mid T_a)\texttt{)} \mid T_0$
Tüübi nimi ja liigiargumendid	T_n	$::=$	$\texttt{!?}x\ \gamma$
Tüübitaseme täht	T_c	$::=$	$\texttt{!}c$
Tüübitaseme täisarv	T_i	$::=$	$\texttt{!}i$

Tüüpide süntaks on sarnane liikide omaga, aga keerulisem selle tõttu, et tüüpide puhul lisanduvad eksplitsiitsed liigiargumendid, mis on liigipolümorfsete tüüpide korral kohustuslikud liigituletuse vältimiseks, ning tüübitaseme primitiivid. Lisaks on olemas süntaktiline suhkur tüübitaseme naturaalarvude jaoks (näiteks teisendatakse `1` tüübiks `!Next !Zr`).

Binaarsete tüübikonstruktorite `Function` ja `Pair` jaoks on defineeritud operaatorid. Mõlemad assotsieeruvad paremale ning paari tüübi operaator `*` on kõrgema prioriteediga kui funktsiooni tüübi operaator `->`.

Tähega n tähistame süntaksi spetsifikatsioonis naturaalarvu (kaasa arvatud null), tähega c tähemärki ja tähega i täisarvu. Tähtede ümber käivad topeltjutumärgid (näiteks `"!"`). Reavahetuse märk on `"\n"`. Hetkel toetab Awful ainult ASCII standardi tähti.

$$\text{Liigiargumendid} \quad \gamma \quad ::= \quad (\texttt{[}L(\texttt{,}L)^*\texttt{]})?$$

Liigiargumentide puhul antakse liigiparameetrite konkreetsed väärtused ette kandilistes sulgudes liigi või tüübi nime järel. Juhul, kui liik või tüüp ei ole liigipolümorfne – ehk ei võta ühtegi liigiargumenti – tuleb kandilised sulud ära jätta.

$$\begin{array}{lllll} \text{Algebraline andmetüüp} & A & ::= & \texttt{Algebraic}\ x\ t\texttt{(}a\texttt{,}\ a(\texttt{,}\ a)^*\texttt{)} \\ \text{Algebralise andmetüübi konstruktor} & a & ::= & x\ T_p^* \end{array}$$

Algebraline andmetüüp, nagu ka struktuur, algab nimest ja tüübimuutujate nimekirjast. Sellele järgneb nimekiri vähemalt kahest konstruktorist. Algebralise andmetüübi konstruktor, erinevalt struktuuri omast, ei anna väljadele nimesid.

$$\begin{array}{lllll} \text{Hargnev andmetüüp} & B & ::= & \texttt{Branching}\ x\texttt{[!}x\ \gamma\texttt{]}t\texttt{(}b\texttt{,}\ b(\texttt{,}\ b)^*\texttt{)} \\ \text{Hargneva andmetüübi konstruktor} & b & ::= & \texttt{!}x\ x^*\ \texttt{->}\ x\ \phi \end{array}$$

Hargnev andmetüüp algab nimest. Sellele järgneb kandilistes sulgudes liik, mille järgi hargnetakse. Kuna hargnev andmetüüp võib hargneda ainult üle liigi, millel on lõplik arv tüübikonstruktoreid, peab see tingimata olema edutamise teel saadud liik ning liigi nimele peab eelnema hüüumärk. Järgneb ülejäänud tüübimuutujate nimekiri ja konstruktorite nimekiri. Konstruktor koosneb edutamise teel saadud tüübi nimest, tüübimuutujate nimekirjast, konstruktori nimest ja väljadest.

$$\begin{array}{lllll} \text{Klass} & C & ::= & \texttt{Class}\ x\texttt{\{}\tau\texttt{\}}(\texttt{<}x\texttt{>})?((M(\texttt{,}\ M)^*))? \\ \text{Meetod} & M & ::= & x\ t\ k\ \texttt{:}\ T \end{array}$$

Tüübiklassi deklaratsioon algab nimest ning tüübimuutujast koos liigiga looksulgudes. Päritav klass on võimalik spetsifitseerida nurksulgudes. Seejärel tuleb meetodite nimekiri, mis võib olla ka tühi, mis juhul tuleb ümarsulud ära jätta.

$$\begin{array}{lllll} \text{Kitsendused} & k & ::= & (\texttt{<}\kappa(\texttt{,}\ \kappa)^*\texttt{>})? \\ \text{Kitsendus} & \kappa & ::= & x\ x \end{array}$$

Kitsendused pannakse nurksulgudesse. Juhul kui kitsedusi ei ole, tuleb nurksulud ära jätta. Kitsendus koosneb klassi nimest ja kitsendatava tüübimuutuja nimest.

$$\text{Definitsioon} \quad F \quad ::= \quad \texttt{Def}\ x\ t\ k\ \phi\ \texttt{:}\ T\ \texttt{=}\ E$$

Definitsioon sisaldab nime, tüübimuutujaid, kitsendusi, argumentide nimekirja, tüüpi ja avaldist. Argumentide nimekiri on süntaktiline suhkur, mis parandab definitsioonide loetavust.

```
Def Id[T : Star](x : T) : T = x
Def Id[T : Star] : T -> T = x -> x
```

Need kaks definitsiooni on ekvivalentsed ning esimene teisendatakse parsimise järel teiseks.

Avaldis	E	$::=$	$E_1 \mid E_0$
Mitteatomaarne avaldis	E_1	$::=$	$\Lambda_1 \mid \alpha \mid \lambda \mid \mu \mid \Psi \mid m$
Atomaarne avaldis	E_0	$::=$	$\Lambda_0 \mid \nu \mid c \mid i$
Sulgudes avaldis	E_p	$::=$	$(E_1) \mid E_0$

Avaldis võib olla funktsiooni rakendamine, lambda-avaldis, mustrisobitusavaldis, let-avaldis, jäägiklassiringi element, muutujanimi, täht või täisarv. Lisaks on keelde sisse ehitatud süntaktiline suhkur listide jaoks.

$$\text{Jäägiklassiringi element} \quad m \quad ::= \quad n \,\#\, n$$

Jäägiklassiringi elemendi (`Modular`) puhul kirjutatakse ringi moodul numbri järele, eraldatuna `#` sümboliga.

$$\text{Muutujanimi} \quad \nu \quad ::= \quad x(\{T\})?([T(,T)^*])?$$

Muutujanimele saab anda eksplitsiitseid tüübiargumente.

$$\text{Funktsiooni rakendamine} \quad \alpha \quad ::= \quad E_p \; E_p \; E_p^*$$

Funktsiooni rakendamine assotsieerub vasakult paremale ning järjekorda saab muuta ümarsulgudega täpselt nagu Haskellis. Erinevalt Haskellist ei luba Awful ebavajalikke sulgusid, mille ära jätmine ei põhjusta muudatusi avaldise tähenduses, näiteks `(0)`. Ainsaks erandiks on sulud jäägiklassiringi elementide ja mittetühjade listide süntaktilise suhkru ümber, kus sulud küll ei osale tehete jäjekorra määramisel, aga see-eest aitavad programmeerijat koodi lugemisel. Näiteks on avaldis `f (0 # 1)` oluliselt paremini loetav kui `f 0 # 1`.

$$\text{Lambda} \quad \lambda \quad ::= \quad x \texttt{ -> } E$$

Lambda-avaldis koosneb muutujanimest, millele järgneb nool ja avaldis.

Mustrisobitus	μ	::=	`Match` E `{`$(M_a \mid M_c \mid M_i \mid M_m)$`}`
	M_a	::=	μ_a`(,` μ_a`)`$^*\delta?$
	μ_a	::=	$x\ x^*$ `->` E
Vaikevalik	δ	::=	`, Default ->` E
	M_c	::=	μ_a`(,` μ_c`)`$^*\delta$
	μ_c	::=	c `->` E
	M_i	::=	μ_a`(,` μ_i`)`$^*\delta$
	μ_i	::=	i `->` E
	M_m	::=	μ_a`(,` μ_i`)`$^*\delta?$
	μ_m	::=	m `->` E

Mustrisobitus algab võtmesõnaga `Match` ja avaldisega, seejärel on looksulgudes harude nimekiri. Juhul, kui tegu on algebraliste andmetüüpide või jäägiklassiringi elementide mustrisobitusega (M_a ja M_m), on vaikevalik (`Default`) mittekohustuslik. Juhul, kui tegu on primitiivide mustrisobitusega (M_c ja M_i), on vaikevalik kohustuslik.

Let-avaldis	Ψ	::=	`Let` ψ`(,`ψ`)`* `In` E
Lokaalne definitsioon	ψ	::=	$x\ x^*$ `=` E

Let-avaldis algab võtmesõnaga `Let`, millele järgneb mittetühi lokaalsete definitsioonide nimekiri, võtmesõna `In` ja tulemuseks olev avaldis. Tegu on järjestikuse let-avaldisega, mis ei võimalda rekursiivseid definitsioone. Parsimise järel teisendatakse let-avaldis lambda-avaldiste rakendamise jadaks. Awfuli let-avaldisest räägime ka alamjaotises 5.7.4.

Tühi list	Λ_0	::=	`List`
Mittetühi list	Λ_1	::=	`List (`E`(,`E`)`*`)`

Listide süntaktiline suhkur algab sõnaga `List` ning järgneb ümarsulgudes komadega eraldatud elementide loetelu. Juhul, kui list on tühi, on kohustuslik ümarsulud ära jätta.

Esindaja	I	::=	`Instance` x{$T_n\ x^*$}k`(`$(m$`(,`m`)`*`)`$)?$
Meetodi teostus	m	::=	$x\ x^*$ `=` E

Esindaja korral tuleb märkida klassi nimi, tüübi nimi koos tüübimuutujatega, kitsenduste nimekiri ja meetodite teostuste nimekiri. Meetodi teostus koosneb meetodi nimest, muutujate nimekirjast ja avaldisest. Võimalus otse meetodi nime järel muutujate nimesid kirjutada, et vähendada lambda-avaldiste arvu, on jällegi süntaktiline suhkur. See teisendatakse enne nime- ja tüübikontrolli lambda-avaldisteks.

Süntaks ei ole taandetundlik. Põhjuseks oli parseri teostuse lihtsus, aga ka soov anda kasutajale vabadus koodi paigutuses. Lisaks vähendab mitte-taandetundlikkus parsimisvigade ohtu suuremate kooditükkide kopeerimisel ühest kohast teise, mis on mugav näiteks koodi refaktoreerides. Peamiseks puuduseks on liigsed sulud ja eraldajad (näiteks `Match` avaldises), mida taandetundlik süntaks oleks aidanud vältida.

4.2 Kommentaarid

Kommentaaride süsteem on sarnane Haskellile ja C keelele, selle vahega, et kasutatakse teistsuguseid sümboleid. Üherealist kommentaari alustatakse graavise sümboliga.

```
Def Id[T : Star](x : T) : T = T `Ühikfunktsioon
```

Mitmerealist kommentaari alustatakse lainelise joone ja kaldkriipsuga, ning lõpetatakse kaldkriipsu ja lainelise joonega. Nagu ka Haskellis võib mitmerealisi kommentaare teineteise sisse paigutada.

```
~/ Awful on funktsionaalne keel. /~
```

4.3 Muutujanimede leksiline struktuur

Keel on tõstutundlik. Erinevalt Haskellist ei ole piiratud, mis nimed peavad algama suur- ja millised väiketähega. Failid, liigid, tüübid, tüübimuutujad, konstruktorid, struktuuride väljad, klassid, definitsioonide nimed ja lokaalsed muutujad võivad kõik alata kas suur- või väiketähega vastavalt kasutaja soovile.

Nimedes on lubatud ülakoma, numbrid (välja arvatud esimeses positsioonis), ladina tähed ja alakriips. ASCII standardis mitte-esinevad sümbolid ei ole nimedes lubatud, küll aga tohib neid kasutada kommentaarides.

Üksik alakriips tähistab, nagu ka Haskellis, nimetamata muutujat.

5 Keele kirjeldus ja koodinäited

Awful on deklaratiivne, puhas, funktsionaalne keel.

Kuna töö teemaks oli tüübisüsteemi arendamine mitte kompileerimisega seotud probleemaatika, sai kirjutatud ainult interpretaator ning kompileerimise võimalust ei ole. Tüübisüsteem on siiski staatiline: koodifaile on võimalik tüübikontrollida ka ilma väärtustajat käivitamata ning tüükontroll toimub enne väärtustamist mitte selle käigus.

Erinevalt Haskellist, mis kasutab laiska väärtustamist, on Awful agara väärtustamisega. Laisa väärtustamise puhul väärtustatakse avaldised alles siis, kui nende tulemusi vaja on, samas kui agar väärtustamine tähendab, et kohe funktsiooni kutsel väärtustatakse kõik argumendid [23]. Agara väärtustamise valiku põhjuseks, nagu ka selle põhjuseks, et keel on interpreteeritud mitte kompileeritud, oli see, et töö fookuses oli tüübisüsteem ning väärtustamise viis on selle seisukohast ebatähtis. Seega sai valik langetatud lihtsama variandi kasuks.

5.1 Awfuli interpretaatori kasutajaliides

Interpretaatori lähtekood, mis on kirjutatud Haskellis, ja ka keelega kaasas käivad teegid, asuvad aadressil

```
github.com/LiisiKerik/Awful
```

Interpretaatori kompileerimiseks tuleb kompileerida fail `Awful.hs`.

```
ghc Awful.hs
```

Awful annab kasutajale kaks käsklust, mida koodifailidele ja avaldistele rakendada: `check`, mis teostab tüübikontrolli, ning `eval`, mis väärtustab avaldise. Käsklus `check` võtab argumendiks failinime ning tüübikontrollib faili sisu ja kõik selle impordid.

```
./Awful check Standard.awf
```

Käsklus `eval` võtab argumendiks nimekirja failinimedest ja avaldise ning tüübikontrollib failid ja kõik nende impordid. Seejärel väärtustatakse avaldis keskkonnas, kus on saadaval kasutaja antud failide nimekirjast leitud definitsioonid.

```
./Awful eval Standard.awf Algebra.awf "Fmap (Add 1) (Complex 2 5)"
```

Interpreteeritav avaldis peab olema tüübist, mis on `Writeable` klassi esindaja, ning tulemus kuvatatakse `Write_Brackets` meetodi abil.

5.2 Koodifailid ja importimine

Kõik Awfuli koodifailid lõpevad laiendiga `.awf`. Faile saab importida kasutades võtmesõna `Load`. Impordid peavad olema faili alguses ja nende omavaheline järjekord ei ole oluline.

```
Load Algebra.awf
Load Standard.awf
```

Ringsõltuvused failide vahel on keelatud. Nende olemasolul antakse kasutajale veateade, kus kuvatakse nimekiri tsüklit moodustavatest failidest.

5.3 Nimekonfliktid

Awfuli nimekontrolli olulisemad aspektid on järgmised:

- Programmis esinevaid nimesid – liikide, tüüpide, konstruktorite, struktuuri väljade, klasside, definitsioonide, tüübimuutujate ja lokaalsete muutujate nimesid – kogutakse ühte nimekontrolli-keskkonda. See tähedab, et sama nime ei tohi jagada ka täiesti erineva otstarbega konstruktsioonid, näiteks tüüp ja definitsioon, kuigi sellele ei oleks põhimõttelisi takistusi.
- Ainsaks erandiks reeglist, et globaalsed nimed ei tohi olenemata otstarbest omavahel ühtida, on struktuuri tüübikonstruktori ja andmekonstruktori nimi. Struktuuri deklareerimisel antakse automaatselt sama nimi nii tüübikonstruktorile kui ka andmekonstruktorile.
- Sama programmi kahes erinevas moodulis ei tohi esineda ühesugust globaalset nime, isegi juhul kui impordisuhted failide vahel on sellised, et tegelikult nimekonflikti ei teki. Tegu oli teostuse lihtsusest lähtuva otsusega.
- Kui moodul `B` impordib (otse või kaudselt) mooduli `A`, tehakse nimekontroll enne mooduli `A` peal. See tähendab, et mõni lokaalne nimi moodulis `A` võib ühtida mõne globaalse nimega moodulis `B`, aga mitte vastupidi.
- Erinevalt Haskellist ei luba Awful muutujate varjutamist (ing. k. *shadowing*). Näiteks lambda-avaldis `f = \x -> \x -> x` on Haskellis lubatud (olenevalt kompilaatori seadistustest võib see resulteeruda hoiatuses). Sisemine `x` varjutab välimise ning avaldise `f 0 1` väärtustamise tulemuseks on `1`. Samas Awfulis on analoogne avaldis `x -> x -> x` keelatud.

5.4 Liigid ja andmetüübid

Haskellis on olemas tavalised algebralised andmetüübid ja üldistatud algebralised andmetüübid. Awfuli lähenemine andmetüüpidele on teistsugune. Keel pakub kasutajale kolme erinevat viisi uute andmetüüpide loomiseks: struktuurid, algebralised andmetüübid (mis mõnevõrra erinevad Haskelli omadest) ja hargnevad andmetüübid. Kõigil on erinev otstarve.

Awful nõuab, et andmetüübid oleks deklareeritud enne klasse ja definitsioone. Andmetüüpide deklaratsioonide omavaheline järjekord ei ole aga oluline isegi kui nende vahel on sõltuvusi.

5.4.1 Liigid ja edutamine

Keelde on sisse ehitatud liigid `Star` ja `Arrow`, mis on analoogsed vastavalt Haskelli liikidega `*` ja `->`. Liigi `Arrow` jaoks on sisse ehitatud binaarne operaator `->`. Lisaks on Awfuli liigi- ja tüübisüsteemi täiendatud edutamise abil. Edutamisest on antud põhjalik ülevaade peatükis 3 ning Haskelli ja Awfuli edutamise erinevusi käsitleb alamjaotis 3.4.2.

5.4.2 Struktuurid

Struktuurid on mõeldud andmetüüpide jaoks, millel on täpselt üks konstruktor. Andmekonstruktori nimi ühtib automaatselt tüübi nimega. Struktuuri väljadele on kohustuslik nimed anda.

```
Struct Complex[T : Star](Real : T, Imaginary : T)
```

Juhul, kui struktuur ei kasuta tüübipolümorfismi, tuleb kandilised sulud lihtsalt ära jätta. Sama kehtib ka struktuuri väljade kohta: juhul kui struktuuril ei ole ühtegi välja, tuleb ära jätta ümarsulud väljade loeteluga.

```
Struct Trivial
```

5.4.3 Algebralised andmetüübid

Algebralised andmetüübid sarnanevad Haskelli omadele, selle vahega, et kasutajalt nõutakse vähemalt kahte konstruktorit ja ühegi andmekonstruktori nimi ei tohi tüübi nimega ühtida. Juhul, kui kasutaja tahab kirjutada ühe andmekonstruktoriga andmetüüpi, on selleks struktuurid.

Konstruktorite loetelu käib tüübimuutujate nimekirja järel ümarsulgudes.

```
Algebraic Either[T : Star, U : Star](Left T, Right U)
```

Kuna algebraliste andmetüüpide puhul on andmetel kaks või enam erinevat võimalikku kuju, ei ole nende jaoks antud võimalust spetsifitseerida väljade nimesid – need funktsioonid ei oleks tüübiturvalise. Kui struktuurist saab andmed kätte väljade nimede abil, siis algebraliste andmetüüpide jaoks tuleb kasutada mustrisobitusavaldist, mida on kirjeldatud alamjaotises 5.7.3.

5.4.4 Keelde sisse ehitatud andmetüübid

Lisaks tähtedele (`Char`), funktsioonidele (`Function`), piiramata suurusega täisarvudele (`Int`) ja jäägiklassiringidele (`Modular`) on Awfulis ka ilma standardteeki importimata saadaval järgmised andmetüübid:

```
Algebraic Comparison(LT, EQ, GT)
Algebraic List[T : Star](Empty_List, Construct_List T (List T))
Algebraic Logical(False, True)
Algebraic Maybe[T : Star](Nothing, Wrap T)
Algebraic Nat(Zr, Next Nat)
Struct Pair[T : Star, U : Star](First : T, Second : U)
```

Väiksem-võrdne-suurem võrdlustulemuse tüüp `Comparison` on keelde sisse ehitatud, sest keelde on sisse ehitatud ka primitiivide (tähtede ja täisarvude) võrdlus. Listid, loogikaväärtused ja paarid on keelde sisse ehitatud põhjusel, et neid on vaja `Writeable` klassi jaoks, mille kaudu on teostatud avaldise väärtustamise tulemuste kasutajasõbralik esitamine. `Maybe` on lisatud sisse ehitatud klassi `Field` (korpus) jaoks, sest pöördelemendi tehte tulemus antakse `Maybe` monaadi all, kuna kõik korpused peale triviaalse sisaldavad mittepööratavaid elemente. Naturaalarvude tüüp `Nat` on lisatud, sest jäägiklassiringid nõuavad tüübitaseme naturaalarvusid.

Kuna kõik nimetatud struktuurid ja algebralised andmetüübid on edutatavad, kaasneb nendega ka vastavate edutamise teel saadud liikide ja tüüpide kasutamise võimalus.

5.4.5 Hargnevad andmetüübid

Kui Awfuli struktuurid ja algebralised andmetüübid on võrdlemisi sarnased Haskelli algebralistele andmetüüpidele mõningate väikeste erinevuste ja piirangutega, siis hargnevad andmetüübid on oluliselt teistsugused. Seega selgitame neid põhjalikumalt.

Hargnevad andmetüübid on mõeldud asendama osasid üldistatud algebraliste andmetüüpide pakutavaid võimalusi, ilma samas tüübisüsteemile oluliselt keerukust lisamata. Meenutame peatükis 3 toodud näiteid üldistatud algebraliste andmetüüpide rakendustest: nende abil saab konstrueerida näiteks staatilise pikkusega vektoreid, üldistatud ennikuid ja mitmemõõtmelisi staatilise suurusega tabeleid. Loetletud struktuurid ei vaja tegelikult üldistatud algebraliste andmetüüpide suurt paindlikkust, vaid järgivad väga kitsast spetsiifilist mustrit.

Olgu meil edutatav andmetüüp nimega D, millel on l tüübimuutujat ja m andmekonstruktorit.

$$\texttt{data}\ D\ t_1 \ldots t_l = C_1\ T_{11} \ldots T_{1n_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_m\ T_{m1} \ldots T_{mn_m}$$

Selle abil konstrueeritud üldistatud algebraline andmetüüp on tihti järgmisel kujul:

$$
\begin{aligned}
&\texttt{data}\ G\ \texttt{::}\ D\ K_1 \ldots K_l\ \texttt{->}\ L_1\ \texttt{->}\ \ldots\ \texttt{->}\ L_a\ \texttt{where}\\
&\quad E_1\ \texttt{::}\ U_{11}\ \texttt{->}\ldots\texttt{->}\ U_{1b_1}\ \texttt{->}\ G\ \texttt{(}C_1\ v_{11}\ \ldots v_{1n_1}\texttt{)}\ u_1 \ldots u_a\\
&\quad \ldots\\
&\quad E_m\ \texttt{::}\ U_{m1}\ \texttt{->}\ldots\texttt{->}\ U_{mb_m}\ \texttt{->}\ G\ \texttt{(}C_1\ v_{m1}\ \ldots v_{mn_m}\texttt{)}\ u_1 \ldots u_a
\end{aligned}
$$

Tüübikonstruktori G esimene argument on liigist D (rakendatuna mingitele teistele liikidele juhul kui D võtab argumente). Iga D andmekonstruktori C_i kohta luuakse üks G andmekonstruktor E_i. Tüübimuutujad $\{u_1, \ldots, u_a, v_{i1}, \ldots v_{in_i}\}$ on kõik erinevad ning argumentide tüübid $(U_{i1}, \ldots, U_{ib_i})$ ei sisalda ühtegi tüübimuutujat peale nende, mida resultaattüübis mainitakse. Andmekonstruktori E_i resultaattüübi esimene argument on D vastav konstruktor C_i rakendatuna sobivale arvule tüübimuutujatele ning ülejäänud argumendid on lihtsalt tüübimuutujad. Kõik G andmekonstruktorite resultaatttüübid näevad välja identsed kui välja arvata esimene argument.

Awful võimaldab hargnevate andmetüüpide abil asendada selliseid üldistatud algebralisi andmetüüpe, mis vastavad kõigile ülalpool loetletud kitsendustele. Ütleme, et andmetüüp G *hargneb* tüübi D järgi. Tüüpi G saab Awfulis kirja panna järgnevalt:

$$
\begin{aligned}
&\texttt{Branching}\ G\texttt{[}D\ K_1 \ldots K_l\texttt{][}u_1\ \texttt{:}\ L_1\texttt{,}\ \ldots\texttt{,}\ u_a\ \texttt{:}\ L_a\texttt{](}\\
&\quad C_1\ v_{11} \ldots v_{1n_1}\ \texttt{->}\ E_1\texttt{(}F_{11}\ \texttt{:}\ U_{11}\texttt{,}\ \ldots\texttt{,}\ F_{1b_1}\ \texttt{:}\ U_{1b_1}\texttt{),}\\
&\quad \ldots\texttt{,}\\
&\quad C_m\ v_{m1} \ldots v_{mn_m}\ \texttt{->}\ E_m\texttt{(}F_{m1}\ \texttt{:}\ U_{m1}\texttt{,}\ \ldots\texttt{,}\ F_{mb_m}\ \texttt{:}\ U_{mb_m}\texttt{))}
\end{aligned}
$$

Toome esimeseks hargnevate andmetüüpide näiteks staatilise pikkusega vektorid.

```
Branching Array[!Nat][T : Star](
  !Zr -> Empty_Array,
  !Next N -> Construct_Array(Head : T, Tail : Array N T))
```

Esimesed kandilised sulud näitavad, millise liigi järgi hargnemine toimub. Antud juhul toimub hargnemine naturaalarvude järgi. Seejärel loetleme teised vajalikud tüübimuutujad koos liikidega: antud juhul on meil üks tüübimuutuja `T`, mis tähistab vektori elementide tüüpi. Seejärel loome naturaalarvu-liigi kummagi tüübikonstruktori jaoks ühe andmekonstruktori. Nullile vastab tühi vektor; mittenullile vastab vektor, mis koosneb peast ja ühe võrra lühemast sabast.

Nagu ka struktuuride korral on kohustuslik ette anda väljade nimed. Hargneva andmetüübi esimese tüübiargumendi fikseerimisel saame ühe kindla struktuuri ning seega on väljade küsimine alati tüübiturvaline.

Toome teiseks näiteks üldistatud ennikud.

```
Branching Tuple[!List Star](
  !Empty_List -> Empty_Tuple,
  !Construct_List T L ->
    Construct_Tuple(Head_Tuple : T, Tail_Tuple : Tuple L))
```

Hargneme üle listide. Kuna Awfulis puudub liigituletus, on tarvis märkida, et hargneme üle listide, mille elemendid on tüübid liigist `Star`. Listidel on kaks konstruktorit ja ka üldistatud ennikul on kaks konstruktorit. Tühjale tüüpide listile vastab tühi ennik. Tüüpide listile, mis koosneb tüübist `T` ja sabast `L`, vastab ennik, mille pea on tüüpi `L` ja saba on ühe võrra lühem ennik, mis koosneb tüüpidest `L`.

Hargnevate andmetüüpide peamine eelis on see, et teades esimest tüübiargumenti, näiteks teades, kas vektor on pikkusega `!Zr` või `!Next N`, teame kohe, millise konstruktori abil on andmed konstrueeritud ja milliseid väljasid on sealt võimalik kätte saada.

Hargnevate andmetüüpide seatavatel rangetel piirangutel on ka olulisi puudusi. Üheks lihtsaks näiteks on üldistatud `Either`. Kui meil on võimalus kirjutada suvalise arvu tüüpide konjunktsiooni, tekib küsimus, kas saaksime kirjutada ka suvalise arvu tüüpide disjunktsiooni. Haskellis on see võimalik.

```
data Either' :: [*] -> * where
  Left' :: t -> Either' (t : l)
  Right' :: Either' l -> Either' (t : l)
```

Paneme tähele, et see üldistatud algebraline andmetüüp ei vasta hargnevatele andmetüüpidele seatud kitsendustele. Mõlema konstruktori resultaattüübis on esimene argument mittetühi list ja väärtust tüüpi `Either' []` ei ole üleüldse võimalik konstrueerida. Tegu on kasuliku andmestruktuuriga, mida Awfuli hargnevad andmetüübid ei võimalda.

5.5 Tüübiklassid

Uue tüübiklassi loomine käib võtmesõnaga `Class`, millele järgneb klassi nimi, tüübimuutuja nimi ja liik, päritav klass (kui seda on), ja nimekiri meetoditest koos tüüpidega.

```
Class Functor{F : Star -> Star}(
  Fmap[T : Star, U : Star] : (T -> U) -> F T -> F U)
```

Vahel on kasulikud ka ilma meetoditeta klassid. Sisuliselt on tegu predikaatidega tüüpide peal ja nad on kasulikud kirjeldamaks tüüpide omadusi. Üks lihtne näide on klass `Commutative`, mille esindajat saab kirjutada märkimaks, et tüüp, mis on ring, on lisaks teatud tingimustel kommutatiivne. Kuna antud juhul ei ole tüübimuutuja nimetamine vajalik, lubab Awful selle asemele kirjutada alakriipsu.

```
Class Commutative{_ : Star}<Ring>
```

Siinkohal tasub märkida, et täpselt nagu ka teiste klassidega, ei kontrollita selliste predikaatide puhul kasutaja kirjutatud klassiesindajate vastavust nõutud kitsendustele. Awful ei ole tõestusassistent ega paku võimalusi selliste omaduste tõestamiseks. See, et tüüp kuulub klassi `Commutative`, ei ole garantii, et tegu on tõepoolest kommutatiivse ringiga, juhul kui programmeerija on esindajaid defineerides vea teinud.

5.5.1 Pärilus

Awful võimaldab ainult ühest pärilust. Mitmene pärilus on plaanis teostada edasise töö käigus.

Seda, et üks klass pärib teist, märgitakse nurksulgude abil:

```
Class Finite{T : Star}<Ord>(All : List T)
```

Näiteks antud juhul eeldab klass `Finite` (mis lubab `All` meetodi abil saada list lõpliku andmetüübi kõigist võimalikest väärtustest) klassi `Ord`.

5.5.2 Klassi meetodite tüübikitsendused

Vahel on vaja kirjutada klassi, mille meetodid sisaldavad tüübikitsendusi. Hea näide on klass `Mat`, mis on mõeldud üldistamaks erinevaid ruutmaatrikseid ning sisaldab endas adjunktmaatriksi, karakteristliku polünoomi, determinandi ja vektoriga korrutamise meetodeid. Tema meetodites esinevaid tüübimuutujaid on vaja kitsendada kitsendustega `Ring` ja `Commutative`, sest näiteks maatriksi ja vektori korrutamist ei ole võimalik defineerida, kui allolev struktuur ei ole ring, ja determinant nõuab kommutatiivset ringi.

```
Class Mat{F : !Nat -> Star -> Star}(
  Adj[N : !Nat, T : Star]<Commutative T> : F N T -> F N T,
  Adj_Det[N : !Nat, T : Star]<Commutative T> : F N T -> F N T * T,
  Characteristic[N : !Nat, T : Star]<Commutative T> :
    F N T -> Array (!Next N) T,
  Det[N : !Nat, T : Star]<Commutative T> : F N T -> T,
  Matrix_by_vector[N : !Nat, T : Star]<Ring T> :
    F N T -> Array N T -> Array N T,
  Vector_by_matrix[N : !Nat, T : Star]<Ring T> :
    Array N T -> F N T -> Array N T)
```

5.5.3 Keelde sisse ehitatud klassid

Viis klassi on saadaval ka ilma standardteeki importimata:

```
Class Field{T : Star}<Ring>(Inverse : T -> Maybe T)
Class Nonzero{N : !Nat}(Div' : Int -> Int)
Class Ord{T : Star}(Compare : T -> T -> Comparison)
Class Ring{T : Star}(
  Add : T -> T -> T,
  Convert : Int -> T,
  Multiply : T -> T -> T,
  Negate : T -> T)
Class Writeable{T : Star}(Write_Brackets : T -> List Char * Logical)
```

Klass `Ord` on sarnane Haskelli samanimelise klassiga, selle vahega, et Awfulis ei ole klassi `Eq`.

Klass `Nonzero` puhul on esindaja defineeritud suvalise positiivse tüübitaseme naturaalarvu `!Next N` jaoks, aga mitte nulli jaoks. Lisaks sisaldab see klass meetodit täisarvuliseks jagamiseks tüübitaseme naturaalarvuga. Selle meetodi kasutusest toome näite alamjaotises 5.6.2.

Klass `Writeable`, sarnaselt Haskelli klassiga `Show`, on mõeldud teisendamaks andmestruktuure sõnedeks. Meetod `Write_Brackets` tagastab paari sõnest ja loogikaväärtusest. Loogikaväärtus näitab, kas avaldisele on sulgusid ümber vaja või mitte sellisel juhul, kui ta on mõne teise avaldise komponent. Meetodi `Write_Brackets` kaudu on teostatud avaldiste väärtustamise tulemuste kuvamine. Seetõttu peab kasutaja `eval` käsklust tarvitades hoolitsema selle eest, et väärtustatava avaldise tüüp oleks klassi `Writeable` esindaja.

Klassid `Ring` ja `Field` on tuttavad algebrast: tegu on (mitte tingimata kommutatiivse) ringi ja korpusega. Klassi `Ring` meetod `Convert`, mis teisendab täisarvu ringi elemendiks, on mõeldud defineerimaks nulli (`Convert 0`) ja ühikut (`Convert 1`), aga ka selleks, et oleks mugav suvalises ringis elementi täisarvulise konstandiga korrutada.

5.6 Definitsioonid ja esindajad

Nagu ka andmetüüpide ja klasside korral, ei ole definitsioonide ja esindajate omavaheline järjekord oluline.

5.6.1 Definitsioonid

Definitsioon algab võtmesõnaga `Def`, millele järgneb funktsiooni nimi, tüübimuutujate nimekiri, kitsenduste nimekiri, argumentide nimekiri, tulemuse tüüp ja avaldis. Ühikfunktsiooni definitsioon näeb välja selline:

```
Def Id[T : Star](x : T) : T = x
```

Toome ka ühe näite definitsioonist, kus esineb kitsendusi. Juhul, kui `T` on ring, saame tüübi `T` jaoks `Convert` meetodiga defineerida nulli.

```
Def Zero[T : Star]<Ring T> : T = Convert 0
```

5.6.2 Tehted primitiividega

Keelde on sisse ehitatud järgmised klasside `Ord`, `Ring` ja `Writeable` esindajad:

	`Field`	`Ord`	`Ring`	`Writeable`
`Char`		✓		
`Int`		✓	✓	✓
`Modular`	✓	✓	✓	✓

Tüüp `Modular N` on defineeritud klasside `Field` ja `Ring` esindajaks ainult sellisel juhul, kui `N` ei ole null ehk kuulub klassi `Nonzero`. Selle põhjuseks on klassi `Ring` meetod `Convert`, mis teisendab täisarvu vastava ringi elemendiks. Tüübi `Modular 0` ehk tühja ringi jaoks seda meetodit defineerida ei saa.

Lisaks on keelde sisse ehitatud kolm funktsiooni täisarvulise jagatise ja jäägi leidmiseks.

```
Div : Int -> Int -> Maybe Int
Mod : Int -> Int -> Maybe Int
Div'[N : !Nat]<Nonzero N> : Int -> Int
```

Funktsioonid `Div` ja `Mod` annavad vastavalt täisarvulise jagamise tulemuse ja jäägi. Tulemus on tüübiturvalisuse huvides `Maybe` monaadi all juhuks kui jagaja on null. Meetod `Div'` puhul on jagatav tavaline täisarv, aga jagaja on positiivne tüübitaseme naturaalarv, mistõttu on garanteeritud, et nulliga jagamist ei toimu. Seda meetodit saab programmeerija kasutada juhtudel, kus ta tahab täisarvu konstandiga jagada.

Võib tekkida küsimus, miks ei ole klassis `Nonzero` analoogset meetodit `Mod'`, mis leiaks täisarvulise jagamise jäägi juhul, kui jagaja on positiivne tüübitaseme naturaalarv `N`. Sellise meetodi eeliseks oleks võimalus anda tulemuseks mitte `Int`, vaid `Modular N`. Põhjuseks, miks sellist meetodit ei ole, on see, et klassis `Ring` leiduv meetod `Convert`, mille abil saab täisarvu jäägiklassiringi elemendiks teisendada, täidab täpselt sama otstarvet.

Toome siinkohal näite meetodite `Div'` ja `Convert` kasutusest. Algebrateegis `Algebra.awf` defineeritud funktsioon `Pow'` võtab argumendiks suvalise ringi elemendi `x` ja mittenegatiivse täisarvu `i` ning arvutab `x` astmes `i`. Selleks, et funktsiooni keerukus oleks lineaarse asemel logaritmiline, kasutame astme täisarvulist jagamist kahega.

```
Def Pow'[T : Star]<Ring T>(x : T, i : Int) : T =
  Match i {
    0 -> Identity,
    Default ->
      Let
        y = Pow' x (Div'{2} i),
        z = Multiply y y
      In
        Match Convert i {0 # 2 -> z, 1 # 2 -> Multiply x z}}
```

Funktsiooni kutse `Div'{2}` sisaldab tüübiargumendi – antud juhul tüübitaseme naturaalarvu – eksplitsiitset täpsustamist. Sellest räägime täpsemalt alamjaotises 5.7.5.

5.6.3 Esindajad

Esindajat deklareeritakse, sarnaselt Haskellile, sõnaga `Instance`.

```
Instance Ord{Trivial}(Compare _ _ = EQ)
```

Tüübi nimi, mille jaoks esindajad defineeritakse, ning vajaduse korral ka tüübimuutujad, käivad looksulgudesse. Nurksulgudes saab anda tüübimuutujate kitsendusi.

```
Instance Finite{Pair T U}<Finite T, Finite U>(All = Zip Pair All All)
```

Juhul, kui mõnele tüübimuutujale ei ole tarvis kitsendusi anda, saab selle asemele kirjutada alakriipsu.

```
Instance Functor{Function _}(Fmap f g x = f (g x))
```

Juhul, kui tegu on ilma meetoditeta klassiga, jäetakse ära ümarsulud meetodite nimekirjaga.

```
Instance Commutative{Int}
```

Awfulis on klassi esindajate üks eripära see, et meetodid tuleb kirja panna täpselt sellises järjekorras nagu nad on klassideklaratsioonis. Tegu oli otsusega, mis lähtus kontrolli teostuse lihtsusest.

5.7 Avaldised

Awfuli avaldis võib olla muutujanimi, primitiiv, funktsiooni rakendamine, lambda-avaldis, mustrisobitusavaldis või let-avaldis.

Lisaks on keelde sisse ehitatud süntaktiline suhkur listide jaoks, mis algab märksõnaga `List`. Tühja listi puhul piisabki ainult sellest märksõnast. Mittetühja listi puhul käib elementide nimekiri ümarsulgudesse ning elemendid eraldatakse komadega, näiteks `List (x)` ja `List (x, y)`.

Käesolevas jaotises räägime põhjalikumalt defineerimata käitumisest, lambda-avaldistest, mustrisobitusest ja let-avaldisest.

5.7.1 Defineerimata käitumine

Defineerimata käitumist saab märkida erilise muutuja abil, mille nimi on `Crash`. Sarnaselt Haskelli muutujaga `undefined` on `Crash` parameetriliselt polümorfne üle suvalise tüübi liigist `Star` (`*`). Seda saab panna suvalisse kohta avaldises veel kirjutamata kooditüki asenduseks, või luua funktsioone, mille käitumine ongi osade sisendite korral defineerimata.

Haskelli `undefined` ja Awfuli `Crash` käitumine väärtustamisel on aga erinev selle tõttu, et Haskell on laisk ja Awful on agar keel. Näiteks Haskelli avaldis `fst (0, undefined)` annab `0`, aga Awfuli analoogne avaldis `First (Pair 0 Crash)` resulteerub tulemuses `Crash`.

5.7.2 Lambda-avaldis

Funktsionaalse programmeerimiskeelena sisaldab Awful loomulikult lambda-avaldisi. Mustrisobituse võimalust funktsiooni argumentides ei ole veel teostatud, aga seda on tulevikus plaanis lisada. Süntaks on sarnane Haskelli omaga, aga ei ole vajalik \ sümbol enne muutujanime. Toome siinkohal näite lambda-avaldist kasutavast funktsioonist.

```
Def Div_left[T : Star]<Field T>(x : T, y : T) : Maybe T =
  Fmap (z -> Multiply z y) (Inverse x)
```

Süntaktiline suhkur korraga kahe muutuja andmiseks Awfulis puudub. Kui Haskellis saab kirjutada `\ x y -> x`, siis Awfulis peab kirjutama `x -> y -> x`.

5.7.3 Mustrisobitus

Mustrisobitus töötab tähtede, täisarvude, jäägiklassiringi elementide ja algebraliste andmetüüpide peal.

```
Def Not(x : Logical) : Logical = Match x {False -> True, True -> False}
```

Saab kirjutada ka `Default` juhu, mida kasutatakse, kui ükski eelnev variant ei sobinud. See vaikevalik tuleb alati kirjutada mustrisobituse viimaseks juhuks.

```
Def Max[T : Star]<Ord T>(x : T, y : T) : T =
  Match Compare x y {LT -> y, Default -> x}
```

Turvalisuse tagamiseks kontrollitakse iga `Match` avaldise puhul, kas kõik võimalikud juhud on käsitletud. Kattuvad või puuduvad juhud annavad vea. See tähendab, et `Default` juhtum on kohustuslik tähtede ja täisarvude mustrisobituses ning ka jäägiklassiringi elementide ja algebraliste andmetüüpide jaoks juhul, kui eelnevalt ei ole kõiki võimalikke harusid käsitletud.

Erinevalt Haskellist ei luba Awful kirjutada keerulisema struktuuriga mustrisobitust, näiteks sobitada sama `Match` avaldisega tühja, 1-elemendilist ja 2-elemendilist listi. Sellise otsuse põhjuseks oli teostuse lihtsus.

Samuti ei luba Awful `Match` avaldist üle struktuuri (selle asemel tuleb kasutada väljade nimesid, ja tulevikus on plaanis lisada võimalus teha struktuuride mustrisobitust otse funktsiooni argumendis) ega üle hargneva andmetüübi (selle asemel on võimalik kasutada tüübiklasse viisil, mis on kirjeldatud jaotises 5.8).

5.7.4 Let-avaldis

Awfulis on olemas järjestikune let-avaldis. Let-avaldise lokaalsed definitsioonid peavad olema kirja pandud sellises järjekorras, et iga definitsioon võib kasutada ainult eelnevaid definitsioone aga mitte iseennast ega järgnevaid definitsioone. Toome näite let-avaldise kasutusest standardteegis. Funktsioon `Brackets` võtab argumendiks klassi `Writeable` esindaja ning annab tulemuseks sõne, mis on olenevalt vajadusest sulgudega ümbritsetud või mitte.

```
Def Brackets[T : Star]<Writeable T>(x : T) : List Char =
  Let
    y = Write_Brackets x
  In
    Match Second y {
      False -> First y,
      True -> Cat (Lift "(") (Cat (First y) (Lift ")"))}
```

Let-avaldised on Awfulis süntaktiline suhkur, mis teisendatakse parsimise järel lambda-avaldiste rakendamise jadaks. Viime näitena läbi kahte lokaalset definitsiooni sisaldava let-avaldise teisenduse.

```
Let f x = Add x 1, y = 2 In f y
```

Alguses viiakse let-avaldis kujule, kus iga definitsioon on eraldi let-avaldises. Peame meeles, et definitsioon `f x = Add x 1` tähendab tegelikult `f = x -> Add x 1`.

```
Let f = x -> Add x 1 In Let y = 2 In f y
```

Seejärel teisendame iga let-avaldise `Let` x `=` e_0 `In` e_1 kujule `(`x `->` e_1`)` e_0.

```
(f -> (y -> f y) 2) (x -> Add x 1)
```

Awfuli let-avaldised ei luba defineerida polümorfseid funktsioone. Põhjuseid on lähemalt selgitatud alamjaotises 6.3.3. Seni aga toome näite polümorfsest let-avaldisest, mida Awfuli tüübikontroll ei luba.

```
Def Bug : Char * Int = Let Id x = x In Pair (Id "A") (Id 0)
```

Antud näide töötaks juhul kui lokaalselt defineeritud funktsiooni `Id` kasutataks kas ainult tähtede või ainult täisarvude peal. Kasutaja on aga eeldanud, et `Id` on parameetriliselt polümorfne ühikfunktsioon, mis töötab suvalise tüübi peal, ja on seda kasutanud nii tähe kui ka täisarvu peal. Kuna let-avaldise all defineeritud funktsioonid on monomorfsed, üritab tüübikontrollija unifitseerida tüüpe `Char` ja `Int` ning tulemuseks on tüübiviga.

5.7.5 Tüübiargumentide eksplitsiitne edastamine

On juhtumeid, kus tüübituletus ei ole võimalik ja ei aita ka standartse Haskelli stiilis tüübiannotatsioonid avaldiste sees, näiteks `show (read "0" :: Int)`. Awful võimaldab avaldises esineva polümorfse funktsiooni tüüpi täpsustada eksplitsiitsete tüübiargumentide abil. Pöördume tagasi jaotises 2.3 toodud näite juurde meetodist, mille teostust ei ole võimalik tüübist tuletada.

```
Class Nonzero{N : !Nat}(Div' : Int -> Int)
```

Kuna Awful võimaldab meetodi teostust eksplitsiitselt täpsustada, ei ole tüübi mittetuletatavus enam takistuseks meetodi kasutamisele. Meetodi teostust täpsustatakse Awfulis tüübiga loogeliste sulgude vahel, näiteks antud juhul oleks kahega jagamiseks vaja valida meetodi `Div'` teostus tüübi `2` jaoks: `Div'{2}`.

Paneme tähele, et meetod `Div'` võib olla kasutuses mitte ainult otse, vaid ka kaudselt mõne muu funktsiooni kaudu.

```
Def Div_5[N : !Nat]<Nonzero N> : Int = Div'{N} 5
```

Juhul, kui kutsutakse funktsiooni `Div_5`, ei ole võimalik tuletada, mis on tüübimuutuja `N` väärtus, ja seega ei ole võimalik tuletada ka seda, millist meetodi `Div'` teostust kasutab `Div'{N}`. Seega pakub Awful võimalust täpsustada mitte ainult meetodi teostust, vaid ka tavalisi tüübimuutujaid, näiteks `Div_5[2]`.

Kui funktsioon on mõne klassi meetod ja on lisaks parameetriliselt polümorfne, näiteks `Fmap`, saab kasutaja ise valida, kas ta spetsifitseerib ainult meetodi teostuse, ainult tavalised tüübimuutujad või mõlemad. Loomulikult on alati ka võimalus tüübiargumente mitte eksplitsiitselt edastada, kui tegu on avaldisega, mille peal on võimalik tüübituletus.

```
Fmap (Add 1) List
Fmap{List} (Add 1) Empty
Fmap[Int, Int] Id List
Fmap{List}[Int, Int] Id Empty
```

Vaatleme, kuidas Awfuli tüübiargumentide eksplitsiitne edastamine võrdleb teiste keelte omaga.

Standartne Haskell ei võimalda tüübiargumentide eksplitsiitset täpsustamist, aga see on võimalik kasutades laiendust `TypeApplications` [4]. Tüübiargumente saab anda sümboliga `@`, näiteks `show (read @Int "0")`. Haskell lubab edastada ainult osad tüübiargumendid. Awfuli puhul tuleb edastada kõik tüübiargumendid või mitte ühtegi ja ainsaks erandiks on tüübiklassi meetodi teostuse spetsifitseerimine, mida saab teha sõltumatult ülejäänud tüübimuutujate täpsustamisest.

Awful võtab tüübiargumentide edastamise süntaksis eeskuju hoopis sellistest keeltest nagu Visual Basic [9] ja C# [2], kus polümorfsete ehk geneeriliste (ing. k. *generic*) funktsioonide

kasutamisel on tüübiparameetrite nimekiri sulgude sees ja eraldatud komadega. Käesoleva töö arutori arvates on selline süntaks loetavam ja kasutajasõbralikum, sest tüübiargumendid ja tavalised argumendid on visuaalselt paremini eristatud.

5.8 Tüübiklasside kasutamine tüübimuutujate mustrisobituseks

Vaatlesime, kuidas luua hargnevate andmetüüpide abil staatilise pikkusega vektoreid ja üldistatud ennikuid. Ainult andmestruktuuride deklareerimisest aga ei piisa – soovime kirjutada funktsioone, mis nende andmestruktuuride peal töötaks. Näiteks tahame teostada `Fmap` meetodit vektorite jaoks.

Vektorite struktuur on sarnane listidele, selle vahega, et pikkus on tüübi osa. Meenutame, kuidas on teostatud listide `Fmap`.

```
Instance Functor{List}(
  Fmap f x =
    Match x {
      Empty_List -> List,
      Construct_List y z -> Construct_List (f y) (Fmap f z)})
```

Oleks loogiline analoogselt teostada ka vektorite `Fmap` meetodit. Haskellis ongi see võimalik.

```
instance Functor (Array n) where
  fmap f x =
    case x of
      Empty_Array -> Empty_Array,
      Construct_Array y z -> Construct_Array (f y) (fmap f z)
```

Awful aga sellist teostust ei luba, sest `Array` on hargnev andmetüüp ja Awfuli `Match` avaldis töötab ainult tavaliste algebraliste andmetüüpide mitte hargnevate andmetüüpide peal. Ning ka Haskellis on osadel juhtudel `case` avaldise kasutamisele takistusi. Üks näide on klassi `Applicative` meetod `pure`, mis on tüüpi `t -> f t`. Antud meetodil ei ole ühtegi argumenti, mille peal mustrisobitust teostada.

Üks variant oleks mustrisobitus tüübimuutuja `n` peal, mis tähistab vektori pikkust. Ei Haskell ega Awful ei toeta hetkel sellist lähenemist. Seni saab aga ajutise lahendusena kasutada vahendit, mis on keeles juba olemas – tüübiklasse.

```
Class Functor_Array{N : !Nat}(
  Fmap_Array[T : Star, U : Star] : (T -> U) -> Array N T -> Array N U)
Instance Functor_Array{!Zr}(Fmap_Array _ _ = Empty_Array)
Instance Functor_Array{!Next N}<Functor_Array N>(
  Fmap_Array f a = Construct_Array (f (Head a)) (Fmap f (Tail a)))
Instance Functor{Array N}<Functor_Array N>(Fmap = Fmap_Array)
```

Loome abiklassi `Functor_Array` üle naturaalarvulise tüübimuutuja. See sisaldab sama meetodit mis `Functor`, ainult et mitte suvalise funktori vaid vastava pikkusega vektori jaoks. Seejärel loome abiklassi esindajad nii nulli kui ka mittenulli jaoks. Lõpuks defineerime vektori klassi `Functor` esindajaks, kasutades `Functor_Array` meetodit.

Sellel lähenemisel on kaks peamist puudust. Esiteks peab programmeerija kirjutama palju abiklasse. Teiseks resulteeruvad need abiklassid ebavajalikes kitsendustes.

```
Def Square_Array[N : !Nat, T : Star]<Functor_Array N, Ring T> :
  Array N T -> Array N T =
    Fmap Square
```

Siinkohal oleks kitsendus `Functor_Array N` tegelikult täiesti ebavajalik. Kõik naturaalarvulised tüübid on klassi `Functor_Array` esindajad ning seega tüübimuutuja `N` võiks antud juhul täiesti kitsendamata olla. Oleks hea, kui oleks võimalik vektorite ja teiste hargnevate andmetüüpide funktsioone teostada otse tüübimuutuja peal mustrisobitust tehes – ning tulevikus on plaanis Awfulit vastavalt täiendada. Sellest ideest räägime põhjalikumalt jaotises 7.1.

Toome veel ühe näite sellest, kuidas kasutada tüübiklasse kirjutamaks funktsioone, mis töötavad hargnevate andmetüüpide peal. Järgnev kood võimaldab defineerida üldistatud ennikud klassi `Ord` esindajaks:

```
Class Ord_Tuple{L : !List Star}(
  Compare_Tuple : Tuple L -> Tuple L -> Comparison)
Instance Ord{Tuple L}<Ord_Tuple L>(Compare = Compare_Tuple)
Instance Ord_Tuple{!Empty_List[Star]}(Compare_Tuple _ _ = EQ)
Instance Ord_Tuple{!Construct_List[Star] T L}<Ord T, Ord_Tuple L>(
  Compare_Tuple x y =
    Compare
      (Pair (Head_Tuple x) (Tail_Tuple x))
      (Pair (Head_Tuple y) (Tail_Tuple y)))
```

Kitsendus `Ord_Tuple L` kannab teavet selle kohta, et kõik listi `L` kuuluvad tüübid kuuluvad klassi `Ord`. Seda kitsendust, erinevalt kõigi naturaalarvude peal kehtivatest kitsendustest, ei oleks enam võimalik lihtsalt ära jätta.

6 Teostuse detailid

Awfuli interpretaator on kirjutatud Haskellis. Interpretaator on võrdlemisi tüüpilise arhitektuuriga, koosnedes lekserist, parserist, süntaktilist suhkrut eemaldavast vahesammust, nimekontollijast, tüübikontrollijast ja väärtustajast. Lisaks on eraldi moodul, mis vastutab kasutajaliidese ja impordite eest.

6.1 Parser

Awful kasutab aplikatiivset parsimist. Selles jaotises refereerime lühidalt monaadilist [13] ja aplikatiivset [16] parsimist ning selgitame, kuidas Awfuli parser on teostatud.

6.1.1 Parserite andmetüüp

Selleks, et saaks kasutada aplikatiivset või monaadilist parsimist, tuleb kirjutada parserite andmetüüp. Toome siinkohal parserite andmetüübi, mis lisaks erinevatele võimalikele väljundtüüpidele ei piira ka sisendi tüüpi ega veateadete andmise viisi.

```
newtype Parser s m t = Parser {parser :: s -> m (t, s)}
```

Selline parser võtab sisendteksti tüüpi `s` ning annab tulemuseks väljundi tüüpi `t` ja parsimata jäänud osa sisendekstist, kusjuures väljund antakse monaadi `m` all.

Sisend `s` võib olla tähtede nimekiri, aga võib olla ka mõni keerulisem struktuur. Praktikas on oluliselt lihtsam parsida teksti, mille peal on eelnevalt tehtud leksiline analüüs, et parser ei peaks tegelema selliste detailidega nagu nimede ja arvude tähthaaval kokku korjamine või kommentaarid. Lisaks parsitavale tekstile võib olla kasulik kaasas kanda ka lisainfot, näiteks teavet selle kohta, kui kaugele on teksti ebaõnnestunud parsimiskatsete käigus ette vaadatud.

Monaad `m` on vajalik selle tõttu, et parsimine võib ka ebaõnnestuda või rohkem kui ühe tulemuse anda. See monaad võib olla näiteks `Maybe` (juhul kui meid ei huvita, miks täpselt parsimine ebaõnnestus) või `Either e` (juhul kui vajame veateateid tüüpi `e`). Aga monaad `m` võib olla ka näiteks list, juhul kui on tarvis mitmest parsimist.

6.1.2 Monaadiline parsimine

Parsereid saab vaadelda monaadidena [13].

Juhul, kui `m` on funktor, on tüüp `Parser s m` samuti funktor. Intuitiivselt tähendab see, et tüüpi `t` väljundiga parseri puhul saame väljundile eduka parsimise korral rakendada funktsiooni tüüpi `t -> u` ja saame niiviisi parseri väljundtüübiga `u`.

```
instance Functor m => Functor (Parser s m) where
  fmap f (Parser p) = Parser (\x -> first f <$> p x)
```

Funktsioon `first` moodulist `Data.Bifunctor` rakendab funktsiooni paari (või mõne muu bifunktori) esimesele elemendile.

Juhul, kui `m` on monaad, on tüüp `Parser s m` samuti monaad. Operaator `>>=`, mis võtab argumendiks parseri `p` ja funktsiooni `f`, rakendab sisendtekstile parserit `p` ning, saades tulemuseks väärtuse `r` ja teksti `x`, parsib teksti `x` parseriga `f r`. See tähendab, et monaadilise parsimise puhul saab parsimise edasine käik sõltuda parsimistulemusest.

Meetod `return` teeb väärtusest `r` parseri, mis ei muuda sisendteksti ja annab parsimise tulemuseks sellesama väärtuse `r`.

```
instance Monad m => Monad (Parser s m) where
  Parser p >>= f = Parser (p >=> \(r, x) -> parser (f r) x)
  return r = Parser (\x -> return (r, x))
```

Operaator `>=>`, mis on defineeritud teegis `Control.Monad`, käitub järgmiselt:

```
p >=> f = \x -> p x >>= f
```

Monaad võimaldab muuhulgas parserite järjestikust kombineerimist. Juhul, kui soovime parsida teksti alguses parseriga `p`, mille väljund on tüüpi `t`, ja seejärel parsida ülejäänud teksti parseriga `q`, mille väljund on tüüpi `u`, ning võtta tulemused kokku üheks väärtuseks funktsiooni `f :: t -> u -> v` abil, saab seda teha parseriga `ap (fmap f p) q`. Funktsioon `ap` on tüüpi `m (t -> u) -> m t -> m u`, kus `m` on monaad.

Lisaks parserite järjestikusele kombineerimisele on tarvis ka võimalust kahe või enama parseri vahel valida. Seda on võimalik teha klassi `MonadPlus` abil, mis on `Monad` alamklass. `MonadPlus` sisaldab assotsiatiivset operatsiooni `mplus` ja selle operatsiooni ühikut `mzero`.

```
instance MonadPlus m => MonadPlus (Parser s m) where
  mplus (Parser p) (Parser q) = Parser (\x -> mplus (p x) (q x))
  mzero = Parser (\_ -> mzero)
```

Meetod `mplus` võtab argumendiks kaks parserit ja üritab rakendada esimest. Kui esimene parser ebaõnnestub, üritatakse rakendada teist. `mzero` on parser, mis alati ebaõnnestub.

6.1.3 Aplikatiivne parsimine

Iga tüübikonstruktor, mis on monaad, on ka aplikatiivne funktor. Ka parserid on aplikatiivsed funktorid [16].

```
instance Monad m => Applicative (Parser s m) where
  Parser p <*> Parser q = Parser (p >=> \(f, x) -> first f <$> q x)
  pure r = Parser (\x -> pure (r, x))
```

Loetleme siinkohal `Applicative` klassi esindajate jaoks saada olevaid operaatoreid, mida saab kasutada parserite järjestikuseks kompositsiooniks.

- `(<$>) :: Functor f => (t -> u) -> f t -> f u`

  Tegu on meetodi `fmap` operaatorkujuga.

- `(<*>) :: Applicative f => f (t -> u) -> f t -> f u`

  Parserite järjestikust kompositsiooni `ap (fmap f p) q` on aplikatiivse funktori operaatorite abil võimalik kirja panna kui `f <$> p <*> q`.

- `(<$) :: Functor f => t -> f u -> f t`

  See `fmap` sarnane operaator on kasulik juhul, kui meid huvitab teiseks argumendiks oleva parseri õnnestumine või ebaõnnestumine, aga mitte selle tulemus.

- `(<*) :: Applicative f => f t -> f u -> f t`
  `(*>) :: Applicative f => f t -> f u -> f u`

  Need `(<*>)` sarnased operaatorid eiravad vastavalt teise ja esimese parseri parsimistulemust. Operaatorid `<$`, `<*` ja `*>` on kasulikud parsimaks näiteks võtmesõnu, operaatoreid, sulgusid ja eraldajaid.

Klassi `MonadPlus` asemel saab erinevate alternatiivide vahel valimiseks kasutada ka klassi `Alternative`, mis eeldab klassi `Applicative`. Klass `Applicative` on peaaegu identne klassiga `MonadPlus`. Ainsaks sisuliseks vaheks on see, et kui `MonadPlus` eelduseks on `Monad`, siis `Alternative` eelduseks on `Applicative`, mis on vähem piirav eeldus.

Klassis `Alternative` on funktsiooni `mplus` ja konstandi `mzero` asemel samade tüüpidega operaator `<|>` ja konstant `empty`.

```
instance (Alternative m, Monad m) => Alternative (Parser s m) where
  Parser p <|> Parser q = Parser (\x -> p x <|> q x)
  empty = Parser (\_ -> empty)
```

Aplikatiivne parsimine on vähem võimas kui monaadiline. Komponeerides parsereid operaatori `>>=` abil võib teise parseri käitumine oleneda esimese parseri tagastatud parsimistulemusest; aplikatiivse parsimise korral ei ole see võimalik. Siiski on aplikatiivne parsimine paljudeks rakendusteks piisav ning üks oluline eelis aplikatiivselt kirjutatud parserite juures on see, et parseri teostus näeb välja väga sarnane grammatika formaalse spetsifikatsiooniga.

6.1.4 Awfuli parseri teostus

Awful kasutab aplikatiivset parsimist. Awfuli parser järgib alamjaotises 6.1.1 kirjeldatud mustrit, et parser on funktsioon, mis tagastab monaadi all paari parsimistulemisest ning järelejäänud sisendist. Võtame kasutusele tüübisünonüümi `Parser'`, mis täpsustab konkreetse sisendtüübi ja veateadete andmise viisi.

```
type Parser' = Parser State (Either Location)
```

Sisendiks on tüüp nimega `State`, mis koosneb lekseri väljastatud süntaksiüksuste (ing. k. *token*) nimekirjast ja lisaks asukohast, mis ütleb, kui kaugele on teksti ette vaadatud. See teave on vajalik andmaks võimalikult täpse asukohaga veateateid.

```
data State = State Tokens Location
```

Tulemus antakse `Either Location` monaadi all. Veateate tüübiks on seega `Location`, mis ütleb, kust alates ei õnnestunud teksti enam edasi parsida.

Parserite järjest komponeerimine (ning seega ka klassi `Applicative` teostus) on täpselt sama mis jaotises 6.1.3. Küll aga muudab vigade asukohtade leidmine keerulisemaks valiku alternatiivide vahel [15]. Selleks, et tüübisünonüümi `Parser'` saaks defineerida klassi `Alternative` esindajaks, mis varjutaks tüübi `Parser` vastavat esindajat, kasutame laiendusi `FlexibleInstances`, `OverlappingInstances` ja `TypeSynonymInstances`. Valik kahe võimaliku parsimisviisi vahel ja alati ebaõnnestuv parser on Awfuli jaoks teostatud järgnevalt:

```
instance Alternative Parser' where
  Parser p <|> Parser q =
    Parser
      (\x ->
        case p x of
          Left l -> q (update_location x l)
          Right r -> Right r)
  empty = Parser (\(State _ l) -> Left l)
```

Kui toimub valik kahe alternatiivi vahel, proovime enne esimest parserit. Juhul, kui see õnnestub, tagastamegi vastava tulemuse. Juhul, kui esimene parser ebaõnnestub, tunduks loomulik lihtsalt samale sisendile teist parserit rakendada. Siis aga tekib probleem veateadete asukohtadega. Juhul, kui esimene parser jõuab kaugemale kui teine, aga mõlemad ebaõnnestuvad, ütleks selline süsteem, et viga juhtus varem kui tegelikult. Näiteks kui grammatika lubab sõnet `"Catamorphism"` või `"Category"`, aga parser saab sõne `"Catastrophe"`, tahaksime kasutajale öelda, et viga juhtus tähemärgil `5`, kus ebaõnnestus sõne `"Catamorphism"` parsimine, mitte tähemärgil `4`, kus ebaõnnestus sõne `"Category"` parsimine.

Ka kahest vea-asukohast suurima leidmine ei aita. Oletame, et tahame parsida teksti reegliga `many p <* end`, kus `p` parsib sõnet `"aa"` ja `end` nõuab, et kogu tekst oleks parsitud. Siis teksti `"ab"` korral antaks vea asukohaks `1` mitte `2`. Parser `p` ebaõnnestub tähemärgil `2`. Parser `many p` õnnestub, andes tulemuseks tühja listi, ja jätab jäägiks kogu teksti `"ab"`. Rakendades sellele parserit `end`, mis ei mäleta, kui kaugele parser `p` jõudis, tulebki välja, nagu oleks viga kohe teksti alguses, kuigi tegelikult jõudis parser `p` kaugemale.

Selleks jätabki Awfuli parser alati meelde kõige suurema asukoha, kuhu mõnel katsel edukalt jõuti, ning see asukoht väljastataksegi programmeerijale veateates. Uue kõige kaugema vea-asukoha arvutamist teostab funktsioon `update_location`.

```
update_location :: State -> Location -> State
update_location (State a b) c = State a (max b c)
```

6.2 Nimekontroll

Awful paneb nimekontrollil kõik nimed ühte keskkonda. Iga faili läbib nimekontrollija kaks korda. Esimesel läbimisel korjatakse keskkonda kõik globaalsed nimed:

- Andmetüüpide nimed
- Andmekonstruktorite nimed
- Struktuuride ja hargnevate andmetüüpide väljade nimed
- Klasside nimed
- Meetodite nimed
- Definitsioonide nimed

Teisel läbimisel kontrollitakse lokaalseid nimesid:

- Iga andmetüübi jaoks kontrollitakse, kas kõik tüübimuutujate nimed on erinevad globaalsetest nimedest ja teineteisest.
    - Iga hargneva andmetüübi haru jaoks kontrollitakse, kas kõik antud harus sisse toodud tüübimuutujate nimed on erinevad globaalsetest nimedest, kogu hargneva andmetüübi jaoks kehtivatest tüübimuutujatest ja teineteisest.
- Iga klassi jaoks kontrollitakse, kas klassi päises deklareeritud tüübimuutuja on erinev globaalsetest nimedest.
    - Iga meetodi jaoks kontrollitakse, kas kõik tüübimuutujate nimed on erinevad globaalsetest nimedest, klassi päises deklareeritud tüübimuutujast ja teineteisest.

- Iga definitsiooni ja esindaja jaoks kontrollitakse, kas kõik tüübimuutujate nimed on erinevad globaalsetest nimedest ja teineteisest.
    - Iga avaldise jaoks kontrollitakse, kas iga uus sisse toodud lokaalne muutuja on erinev teistest antud skoobis kehtivatest nimedest.
        * Funktsiooni rakendamise korral kontrollitakse seda tingimust eraldi funktsiooni jaoks ja seejärel argumendi jaoks.
        * Lambda-avaldise korral kontrollitakse, kas muutuja nimi on erinev teistest antud skoobis kehtivatest nimedest. Seejärel kontrollitakse lambda-avaldise keha uue keskkonna all, mis on saadud vanast keskkonnast vastava muutuja lisamise teel.
        * Mustrisobitus-avaldise korral kontrollitakse sobitatav avaldis. Seejärel kontrollitakse kõik harud.
            · Iga haru jaoks kontrollitakse, kas mustrisobituse jaoks sisse toodud muutujate nimed on erinevad teistest antud skoobis kehtivatest nimedest ja teineteisest.
            · Haru resultaadiks olevat avaldist kontrollitakse uues keskkonnas, kuhu on lisatud ka mustrisobituse käigus sisse toodud lokaalsete muutujate nimed.

6.3 Tüübikontroll

Awfuli fail koosneb andmetüüpide deklaratsioonidest, klasside deklaratsioonidest ning definitsioonidest, sellises järjekorras. Kirjeldame, kuidas Awfuli tüübikontrollija neid keelekonstruktsioone käitleb.

6.3.1 Andmetüübid

Alustuseks sorteerib tüübikontrollija kõik failis esinevad andmetüübid edutatavateks ja mitteedutatavateks. Seejärel kontrollitakse ja kogutakse kogu vajalik teave alguses edutatavate ja seejärel mitteedutatavate andmetüüpide kohta. Selline läbimise järjekord on loogiline, sest edutatav andmetüüp ei tohi sõltuda ühestki mitteedutatavast andmetüübist, samas kui mitteedutatud andmetüübid võivad kasutada edutatavaid andmetüüpe nii tavalisel kui ka edutatud kujul.

Nii edutatavad kui ka mitteedutatavad andmetüübid läbitakse kaks korda. Kuna andmetüübid võivad teineteisest sõltuda, ei ole võimalik andmekonstruktorite argumentide tüüpide korrektsust kontrollida ühe läbimisega, sest selleks on tarvis teavet kõigi andmetüüpide kohta.

Esimesel läbimisel kontrollitakse, kas tüübimuutujate liigid on korrektsed. Kogutakse kokku teave tüübikonstruktorite liikide kohta. Ühtlasi lisatakse väärtustamiskeskkonda

kõik andmekonstruktorid ja väljade nimed (kuigi nende tüüpide korrektsust kontrollitakse alles teise läbimise käigus).

Hargnevate andmetüüpide puhul kontrollitakse lisaks tüübimuutujate liikide korrektsusele ka seda, et andmetüüp hargneks üle mõne liigi, mis on edutatud algebraline andmetüüp, ning et oleks kirjas täpselt üks haru iga vastava liigi tüübikonstruktori jaoks.

Teisel läbimisel, kui meil on juba olemas info kõigi tüüpide ja liikide kohta, kontrollitakse, kas andmestruktuuride väljade tüübid on korrektsed. Tüüpimiskeskkonda lisatakse andmekonstruktorite ja väljade tüübid.

6.3.2 Klassid

Andmetüüpide järel läbib Awfuli tüüpija klassid, kuna klassid kasutavad andmetüüpe aga mitte vastupidi. Klasse tuleb samuti läbida kaks korda, sest klasside meetodid võivad sisaldada kitsendusi ning seega on meetodeid kontrollides vaja teavet ka kõigi teiste klasside kohta.

Klasside esimesel läbimisel kontrollitakse, et liigid, üle mille klassid on deklareeritud, oleks korrektsed, ning kogutakse kokku teave klasside liikide kohta. Kogutakse kokku pärilust puudutav teave ning kontrollitakse, et ei esineks ringpärilust.

Iga meetodi jaoks kontrollitakse esimesel läbimisel, et tüübimuutujate liigid ja meetodi tüüp oleks korretksed.

Teisel läbimisel kontrollitakse, et klassid päriks ainult sama liiki klasse. Seda tuleb teha teisel läbimisel, kuna Awful ei nõua klasside kirja panemist pärilushierarhia järjekorras ja seega on päritavate klasside liikide sobivuse tuvastamiseks vaja enne koguda kokku kõigi klasside info.

Lisaks kontrollitakse teisel läbimisel meetodite kitsendusi, ehk seda, et kitsendused oleks rakendatud sobivat liiki tüübiparameetritele. Pärast meetodi kitsenduste kontrollimist lisatakse meetodi tüüp tüüpimiskeskkonda.

6.3.3 Definitsioonid, esindajad, avaldised ja tüübituletus

Andmetüüpide ja klasside järel liigub tüübikontrollija definitsioonide ja esindajate juurde. Definitsioonid ja esindajad kontrollitakse viimasena, sest nad kasutavad nii andmetüüpe kui ka klasse, samas kui andmetüübid ja klassid ei ole definitsioonidest kuidagi mõjutatud.

Ka definitsioonid ja esindajad läbitakse kaks korda, sest võib esineda rekursiooni ning avaldise tüübikontrolliks on tarvis teada kõigi definitsioonide tüüpe. Kitsenduste kontrolli jaoks on vaja teada, milliseid esindajaid programmis leidub.

Definitsioonide esimesel läbimisel kontrollitakse, et tüübimuutujate liigid ja kitsendused ning definitsiooni tüüp oleks korrektsed. Tüübikeskkonda lisatakse teave definitsiooni tüübi kohta.

Esindajate esimesel läbimisel tagatakse, et ei oleks deklareeritud korduvaid esindajaid. Iga esindaja puhul kontrollitakse esiteks, et klass ja tüübikonstruktor, millele üritatakse esindajat kirjutada, tõepoolest eksisteerivad. Kontrollitakse, kas tüüp, mida programmeerija

üritab klassi esindajaks defineerida, on antud klassi jaoks õiget liiki. Kontrollitakse, et esindajas leiduks õiges järjekorras kõik klassi meetodid.

Definitsioonide ja esindajate teisel läbimisel tüübitakse avaldised, kasutades esimesel läbimisel kogutud teavet definitsioonide tüüpide ning programmis leiduvate esindajate kohta.

Lisaks kontrollitakse teisel läbimisel, et kirjutatud esindajad rahuldaks pärilussuhteid. Juhul, kui tüüp T on teatud kitsendustel klassi C esindaja ning C on klassi B alamklass, siis tüüp T peab samadel või nõrgematel kitsendustel olema ka klassi B esindaja.

Awful võimaldab parameetrilist polümorfismi. Sarnaselt Haskelliga [5] ei nõua Awful polümorfsete funktsioonide kutsel tüübiargumentide eksplitsiitset edastamist. Selle tõttu on vajalik tüübituletus.

Avaldise tüüpimisel koostab Awful kaks võrrandisüsteemi. Esimene võrrandisüsteem koosneb tüüpide paaridest ja tähistab võrdusi tüüpide vahel. Teine võrrandisüsteem koosneb sõnede ja tüüpide paaridest ning tähistab kitsendusi. Kirjeldame lühidalt, kuidas Awfuli interpretaator avaldisi tüübib ja võrrandisüsteemid koostab. Olgu tüübitava avaldise tüübiks T.

- Primitiivide korral lisatakse süsteemi võrrand, et T on ekvivalentne vastava primitiivi tüübiga.

- Muutujanime x korral otsitakse nimi tüüpimiskeskkonnast üles. Juhul, kui seda ei leita, antakse kasutajale veateade, et muutuja x vastavas asukohas on defineerimata. Kui x leitakse tüüpimiskeskkonnast ning tema tüüp on U, kusjuures U võtab n tüübimuutujat, tuuakse sisse n uut tüübimuutujat $V_1 \ldots V_n$. Süsteemi märgitakse, et tüüp T võrdub tüübiga $U\ V_1 \ldots V_n$.

  Muutuja x tüüp võib sisaldada ka kitsendusi. Juhul, kui x tüübimuutuja number i on kitsendatud klassiga C, lisatakse kitsenduste võrrandisüsteemi kitsendus $C\ V_i$.

- Funktsiooni rakendamise avaldise $e_0\ e_1$ jaoks tuuakse sisse uus tüübimuutuja U, millega tähistatakse argumendi tüüpi. Avaldis e_0 tüübitakse arvestusega, et tema tüüp peab olema `Function` $U\ T$, ja avaldis e_1 tüübitakse arvestusega, et tema tüüp peab olema U.

- Lambda-avaldise x `->` e korral tuuakse sisse kaks uut tüübimuutujat U ja V ning süsteemi märgitakse, et tüüp T võrdub tüübiga `Function` $U\ V$. Seejärel tüübitakse avaldis e keskkonnas, kuhu on lisatud teave, et muutuja x tüüp on U, arvestusega, et avaldise tüüp peab olema V.

- Mustrisobitusavaldise puhul üle avaldise e kontrollitakse, et kõik mustrid oleks mõeldud sama tüüpi avaldise jaoks. Mustrid peavad kõik olema kas täisarvud, tähed, sama algebralise andmetüübi konstruktorid või sama mooduliga jäärgiklassiringi elemendid.

Ühtlasi kontrollitakse ka seda, et ei oleks kattuvaid mustreid ning et juhul, kui ei ole loetletud kõik võimalikud harud, leiduks lõpus ka vaikevalik.

Mustritest saab leida, mis on avaldise e tüüp. Juhul, kui tegu on primitiivide mustrisobitusega, saab avaldist e tüüpida arvestusega, et tema tüüp on vastava primitiivi tüüp. Juhul, kui tegu algebralise andmetüübi U mustrisobitusega ja tüübil U on n parameetrit, tuleb algebralise andmetüübi tüübiparameetrite jaoks sisse tuua tüübimuutujad $V_1 \ldots V_n$ ning e tüübitakse arvestusega, et tema tüüp on $U\ V_1 \ldots V_n$. See on mõnevõrra sarnane muutujanime tüüpimisega.

Tüüpida tuleb ka harude tulemuseks olevad avaldised. Kuna mustrisobituse tulemuse tüüp peab olema T, tuleb iga haru tulemuseks olev avaldis tüüpida arvestusega, et tema tüüp on T. Algebraliste andmetüüpide mustrisobituse korral lisanduvad igas harus keskkonda ka mustrisobituse käigus sisse toodud lokaalsed muutujad.

Awfuli parameetriline polümorfism on teatud määral sarnane Hindley-Milneri tüübisüsteemiga [17], mida kasutab ka Haskell [5]. Awful on vähem võimas ja ei luba let-avaldise sees polümorfset funktsiooni defineerida. Awfulis saab polümorfseid funktsioone defineerida ainult globaalsel tasemel ja neile on kohustuslik lisada tüübisignatuur. Kuna Awfuli let-avaldis on järjestikune ja võimaldab ainult monomorfseid väärtusi defineerida, teisendatakse see parsimise järel lambda-avaldiste rakendamise jadaks ning seda ei ole tüübikontrolli käigus vaja eraldi käsitleda.

Let-avaldise all polmorfsete funktsioonide defineerimine ei ole praktikas kuigi sageli kasulik ning Hindley-Milneri tüübisüsteemi täiendamine lisavõimalustega nagu näiteks tüübiklassid ja üldistatud algebralised andmetüübid muudab selliste üldistatud let-avaldiste tüübituletuse oluliselt keerulisemaks. Selle tõttu on tehtud ettepanek Haskellist let-avaldiste all polümorfsete funktsioonide defineerimise võimalus eemaldada [21] ja GHC keelab üldistatud algebraliste andmetüüpide laienduse kasutamise korral polümorfsed let-avaldised, mis kasutavad lokaalseid muutujaid [7].

Ka Awfulis on tüübiklassid. Hargnevate andmetüüpide kasutajasõbralikkuse huvides on tulevikus plaanis teostada mustrisobitus tüübimuutujate peal, mis sarnaselt üldistatud algebraliste andmetüüpide mustrisobitusega toob kaasa lokaalsed kitsendused. Lisaks ei ole veel teada, kuidas üldistatud let-avaldise tüübituletus ühilduks Awfulis tulevikus plaanitava tüübisüsteemi täiendusega: ideega käsitleda klasse kui alamliike, mida kirjeldame jaotises 7.7. Seega võib väita, et lokaalsete polümorfsete funktsioonide puudumine on pigem eelis kui puudus.

6.4 Väärtustamine

Awfulis kustutatakse tüübikontrolli järel avaldiste tüübid. Väärtustaja kasutab avaldisi, mille küljes ei ole teavet tüüpide kohta.

6.4.1 Struktuurid

Väärtustajas on struktuur esitatud tüübi `Map` abil. Struktuur on esitatud kui `Map`, kus sõnedele (väljade nimedele) vastavad avaldised. Struktuuri välja leidmiseks kasutab väärtustaja `Data.Map` mooduli meetodit `lookup`.

6.4.2 Algebralised andmetüübid

Algebralise andmetüübi esitus väärtustajas on list väljade sisust, millele on lisatud konstruktori nimi. Konstruktori nimi on algebraliste andmetüüpide puhul erinevalt struktuurist vajalik, sest seda kasutatakse mustrisobitusel õige haru leidmiseks.

6.4.3 Hargnevad andmetüübid

Kuna hargnevate andmetüüpide peal ei tehta mustrisobitust ja nende puhul on vaja väljade nimesid, siis nende esitamiseks kasutab väärtustaja sama viisi mis struktuuride jaoks.

6.4.4 Funktsiooni rakendamine

Awful on agar keel. See tähendab, et juba enne funktsiooni rakendamise tulemuste arvutamist väärtustatakse kõik argumendid.

6.4.5 Defineerimata käitumine

Muutuja `Crash` on keelde sisse ehitatud muutuja, mis esineb tüüpimiskeskkonnas, aga seda ei ole lisatud väärtustamiskeskkonda. Väärtustaja annab tulemuse `Maybe` monaadi all, ning kui muutuja väärtustamiskeskkonnast leidmine ebaõnnestub (mis võib juhtuda ainult muutuja `Crash` korral), antakse väärtustamise tulemuseks `Nothing` ja kasutajale kuvatakse, et väärtustamise lõpptulemuseks oli `Crash`.

6.4.6 Lambda-avaldised

Awfuli väärtustaja ei võimalda hetkel väärtustamiskeskkonda lokaalseid muutujaid lisada. Seetõttu on lambda-avaldise rakendamine teostatud asendamise abil: väärtustatakse argument, funktsiooni kehas asendatakse muutuja kõik esinemised väärtustatud argumendiga, ja seejärel väärtustatakse tulemuseks olev avaldis.

6.4.7 Mustrisobitus

Mustrisobitusavaldis on väärtustajas esitatud tüübi `Map` abil. Võtmeteks on primitiivide puhul vastavad primitiivid ja algebraliste andmetüüpide korral konstruktori nimesid tähistavad sõned. Alguses väärtustatakse sobitatav avaldis. Seejärel leitakse õige haru meetodi `lookup` abil. Juhul, kui tegu on mustrisobitusega, mis toob sisse uusi lokaalseid muutujaid,

asendatakse need muutujad vastavate avaldistega. Seejärel väärtustatakse tulemuseks olev avaldis.

6.4.8 Meetodid ja kitsendused

Kuna Awfuli väärtustajal ei ole teavet tüüpide kohta, ei saa meetodite puhul väärtustamise ajal otsustada, millist teostust on tarvis. Näiteks nähes nime `Write_Brackets`, ei teaks väärtustaja, kas tegu on tähemärkide, täisarvude või mõne muu tüübi kirjutamise meetodiga.

Jaotistes 2.1 ja 2.2 selgitasime, kuidas tüübiklassid võimaldavad anda sarnase otstarbega funktsioonidele sama nime ning lubavad vältida osade argumentide kirjutamist. Selleks, et saaks väärtustada ilma tüübiannotatsioonideta avaldisi, teostatakse tüübikontrolli järel vastupidine protsess [22]. Meetodite nimed täpsustatakse, nii et iga tüübi jaoks, mis on klassi esindaja, oleks meetodil unikaalne nimi. Kitsendused teisendatakse argumentideks. Kitsendatud funktsioonidele antakse lisa-argumendid.

Näiteks kui meil on klass

```
Class C{T : Star}(f : T -> T)
```

ning täisarvud ja kompleksarvud on selle klassi esindajad

```
Instance C{Int}(f = Negate)
Instance C{Complex T}<C T>(f = Complex (f (Real x)) (f (Imaginary x)))
```

siis avaldis

```
f (Complex 1 0)
```

teisendatakse enne väärtustamist järgmisele kujule:

```
f{Complex} f{Int} (Complex 1 0)
```

Kui kasutaja jaoks on meetod `f` ühe argumendiga funktsioon, siis väärtustaja näeb kitsendusi lisa-argumentidena. Näiteks meetodi `f` teostus kompleksarvude jaoks näeb väärtustajas välja järgnev:

```
f{Complex} f{T} x = Complex (f{T} (Real x)) (f{T} (Imaginary x))
```

7 Edasine töö

Awfuli peamisteks edasisteks eesmärkideks on võimalus kohelda liike kui kategooriaid ja klasse kui alamliike. Lisaks on kavas ka väiksemad täiendused parandamaks keele kasutuskõlblikkust.

7.1 Mustrisobitus tüübimuutujate peal

Hetkel on Awfulis ainus viis hargnevate andmetüüpide peal funktsioone teostada abitüübiklassid, mida kirjeldasime jaotises 5.8. Selle asemel on tulevikus kavas võimaldada mustrisobituavaldist tüübimuutujate peal. Mustrisobitus tüübimuutuja mitte andmetüübi enda peal võimaldaks ilma abiklasse kirjutamata ka selliseid meetodeid, millel ei ole argumente ja mille jaoks ei ole mustrisobitus argumentide peal seega põhimõtteliselt võimalik. Toome siinkohal näite vektorite tüübist kui `Applicative` klassi esindajast.

```
Instance Applicative{Array N}(
  Apply =
    Match N {
      !Zr -> _ -> _ -> Empty_Array,
      !Next N' ->
        Construct_Array f x -> Construct_Array y z ->
          Construct_Array (f y) (Apply x z)},
  Lift x =
    Match N {
      !Zr -> Empty_Array,
      !Next N' -> Construct_Array x (Lift x)})
```

Mustrisobitus tüübimuutujate peal ei pea tingimata tähendama tüüpide kaasas kandmist väärtustamise ajal. Mustrisobitust tüübimuutujate peal oleks tüüpimise käigus võimalik teisendada abitüübiklasside abil töötavaks lahenduseks, mida kirjeldasime jaotises 5.8. Sellisel juhul oleks tegu kasutajasõbralikuma süntaktilise suhkruga praegusele lahendusele.

7.2 Operaatorid

Magistritöö raames keskendus töö tüübiklasside ja edutamise arendamisele ning töö skoobist jäid välja mõned keele võimalusi otseselt mitte laiendavad, aga keele kasutusmugavuse ja koodi loetavuse seisukohast olulised aspektid. Hetkel on Awfuli süntaksi kõige suuremaks probleemiks operaatorite puudumine. See põhjustab pikemate avaldiste raskestiloetavust. On plaanis lisada võimalus lubada kasutajal binaarseid operaatoreid defineerida.

7.3 Struktuuride mustrisobitus

Operaatorite puudumise järel on Awfuli liigpikkade avaldiste ja halva loetavuse teine olulisim allikas struktuuride mustrisobituse puudumine. Selle tõttu on vaja sageli väljade nimesid kasutada.

```
Instance Bifunctor{Pair}(Bimap f g x = Pair (f (First x)) (g (Second x)))
```

Tulevikus on plaanis lisada struktuuride mustrisobituse võimalus:

```
Instance Bifunctor{Pair}(Bimap f g (Pair x y) = Pair (f x) (g y))
```

7.4 Detailsemad veateated

Hetkel antakse tüübivigade puhul ainult funktsioon, kus viga tekib. Ei öelda täpsemat põhjust. Tulevikus on plaanis veateatele lisada teave selle kohta, millise kahe tüübi unifitseerimine ebaõnnestus.

Teine veateadete liik, mida saaks oluliselt informatiivsemaks muuta, on veateated, mis tulenevad sellest, et kitsendatud meetodit üritatakse kutsuda väärtuse peal, mille tüübi jaoks need klassikitsendused ei kehti.

```
Def Max[T : Star](x : T, y : T) : T =
  Match Compare x y {LT -> y, Default -> x}
```

Antud näites on kasutaja unustanud kirjutada tüübikitsenduse, mistõttu funktsioon ei tüüpu. Hetkel öeldakse veateates, et vea põhjustas klass `Ord`, aga kasutajale ei öelda, et vea põhjustas meetod `Compare` ja et tüüp, mille jaoks klassikitsendus täitmata on, on `T`. Kasutaja peab ise välja mõtlema, kas viga tuleb puuduvast tüübimuutuja kitsendusest või hoopis kirjutamata jäänud klassi esindajast. Veateade, mis ütleks, mis tüübi või tüübimuutuja jaoks on kitsendus rahuldamata, aitaks seda laadi vigu kiiremini parandada.

7.5 Mitmene pärilus

Awfuli tüübiklasside kõige olulisem puudus võrreldes Haskelli omadega on mitmese päriluse puudumine. Edasise töö käigus on plaanis täiendada Awfuli tüübiklasse toetamaks mitmest pärilust.

7.6 Liik kui kategooria

Tüüpe saab vaadelda kui kategooriat. Kategooria objektid on tüübid ning nooled tüüpide `T` ja `U` vahel on funktsioonid tüüpi `Function T U`. Noolte kompositsiooniks on funktsioonide kompositsioon ja iga objekti (tüübi) ühiknool on vastava tüübi peal opereeriv

ühikfunktsioon [14, 12]. Selline vaade tüüpidele kui kategooriale on võimalik ka Awfuli puhul.

Kategooriatena saab aga vaadelda ka teisi liike peale liigi `Star`, näiteks paaride liiki (mis annab meile korrutiskategooria) ja liiki `Arrow`.

7.6.1 Kategooriate defineerimiseks vajalikud keele täiendused

Selleks, et liikide jaoks saaks defineerida noolte tüüpi, noolte kompositsiooni ja ühiknoolt, oleks Awfulit vaja täiendada järgmistel viisidel.

- Haskellis saab kategooriate defineerimiseks kasutada tüübiklasse, kuna Haskell võimaldab luua tüübiklasse, mis töötavad liikide mitte tüüpide peal, ning võimaldab tüübiklassi sees tüüpe defineerida. Awfuli tüübiklassid selliseid võimalusi hetkel ei paku. Üks võimalus Awfulis kategooriaid defineerida oleks tüübiklasse vastavalt täiendada. Teine võimalus oleks luua uus konstruktsioon tüübiklassidele sarnase süntaksiga, mis oleks mõeldud spetsiaalselt kategooriate defineerimiseks. Liigi `Star` kategooria võik selles süntaksis välja näha näiteks niiviisi:

  ```
  Category{Star}(
    T -> U = Function T U,
    Compose f g x = f (g x),
    Id x = x)
  ```

  Eemaldame keelest süntaktilise suhkru, kus operaator `->` tähistab tüüpi `Function`. Üldistame selle operaatori tähistama suvalise liigi noole tüüpi. Meetod `Compose` tähistab noolte kompositsiooni ja meetod `Id` tähistab ühiknoolt.

- Nagu näeme järgmistes jaotistes, on paljude kasulike kategooriate noolte tüübid liigipolümorfsed. Awfuli tüübisüsteemi on tegelikult juba ehitatud tugi liigipolümorfsete tüüpide jaoks, sest liigipolümorfsed tüübid on parametriseeritud andmetüüpide, näiteks listide edutamise vältimatu tagajärg. Tegu ei ole aga võimalusega, mida kasutajal on võimalik otse kasutada uute andmetüüpide loomisel – süntaks ei võimalda liigimuutujaid kasutada.

  Keelt oleks tarvis täiendada viisil, mis lubaks kasutajal liigipolümorfseid andmetüüpe luua. Plaanis on lubada liigimuutujate sisse toomine topeltkandiliste sulgude abil, näiteks `[[K, L]]`. Süntaks võtab eeskuju tüübimuutujate omast, selle vahega, et kui tüübimuutujate puhul tuleb märkida liigid, siis liigimuutujate puhul ei ole sortide täpsustamist tarvis. Kuna ei ole plaanis lubada polümorfismi üle argumente nõudvate liigikonstruktorite, on kõik liigimuutujad ühest sordist.

  Lisaks on vaja sisse ehitada liigipolümorfsete funktsioonide tugi, mida hetkel ei ole. Seda on tarvis põhjusel, et liigipolümorfsete tüüpide andmekonstruktorid ja väljad on samuti liigipolümorfsed.

- Samuti selgub järgmiste jaotiste näiteid vaadeldes, et praegused võimalused tüübimuutujate sisse toomiseks ei ole piisavad. Vahel, kui meil on ainult ühe tüübikonstruktoriga liik, on kasulik teostada tüübi mustrisobitust otse seal, kus deklareerime tüübimuutujaid. Toome näiteks liigi `!Wrapper`, millel on ainult üks tüübikonstruktor `!Wrapper`. Tahame struktuuri `Example` puhul kirjutada mitte `[W : !Wrapper K]`, vaid `[!Wrapper T : !Wrapper K]`, selleks, et saaks tüübi `!Wrapper` argumenti kasutada.

  ```
  Struct Wrapper[T : Star](Unwrap : T)
  Struct Example[[K]][!Wrapper T : !Wrapper K](f : T -> T)
  ```

- Tüübikonstruktorite liigi `Arrow` kategooria kirjutamisel tekib vajadus anda andmekonstruktorile polümorfset argumenti. Seega peab keel lubama tüüpe, mis sisaldavad mitte ainult kogu tüübi peal kehtivaid üldsuse kvantoreid, vaid ka üldsuse kvantoreid, mis kehtivad ainult ühe funktsiooniargumendi peal.

  ```
  Fun_Hom[F : Star -> Star, G : Star -> Star] :
    (Forall [T : Star] F T -> G T) -> Fun_Hom F G
  ```

Järgnevates alamjaotistes toome mõned näited kategooriatest, mida võiks Awfulis defineerida saada.

7.6.2 Vastandkategooria

Kategooria `K` vastandkategooria on kategooria, mis sisaldab samu objekte (tüüpe), aga kõik nooled on ümber pööratud. Vastandkategooria jaoks loome struktuuri

```
Struct Opp[T : Star](Opp' : T)
```

Edutamise teel saadakse sellest ühe argumendiga liigikonstruktor `!Opp` ning liigipolümorfne tüübikonstruktor `!Opp` liiki `K -> !Opp K`, kus `K` on suvaline liik. Lisaks objektidele on kategooria jaoks vaja ka nooli. Loome noolte jaoks tüübi

```
Struct Opp_Hom[[K]][!Opp T : !Opp K, !Opp U : !Opp K](Opp_Hom' : U -> T)
```

Juhul, kui liik `K` on kategooria, on kategooria ka liik `!Opp K`.

```
Category{!Opp K}<Category K>(
  T -> U = Opp_Hom T U,
  Compose (Opp_Hom f) (Opp_Hom g) = Opp_Hom (Compose g f),
  Id = Opp_Hom Id)
```

7.6.3 Korrutiskategooria

Korrutiskategooria moodustatakse kahest kategooriast `K` ja `L` niiviisi, et uue kategooria objektid (tüübid) on paarid, mille esimene element on kategooriast `K` ja teine element kategooriast `L`. Korrutiskategooria nooled on paarid kategooria `K` ja kategooria `L` nooltest.

Meil on juba olemas paaride tüüp ja andmekonstruktor. Sellest saab edutamise teel kahe argumendiga liigikonstruktori `!Pair` ja tüübikonstruktori `!Pair` liiki `K -> L -> !Pair K L`, kus `K` ja `L` on suvalised liigid. Loome paaride noolte tüübi.

```
Struct Pair_Hom[[K, L]][!Pair T U : !Pair K L, !Pair V W : !Pair K L](
  First_Hom : T -> V,
  Second_Hom : U -> W)
```

Paaride kategooria kood näeks välja niiviisi:

```
Category{!Pair K L}<Category K, Category L>(
  T -> U = Pair_Hom T U,
  Compose (Pair_Hom f g) (Pair_Hom h i) =
    Pair_Hom (Compose f h) (Compose g i),
  Id = Opp_Hom Id)
```

7.6.4 Tüübikonstruktorite kategooria

Ka tüübikonstruktorid on kategooria, eeldusel, et tulemuseks olevad tüübid moodustavad kategooria. Loome tüübikonstruktorite noolte tüübi, mis on liigipolümorfne üle tüübikonstruktori argumendi ja tulemuse liikide `K` ja `L` ning võtab kaks argumenti `F` ja `G` liigist `K -> L`. Andmekonstruktor võtab argumendiks ühe funktsiooni, mis on polümorfne üle tüübi `T` liigist `K` ja mille tüüp on `F T -> G T`.

```
Struct Fun_Hom[[K, L]][F : K -> L, G : K -> L](Fun_Hom'[T : K] : F T -> G T)
Category{Arrow K L}<Category L>(
  T -> U = Fun_Hom T U,
  Compose (Fun_Hom f) (Fun_Hom g) = Fun_Hom (Compose f g),
  Id = Fun_Hom Id)
```

7.7 Klass kui alamliik

Funktsioonid vajavad sageli kitsendusi. Näiteks kui meil on polünoomide tüüp `Polynomial T`, siis enamik kasulikke definitsioone ja klasse, mida polünoomide jaoks kirjutada saab, nõuavad, et tüüp `T` oleks kommutatiivne ring.

Üks kasulik funktsioon, mida saab kirjutada tüübi `F T` jaoks, kus `F` on funktor ja `T` on ring, on kõigi elementide sama väärtusega korrutamine. Niiviisi saame kirjutada skalaariga korrutamise funktsiooni paljude vektorruumide jaoks, näiteks kompleksarvud, maatriksid,

kvaternionid ja staatilise pikkusega vektorid. Toome siinkohal funktsiooni, mis korrutab kõik `y` elemendid vasakult väärtusega `x`.

```
Def Multiply_left[T : Star, F : Star -> Star]<Functor F, Ring T>
  (x : T, y : F T) : F T =
    Fmap (Multiply x) y
```

Ka polünoome saab arvuga läbi korrutada. Kuid erinevalt vektoritest ja maatriksitest, millel on kindel struktuur, mis jääb skalaariga korrutamisel muutumatuks, koosneb polünoom kordajate listist, ning selle listi pikkus võib arvuga korrutamise käigus muutuda, juhul kui see arv annab kõige kõrgema liikme kordajaga korrutades tulemuseks nulli.

Võtame näiteks polünoomi $p(x) = 2x + 3$ jäägiklassiringis mooduliga 4. Korrutades polünoomi p väärtusega 2, saame tulemuseks $3p(x) = 2$, sest 2 ja 2 on ringis mooduliga 4 nullitegurid. Algse polünoomi kordajate list on pikkusega 2, aga tulemuse kordajate listi pikkus on kõigest 1. Seega võiks `Fmap f` polünoomide korral pärast listi igale elemendile funktsiooni `f` rakendamist eemaldada kõrgematesse kordajatesse tekkinud nullid, et polünoom oleks ka pärast arvuga korrutamist normaalkujul. Seda aga ei ole võimalik saavutada standartses Haskellis ega ka Awfulis, sest meetodi `Fmap` tüüp on `(T -> U) -> F T -> F U`, kus `F` on funktor ning `T` ja `U` on kitsendamata tüübimuutujad. Samas `Fmap` meetodi teostus polünoomide jaoks, mis kustutaks pärast funktsiooni rakendamist ebavajalikud nullid, nõuaks, et `U` oleks ring (et oleks võimalik nulliga võrdlemine).

Teine näide tüübikonstruktorist, mille puhul nõuaks klasside meetodid lisakitsendusi, on `Set` [18]. Kuna `Set T` peal töötavad funktsioonid eeldavad, et `T` kuulub klassi `Ord`, tekib tüübikonstruktori `Set` jaoks `Fmap` meetodi kirjutamisel sama probleem mis polünoomide puhul.

7.7.1 Kitsenduste liik Haskellis

Üks võimalik lahendus antud probleemile Haskellis on kitsenduste liik `Constraint`. Näiteks saab Haskellis üldisema funktorite tüübi kirja panna järgmisel viisil [18]:

```
class Functor' f where
  type FConstraint f t :: Constraint
  fmap' :: (FConstraint f t, FConstraint f u) => (t -> u) -> f t -> f u
```

Klass `Functor'` sisaldab tüüpi `FConstraint`, mis võimaldab anda `f` argumendile kitsendusi. Kõik tüübikonstruktorid, mis on klassi `Functor` esindajad, on ka klassi `Functor'` esindajad. Lisaks saab klassi `Functor'` esindajaks defineerida selliseid tüübikonstruktoreid nagu `Set`, mille operatsioonid seavad tüübiargumendile kitsendusi.

```
instance Functor' Set where
  type FConstraint Set t = Ord t1
  fmap' f x = fromList (fmap f (elems x))
```

7.7.2 Klassikitsendused kui liigi osa

Liigi `K` peal töötavat klassi saab vaadelda liigi `K` alamliigina, ehk kitsendusi saab vaadata kui liigi osa. Muuhulgas võimaldab selline lähenemine teha kitsendatud operatsioonidega tüüpe tüübiklasside esindajateks ilma eelmises jaotises mainitud kitsenduste liiki kasutusele võtmata.

Toome näite sellest, kuidas see võiks toimida klassi `Functor` jaoks. Muudame klassi `Functor` liigipolümorfseks, andes talle liigimuutujad `K` ja `L`. Tüübimuutuja `F` liik on nüüd mitte `Star -> Star` vaid `K -> L` arvestamaks sellega, et funktor ei pruugi olla liigi `Star` endofunktor [18], vaid võib olla funktor kahe suvalise liigi vahel, mille jaoks on defineeritud kategooria koos vastava noole tüübiga.

Meetodi `Fmap` teostus on sarnane mis enne, selle vahega, et tüübimuutujate `T` ja `U` liik on nüüd `K` mitte `Star`. Meetod `Fmap` tavalise funktori korral on meetod, mis võtab argumendiks funktsioon tüüpi `Function T U` ja teeb sellest funktsiooni tüüpi `Function (F T) (F U)`. Üldistatult saab seda vaadata kui meetodit, mis võtab argumendiks liigi `K` noole tüüpide `T` ja `U` jaoks ning tagastab liigi `L` noole tüüpide `F T` ja `F U` jaoks.

```
Class Functor[[K, L]]{F : K -> L}(
  Fmap[T : K, U : K] : (T -> U) -> F T -> F U)
```

Juhul, kui kohtleme klasse alamliikidena, saab tüübikonstruktori `Set` liiki panna kirja kui `Ord -> Star`, mis võimaldab defineerida tüüpi `Set` klassi `Functor` esindajaks.

8 Kokkuvõte

Käesoleva töö raames loodud keel Awful on staatiliselt tüübitud puhas funktsionaalne programmeerimiskeel. Töö käigus sai teostatud Awfuli interpretaator ning kirjutatud keele süntaksi formaalne spetsifikatsioon ja keele kirjeldus koos ülevaatega vajalikest programmeerimiskeelte alastest taustateadmistest.

Erinevalt Haskellist, kus on kasutusel tavalised algebralised andmetüübid ning üldistatud algebralised andmetüübid, on Awfulis struktuurid, mis on mõeldud ühe konstruktoriga tüüpide jaoks, algebralised andmetüübid, mis sarnanevad Haskelli algebralistele andmetüüpidele selle vahega, et nõuavad vähemalt kahte konstruktorit, ning hargnevad andmetüübid, mis võimaldavad teatud piiratud juhtudel asendada üldistatud algebralisi andmetüüpe ilma tüübisüsteemi oluliselt täiendamata.

Awfuli tüübi- ja liigisüsteemi on täiendatud andmetüüpide edutamisega võttes eeskuju Haskellist. Edutamine muudab tüübisüsteemi rikkalikumaks, võimaldades tüübiturvaliselt luua näiteks tüübitaseme naturaalarvusid ja liste. Käesolev töö näitab, et edutatud liikidele ja tüüpidele leidub kasulikke rakendusi ka keeles, milles puuduvad üldistatud algebralised andmetüübid. Awfuli hargnevad admetüübid võimaldavad paljusid kasulikke edutamise rakendusi, näiteks staatilise pikkusega vektoreid ja üldistatud ennikuid.

Awfulis on olemas parameetriline polümorfism ja *ad hoc* polümorfism. Awfuli tüübiklassid võtavad üldjoontes eeskuju standartsest Haskellist. Ainus olulisem erinevus on mitmese päriluse puudumine, mis on kavas teostada edasise töö käigus. Käesolev töö näitab muuhulgas, kuidas tüübiklasse saab kasutada loomaks hargnevate andmetüüpide peal töötavad funktsioone, juhul kui keel ei toeta mustrisobitust tüübimuutujate ega hargnevate andmetüüpide peal. Tegu on võttega, mida saab kasutada ka Haskellis nendel juhtudel, kus üldistatud algebraliste andmetüüpide jaoks on vaja kirjutada meetodeid, mis ei võimalda mustrisobitust argumendi peal, näiteks klassi `Applicative` meetod `pure`.

Edasise töö käigus on plaanis täiendada keele tüübisüsteemi käsitlemaks liike kui kategooriaid ja klasse kui alamliike. See laseks üldistada teatud tüübiklasse (näiteks `Functor`) ja võimaldaks muuhulgas lahendada probleemi kitsendatud meetoditega, näiteks meetodi `Fmap` teostus tüübi `Set` jaoks, mis nõuab Haskellis kitsenduste liiki. Lisaks on kavas teostada mustrisobitus tüübimuutujate peal (eesmärgiga parandada hargnevate andmetüüpide kasutajasõbralikkust), mitmene pärilus ning mõned väiksemad täiendused, peamiselt kasutusmugavuse ja süntaksi vallas: operaatorid, struktuuride mustrisobitus ja detailsemad veateated.

Viidatud kirjandus

[1] Agda Language Reference (agda.readthedocs.io/en/v2.5.3/language, 14.04.2018).

[2] C# Guide (docs.microsoft.com/en-us/dotnet/csharp, 18.05.2018).

[3] The Coq Proof Assistant Reference Manual (coq.inria.fr/distrib/current/refman, 14.04.2018).

[4] Glasgow Haskell Compiler User's Guide (downloads.haskell.org/~ghc/latest/docs/html/users_guide, 02.04.2018).

[5] Haskell 2010 Language Report (haskell.org/onlinereport/haskell2010, 16.05.2018).

[6] The Idris Tutorial (docs.idris-lang.org/en/latest/tutorial, 14.04.2018).

[7] Let Generalisation in GHC 7.0 (ghc.haskell.org/trac/ghc/blog/letgeneralisationinghc7, 20.05.2018).

[8] Python Documentation (python.org/doc, 15.05.2018).

[9] Visual Basic Guide (docs.microsoft.com/en-us/dotnet/visual-basic, 11.05.2018).

[10] Wolfram Language Documentation (reference.wolfram.com/language, 26.04.2018).

[11] James Cheney and Ralf Hinze. First-class Phantom Types. Technical report, Cornell University, 2003.

[12] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and Loose Reasoning is Morally Correct. *SIGPLAN Not.*, 41(1):206–217, January 2006.

[13] Graham Hutton and Erik Meijer. Functional Pearl: Monadic Parsing in Haskell. 8, 07 1998.

[14] Joachim Lambek. Cartesian Closed Categories and Typed Lambda-Calculi. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, pages 136–175, London, UK, UK, 1986. Springer-Verlag.

[15] André Murbach Maidl, Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. Error Reporting in Parsing Expression Grammars. *Science of Computer Programming*, 132:129 – 140, 2016. Selected and extended papers from SBLP 2013.

[16] Conor Mcbride and Ross Paterson. Applicative Programming with Effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[17] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[18] Dominic A. Orchard and Alan Mycroft. Categorical Programming for Data Types with Restricted Parametricity. 2012.

[19] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[20] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000.

[21] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let Should Not Be Generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 39–50, New York, NY, USA, 2010. ACM.

[22] P. Wadler and S. Blott. How to Make *Ad-hoc* Polymorphism Less *Ad Hoc*. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[23] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971.

[24] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Liisi Kerik**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

   Funktsionaalse programmeerimiskeele liigisüsteem

   mille juhendaja on Härmel Nestra

   1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

   1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace'i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 21.05.2018