Astmelisustatud programmeerimine ja *MetaML*

Henri Lakk
henri.lakk@ut.ee
Programmeerimiskeelte semantika uurimisseminar
Arvutiteaduste Instituut, Tartu Ülikool

Kokkuvõte

Astmelisustatud programmeerimise (*staged programming*) üldine mõte, eriti programmeerimiskeelel *MetaML*, on luua tõhusaid programme. Astmelisustamise eesmärgiks pole programmid, mis arvutavad "korrektse" väljundi, vaid parem kontroll ressursside (nii aja kui mahu) üle. Selle saavutamiseks kasutatakse erilisi annotatsioone, et juhtida alam- avaldiste väärtustamise järjekorda.

Käesolev töö annab ülevaate *MetaML* keele programmide, andmetüüpide ja andmete astmelisustamisest ning tüüpimise süsteemist. Kirjeldatakse astmelisustamise eeliseid ja olemasolul viidatakse ka teadaolevatele probleemidele.

1 Sissejuhatus

Antud töö põhineb peamiselt kahel Tim Sherad'i artiklil programmide ning andmestruktuuride astmelisustamisest ja keelest *MetaML* [1, 2].

Lambda-arvutusest on teada, et väärtustamiste järjekorrast sõltub redutseerimisel tehtavate sammude arv. Kuna tehtavate sammude hulgast sõltub reduseerimiseks kasutavate ressursside suurus, annab väärtustuste järjekorra muutmise võimalus programmeerijale parema kontrolli ressursside kasutamise üle.

Tavaliselt on programmeerimiskeeltel kindel fikseeritud avaldiste väärtustamise strateegia (agarad ja laisad keeled), kus programmeerijal on vähe võimalusi väärtustamise järjekorda ilmutatud kujul muuta. Selle probleemi lahendamiseks on loodud spetsiaalsed astmelisustamise annotatsioonid, mis võimaldavad programmeerijatel täpselt muuta fikseeritud väärtustamise järjekorra reegleid vastavalt vajadusele.

MetaML lisab astmelisustamise annotatsioone *Standard ML* [1] [6] keelele. Astmelisustamise annotatsioonid jagavad programmi (funktsiooni) alam-

[1] Standard *ML* on universaalne, modulaarne, funktsionaalne programmeerimiskeel, millel on kompileerimisaegne tüübikontroll ja tüübi tuletamine (*type-inference*). Keel on populaarne kompilaatorite loojate, programmeerimiskeelte uurijate ja ka teoreemide automaattõestuste arendamisel [3].

avaldised etappideks (astmeteks) ning võimaldavad programmeerijal nende astmete käivitamise ja väärtustamise järjekorda juhtida. See võimaldab suurendada programmi efektiivsust, kuna arendaja saab eemaldada üleliigseid arvutusi.

2 Astmelisustamine

Palju vastandumist laisa väärtustuse keelte (nagu *Haskell*) ja range väärtustusega keelte (nagu *Standard ML*) vahel on tulnud tunnetuslikust väärtustuse ülemlikkusest teise ees. Funktsionaalsete andmestruktuuride uuringud on aga näidanud, et ükski kindel väärtustuse järjekord pole kõikides olukordades teistest parem [5].

Ka varasemalt on tehtud katseid väärtustus järjekorra manipuleerimiseks. Ranguse annotatsioonid realiseerivad laiskades keeltes ajutiselt agara väärtustamise ning "sunni" ja "viivita" annotatsioonid vastavalt agarates keeltes laisa väärtustamise. Agarates keeltes on võimalik simuleerida laiska väärtustamist kasutades lambda- abstraktsioonide viivituse efekti. Alljärgnev näide illustreerib tüüpilist laisa väärtustusega listi agaras keeles:

```
datatype 'a lazylist =
            lazyNil | lazyCons of 'a * (unit -> 'a lazylist);

fun count n = lazyCons(n, fn () => count (n+1))
```

Näide 1: Laisa väärtustusega list ranges keeles

Antud juhul on listi sabaks funktsioon, mis sunnib listi saba väärtustamist alles siis, kui antud funktsiooni rakendatakse.

Lambda-avaldiste kasutamisel on nii poolt- kui ka vastuargumente. Lambda-avaldiste heaks omaduseks on võimalus luua abstraktsioone, mida saab korduvalt taaskasutada. Nagu näitest 1 näha, võimaldavad need konstruktsioonid programmeerijal luua laisa väärtustusega lõpmatuid andmestruktuure isegi agarates keeltes. Sellisel lähenemisel on aga piirang - pole võimalik teha lambda-aluseid arvutusi, enne kui lambdat ennast pole rakendatud. See oluline piirang kehtib samaväärselt nii laiskade kui ka agarate keelte jaoks. Mõnikord on aga just sellist lambda-alust arvutust tarvis, kuid selle saavutamiseks puuduvad vahendid. Antud olukorra selgitamiseks on toodud järgnev näide:

```
fun power n = (fn x => if n=0 then 1 else x * (power (n-1) x))

map (power 2)  [1,2,3,4,5]
```

Näide 2: Funktsioon `power` ilma astmelisustamise annotatsioonideta

Viimane defineerib universaalse astendamise funktsiooni `power` ning ühe pisikese programmi, kus funktsioon `power` on kitsendatud ruutu tõstmiseks (eksponent `n` on fikseeritud arvuks 2), mida rakendatakse korduvalt `map`

funktsiooni poolt. Kõige efektiivsem strateegia oleks, kui funktsioon `power 2` avaldatakse ainult korra. Kuna aga funktsiooni avaldamise tulemuseks on lambda- avaldis, siis reaalselt seda siiski ei tehta. Kasutades kommentaarides [2] toodud reduktsiooni samme saame:

```
map (power 2) [1,2,3,4,5]
       (* avalda definitsioon *)
map (fn x => if 2=0 then 1 else x * (power (2-1) x))
    [1,2,3,4,5]
       (* teosta lambda-alune if  *)
map (fn x => x * (power (2-1) x)) [1,2,3,4,5]
       (* avalda power uuesti *)
map (fn x => x * ((fn x => if 1=0 then 1 else x * (power (1-1)
    x)) x))
        [1,2,3,4,5]
       (* kasutades beeta reeglit rakenda ilmutatud kujul
          lambda x-le *)
map (fn x => x * (if 1=0 then 1 else x * (power (1-1) x)))
    [1,2,3,4,5]
       (* teosta if *)
map (fn x => x * (x * (power (1-1) x))) [1,2,3,4,5]
       (* avalda power uuesti *)
map (fn x => x * (x * (fn x => if 0=0 then 1 else x * (power
    (0-1) x)) x))
        [1,2,3,4,5]
       (* kasutades beeta reeglit rakenda ilmutatud kujul
          lambda x-le *)
map (fn x => x * (x * (if 0=0 then 1 else x * (power (0-1) x)))
    )
        [1,2,3,4,5]
       (* teosta if *)
map (fn x => x * (x * 1)) 1,2,3,4,5]
       (* rakenda map funktsioon *)
[ 1,4,9,16,25]
```

Näide 3: Astmelisustamise annotatsioonideta funktsiooni `power` redutseerimine

Alles pärast funktsiooni `power` täielikku avaldamist saame korduvalt rakendada ruututõstmise funktsiooni kasutades selleks `map` funktsiooni. Funktsiooni `power` saab avaldada, kui rakendada redutseerimise reegleid lambda all. Selline lähenemine säästaks mitmekordsest redutseerimisest, kui rakendatakse funktsiooni `power 2`.

Keeles *MetaML* kasutatakse erilisi annotatsioone eelnevalt kirjeldatud probleemi lahendamiseks. Antud annotatsioonid võimaldavad viivitada avaldise redutseerimist, avaldiste liitmist üheks suuremaks avaldiseks ja üks, mis sunnib viivitatud avaldise redutseerimist.

- **Metasulud** (`< >`) viivitavad sulgude vahel oleva avaldise redutseerimist (vt lk 5).

[2] Kommentaarid asuvad märgipaaride `(*` ja `*)` vahel.

- **Paooperaator** (`~`) leevendab redutseerimist viivitava reeglite kehtivust alam- avaldistele. Antud operaatori puhul kehtib reegel avaldis `<...~<e> ...>` redutseerub avaldiseks `<...e ...>`, mida nimetatakse esimeseks koodi elimineerimise reegliks (vt lk 7).

- **`run` operaator** sunnib tema mõjus oleva koodijupi väärtustamist. Antud operaatori puhul kehtib reegel, kus avaldis `run < e >` redutseerub avaldiseks `e` , kui `e` ei sisalda paooperaatorit. Viimast redutseerimist nimetatakse koodi elimineerimise teiseks reegliks. (vt lk 8).

Esimene koodi elimineerimise reegel on vahend, mis teeb *MetaML* keele paindlikuks. Paooperaator võib asuda kõikjal, isegi lambda all. Näiteks avaldis `<fn x => ...~e ...>` sunnib alam-avaldise `e` väärtustamist, millega vastasel juhul oleks viivitatud kuni lambda rakendamiseni.

Kui kõik metasulud on eemaldatud rakendatakse vaikimisi väärtustamise strateegiat järelejäänud termile. Kasutades neid annotatsioone võib kirjutada funktsiooni `power` ümber järgnevalt:

```
fun power n =  fn x => if n=0
                       then <1>
                       else < ~x * ~(power (n-1) x) >)

map (run <fn z => ~(power 2 <z>)>)  [1,2,3,4,5]
```

Näide 4: Funktsiooni `power` astmelisustatud versioon

Kasutades vaikimisi range väärtustuse (*leftmost* ja *innermost*) strateegiat, kuid järgides samaaegselt annotatsioonide järjestust saame reduktsioonid järgnevalt:

```
map (run <fn z => ~(power 2 <z>)>)  [1,2,3,4,5]

map (run <fn z => ~(if 2=0 then <1> else < ~<z> * ~(power (2-1)
     <z>) >)>)
      [1,2,3,4,5]

map (run <fn z => ~< ~<z> * ~(power (2-1) <z>) >>)
    [1,2,3,4,5]

map (run <fn z => ~< z * ~(power (2-1) <z>) >>)   [1,2,3,4,5]

map (run <fn z => ~< z * ~(if 1=0 then <1>
                                  else < ~<z> * ~(power (1-1)
                                     <z>) >) >>)
      [1,2,3,4,5]

map (run <fn z => ~< z * ~< ~<z> * ~(power (1-1) <z>) >>>)
    [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * ~(power (1-1) <z>) >>>)
    [1,2,3,4,5]
```

```
map (run <fn z => ~< z * ~< z * ~(power 0 <z>) >>>)
    [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * ~<1> >>>)    [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * 1 >>>)    [1,2,3,4,5]

map (run <fn z => ~< z * z * 1 >>)    [1,2,3,4,5]

map (run <fn z => z * z * 1 >)  [1,2,3,4,5]

map (fn z => z * z * 1)  [1,2,3,4,5]

[1,4,9,16,25]
```

Näide 5: Astmelisustatud funktsiooni `power` redutseerimine

See toodud idee on astmelisustatud programmeerimise alus, millisel viisil programme koostatakse ja kasutatakse. Lambda-aluste reduktsioonide järjekorra juhtimise võimalus teeb antud paradigma oluliselt võimsamaks võrreldes traditsiooniliste paradigmadega, mis kasutavad üht fikseeritud väärtustuse strateegiat. Astmelisustatud programmeerimise paradigma ei võimalda koostada rohkem programme, kuid selle asemel võimaldab kõikidel programmidel juhtida nende endi väärtustuse järjekorda, andes sellega neile suurema kontrolli oma arvutuslike ressursside kasutamise üle.

3 *MetaML* tutvusus

MetaML on meta-programmeerimise süsteem. Keeles *MetaML* kodeeritakse meta-keelseid (ik *m*eta-language - keel, mis kirjeldab manipulatsioone) programme, mis manipuleerivad objekt-keelseid (ik *o*bject language - keel, mida manipuleeritakse) programme. *MetaML* keele meta- ja objekt-keeleks on *Standard ML*. Viimasest tulenevalt on kõik *Standard ML* keelsed programmid ka *MetaML* programmideks, kus aga ei sooritata manipulatsioone objektide tasemel. *MetaML* on mõeldud *Standard ML* keele konservatiivseks laienduseks - sellel on sama süntaks ja semantika, millele on lisatud mõned meta-taseme programmeerimise konstruktsioonid (astmelisustamise annotatsioonid). [4]

3.1 Nurksulgude operaator ehk metasulud

MetaML keeles kirjutatakse esimese astme avaldis kahe meta-sulu vahele. Näiteks, alljärgnev *MetaML* sessioon illustreerib konstandi `23` astmelisutamist:

```
-| <23>;
val it = <23> : <int>
```

Näide 6: Konstandi astmelisustamine

Avaldisel `<23>` on tüüp `<int>`, kus tüübid `int` ja `<int>` ei ole samad. Kui kasutada `<int>` väärtust täisarvuna tekib tüübikontrolli faasis viga:

```
-| <23> + 2;
Type Error:
Cannot unify the types: in type application the type
    constructors
do not match: int is not equal to <int>
in expression: (<23>,2)
```

Näide 7: Astmelisustamise tüüpimine

Vaatleme järgnevat koodi, kus `length` on juba eelnevalt defineeritud funktsioon.

```
-| <length 1,2]>;
val it = <%length 1,2]> : <int>
```

Näide 8: Vabade muutujata astmete vaheline säilivus

Sümbol `%` tagastatud väärtuses tähistab asjaolu, et `length` on muudetud väärtusest tegelikuks koodijupiks. Seda nimetatakse vabade muutujate või astmete vahelise säilivuse leksiliseks tuvastamiseks. Kuna *MetaML* keele operaatorid (näiteks + ja *) on analoogselt identifikaatorid, siis väga sageli on ka nende ees `%`, kui vastavat koodi kuvatakse.

Kõikide avaldiste väärtustamist, sealhulgas ka funktsionaalavaldisi, on võimalik viivitada. Vaatleme järgnevaid näiteid:

```
-| val idCode = <fn x => x>;
val idCode = <fn a => a> : ['b].<'b -> 'b>
-| <fn n => n + 1>;
val it = <fn a => a %+ 1> : <int -> int>
```

Näide 9: Funktsioonide tüüpimine

Muutuja `idCode` esitab puhast *MetaML* funktsiooni, mille tüübiks on `['b].<'b->'b>`, mis on polümorfne kood, mis sisaldab polümorfseid muutujaid `'b`. Nagu näitest näha, nimetab kuvamis-mehhanism seotud muutujad ümber, millest tulenevalt `<fn a => a>`. Avaldis `<fn n => n + 1>` kujutab endast funktsiooni, mis saab parameetriks täisarvu ja tagastab (ühe võrra suurema) täisarvu.

Iga koodijupi *tase* (*aste*) on võrdne teda ümbritsevate nurksulu paaride arv miinus teda ümbritsevate paooperaatorite arv. Lihtsad väärtused nagu 13 ja `(fn n => n + 1)` on 0-taseme kood. Avaldisees `<fn n => n + 1>` on funktsioon ühe paari nurksulgude vahel, seega on tegemist 1. taseme koodiga. Termis `<fn n => <n + 1> >` on alam- term `n + 1` teise taseme koodijupp.

```
-| <fn n => <n + 1>>;
val it = <fn a => <a %+ 1>> : <int -> <int>>
```

Näide 10: Kolmetasemeline programm

Avaldis `<fn n => <n + 1> >;` tähistab kolme tasemelist programmi. Tasemel 0 on tegemist andmetega, mis esitavad programmi (`<fn n => <n + 1> >;`). Selle programmi käivitamise tulemusel tagastatakse funktsioon, mida on võimalik kasutada 1. tasemel (`fn n => <n + 1>;`). Kui saadud funktsiooni rakendada täisarvule, tagastab see uue koodijupi, mida on võimalik kasutada 2. tasemel (`<%n %+ 1>`). [1]

3.2 Paooperaator

Nurksulgudes olevaid avaldisi võib vaadelda kui avaldisi, mille väärtustamist/ arvutamist viivitatakse. Sageli on aga mõistlik lubada teatuid reduktsiooni samme mõne suure nurksulgude vahel oleva avaldise sees, selle konstrueerimise ajal. Seda võimaldab *MetaML* kasutades prefiks paooperaatorit, milleks on tilde (~). Kuna tilde peab kindlasti asuma nurksulgude vahel, saab teda kasutada ainult esimesel ja sellest kõrgematel tasemetel. Vaatleme järgnevat funktsiooni `pair`.

```
-| fun pair x = <( ~x , ~x )>;
val pair = Fn : ['b].<'b> -> <('b * 'b)>
```

Näide 11: Funktsioon `pair`

Funktsioon `pair` saab sisendiks koodijupi, mis on tüüpi `<'b>` ja genereerib selle alusel uue koodijupi, mille tüüp on `<('b * 'b)>`. Sisendkood x teisendatakse koodiks *(x, x)*. Selle saavutamiseks peame lisama koodi x tulemuse kahte osasse, mida saavutatakse funktsiooni `pair` definitsioonis, kasutades paooperaatorit sümboli `x` ees.

Kui esimese taseme nurksulgude vahel esineb alam-avaldis `~e`, siis väärtustab süsteem `e` väärtuse koodijupiks `<v>` ning seejärel asendatakse antud avaldis avaldisega `v`. Antud protsessi nimetatakse esimeseks elimineerimise reegliks (avaldis `~<e>` asendub avaldiseks `e`).

```
-| (pair <17-4>);
val it = <(17 %- 4,17 %- 4)> : <(int * int)>
```

Näide 12: Funktsiooni `pair` kasutamise näide

Viimase avaldise redutseerimise sammud on järgnevad:

```
pair <17 %- 4>

<( ~<17 %- 4>, ~<17 %- 4> )>

<( 17 %- 4, 17 %- 4 )>
```

Näide 13: Näites 12 toodud avaldise redutseerimine

3.3 `run` operaator

Operaator `run` on astmelisustamise annotatsioon, mis ilmutatud kujul märgib, millal viivitatud arvutus tuleb sooritada (näiteks koodijupp).

```
-| val z = <27 - 15>;
val z = <27 %- 15> : <int>

-| run z;
val it = 12 : int
```

Näide 14: Operaator `run`

Operaator `run` võimaldab redutseerida koodi väärtuseni, mis saadakse antud koodi käivitamisel. Ühtegi arvutust enam edasi ei lükata ja tulemuseks on konkreetne väärtus. Teist koodi elimineerimise reeglit (vastavalt avaldis `run <e>` väärtustatakse avaldiseks `e`) saab kasutada ainult eeldusel, et `e` ei sisalda paooperaatoriga märgitud alam-avaldisi. Kui `e` aga sisaldab paooperaatoriga märgitud alam-avaldisi, siis tuleb nendele rakendada koodi esimest elimineerimise reeglit. Tegemist on väga olulise reegliga, kuna see nõuab, et kogu käivitatav kood (avaldis) peab olema "täielikult avaldatud"enne käivitamist.

Antud protsessi illustreerivad järgnevad reduktsioonid:

```
run <1 + ~( (fn x => x) <2+3> )>

run <1 + ~( <2+3> ) >

run <1 + ~<2+3> >

run <1 + 2+3 >

1 + 2+3

6
```

Näide 15: Operaatori `run` redutseerimine

3.4 `lift` operaator

Operaator `lift` sarnaneb metasulgudega teisendades avaldise koodijupiks. Antud operaator aga erineb metasulgudest, kuna enne sisendi viivitamist redutseerib ta selle. Antud erinevust illustreerib järgnev näide:

```
-| <4+1>;
val it = <4 %+ 1> : <int> (* k"aivitamist ei toimu *)

-| lift (4+1);             (* k"aivitatakse 4+1 *)
val it = <5> : <int>
```

Näide 16: Operaatori `lift` ja metasulgude erinevus

Operaatorit `lift` saab kasutada eelmise lõigu kahe astmelise avaldise arusaadavamaks tegemiseks. Kasutades `lift` operaatorit 2. astmes väärtustub seotud muutuja n literaal konstandina(väärtus 6) leksiliselt tuvastatud konstandi asemel (`%n` nagu eelnevas näites 16).

```
-| val x = <fn n => < ~(lift n) + 1> >;
val x = <fn a => <~(lift a) %+ 1> > : <int -> <int>>

-| val y = run x;
val y = fn : int -> <int>

-| val z = y 6;
val z = <6 %+ 1> : <int>

-| run z
val it = 7 : int
```

Näide 17: Operaator lift kasutamise näide

Tuleb märkida, et operaatorit `lift` pole võimalik rakendada kõrgemat järku (näiteks funktsionaalsetel) argumentidel, kuna seda pole neil defineeritud.

3.5 Vabade muutujate leksiline tuvastamine

Vabade muutujata leksilist tuvastamist kasutatakse varajasemalt eelnevates tasemetes defineeritud väärtustele viitamiseks.

```
-| val n = 10;
val n = 10 : int

-| val oder = <(n,3)>;
val oder = <(%n,3)> : <(int * int)>
```

Näide 18: Vabade muutujate leksiline tuvastamine

Antud näites muutuja `n` väärtustatakse tasemelt 0, aga `oder` väärtustuses, kus `n` esineb vabalt, viidatakse sellele 1. tasemelt. Käivitusajal, kui avaldist `<(n,3)>` väärtustatakse, peab süsteem arvutama koodijuppi, mis on seotud `n` väärtusega. Nimetatud koodijupi näol on tegemist konstandiga, kuna on teada, et `n` väärtuseks on 10. Seda nimetatakse astmete vaheliseks säilivuseks. Koodi kuvamise mehhanism väljastab kõik vabade muutujate leksiliselt tuvastatud konstandid annotatsiooniga `%`, millele järgneb antud vaba muutuja nimi, mille väärtust kasutati antud konstandi leidmiseks. Kõik vabad muutujad, sõltumata tüübist, metasulgude vahel konstrueerivad sellised konstandid. Nõnda moodustatakse funktsioonisest koodijupid.

Operaatori `lift` ja leksilise vabade muutujate erinevuseks on asjaolu, et operaatorit `lift` ei saa rakendada funktsionaalsetel väärtustel. Viimane on tingitud, kuna operaator `lift` peab konstrueerima avaldise, mis väärtustades tagastab sama väärtuse. Funktsioonide puhul pole see alati võimalik.

Kasutades tasemete vahelist säilivust on võimalik genereerida funktsioonist koodijupp. Tasemete vaheline säilivus konstrueerib konstandi, mida pole tarvis väärtustada, kui seda lõpuks käivitatakse. See annab võimaluse genereerida koodi ka funktsioonide jaoks [1]:

```
-| val inc = fn a => a+1;
val inc = fn : int -> int

-| val encodeInc = <inc 5>;
val encodeInc = <%inc 5> : <int>

-| run encodeInc;
val it = 6 : int
```

Näide 19: Tasemete vaheline täilivus

4 Andmete ja andmetüüpide astmelisustamine

4.1 Andmete astmelisustamine

Osutub, et lisaks programmi avaldiste väärtustuse järjekorra juhtimisele on efektiivsuse huvides otstarbekas kasutada astmelisustamist ka andmetüüpidel. Vaatleme järgnevat kahe listi ühendamise *MetaML* näidet:

```
(* app: 'a list -> 'a list -> 'a list *)
fun app [] ys       = ys
  | app (x::xs) ys = x :: app xs ys
```

Näide 20: Kahe listi konkateneerimise algne funktsioon `app`

Toodud funktsioon on defineeritud induktiivselt esimese argumendi suhtes, mida on võimalik ära kasutada lisades astmelisustamise annotatsioone:

```
(* app: 'a list -> <'a list> -> <'a list> *)
fun app1 [] ys       = ys
  | app1 (x::xs) ys = <x :: ~(app1 xs ys)>
```

Näide 21: Kahe listi konkateneerimise funktsioon `app1`

Antud funktsioon `app1` on tööajal koodigeneraator, kui käivitada see kahe argumendiga. Näiteks:

```
metaml> app1 [1,2] <[4,5]>;
<[1,2,4,5]>

metaml> <fn z => ~(app1 [3,6] <z>)>;
<fn z => 3 :: 6 :: z>
```

Näide 22: Kahe listi konkateneerimise funktsiooni `app1` rakendamine

Funktsioon `app1` töötab järgmiselt: kuna funktsiooni `app1` esimene argument on teada koodi genereerimise ajal (näiteks [1,2]) saab kasutada rekursiivset algoritmi osa vastava koodijupi genereerimiseks. Näite 22 2. pooles kasutatakse funktsiooni `app1` teise funktsiooni defineerimisel. Kui antud funktsiooni rakendada tööajal, teostab see listide liitmise üleliigse rekursioonita.

See sarnaneb protsessile, mida teevad osalised väärtustajad nagu *CMIX*. Programmeerija üleandeks on tuvastada staatilised ja dünaamilised sisendid. Osaline väärtustaja teeb mõned analüüsid, mis määravad andmed, mida on võimalik arvutada staatiliselt ja mida tuleb arvutada töötamise ajal. *MetaML* keeles peab programmeerija ka selle analüüsi iseseisvalt teostama ning lisama vastavaid annotatsioone.

Toodud näidet aga on võimalik veelgi enam täiustada. Funktsiooni `app1` definitsioon ei kajasta, et lõplik resultaat pole täielikult dünaamiline - vastuse kogu esimene ots on teada. Selle probleemi lahendamiseks tuleb kasutada osaliselt staatilist listi. Koostame uue listi tüübi ja sellele ka listide liitmisest uue versiooni:

```
datatype 'a list1 = Nil
                  | Cons ot ('a * 'a list1)
                  | Later of <'a list>

(* app2: 'a list -> 'a list1 -> 'a list1 *)
fun app2 [] ys       = ys
  | app2 (x::xs) ys = Cons (x, app2 xs ys)
```

Näide 23: Listide konkateneerimise funktsioon `app2`

Oluline on tähelepanu juhtida asjaolule, et viimane näide kasutab samaaegselt uut tüüpi (`list1`) kui ka vana stiili (`list`) liste. Siinkohal on tegemist astmelisustatud andmetüübi, mitte aga astmelisustatud funktsiooniga. Funktsiooni `app2` definitsioon on identne esialgse funktsiooni `app` definitsiooniga, kui jätta kõrvale astmelisustatud andmetüübi konstruktori kasutamine. Tüüp `list1` sisaldab kõiki staatilisi ja dünaamilisi liste ning sealjuures ka osaliselt staatilisi liste. Eksisteerib palju erinevaid liste, mis on täielikult staatiliste (`'a list`) ja täielikult dünaamiliste (`<'a list>`) listide vahepealsed, mida `list1` on suuteline eristama.

Algebralisi andmestruktuure, mis sisaldavad dünaamilist informatsiooni nimetatakse osaliselt staatiliseks andmetüübiks. Osaliselt staatilised andmetüübid erinevad osaliselt staatilistest andmetest. Osaliselt staatilised andmed on parameetritega andmetüübid, kus parameetriks on programmi kood. Antud olukorras on andmestruktuur teada, kuid mõned elemendid selles on dünaamilised. Näiteks on `<int> list` osaliselt staatilised andmed. Osaliselt staatiliste andmestruktuuride puhul on osa (või kogu) struktuurist teadmata. Näide sellise andmestruktuuri kohta on `int list1`.

4.2 Andmetüüpide astmelisustamine

Andmetüüpide astmelisutamise üheks viisiks on andmetüübile lisa konstruktorite lisamine. Näiteks võib tuua *ML* keele tüübi `sum` astmelisutatud versiooni:

```
datatype ('a, 'b) Either
    = InLeft of 'a
    | InRight of 'b
    | InLeftC of <'a>
    | InRightC of <'a>
    | SumC of <('a,'b) sum>
```

Näide 24: Astmelisustatud andmetüüp `Either`

Lisatud konstruktorid võimaldavad osaliselt staatilisi olekuid `Either` tüübist, ning võimaldavad lisa informatsiooni rakendamist koodi genereerimise ajal. Esimesed kaks konstruktorit on vastavalt tüübi `sum` konstruktorid `Left` ja `Right`. Need konstruktorid käsitlevad juhtumeid, kui kogu informatsioon on staatiliselt olemas - nii kuju kui ka sisu. Väga tõenäoline on aga juhtum, kus on teada ainult väärtuse kuju, kuid puudub tegelik sisu. Selliste olukordade käsitlemiseks on konstruktorid `InLeftC` ja `InRightC`. Alljärgnev on näide, kuidas sellist informatsiooni on võimalik kasutada:

```
(* f: <('a,'b) sum> -> (('a.'b) Either -> <c>) -> <c> *)
fun f x k = <case ~x of
               Left  c => ~(k (InLeftC  <c>))
             | Right c => ~(k (InRightC <c>)) >
```

Näide 25: Astmelisustatud andmetüübi kasutamine

Funktsioon `f` teostab töötamis-aegse kontrolli ja edastab osaliselt teadaoleva informatsiooni jätku `k`. Nüüd võib `k` koostada paremat koodi, kuna nüüd on teada andmete kuju koodi genereerimise ajal.

Lõpuks on aga võimalus, et midagi pole teada väärtuse kohta. Selliseid väärtuseid luuakse kasutades konstruktorit `SumC`, mis sisaldab koodi, mis omakorda leiab väärtuse.

Produktsioone on võimalik astmelisustada sarnasel viisil:

```
datatype ('a,'b) Pair
    = Pair   of ('a *'b)
    | PairL  of (<'a> *'b)
    | PairR  of ('a * <'b>)
    | PairLR of (<'a> * <'b>)
    | PairC  of <'a *'b>
```

Näide 26: Produktsioonide astmelisustamine

Konstruktor `Pair` koostab täielikult staatilise paari. Konstruktorid `PairL`, `PairR` ja `PairLR` koostavad paari, milles vastavalt vasak, parem või mõlemad

komponendid on dünaamilised. Võib tunduda, et konstruktor `PairC` on üleliigne, kuna alati on teada paari kuju kasutades selleks konstruktorit `PairLR`. Nende kahe konstruktori erinevust illustreerib järgnev näide:

```
fun loop() = loop()

fun f (PairLR (x,y))    = x
  | f (PairC c)         = < #1 ~c>

metaml> run (f (PairLR (<1>,<loop()>)))
1
metaml> run (f (PairC <(1,loop())>))
... Lõputu tsükkel ...
```

Näide 27: Produktsiooni astmelisustatud tüübi `Pair` konstruktorite `PairLR` ja `PairC` erinevus

Nende kahe konstruktori vahe ilmneb, kui käsitleda üht `PairLR` väärtust. Sellisel juhul on vaatluse all kaks koodijuppi, mida võidakse käivitada eraldi, et genereerida paari komponendid. Näiteks, kui pole vajadust arvutada teist komponenti, pole ka vajadust käivitada vastavat koodi. Vastupidiselt, kui käsitleda `PairC` väärtust, siis on vaid üks koodijupp, mis käivitades loob mõlemad paari väärtused korraga.

Sarnaselt eelnevaga on võimalik astmelisustada liste. Järgnevalt on defineeritud rekursiivne listide andmestruktuur:

```
datatype 'a List
    = Nil
    | Cons of ('a * 'a List)
    | ConsL of (<'a> * 'a List)
    | ConsR of ('a * <'a List>)
    | ConsLR of (<'a> * <'a List>)
    | ConsC of <'a * 'a List>
    | Cons1 of <'a * 'a List>
    | ConsL1 of < <'a> * 'a List >
    | ConsR1 of <'a * <'a List> >
    | ConsLR1 of < <'a> * <'a List> >
    | ConsC1 of < <'a * 'a List > >
```

Näide 28: Astmelisustatud listide rekursiivne definitsioon

Antud näites seostatakse `Nil` konstruktori `Left` rolliga ja `Cons` konstruktori `Right` rolliga. Uue astmeliustatud tüübi suurus kasvab ruutkeerukusega algsete komponentide ja alam-komponentide arvu suhtes. Väiksemate andmetüüpide ja nende andmetüüpide jaoks, kus igal konstruktoril on 0 või ainult üks alam-komponent, on selline lähenemine mõistlik. Kuid nagu näites 28 näha, on isegi listide, millel on algselt ainult kaks konstruktorit, muutub astmelisustamine kohmakaks. Esiteks sisaldab toodud andmetüüp korduseid (nt `ConsC` ja `Cons1`) ning mõned konstruktorid on vajalikud ainult juhul kui käsitleda liste rohkemas kui kahes arvutamise astmes.

Alternatiiv eelnevale listide astmelisustamisele on kasutada sama mustrit nagu eelnevalt tüübis `list1`. Iga andmetüübi kohta tuleb luua uus varjestav versioon, mis eksisteerib koos esialgse andmetüübiga. Varjestaval tüübil on üks lisa konstruktor. Järgnevates näidetes lisatakse varjestava tüübi nime ja konstruktorite lõppu sufiks "S":

```
datatype 'a List  = Nil
                  | Cons of 'a * 'a List;
datatype 'a ListS = NilS
                  | ConsS of 'a * 'a ListS
                  | ListS of <'a List>;
```

Näide 29: Alternatiivne astmelisustatud listi andmetüüp

Selles näites on varjestava tüübi lisa konstruktoriks ainus komponent, mis on esialgset tüüpi kood. Tegemist on tüübiga, mis katab enamuse, aga mitte kõiki, listide astendamisi. Konkreetseks olukorraks, mida antud tüüp ei kata, kui on teada, et struktuur on tüüpi `Cons`, kuid argument on dünaamiline paar.

5 Mustrite tuvastus

Termide teisendamise süsteemi võib vaadelda kui produktsioonide hulka, millel on nii vasak kui parem pool. Mõlemad pooled koosnevad mustritest, kus muster on term, mis sisaldab alam-termidena spetsiaalseid mustrite tuvastamise muutujaid.

Reeglit $(a \Rightarrow b)$ võib rakendada termile t kui t alam- term s vastab mustrile a mingi substitutsiooni σ alusel. Reeglit rakendades asendatakse termi t sees olev alam-term s termiga σb ja saadakse tulemus t'. Öeldakse, et term t teisendub (ühe sammuga) termiks t' ja seda tähistatakse $t \Rightarrow t'$. Esitame monoidi reeglid:

$$\begin{aligned} r_1 &: \quad x + 0 \Rightarrow x \\ r_2 &: \quad 0 + x \Rightarrow x \\ r_3 &: \quad x + (y + z) \Rightarrow (x + y) + z \end{aligned}$$

Reeglite muutujad x, y ja z on mustriks suvalisele termile. Kui mõni muutuja esineb reegli vasakus pooles enam kui ühel korral, siis mustri sobimiseks peavad olema kõik antud muutujaga kohakuti olevad termit olema identsed. Selline olukord tekib, kui mustris on üks muutuja enam kui ühe korra esindatud, nagu näiteks $(x + x)$.

Üldiselt reeglid ei muutu süsteemi töö käigus. Samas on aga on kõige lihtsam mustrite tuvastamise funktsioon samaaegselt ka seos (reegel) vaadeldava ja võrreldava mustri vahel. Viimane omadus annab võimaluse astmelisustamiseks: esimeses astmes on võimalik ”spetsialiseeruda“ ja eemaldada üleliigsete seoste moodustamist.

Alljärgnevas näites on toodud astmelisutamata termide mustrite tuvastuse *MetaML* näite programm. Termide ja mustrite esitamiseks kasutatakse andmetüüpe vastavalt `Term` ja `Pat`. Andmetüübi `Pat` struktuur sarnaneb termide struktuurile, kuid `Pat` andmetüübil on üks lisa konstruktor `VarP`, mida kasutatakse mustri muutujate esitamiseks.

Substitutsioon on kujutatud listina paaridest (`string` × `Term`). Mustri tuvastamine õnnestub vaste leidmisega, või ebaõnnestub, mis on käsitletud andmetüübis `Result`. Andmetüübid `Subst` ja `Result` on defineeritud koos termi parameetriga, mille olemasolu läheb tarvis siis, kui asendada `Term` andmetüüp astmelisustatud versiooniga.

```
datatype Term = Int of int  | Con  of (string * Term * Term)
datatype Pat  = IntP of int | ConP of (string * Pat  * Pat ) |
    VarP of string;
type 'a Subst       = (string * 'a) list
datatype 'a Result  = Fail | Ok of 'a Subst

(* Term -> Term -> bool *)
fun termEq (Int x) (Int y) = x = y
  | termEq (Con (con1,x1,y1)) (Con (con2,x2,y2))
        = (con1 = con2) andalso (termEq x1 x2) andalso (termEq
            y1 y2)

fun match1 (VarP) t env k =
      (case lookup x env of
         NONE    => k (Ok ((x,t)::env))
       | SOME t' => if termEq t t' then k (Ok env) else k Fail)
  | match1 (IntP x) (Int y) env k =
      if x = y then k (Ok env) else k Fail
  | match1 (ConP (conp,xp,yp)) (Con (con,x,y)) env k =
      if conp = con
         then match1 xp x env (fn Fail    => k Fail
                                | Ok env' => match1 yp y env"a
                                    k)
         else k Fail
  | match1 _ _ env k = k Fail
```

Näide 30: Andmetüübid `Term` ja `Pat` ning funktsioon `match1`

Funktsioon `match1` saab parameetriks mustri, termi, mida võrrelda mustriga, substitutsiooni ja jätku (*continuation*). Jätku parameeter on tüüpi `Term Result`. Funktsioon võrdleb termi ja mustrit komponentide kaupa. Kui struktuurid on ühilduvad, moodustatakse `Ok` resultaat ja sellele rakendatakse jätku. Kui struktuurid ei ole ühilduvad, siis rakendatakse jätku termile `Fail`.

Kui `VarP` juhtumi käsitlemisel muutuja juba esineb substitutsioonis, peame kontrollima termi, mis on selle muutujaga seotud oleks sama struktuuriga. Selle jaoks kasutatakse termide võrdsuse kontrollimise funktsiooni `termEq`. Selline olukord võib juhtuda, kui muster pole lineaarne, ehk sama mustri muutujat kasutatakse mustri mitmes osas, nagu $(x + x)$.

Funktsiooni `match1` võib kasutada rakendades talle mustri, termi, tühja

listi ja identiteedi jätku (*identity continuation*):

```
fun test p t = match1 p t [] id
```

Näide 31: Funktsiooni `mach1` kasutamine

Astmelisustamise võimekuse demonstreerimiseks võib funktsiooni `match1` ümber kirjutada kasutades sealjuures astmelisustamise annotatsioone:

```
fun match2 (VarP x) t env k =
      (case lookup x env of
         NONE    => k (Ok ((x,t)::env))
       | SOME t2 => <if termEq ~t ~t2 then ~(k (Ok env)) else
          ~(k Fail)>)
  | match2 (IntP x) term env k =
      <case ~term of
         Int i => if i = ~(lift x) then ~(k (Ok env)) else ~(k
            Fail)
       | Con (con,x,t) => ~(k Fail) >
  | match2 (ConP (conp,xp,yp)) term env k =
      <case ~term of
         Int i => ~(k Fail)
       | Con (con,x,y) => if con = ~(lift conp)
           then ~(match2 xp <x> env (fn Fail    => Fail
                                      | Ok env' => match2 yp <y
                                         > env' k))
           else ~(k Fail)>
  | match2 _ _ env k = k Fail
```

Näide 32: Näites 30 toodud funktsiooni `match1` astmelisustatud versioon `match2`

Antud versioon mustrite tuvastamisest (`match2`) saab sisendiks tundmatu termi `t`. Kuna termi `t` näol on tegemist koodijupiga, mis kujutab endast arvutust, mille käivitamise tulemuse tüüp on `Term` (vastandiks kompileerimise ajal teadaolev term), mille struktuuri pole võimalik määrata. Parim, mida antud olukorras teha annab on termi `t` lisamine genereeritud koodi nõnda, et selle struktuur selgitatakse välja programmi tööajal.

Kui rakendada funktsiooni `match2` mustrile, tühjale substitutsioonile ja sobivale jätkule genereeritakse tuvastus funktsioon, millest on eemaldatud üleliigne interpreteerimine, mis tuleb mustri läbimisest.

```
metaml> p1;
val it = ConP ("+",VarP "x",VarP "x") : Pat;

metaml> val matchfun = match2 p1;
val matchfun =
<(fn Int d => NONE
   | Con(c,b,a) =>
       if c = "+"
          then if termEq a b then SOME ([("x",b)]) else NONE
          else NONE)>
 : <Term -> (string * Term) list option>
```

Näide 33: Funktsiooni `match2` rakendamine

Rakendades funktsiooni `match2` suurematel mustritel, genereeritakse keerulisemad tuvastuse funktsioonid, kuid üldine printsiip jääb samaks.

6 Astmelisustatud programmide tüüpimine

Meta-operaatorid lisavad teatava keerukuse *Standard ML* keele tüüpimise reeglitele. *MetaML* tüübi kontrollimise mehhanism abistab programmeerijat mitme-tasemelise avaldise tüüpide järjepidavuse kontrollimisel:

- Igale kõrgema taseme avaldisele antakse kindel kooditüüp. Antud kooditüüp sisaldab informatsiooni, mis täielikult kirjeldab viivitatud koodi tüüpi. Keele *MetaML* semantika eeldab, et iga koodjupp on turvaliselt käivitatav keskkonnas, milles pole seoseid vabade muutujatega. Enamus vabu muutujaid teisendatakse astmete vahel säilivateks konstantideks, kuid metasulgude ja `run` operaatori omavaheline käitumine võimaldab mõnel vabal muutujal antud mehhanismist välja jääda. Viimase näiteks on paooperaatori ja `run` operaatori vahelise seose tüüpimine. Probleemseks osutub programm `<fn x =>  (run <x>)>` mis põhjustab töökäigus vea.

```
-| <fn x => ~(run <x>)>;
Error: The term:
x
in file 'top level' 18 - 19
variable bound in phase 1 used too early in phase 1
```

Näide 34: Paooperaatori ja `run` operaatori tüüpimise probleem

Antud viga tuleneb operaatorist `run n`, mis eemaldab metasulud ilma taseme n vähendamiseta. See on normaalne asjade kulg ja tegemist on korrektse operatsiooniga. Erandiks on juhtum, kus `run` operaatorit rakendatakse koodil, mis sisaldab vabu muutujaid, mis on seotud kõrgemal tasemel. Toodud näide 34 illustreeribki kirjeldatud olukorda, kus `x` on seotud esimesel tasemel, aga `run` käivitatakse tasemel 0.

Andmetüübi disainimine, mis suudaks selliseid probleeme lahendada on väga keeruline. On loodud süsteeme, selliste probleemide tuvastamiseks, aga paraku nende kasutamise tulemusena märgitakse vigaseks ka mõned korrektsed programmid, või nendes on tarvilik kasutada lisa annotatsioone.

MetaML keele loojad on selliste vigade suhtes võtnud seisukoha, et need on sarnased tühja listi pea või saba võtmise operatsioonidega: need on tüübitavad, aga nad tekitavad vea tööajal. Selliste vigade vältimine on programmeerija kohustuseks.

- Süsteem kontrollib kõrgema-astmelisuse avaldiste kooskõla. Sellega on mõeldud, et iga viivitatud programm iseenesest on alati korrektselt tüübitud. Näiteks avaldis `<1+true>` pole lubatud, kuigi seda võiks pidada 0-astme koodijupiks. See punkt tagab, et suvaline korrektselt tüübitud avaldis ei saa muutuda vigaseks üheski käivitamise astmes.

- Mitme-astmelise avaldiste faaside vahelist säilivust kontrollib tüübi kontrollija. See tagab, et programmeerija ei saaks kasutada muutujaid astmetes, enne kui need on defineeritud. Näiteks, pole lubatud avaldis `<fn x => ~x>`.

MetaML omistab igale meta-sulgude avaldisele tüübi $\langle\alpha\rangle$. Nurksulud näitavad, et tegemist on taseme 1 avaldisega. Taseme n avaldised on ümbritsetud n nurksulu paariga. Tase 0 avaldistel pole ümbritsetud nurksulgudega ning neile omistatakse tavaline tüüp.

```
-| <2>;
val it = <2> : <int>
```

Näide 35: Esimese astme konstant

Näites 35 on tegemist tase 1 avaldisega kus α `= int` . Tüübis $\langle\alpha\rangle$ näitab α tüüpi, mis saadakse vastava astme käivitamisel. Sellest tulenevalt on tüübi `<2>` tasemel 1 tüübiks `int`.

Sarnaselt eelnevale on võimalik tüüpida ka funktsioone:

```
-| <fn x => x+5>;
val it = <(fn a => a %+ 5)> : <int -> int>

-| fn => <x+5>;
val it = fn : int -> <int>
```

Näide 36: Esimese astme funktsioon

Kokkuvõte

Astmelisustatud programmeerimiskeel annab programmeerijatele uue paradigma, mis võimaldab neil luua efektiivsemaid programme. Töös on käsitletud astmelisustamise erinevaid vorme (andmete, andmestruktuuride ja funktsioonide astmelisustamine, mustrite tuvastamine).

Töös antakse ülevaade keele *MetaML* eelistest ja käsitletakse ka selle puuduseid. Suurim puudus ilmneb astmelisustatud programmide tüüpimise juures, mis võimaldab koostada programme, mille tüübikonfliktid selguvad alles käivitamise ajal.

Hoolimata keele realisatsiooni mõnest puudusest on *MetaML* väga paindlik keel, mis annab programmeerijale senisest suurema kontrolli koodi käivitamise kui ka avaldiste väärtustamise järjekorra üle. See omakorda võimaldab programme optimeerida ühtlasi kompileerimisel, kui ka tööajal.

Töö valmimist raskendas *MetaML* kompilaatori arvutisse paigaldamise ebaõnnestumine, mis ei võimaldanud autoril reaalselt testida töös toodud näiteid. Probleem põhjustas asjaolu, et Tim Sheradi koduleheküljelt viidatud *MetaML* lehekülg enam ei eksisteeri. Antud puudusest sai teavitatud ka keele autorit e-posti vahendusel.

Viited

[1] Using *MetaML*: A staged Programming Language. Tim Sherad. AFP 1998,
http://web.cecs.pdx.edu/~sheard/papers/summerschool.ps
(23.11.2008)

[2] Staging Algebraic Datatypes. Tim Sheard, Iavor Diatchki,
http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps (23.11.2008)

[3] Standard ML, Wikipedia
http://en.wikipedia.org/wiki/Standard_ML (15.12.2008)

[4] Introduction to multistaged Programming, Using *MetaML*. Tim Sherad. Zino Benaissa. Matthieu Martel.
http://web.cecs.pdx.edu/~sheard/papers/manual.ps (23.11.2008)

[5] Purely Functional Data Structures. Chis Okasaki. Cambridge University Press, 1998.

[6] Programming in Standard ML. Robert Harper. Carnegie Mellon University, 2005 http://www.cs.cmu.edu/~rwh/smlbook/ (15.12.2008)