TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Tõnn Talvik 132619IAPM

EFEKTIANALÜÜSIDEL PÕHINEVATE PROGRAMMITEISENDUSTE SERTIFITSEERIMINE

Magistritöö

Juhendaja: Tarmo Uustalu
Professor

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Tõnn Talvik

8. mai 2017

Annotatsioon

Tüübi- ja efektisüsteemid võimaldavad programmide dünaamilist käitumist analüüsida staatiliste tehnikatega. Analüüsi tulemust saab kasutada näiteks programmi optimeerimiseks.

Selle töö eesmärgiks on luua sõltuvate tüüpidega programmeerimiskeeles Agda idee tõestuse (*proof-of-concept*) raamistu efektide analüüsiks ja nendel põhinevateks programmiteisendusteks.

Töös vaadeldakse esimese näitekeelena tüübitud lambdaarvutust, mida on laiendatud eranditega. Keele termidele tuletatakse tüübid ning hinnatakse nende võimalikku efekti: õnnestumist, ebaõnnestumist ja staatiliselt määramata efektiga arvutused. Selleks defineeritakse arvutustüübid gradeeritud monaadi abil. Defineeritakse keele semantika ning tuuakse mõned programmilihtsustused tõestades, et need teisendused ei muuda programmi semantilist interpretatsiooni.

Teise näitena kasutatakse mittedeterministlikku keelt. Efektina hinnatakse programmi võimalike tulemuste arvu. Defineeritakse vastav gradeeritud monaad ja viiakse läbi tüübi- ja efektituletus. Näitekeelele antakse semantika ning tuuakse programmiteisenduste näited, ühtlasi tõestades viimaste korrektsust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 43 leheküljel, 4 peatükki, 33 joonist.

Abstract
Certification of effect-analysis based program transformations

Type-and-effect systems are used to statically analyze program dynamic behaviour. This allows to perform certain program optimizations.

Functional languages distinguish between value types and computation types. Monads are used to reason about the latter.

Recent research marries monadic computations and effect systems. A systematic approach has been given using graded monads, which employs preordered monoids.

The goal of this thesis is to give a proof-of-concept framework for effect-analysis based program transformations. The work is carried out in a dependently typed functional programming language called Agda. Such expressive type system allows to provide a proof of program's certification as it is written. Also, Agda itself is an experimental language and such task has not been tried earlier in this language.

The first example language considered is a typed lambda calculus extended with exceptions. The starting point is raw terms for which types and effects can be inferred. Computation types are defined using a graded monad specifically adapted to capture exception effects. The language semantics is given for refined terms. Structural transformations, i.e. weakening and contraction, are described next. Using those, a few example program optimizations, e.g. dead computation and duplicate computation removal, are defined. These transformations are proved to be correct using Agda as a metalanguage.

The second example considers a language which supports non-deterministic choice. Again, starting from raw terms, their types and effects can be inferred. The type of upper bounded vectors is defined to define the semantics of the non-deterministic language. A suitable graded monad is also defined. Proven optimizing transformations include failed computation and duplicate computation removal. Since the base language is the same as for exceptions, much of the framework developed for exceptions can be reused.

The thesis is in Estonian and contains 43 pages of text, 4 chapters, 33 figures.

Sisukord

Jooniste loetelu

1 Sissejuhatus

1.1 Taust

Tüübisüsteem võimaldab vältida programmides teatud käitusvigu. Efektisüsteemi võib vaadelda tüübisüsteemi edasiarendusena, kus lisaks tüüpidele on programm annoteeritud täiendava informatsiooniga, mis kirjeldab programmi käitumist ehk tema efekti käitusfaasis.

Efektisüsteeme on edukalt kasutatud avaldiste rehkendamise ajastamiseks paralleelarvutamisel, kus efektid piiravad arvutuste võimalikku skoopi [1]. Lihtne efektisüsteem on kasutusel ka Javas, kus meetodid on sildistatud eranditega, mis võivad tekkida vastava meetodi käitusel.

Staatilise programmianalüüsiga saab hinnata arvutuste võimalikke efekte. See võimaldab mh viia läbi optimeerivaid programmiteisendusi. Näiteks saab jälgida, milliseid mälupesasid loetakse ja kirjutatakse, ning selle teadmise alusel eemaldada "surnud" (*dead computation*) või liiased arvutused (*duplicated computation*) [2].

Klassikaliselt kasutatakse funktsionaalprogrammeerimises mittepuhaste arvutuse tüüpimiseks monaade, st tüübitud on ka arvutused, mitte ainult väärtused. See lubab arutleda erinevate arvutuste üle nagu näiteks mittedeterminism, erandid, olek jne, mis ei ole võimalik tavalises lambdaarvutuses [3].

Efektisüsteeme saab kohandada ka monaadide jaoks [4]. Süstemaatiline lähenemine monaadide ja efektide kokkupanekuks põhineb parameetrilistel efekti monaadidel ehk gradeeritud monaadidel, mis kasutavad eeljärjestatud monoidi efektide võrdlemiseks [5].

1.2 Ülesande püstitus

Agda on sõltuvate tüüpidega funktsionaalne programmeerimiskeel ja interaktiivne tõestusassistent, mis põhineb intuitsionistlikul tüübiteoorial. Selles kirjutatud programm on tõlgendatav ja automaatselt kontrollitav kui matemaatiline tõestus.

Selle töö eesmärgiks on realiseerida programmeerimiskeeles Agda idee tõendamise (*proof-of-concept*) raamistu efektide analüüsiks ja nendele põhinevateks programmiteisendusteks. Samas raamistus peab saama näidata, et need teisendused on korrektsed.

Agda on eksperimentaalne keel ja sedalaadi ülesande realisatsioon selles keeles on uudne. Uurimuse käigus tahame teada, kas niisugune töö on teostatav mõistliku vaevaga, kui õppimisele kuluv aeg maha arvata.

Teoreetilisel tasemel on uudne, et efektide analüüsid ja optimisatsioonid toimivad keele juures, mis toetab andmetüüpe, milleks antud töös on naturaalarvud lihtsaima näitena.

1.3 Ülevaade tööst

Teises peatükis realiseeritakse näitekeel, mille efektiks on erandid. Järgmiseks defineeritakse selliste efektide hindamine. Seejärel arendatakse näitekeelele tüübisüsteem, mille käigus rafineeritakse keelt lisades selle arvutustele efektid ja tüübid. Edasi antakse rafineeritud keele semantika ning tuuakse mõningased programmiteisendused, näidates, et semantiliselt on algne ja teisendatud programm ekvivalentsed.

Kolmandas peatükis tuuakse efektianalüüs ja optimeerimise näited mittedeterminismi toetava keele kohta, kasutades ära teises peatükis arendatud raamistut.

Töö käigus valminud lähtekood on tulemuste reprodutseerimiseks allalaetav aadressilt `https://github.com/tonn-talvik/msc`. Lähtekoodi kompileerimiseks on kasutatud Agda versiooni 2.5.1.1 koos standardteegi versiooniga 0.12. Mainitud tarkvarapaketid on tasuta installeeritavad Ubuntu 16.04 LTS või teistest varamutest.

2 Erandid

Selles töös vaadeldavaks baaskeeleks on tüübitud lambdaarvutus koos tõeväärtuste, naturaalarvude ja paaridega. Selles peatükis vaadeldakse keele laiendust eranditega.

Keele efekt seisneb selles, et arvutus kas õnnestub, mille korral tagastatakse väärtus, või ebaõnnestub, mille korral väärtust ei teki. Staatilise analüüsiga saab iga arvutuse efekti hinnata järgnevalt: kindel õnnestumine, kindel ebaõnnestumine või staatiliselt teadmata, kas arvutus õnnestub või mitte. Edaspidi öeldakse efekti hinnangu kohta ka lihtsalt efekt.

Järgnevates alapeatükkides defineeritakse selline keel Agdas, konstrueeritakse tüübituletus koos efektianalüüsiga, määratletakse hästi tüübitud termide semantika ning tuuakse mõned optimeerivate programmiteisenduste näited. Ühtlasi näidatakse teisenduste korrektsust.

2.1 Eranditega keel

Näitekeele grammatika saab esitada Backus-Naur kujul (BNF) järgnevalt, kus `t` on tüübid, `v` on väärtused ja `c` on arvutused:

$$\begin{aligned}
&\texttt{t ::= nat | bool | t} \bullet \texttt{t | t} \Rightarrow e \texttt{ / t} && (e \in \texttt{E}) \\
&\texttt{v ::= TT | FF | ZZ | SS v |} \langle \texttt{ v , v } \rangle \texttt{ | FST v | SND v} \\
&\quad\texttt{| VAR } n \texttt{ | LAM t c} && (n \in \mathbb{N}) \\
&\texttt{c ::= VAL v | FAIL t | TRY c WITH c} \\
&\quad\texttt{| IF v THEN c ELSE c | v \$ v | PREC v c c | LET c IN c}
\end{aligned}$$

Agdas vastastikku defineeritud väärtus- ja arvutustüübid on toodud joonisel 1. Lubatud väärtustüübid `VType` on naturaalarvud, tõeväärtused, teiste väärtustüüpide korrutised ja funktsiooniruumid. Arvutustüüpideks on efektiga `E` annoteeritud väärtustüübid. Efekt `E` on defineeritud alapeatükis 2.2.1.

Vastastikku defineeritud väärtus- ja arvutustermid on toodud joonisel 2. Termide konstruktorite nimetamisel on kasutatud suurtähti vältimaks võimalikke nimekonflikte Agda standardfunktsioonidega. Järgnevalt on selgitatud väärtustermi `vTerm` konstruktorite tä-

```
mutual
  data VType : Set where
    nat : VType
    bool : VType
    _●_ : VType → VType → VType
    _⇒_ : VType → CType → VType

  data CType : Set where
    _/_ : E → VType → CType
```

Joonis 1: Näitekeele tüübid.

hendust.

- `TT` ja `FF` koostavad vastavalt tõeväärtused tõene ja väär.
- `ZZ` koostab naturaalarvu 0 ja konstruktor `SS` oma argumendist järgneva naturaalarvu.
- $\langle$`_,_`$\rangle$ koostab oma argumentide paari.
- `FST` ja `SND` koostavad vastavalt argumendina antud paari esimese ja teise projektsiooni.
- `VAR` koostab de Bruijni indeksiga määratud muutuja. Iga selline indeks on naturaalarv, mis näitab seestpoolt mitmendale sidumisele antud muutuja viitab. Antud töös loendatakse sidumisi alates nullist.
- `LAM` on funktsiooniabstraktsiooni konstruktor, seejuures on funktsiooni parameetri väärtustüüp eksplitsiitselt annoteeritud. Funktsiooni kehaks on arvutusterm üle täiendava muutujaga laiendatud skoobi. St lambda seob funktsiooni kehas funktsiooni parameetrile vastava muutuja.

Järgnevalt on selgitatud arvutustermi `cTerm` konstruktorite (jn 2) tähendust ja vastavas arvutuses kätketud efekti.

- `VAL` tähistab õnnestunud arvutust, seejuures arvutuse tulemuseks on väärtustermiga antud konstruktori argument.
- `FAIL` tähistab arvutuse, mille väärtustüüp on eksplitsiitselt annoteeritud, ebaõnnestumist.
- `TRY_WITH_` on erandikäsitlejaga arvutus: kogu arvutuse tulemuseks on esimese argumendina antud termi arvutus, kui see õnnestub, vastasel korral aga teise argumendina antud termi arvutus.

```
mutual
  data vTerm : Set where
    TT FF : vTerm
    ZZ : vTerm
    SS : vTerm → vTerm
    ⟨_,_⟩ : vTerm → vTerm → vTerm
    FST SND : vTerm → vTerm
    VAR : ℕ → vTerm
    LAM : VType → cTerm → vTerm

  data cTerm : Set where
    VAL : vTerm → cTerm
    FAIL : VType → cTerm
    TRY_WITH_ : cTerm → cTerm → cTerm
    IF_THEN_ELSE_ : vTerm → cTerm → cTerm → cTerm
    _$_ : vTerm → vTerm → cTerm
    PREC : vTerm → cTerm → cTerm → cTerm
    LET_IN_ : cTerm → cTerm → cTerm
```

Joonis 2: Eranditega keele väärtus- ja arvutustermid.

- `IF_THEN_ELSE_` on valikuline arvutus: vastavalt väärtustermi tõeväärtusele on tulemuseks kas esimese (tõene haru) või teise (väär haru) arvutustermiga antud arvutus.
- `_$_` on esimese väärtustermiga antud funktsiooni rakendamine teise väärtustermiga antud väärtusele, kusjuures rakendamise efektiks on funktsiooni kehas peituv efekt.
- `PREC` on primitiivne rekursioon, mille korduste arv on määratud väärtustermiga. Esimene arvutusterm vastab rekursiooni baasile ja teine sammule, kusjuures sammuks on akumulaatori ja sammuloenduri parameetritega funktsioon. Kogu arvutuse efekt vastab kõigi osaarvutuste järjestikku sooritamisele.
- `LET_IN_` lisab esimese arvutustermiga antud väärtuse teise arvutustermi kontekstis esimeseks muutujaks. Arvutuse efekt vastab osaarvutuste järjestikku sooritamisele.

Joonisel 3 on toodud kahe naturaalarvu liitmise funktsioon väärtustermina `ADD` ning naturaalarvude 3 ja 4 liitmine arvutustermina `ADD-3-and-4`. Lisaks on toodud näide arvutustermist `BAD-ONE`, mida annab konstrueerida, kuid mis ei oma sisu: naturaalarvu null ei saa rakendada tõeväärtusele tõene. Sellised halvasti tüübitud termid tuvastatakse tüübituletusega (alaptk 2.3). Harjumuspärasemas süntaksis, kus de Bruijni indeksite asemel kasutatakse muutujate nimesid, saab need termid esitada järgnevalt:

$$\begin{aligned}\text{ADD} &:= \lambda x^{\mathbb{N}}.\text{val}\ (\lambda y^{\mathbb{N}}.\text{prec}\ y\ (\text{val}\ x)\ ((acc, i).\text{val}\ (\text{succ}\ acc)))\\ \text{ADD-3-and-4} &:= \text{let}\ f = \text{ADD}\ 3\ \text{in}\ f\ 4\\ \text{BAD-ONE} &:= 0\ \text{true}\end{aligned}$$

```
ADD : vTerm
ADD = LAM nat
          (VAL (LAM nat
                    (PREC (VAR 0)
                          (VAL (VAR 1))
                          (VAL (SS (VAR 0))))))

ADD-3-and-4 : cTerm
ADD-3-and-4 = LET ADD $ (SS (SS (SS ZZ)))
              IN VAR 0 $ (SS (SS (SS (SS ZZ))))

BAD-ONE : cTerm
BAD-ONE = ZZ $ TT

CMPLX : cTerm
CMPLX = LET TRY
               IF VAR 0
               THEN VAL (VAR 0)
               ELSE FAIL nat
            WITH VAL ZZ
        IN VAL (VAR 1)

SMPL : cTerm
SMPL = VAL (VAR 0)
```

Joonis 3: Näidisavaldised eranditega keeles.

Arvutusterm `CMPLX` (jn 3) eeldab, et kontekstis on vähemalt üks tõeväärtustüüpi muutuja. Vastavalt selle muutuja väärtusele arvutatakse tingimuslause üks harudest: tõese korral tagastatakse selle muutuja väärtus, väära korral aga naturaalarvu tüüpi arvutus ebaõnnestub. Antud töös loetakse tõeväärtused naturaalarvude alamtüübiks ja seega on selline tingimuslause hästi tüübitud. Edasi lisatakse sellele tingimuslause arvutusel erandikäsitleja, mis tagastab väärtuse null. Kogu erandikäsitlejaga arvutus on seotud arvutusse, mis tagastab konteksti teise muutuja, milleks on esialgse konteksti esimene muutuja.

Arvutusterm `SMPL` (jn 3) tagastab lihtsalt kontekstis oleva esimese muutuja. Alapeatükis 2.5 näidatakse, et termid `CMPLX` ja `SMPL` on semantiliselt ekvivalentsed kontekstis, kus on ainult tõeväärtustüüpi muutuja.

2.2 Erandite gradeering

Selles alapeatükis defineeritakse erandite efekti hinnangud, operatsioonid hinnangutel ja hinnangute omavaheline järjestus. Sellega võimaldatakse efektihinnangutega varustatud arvutustüübid ja nende alamtüüpimiine. Ühtlasi näidatakse, et selline hindamine rahuldab eeljärjestatud monoidi ja gradeeritud monaadi omadusi, millele tuginevad semantika (alaptk 2.4) ja optimisatsioonid (alaptk 2.5).

2.2.1 Erandite efektid

Erandite efektide tüüp `Exc` on toodud joonisel 4: konstruktor `err` vastab arvutuse ebaõnnestumisele, konstruktor `ok` arvutuse õnnestumisele ja konstruktor `errok` arvutusele, mille kohta pole teada, kas see õnnestub või mitte.

Efektide korrutamise tehe `_·_` (jn 4) vastab kahe arvutuse järjestikusele sooritamisele. Kui esimene osaarvutus õnnestub, siis kogu arvutuse efekt on määratud teise osaarvutuse efektiga. Kui üks osaarvutustest ebaõnnestub, siis ebaõnnestub kogu arvutus. Ülejäänud juhtudel puudub teadmine arvutuse õnnestumisest või ebaõnnestumisest. Efektide korrutamist kasutatakse `LET_IN_` arvutuse tüüpimisel (alaptk 2.1).

Erandikäsitleja võib parandada kogu arvutuse efekti hinnangut. Põhiarvutuse ja erandikäsitleja efektide kombineerimise tehe `_◇_` on defineeritud joonisel 4. Kui põhiarvutus ebaõnnestub, siis on kogu arvutuse efekt määratud erandikäsitleja efektiga. Põhiarvutuse õnnestumisel on kogu arvutus õnnestunud ja erandikäsitlejat ei arvutata. Kui põhiarvutuse õnnestumine pole teada, aga erandikäsitleja kindlasti õnnestub, siis õnnestub ka kogu arvutus. Ülejäänud juhtudel pole teada, kas kogu arvutus tervikuna õnnestub või mitte. Niisugune efekti hinnangu parandus leiab aset `TRY_WITH_` arvutuse tüüpimisel (alaptk 2.1).

Hinnangute hulga `Exc` konstruktorid moodustavad järgneva võre:

$$\begin{tikzcd} & \text{errok} \arrow[dl, dash] \arrow[dr, dash] & \\ \text{err} & & \text{ok} \end{tikzcd}$$

Hinnangute osaline järjestusseos `_⊑_` on toodud joonisel 5. See seos on definitsiooni järgi refleksiivne `⊑-refl`. Transitiivsus `⊑-trans` on tõestatav argumentide kuju juhtude läbivaatuse abil. Transitiivsuse seost oleks võimalik kodeerida järjestusseose konstruktorina, kuid see pole otstarbekas, kuna hilisemates tõestuses tekivad sellest täiendavad juhud, mida peab analüüsima.

Loomuliku tehete viisil saab defineerida kahe erandihinnangu ülemise ja alumise raja ning näidata nende sümmeetrilisust. Lihtsuse huvides on toodud ainult vastavad tüübisignatuurid, aga mitte definitsioonid (jn 5). Kuna kahel hinnangul ei pruugi alumist raja leiduda, siis on `_⊓_` tulemus mähitud `Maybe` andmetüüpi.

```
data Exc : Set where
  err : Exc
  ok : Exc
  errok : Exc

_·_ : Exc → Exc → Exc
ok · e = e
err · e = err
errok · err = err
errok · ok = errok
errok · errok = errok

_◇_ : Exc → Exc → Exc
err ◇ e' = e'
ok ◇ _ = ok
errok ◇ ok = ok
errok ◇ _ = errok
```

Joonis 4: Erandite efektid ja operatsioonid nendel.

```
data _⊑_ : Exc → Exc → Set where
  ⊑-refl : {e : Exc} → e ⊑ e
  err⊑errok : err ⊑ errok
  ok⊑errok : ok ⊑ errok

⊑-trans : {e e' e'' : Exc} → e ⊑ e' → e' ⊑ e'' → e ⊑ e''
⊑-trans ⊑-refl q = q
⊑-trans err⊑errok ⊑-refl = err⊑errok
⊑-trans ok⊑errok ⊑-refl = ok⊑errok

_⊔_ : Exc → Exc → Exc
--proof omitted
_⊓_ : Exc → Exc → Maybe Exc
--proof omitted
⊔-sym : (e e' : Exc) → e ⊔ e' ≡ e' ⊔ e
--proof omitted
⊓-sym : (e e' : Exc) → e ⊓ e' ≡ e' ⊓ e
--proof omitted

lub : (e e' : Exc) → e ⊑ (e ⊔ e')
--proof omitted
glb : (e e' : Exc) {e'' : Exc} → e ⊓ e' ≡ just e'' → e'' ⊑ e
--proof omitted
```

Joonis 5: Erandite efektide järjestus.

2.2.2 Eeljärjestatud monoid

Hulka `E`, millel on defineeritud korrutamine `_·_` ja ühikelement `i`, st `i` on ühik korrutamise suhtes nii vasakult `lu` kui ka paremalt `ru`, ning korrutamine on assotsiatiivne `ass`, nimetatakse monoidiks. Kui sellel hulgal on määratud kahekohaline seos `_⊑_`, mis on refleksiivne `⊑-refl` ja transitiivne `⊑-trans`, ning kehtib korrutamise monotoonsus `mon`, siis on tegemist eeljärjestatud monoidiga. Joonisel 6 on toodud eeljärjestatud monoidi kirje tüüp Agdas.

Saab näidata, et erandite efekti hinnangud `Exc`, korrutamine `_·_`, mille ühikuks on konstruktor `ok`, ja osaline järjestusseos `_⊑_` moodustavad eeljärjestatud monoidi. Vasakühiku tõestus tuleneb vahetult korrutamise definitsioonist. Paremühiku tõestamisel tuleb teostada varjatud argumendi konstruktori kuju juhtude läbivaatus ja seejärel lähtuda korrutamise definitsioonist. Assotsiatiivsus tõestatakse sarnaselt kasutades juhtude läbivaatust ja korrutamise definitsiooni. Monotoonsuse tõestuses vaadatakse läbi nii võimalikke efekte kui ka järjestusseost nende vahel. Kõik mainitud tõestused on toodud töö käigus valminud lähtekoodis.

2.2.3 Gradeeritud monaad

Monaad on järgnev kolmik: tüübikonstruktor `T`, ühik η (Haskelli “return”) ja nn Kleisli laiendamise operatsioon ehk sidumine `bind`[1].

```
T : Set → Set
η : {X : Set} → X → T X
bind : {X Y : Set} → (X → T Y) → (T X → T Y)
```

Seejuures peavad olema täidetud kolm monaadi seadust: vasakühiku seadus `mlaw1`, paremühiku seadus `mlaw2` ja assotsiatiivsus `mlaw3`.

```
mlaw1 : {X Y : Set} → (f : X → T Y) → (x : X) → bind f (η x) ≡ f x
mlaw2 : {X : Set} → (c : T X) → c ≡ bind η c
mlaw3 : {X Y Z : Set} → (f : X → T Y) → (g : Y → T Z) → (c : T X) →
        bind g (bind f c) ≡ bind (bind g ∘ f) c
```

Joonisel 7 on toodud eeljärjestatud monoidiga `OM` gradeeritud monaadi kirje tüüp Agdas. Efektiga `E` parametriseeritud tüübikonstruktor `T` koos ühikuga η ja sidumistehtega `bind` moodustab gradeeritud monaadi. Neelduvusega `sub` saab kahe efekti järjestatuse tõestuse-

[1]Antud töös on `bind`-i argumentide järjekord vahetatud võrreldes Haskelliga.

```
record OrderedMonoid : Set where
  field
    E : Set
    _·_ : E → E → E
    i : E

    lu : {e : E } → i · e ≡ e
    ru : {e : E } → e ≡ e · i
    ass : {e e’ e’’ : E} → (e · e’) · e’’ ≡ e · (e’ · e’’)

    _⊑_ : E → E → Set
    ⊑-refl : {e : E} → e ⊑ e
    ⊑-trans : {e e’ e’’ : E} → e ⊑ e’ → e’ ⊑ e’’ → e ⊑ e’’

    mon : {e e’ e’’ e’’’ : E} → e ⊑ e’’ → e’ ⊑ e’’’ → e · e’ ⊑ e’’ · e’’’
```

Joonis 6: Eeljärjestatud monoidi andmetüüp.

le tuginedes luua mingist monaadilisest väärtusest vastavalt suurema efektiga monaadilise väärtuse. Neelduvus `sub` peab olema refleksiivne `sub-refl`, transitiivne `sub-trans` ja sidumise suhtes monotoonne `sub-mon`. Samuti peavad olema täidetud gradeeritud versioonid monaadi seadustest `mlaw1`, `mlaw2` ja `mlaw3`. Viimaste juures on kasutatud neelduvuse erijuhtu `sub-eq` efektide võrdsuse korral pääsemaks mööda Agda tüübisüsteemist: võrdsust ei saa tõestada eri tüüpi elementidele.

Eranditega arvutuste jaoks saab defineerida gradeeritud monaadi, mis on gradeeritud erandiefekti hinnangutega. Joonisel 8 on toodud olulisemad definitsioonid. Tüübikonstruktor `T` on defineeritud erandi hinnangu argumendi kuju järgi: veale `err` vastab üheelemendile tüüp ⊤, õnnestumisele `ok` parameetriga antud tüüp `X` ja hinnangule `errok` vastab tüüp `Maybe X`. Ühikuks η on identsusfunktsioon. Sidumise `bind` definitsioonil on analüüsitud kummagi efekti kuju ning vajadusel ka monaadilise väärtuse kuju. Neelduvus `sub` annab efektide refleksiivsuse `⊑-refl` korral etteantud monaadilise väärtuse `c`. Kui järjestuse tõestuse esimeseks efektis on `err`, siis vastavalt tüübikonstruktori definitsioonile saab argument olla hulga ⊤ ainus element `tt`, millele pannakse vastavusse `nothing`. Kui aga efektiks on `errok`, siis vastav väärtus `x` mähitakse `Maybe X` hulka.

2.3 Tüübi- ja efektituletus

2.3.1 Alamtüübid

Väärtus- ja arvutustüüpide osaline järjestus on vastastikku defineeritud (jn 9). Konstruktoriga `st-bn` loetakse tõeväärtused naturaalarvude alamtüübiks. Kehtib väärtustüü-

```
subeq : {E : Set} → {T : E → Set → Set} → {e e' : E} → {X : Set} →
        e ≡ e' → T e X → T e' X
subeq refl p = p


record GradedMonad : Set where
  field
    OM : OrderedMonoid
  open OrderedMonoid OM
  field

    T : E → Set → Set
    η : {X : Set} → X → T i X
    bind : {e e' : E} {X Y : Set} → (X → T e' Y) → (T e X → T (e · e') Y)

    sub : {e e' : E} {X : Set} → e ⊑ e' → T e X → T e' X

    sub-mon : {e e' e'' e''' : E} {X Y : Set} →
              (p : e ⊑ e'') → (q : e' ⊑ e''') →
              (f : X → T e' Y) → (c : T e X) →
              sub (mon p q) (bind f c) ≡ bind (sub q ∘ f) (sub p c)

  sub-eq : {e e' : E} {X : Set} → e ≡ e' → T e X → T e' X
  sub-eq = subeq {E} {T}

  field
    sub-refl : {e : E} {X : Set} → (c : T e X) → sub ⊑-refl c ≡ c
    sub-trans : {e e' e'' : E} {X : Set} →
                (p : e ⊑ e') → (q : e' ⊑ e'') → (c : T e X) →
                sub q (sub p c) ≡ sub (⊑-trans p q) c

    mlaw1 : {e : E} → {X Y : Set} → (f : X → T e Y) → (x : X) →
            sub-eq lu (bind f (η x)) ≡ f x
    mlaw2 : {e : E} → {X : Set} → (c : T e X) →
            sub-eq ru c ≡ bind η c
    mlaw3 : {e e' e'' : E} → {X Y Z : Set} →
            (f : X → T e' Y) → (g : Y → T e'' Z) → (c : T e X) →
            sub-eq ass (bind g (bind f c)) ≡ bind (bind g ∘ f) c
```

Joonis 7: Gradeeritud monaadi andmetüüp.

```
T : Exc → Set → Set
T err X = ⊤
T ok X = X
T errok X = Maybe X

η : {X : Set} → X → T ok X
η x = x

bind : {e e' : Exc} {X Y : Set} →
       (X → T e' Y) → T e X → T (e · e') Y
bind {err} f c = tt
bind {ok} f c = f c
bind {errok} {err} f c = tt
bind {errok} {ok} f (just x) = just (f x)
bind {errok} {ok} f nothing = nothing
bind {errok} {errok} f (just x) = f x
bind {errok} {errok} f nothing = nothing

sub : {e e' : Exc} {X : Set} → e ⊑ e' → T e X → T e' X
sub ⊑-refl c = c
sub err⊑errok tt = nothing
sub ok⊑errok x = just x
```

Joonis 8: Osa erandite gradeeritud monaadi definitsioonist.

pide refleksiivsus `st-refl`. Kahe väärtustüübi korrutis on teise sarnase korrutise alamtüüp `st-prod`, kui vastavad tegurid on alamtüübid. Funktsiooniruumid on alamtüübid `st-func`, kui tagastustüübid on alamtüübid, ja parameetri tüübid on ülemtüübid. Arvutustüüp on teise arvutustüübi alamtüüp `st-comp`, kui nende efektid ja väärtustüübid on järjestatud.

Väärtus- ja arvutustüüpide alamtüüpimise transitiivsus on tõestatud vastastikku joonisel 9. Kui kahe väärtustüüpide alamtüüpimise väite tõestusest üks on alamtüüpimise refleksiivsuse aksioomi `st-refl` kujul, siis transitiivsuse tõestuseks on teine etteantud tõestus. Kui üks etteantud tõestustest on koostatud reeglist `st-prod`, siis ka teine tõestus peab olema paratamatult samal kujul. Sellisel juhul on transitiivsuse tõestus saadav reegli `st-prod` rakendamisega rekursiivelt määratud tegurite transitiivsuste `st-trans` tõestustele. Kui üks etteantud tõestustest on koostatud reeglist `st-func`, siis on seda paratamatult ka teine tõestus. Sellisel juhul tõestatakse transitiivsus reegli `st-func` rakendamisega rekursiivselt väljakutsutud argumentide transitiivsuse `st-trans` ja kehade arvutustüüpide transitiivsuse `sct-trans` tõestustele. Tähelepanu tuleb seejuures pöörata funktsiooni parameetri alamtüüpimise transitiivsusele, kuna parameetri tüüp on funktsioonitüübis kontravariantne.

Arvutustüüpide alamtüüpimise transitiivsuse `sct-trans` (jn 9) argumendid saavad olla ainult reegli `st-comp` kujul. Transitiivsuse tõestus saadakse reegli `st-comp` rakendamisega

```
mutual
  data _≤V_ : VType → VType → Set where
    st-bn : bool ≤V nat
    st-refl : {σ : VType} → σ ≤V σ
    st-prod : {σ σ' τ τ' : VType} →
              σ ≤V σ' → τ ≤V τ' → σ • τ ≤V σ' • τ'
    st-func : {σ σ' : VType} {τ τ' : CType} →
              σ' ≤V σ → τ ≤C τ' → σ ⇒ τ ≤V σ' ⇒ τ'

  data _≤C_ : CType → CType → Set where
    st-comp : {e e' : E} {σ σ' : VType} →
              e ⊑ e' → σ ≤V σ' → e / σ ≤C e' / σ'

mutual
  st-trans : {σ σ' σ'' : VType} → σ ≤V σ' → σ' ≤V σ'' → σ ≤V σ''
  st-trans st-refl q = q
  st-trans p st-refl = p
  st-trans (st-prod p p') (st-prod q q') = st-prod (st-trans p q)
                                                   (st-trans p' q')
  st-trans (st-func p p') (st-func q q') = st-func (st-trans q p)
                                                   (sct-trans p' q')

  sct-trans : {σ σ' σ'' : CType} → σ ≤C σ' → σ' ≤C σ'' → σ ≤C σ''
  sct-trans (st-comp p q) (st-comp p' q') = st-comp (⊑-trans p p')
                                                    (st-trans q q')
```

Joonis 9: Väärtus- ja arvutustüüpide alamtüüpimine.

efektide järjestuse transitiivsuse `⊑-trans` ja väärtustüüpide alamtüüpimise transitiivsuse `st-trans` tõestustele.

2.3.2 Rafineeritud keel

Joonisel 10 on toodud vastastikku defineeritud rafineeritud väärtus- ja arvutustermid. Võrreldes alaptk 2.1-s toodud toorete termidega, on rafineeritud termid parametriseeritud kontekstiga Γ ning indekseeritud vastavalt väärtus- või arvutustüübiga ehk nad "teavad" oma konteksti ja tüüpi. Kontekst `Ctx` on defineeritud kui väärtusttüüpide list, mille elementide järjekord vastab vabade muutujate sissetoomise järjekorrale.

- Konstruktorid `TT` ja `FF` koostavad tõeväärtustüüpi termid tõeväärtuste tõsi ja väär jaoks.
- Konstruktor `ZZ` koostab naturaalarvu tüüpi termi arvu 0 tähistamiseks. Konstruktor `SS` koostab termi antud naturaalarvu tüüpi termi järgarvu tähistamiseks, mis on samuti naturaalarvu tüüpi.
- `⟨_,_⟩` koostab kahest antud väärtustermist paari, mille tüüp on termide tüüpide korrutis.
- `FST` ja `SND` projekteerivad korrutise tüüpi termist vastavalt esimese või teise korrutatava tüüpi termi.
- `VAR` konstrueerib vaba muutuja; ta võtab tõestuse, et mingi tüüp on konteksti element, ning annab väärtustermi, mille tüüp on kõnealuse elemendiga määratud tüüp.
- `LAM` võtab väärtustüübi ja arvutustermi, mille konteksti on parameetriga antud kontekstiga võrreldes väärtustüübiga laiendatud, ning annab funktsioonile vastava väärtustermi.
- `VCAST` suurendab etteantud väärtustermi tüüpi vastavalt etteantud alamtüüpimistõestusele. See võimaldab eri tüüpi väärtustermide tüüpe ühtlustada, mis on vajalik rafineeritud arvutustermide koostamisel.

Rafineeritud arvutustermid (jn 10) määravad täpselt osaarvutuste efektide kombineerimise.

- `VAL` koostab antud väärtustermist õnnestunud arvutuse.
- `FAIL` koostab etteantud väärtustüüpi ebaõnnestunud arvutuse.

```
Ctx = List VType

mutual
 data VTerm (Γ : Ctx) : VType → Set where
   TT FF : VTerm Γ bool
   ZZ : VTerm Γ nat
   SS : VTerm Γ nat → VTerm Γ nat
   ⟨_,_⟩ : {σ σ' : VType} →
           VTerm Γ σ → VTerm Γ σ' → VTerm Γ (σ • σ')
   FST : {σ σ' : VType} → VTerm Γ (σ • σ') → VTerm Γ σ
   SND : {σ σ' : VType} → VTerm Γ (σ • σ') → VTerm Γ σ'
   VAR : {σ : VType} → σ ∈ Γ → VTerm Γ σ
   LAM : (σ : VType) {τ : CType} →
         CTerm (σ :: Γ) τ → VTerm Γ (σ ⇒ τ)
   VCAST : {σ σ' : VType} → VTerm Γ σ → σ ≤V σ' → VTerm Γ σ'

 data CTerm (Γ : Ctx) : CType → Set where
   VAL : {σ : VType} → VTerm Γ σ → CTerm Γ (ok / σ)
   FAIL : (σ : VType) → CTerm Γ (err / σ)
   TRY_WITH_ : {e e' : E} {σ : VType} → CTerm Γ (e / σ) →
               CTerm Γ (e' / σ) → CTerm Γ (e ◇ e' / σ)
   IF_THEN_ELSE_ : {e e' : E} {σ : VType} → VTerm Γ bool →
               CTerm Γ (e / σ) → CTerm Γ (e' / σ) → CTerm Γ (e ⊔ e' / σ)
   _$_ : {σ : VType} {τ : CType} →
         VTerm Γ (σ ⇒ τ) → VTerm Γ σ → CTerm Γ τ
   PREC : {e e' : E} {σ : VType} → VTerm Γ nat →
          CTerm Γ (e / σ) → CTerm (σ :: nat :: Γ) (e' / σ) →
          e · e' ⊑ e → CTerm Γ (e / σ)
   LET_IN_ : {e e' : E} {σ σ' : VType} → CTerm Γ (e / σ) →
             CTerm (σ :: Γ) (e' / σ') → CTerm Γ (e · e' / σ')
   CCAST : {e e' : E} {σ σ' : VType} → CTerm Γ (e / σ) →
           e / σ ≤C e' / σ' → CTerm Γ (e' / σ')
```

Joonis 10: Eranditega keele rafineeritud termid.

- `TRY_WITH_` parandab põhiarvutustermi efekti erandikäsitleja arvutustermi efektiga. Kitsendusena peavad arvutustermid omama sama väärtustüüpi.
- `IF_THEN_ELSE_` eeldab tõeväärtustüüpi tingimust. Kogu arvutustermi efekt on määratud harude, mille väärtustüübid peavad ühtima, efektide ülemise rajaga.
- `_$_` rakendab esimese väärtustermiga antud funktsiooni teise väärtustermiga antud argumendile, seejuures peavad funktsiooni parameetri ja argumendi väärtustüübid ühtima. Saadud arvutuse efekt ja väärtustüüp on määratud funktsiooni keha arvutustüübiga.
- `PREC` eeldab sammude arvuna naturaalarvude tüüpi väärtustermi. Baasarvutuse väärtustüüp on lisatud koos naturaalarvu tüüpi sammuloenduriga sammu arvutustermi konteksti. Täiendava kitsendusena on nõutud, et baasi efekt oleks sammu efektiga korrutamise eelpüsipunkt.
- `LET_IN_` lisab esimese arvutustermi väärtustüübi teise arvutustermi konteksti. Kogu arvutuse efektiks on kahe arvutustermi efektide korrutis ning väärtustüüp on määratud teise arvutustermi tüübiga.
- `CCAST` suurendab etteantud arvutustermi tüüpi vastavalt alamtüüpimistõestusele.

2.3.3 Termide tüübituletus

Etteantud kontekstis saab väärtustermile tuletada vastava väärtustüübi (jn 11). Kuna term võib olla tüüpimatu, siis on `infer-vtype` tulemus mähitud `Maybe` andmetüüpi. Väärtustüübi tuletamisel lähtutakse väärtustermi kujust.

- `TT` ja `FF` annavad kindlasti tõeväärtustüübi.
- `ZZ` on kindlasti naturaalarvu tüüpi. `SS t` korral tuleb täiendavalt kontrollida, kas term `t` on samas kontekstis naturaalarvu tüüpi. Vastasel korral on term halvasti koostatud ja seda ei saa tüüpida.
- Paari `⟨ t , t' ⟩` tüüp on määratud, kui termide `t` ja `t'` tüübid on samas kontekstis määratud. Paari tüübiks on nende termide tüüpide korrutis. Ülejäänud juhtudel pole paari tüüp määratud.
- `FST t` ja `SND t` on määratud, kui term `t` on paar, st antud kontekstis on ta korrutise tüüpi. Projektsiooni tüübiks on vastavalt esimene või teine tegur.
- `VAR x` korral tuleb kontrollida, et naturaalarv `x` on väiksem kui konteksti Γ pikkus. Selleks on kasutatud lahendajat `_<?_`. Naturaalarvude võrratuse tõestusest `p` on

koostatud konteksti pikkusega piiratud naturaalarv `fromℕ≤ p`, mida kasutatakse muutujale vastava tüübi otsimiseks kontekstist `lkp` Γ.

- `LAM` σ `t` puhul tuleb kontrollida, et arvutustermiga `t` antud funktsiooni keha on hästi tüübitud kontekstis, mida on laiendatud parameetri väärtustüübi σ võrra. Arvutustermi tüübituletus `infer-ctype` on toodud allpool.

Joonisel 12 on toodud etteantud kontekstis arvutustermile tüübi tuletamine. Nagu väärtustermide tüübituletuse puhul, on ka arvutustermide tüübituletus `infer-ctype` tulemus mähitud `Maybe` andmetüüpi. Arvutustüübi tuletamisel lähtutakse arvutustüübi kujust.

- `VAL x` on tüübitud, kui väärtustermi `x` tüübituletus õnnestub. Arvutuse väärtustüübiks on tuletatud tüüp. Efektihinnang `ok` tähistab arvutuse õnnestumist.
- `FAIL` σ on alati väärtustüübi σ ebaõnnestumise tüüpi, mille efektihinnang on `err`.
- `TRY t WITH t'` on tüübitud, kui arvutustermid `t` ja `t'` on hästi tüübitud. Kogu arvutuse tüübiks on põhiarvutuse tüübi τ parandamine erandikäsitleja tüübiga τ`'`. Arvutustüüpide parandus _◇C_ on defineeritud efektide paranduse _◇_ ja väärtustüüpide ülemise raja _⊔V_ abil.
- `IF x THEN t ELSE t'` eeldab, et väärtusterm `x` on tõeväärtustüüpi. Kogu arvutuse tüüp on määratud harude tüüpide τ ja τ`'` ülemise rajaga τ `⊔C` τ`'`.
- `f $ t` korral kontrollitakse, et väärtustermi `f` tüübiks on funktsiooniruum ja väärtustermile `t` tuletatud tüüp on `f` parameetri (ehk uusima vaba muutuja) alamtüüp. Ülejäänud juhtudel ei ole funktsiooni rakendamine hästi tüübitud.
- `PREC x t t'` korral kontrollitakse viit tingimust.
  - Väärtusterm `x` peab olema antud kontekstis naturaalarvu tüüpi.
  - Baasi arvutusterm `t` peab olema antud kontekstis hästi tüübitud.
  - Sammu arvutusterm `t'` peab olema tüübitud kontekstis, kuhu on lisatud naturaalarvu tüüpi sammuloendur ja arvutustermi `t` väärtustüüpi σ akumulaator.
  - Osaarvutustele `t` ja `t'` tuletatud väärtustüübid peavad olema samad. Selleks kasutatakse lahendajat `_≡V?_`.
  - Osaarvutuste `t` ja `t'` efektide korrutis ei tohi olla suurem kui baasi `t` efekt. Seda kontrollitakse lahendajaga `_⊑?_`.

  Kui kõik tingimused kehtivad, siis kogu arvutuse tüüp on määratud baasi efekti ja väärtustüübiga.

```
infer-vtype : Ctx → vTerm → Maybe VType
infer-vtype Γ TT = just bool
infer-vtype Γ FF = just bool
infer-vtype Γ ZZ = just nat
infer-vtype Γ (SS t) with  infer-vtype Γ t
... | just nat = just nat
... | _        = nothing
infer-vtype Γ ⟨ t , t' ⟩ with infer-vtype Γ t | infer-vtype Γ t'
... | just σ | just σ' = just (σ • σ')
... | _      | _       = nothing
infer-vtype Γ (FST t) with infer-vtype Γ t
... | just (σ • _) = just σ
... | _            = nothing
infer-vtype Γ (SND t) with infer-vtype Γ t
... | just (_ • σ') = just σ'
... | _             = nothing
infer-vtype Γ (VAR x) with x <? Γ
... | yes p = just (lkp Γ (fromℕ≤ p))
... | no ¬p = nothing
infer-vtype Γ (LAM σ t) with infer-ctype (σ :: Γ) t
... | just τ = just (σ ⇒ τ)
... | _      = nothing
```

Joonis 11: Eranditega keele väärtustermide tüübituletus.

- `LET t IN t'` on tüübitud, kui arvutusterm `t` on tüübitud antud kontekstis ja arvutusterm `t'` on tüübitud kontekstis, mida on laiendatud `t` väärtustüübi võrra. Arvutuse efektiks on `t` ja `t'` efektide korrutis ning väärtustüübiks `t'` väärtustüüp. Kui üks termidest `t` ja `t'` ei ole hästi tüübitud, siis ei ole ka kogu term tüübitud.

Alapeatükis 2.1 toodud näidisavaldiste (jn 3) tüüpimised on esitatud joonisel 13. Tüüpimine `typing-add` väidab, et väärtusterm `ADD` on hästi tüübitud. Väärtustermi tüübiks on naturaalarvu tüüpi parameetriga funktsiooniruum, mis omakorda tagastab kindlasti funktsiooniruumi, mille parameeter on naturaalarv ja tagastab kindlasti naturaalarvu. Curry'mise tõttu võib seega väärtustermi `ADD` vaadelda kui funktsiooni, mis võtab kaks naturaalarvu ja tagastab naturaalarvu.

Tüüpimine `typing-add-3-and-4` (jn 13) väidab, et arvutusterm `ADD-3-and-4` on hästi tüübitud ning kindlasti tagastab naturaalarvu. Tüüpimine `typing-bad-one` väidab, et arvutustermile `BAD-ONE` ei õnnestunud tüüpi tuletada.

Tüüpimised `typing-cmplx` ja `typing-smpl` väidavad, et termid `CMPLX` ja `SMPL` on hästi tüübitud ning tagastavad kindlasti tõeväärtuse. Kumbki tüübitules teostati kontekstis Γ_0, mis koosneb ühest ainsast tõeväärtustüüpi muutujast.

Kõik mainitud tüüpimise tõestused tulenevad vahetult väärtus- ja arvutustermide tüübitu-

```
infer-ctype : Ctx → cTerm → Maybe CType
infer-ctype Γ (VAL x) with infer-vtype Γ x
... | just σ = just (ok / σ)
... | _      = nothing
infer-ctype Γ (FAIL σ) = just (err / σ)
infer-ctype Γ (TRY t WITH t') with infer-ctype Γ t | infer-ctype Γ t'
... | just τ | just τ' = τ ◇C τ'
... | _      | _       = nothing
infer-ctype Γ (IF x THEN t ELSE t')
    with infer-vtype Γ x | infer-ctype Γ t | infer-ctype Γ t'
... | just bool | just τ | just τ' = τ ⊔C τ'
... | _         | _      | _       = nothing
infer-ctype Γ (f $ t) with infer-vtype Γ f | infer-vtype Γ t
... | just (σ ⇒ τ) | just σ' with σ' ≤V? σ
...                            | yes _ = just τ
...                            | no  _ = nothing
infer-ctype Γ (f $ t) | _ | _ = nothing
infer-ctype Γ (PREC x t t')
    with infer-vtype Γ x
... | just nat with infer-ctype Γ t
...        | nothing = nothing
...        | just (e / σ) with infer-ctype (σ :: nat :: Γ) t'
...                       | nothing = nothing
...                       | just (e' / σ') with e · e' ⊑? e | σ ≡V? σ'
...                                        | yes _ | yes _ = just (e / σ)
...                                        | _     | _     = nothing
infer-ctype Γ (PREC x t t') | _ = nothing
infer-ctype Γ (LET t IN t') with infer-ctype Γ t
... | nothing = nothing
... | just (e / σ) with infer-ctype (σ :: Γ) t'
...                | nothing        = nothing
...                | just (e' / σ') = just (e · e' / σ')
```

Joonis 12: Eranditega keele arvutustermide tüübituletus.

letuste `infer-vtype` ja `infer-ctype` definitsioonidest.

2.3.4 Termide rafineerimine

Kui toorele termile õnnestub mingis kontekstis tuletada tüüp, siis saab sellest termist konstrueerida rafineeritud versiooni, mis "teab" oma konteksti ja tüüpi. Joonisel 14 on defineeritud funktsioon, mis toore termi jaoks tagastab vastava rafineeritud termi tüübi või tüübi $\top$, kui term on tüüpimatu.

Väärtustermide rafineerimine etteantud kontekstis (jn 15) matkib väärtustermide tüübituletust (alaptk 2.3.3).

- `TT` ja `FF` korral konstrueeritakse vastav rafineeritud väärtusterm.
- `ZZ` puhul konstrueeritakse nullile vastav rafineeritud väärtusterm `ZZ`. `SS t` korral kontrollitakse, et väärtusterm `t` on hästi tüübitud ja on naturaalarvu tüüpi. Rafineeritud järgarv `SS` koostatakse termi `t` rafineeringust `u`. Kui väärtustermi `t` tüübituletus ei õnnestu või tuletatud tüüp ei ole naturaalarvude tüüp, siis rafineeringu tulemuseks on tüübi $\top$ ainus element `tt`.
- $\langle$ `t , t'` $\rangle$ korral kontrollitakse, et mõlemad väärtustermid `t` ja `t'` on kontekstis hästi tüübitud ja rafineeritud paar koostatakse rafineeritud termidest `u` ja `u'`.
- `FST t` puhul peab väärtustermile `t` tuletatud tüüp olema korrutistüüp. Rafineeritud projektsiooni saab koostada `t` rafineeringust `u`. `SND t` juhtum on analoogne.
- `VAR x` korral koostatakse tõestusest `p`, mis näitab, et naturaalarv `x` on väiksem kui konteksti Γ pikkus, rafineeritud muutuja tõestusega, et `x`-ile määratud kohal kontekstis Γ on `VAR x` jaoks tuletatud tüüp.
- `LAM` σ `t` juhtumis lisatakse parameetri tüüp σ konteksti ja kontrollitakse arvutustermi `t` hästi-tüübitust. Rafineeritud funktsiooniabstraktsioon koostatakse uues kontekstis rafineeritud arvutusest `u`.

Arvutustermide rafineerimine on toodud joonistel 16 ja 17.

- `VAL t` korral kontrollitakse, et väärtusterm `t` on hästi tüübitud, ja rafineeritud arvutus koostatakse vastavast rafineeritud väärtustermist `u`.
- `FAIL` σ rafineerimisel näidatakse, et selle arvutustermi tüübituletus alati õnnestub.

```
typing-add : infer-vtype [] ADD ≡ just (nat ⇒ ok / (nat ⇒ ok / nat))
typing-add = refl

typing-add-3-and-4 : infer-ctype [] ADD-3-and-4 ≡ just (ok / nat)
typing-add-3-and-4 = refl

typing-bad-one : infer-ctype [] BAD-ONE ≡ nothing
typing-bad-one = refl

Γ₀ = [ bool ]

typing-cmplx : infer-ctype Γ₀ CMPLX ≡ just (ok / bool)
typing-cmplx = refl

typing-smpl : infer-ctype Γ₀ SMPL ≡ just (ok / bool)
typing-smpl = refl
```

Joonis 13: Eranditega keele näidisavaldiste tüüpimine.

```
refined-vterm : Ctx → vTerm → Set
refined-vterm Γ t with infer-vtype Γ t
... | nothing = ⊤
... | just τ = VTerm Γ τ

refined-cterm : Ctx → cTerm → Set
refined-cterm Γ t with infer-ctype Γ t
... | nothing = ⊤
... | just τ = CTerm Γ τ
```

Joonis 14: Väärtus- ja arvutustermide rafineerimiste tüübikonstruktorid.

```
refine-vterm : (Γ : Ctx) (t : vTerm) → refined-vterm Γ t
refine-vterm Γ TT = TT
refine-vterm Γ FF = FF
refine-vterm Γ ZZ = ZZ
refine-vterm Γ (SS t) with infer-vtype Γ t | refine-vterm Γ t
... | just nat | u = SS u
... | just bool | _ = tt
... | just (_ • _) | _ = tt
... | just (_ ⇒ _) | _ = tt
... | nothing | _ = tt
refine-vterm Γ ⟨ t , t' ⟩
    with infer-vtype Γ t | refine-vterm Γ t |
         infer-vtype Γ t' | refine-vterm Γ t'
... | just _  | u | just _  | u' = ⟨ u , u' ⟩
... | just _  | _ | nothing | _  = tt
... | nothing | _ | _       | _  = tt
refine-vterm Γ (FST t) with infer-vtype Γ t | refine-vterm Γ t
... | just nat | _ = tt
... | just bool | _ = tt
... | just (_ • _) | u = FST u
... | just (_ ⇒ _) | _ = tt
... | nothing | _ = tt
refine-vterm Γ (SND t) with infer-vtype Γ t | refine-vterm Γ t
... | just nat | _ = tt
... | just bool | _ = tt
... | just (_ • _) | u = SND u
... | just (_ ⇒ _) | _ = tt
... | nothing | _ = tt
refine-vterm Γ (VAR x) with x <? Γ
... | yes p = VAR (trace Γ (fromℕ≤ p))
... | no _  = tt
refine-vterm Γ (LAM σ t)
    with infer-ctype (σ :: Γ) t | refine-cterm (σ :: Γ) t
... | just _ | u = LAM σ u
... | nothing | u = tt
```

Joonis 15: Eranditega keele väärtustermide rafineerimine.

- `TRY t WITH t'` korral kontrollitakse, et `t` ja `t'` on hästi tüübitud ja tuletatud väärtustüüpidel leidub ülemine raja. Rafineeritud arvutuse konstrueerimiseks suurendatakse rafineeritud osaarvutuste `u` ja `u'` tüüpi ülemise rajani vastavalt alamtüüpimise tõestusele `p`.

- `IF x THEN t ELSE t'` korral peab väärtusterm `x` olema tõeväärtustüüpi ning arvutustermid `t` ja `t'` peavad olema hästi tüübitud. Kui harude `t` ja `t'` arvutuste väärtustüüpidel leidub ülemine raja, siis rafineeritud tingimuslause tingimus on rafineeritud väärtusterm `x'` ja tingimuslause harudes suurendatakse rafineeritud arvutuste `u` ja `u'` tüüpi vastavalt alamtüübi tõestusele `p`. Ülejäänud juhtudel tagastatakse tüübi $\top$ element `tt`.

- `f $ x` korral peab väärtusterm `f` olema funktsiooniruumi tüüpi ja seejuures peab argumendile `x` tuletatud tüüp olema mainitud funktsiooniruumi parameetri tüübi alamtüüp. Rafineeritud funktsiooni `f'` rakendamise koostamisel on rafineeritud argumendi `x'` tüüpi suurendatud vastavalt alamtüübi tõestusele `p`.

- `PREC x t t'` korral kontrollitakse, et väärtusterm `x` on naturaalarvu tüüpi ning baasile vastav arvutus `t` on hästi tüübitud. Seejärel, et sammule vastav arvutus `t'` on hästi tüübitud kontekstis, kuhu on lisatud naturaalarvu tüüpi sammuloendur ning baasi väärtustüübile vastav akumulaator. Viimaks kontrollitakse, et baasi ja sammu efektide korrutamine ei ületaks baasi efekti ning et baasile ja sammule vastavad väärtustüübid langevad kokku. Rafineeritud primitiivse rekursiooni term koostatakse vastavatest rafineeritud termidest `x'`, `u`, `u'` ja efektide korrutamise eelpüsipunkti tõestusest `p`.

- `LET t IN t'` puhul peab osaarvutus `t` olema hästi tüübitud antud kontekstis ja osaarvutus `t'` tüübitud kontekstis, kuhu on lisatud `t`-le tuletatud tüüp σ. Rafineeritud arvutuste sidumine koostatakse rafineeritud osaarvutustest `u` ja `u'`.

2.4 Semantika

Joonisel 18 on toodud vastastikku defineeritud väärtus- ja arvutustüüpide ning konteksti semantiline interpretatsioon metakeeles Agda.

- `nat` interpreteeritakse kui naturaalarvud $\mathbb{N}$ ja `bool` kui tõeväärtused `Bool`.

- σ • σ' korral tehakse rekursiivsed väljakutsed korrutatavatele ning tulemused korrutatakse Agdas _×_.

```
refine-cterm : (Γ : Ctx) (t : cTerm) → refined-cterm Γ t
refine-cterm Γ (VAL t) with infer-vtype Γ t | refine-vterm Γ t
... | nothing | u = tt
... | just _ | u = VAL u
refine-cterm Γ (FAIL σ) with infer-ctype Γ (FAIL σ)
... | _ = FAIL σ
refine-cterm Γ (TRY t WITH t')
    with infer-ctype Γ t | refine-cterm Γ t |
         infer-ctype Γ t' | refine-cterm Γ t'
... | nothing      | _ | _                | _ = tt
... | just _       | _ | nothing          | _ = tt
... | just (e / σ) | u | just (e' / σ') | u'
         with σ ⊔V σ' | inspect (_⊔V_ σ) σ'
...      | nothing | _ = tt
...      | just _ | [ p ] =
                           TRY  CCAST u (⊔V-subtype p)
                           WITH CCAST u' (⊔V-subtype-sym {σ} p)
refine-cterm Γ (IF x THEN t ELSE t')
    with infer-vtype Γ x | refine-vterm Γ x
... | nothing | _ = tt
... | just nat | _ = tt
... | just (_ • _) | _ = tt
... | just (_ ⇒ _) | _ = tt
... | just bool | x'
         with infer-ctype Γ t | refine-cterm Γ t
...      | nothing | u = tt
...      | just (e  / σ) | u
              with infer-ctype Γ t' | refine-cterm Γ t'
...           | nothing | u' = tt
...           | just (e' / σ') | u'
                   with σ ⊔V σ' | inspect (_⊔V_ σ) σ'
...                | nothing | _     = tt
...                | just ⊔σ | [ p ] =
                               IF x' THEN CCAST u (⊔V-subtype p)
                                     ELSE CCAST u' (⊔V-subtype-sym {σ} p)
-- to be continued
```

Joonis 16: Eranditega keele arvutustermide rafineerimine, I osa.

```
-- refine-cterm : (Γ : Ctx) (t : cTerm) → refined-cterm Γ t
refine-cterm Γ (f $ x)
    with infer-vtype Γ f | refine-vterm Γ f |
         infer-vtype Γ x | refine-vterm Γ x
... | nothing    | _ | _ | _ = tt
... | just nat   | _ | _ | _ = tt
... | just bool  | _ | _ | _ = tt
... | just (_ • _) | _ | _ | _ = tt
... | just (_ ⇒ _) | _ | nothing | _ = tt
... | just (σ ⇒ τ) | f' | just σ' | x' with σ' ≤V? σ
...                                    | no _  = tt
...                                    | yes p = f' $ VCAST x' p
refine-cterm Γ (PREC x t t')  with infer-vtype Γ x | refine-vterm Γ x
... | nothing | _  = tt
... | just bool | _ = tt
... | just (_ • _) | _ = tt
... | just (_ ⇒ _) | _ = tt
... | just nat | x'
        with infer-ctype Γ t | refine-cterm Γ t
...     | nothing | _ = tt
...     | just (e / σ) | u
            with infer-ctype (σ :: nat :: Γ) t' |
                 refine-cterm (σ :: nat :: Γ) t'
...         | nothing | _ = tt
...         | just (e' / σ') | u' with e · e' ⊑? e | σ ≡V? σ'
...                                  | no  _ | _    = tt
...                                  | yes _ | no _ = tt
refine-cterm Γ (PREC x t t')
    | just nat | x'
        | just (e / σ) | u
            | just (e' / .σ) | u' | yes p | yes refl = PREC x' u u' p
refine-cterm Γ (LET t IN t') with infer-ctype Γ t | refine-cterm Γ t
... | nothing | _  = tt
... | just (e / σ) | u with infer-ctype (σ :: Γ) t' |
                            refine-cterm (σ :: Γ) t'
...                    | nothing        | _  = tt
...                    | just (e' / σ') | u' = LET u IN u'
```

Joonis 17: Eranditega keele arvutustermide rafineerimine, II osa.

- $\sigma \Rightarrow \tau$ interpretatsioon vastab Agda funktsiooniruumile, mille parameetri ja tulemuse tüüp on interpreteeritud vastavalt väärtustüübist σ ja arvutustüübist τ.
- Arvutustüübi `e / σ` interpreteerimiseks rakendatakse gradeeritud monaadi tüübikonstruktorit `T` efektile `e` ja väärtustüübi σ interpretatsioonile.
- Tühi kontekst vastab üheelemendilisele tüübile `⊤`. Mittetühja konteksti pea interpreteeritakse ja korrutatakse rekursiivselt interpreteeritud sabaga.

Joonisel 19 on toodud rafineeritud väärtustermi interpretatsioon antud konteksti interpretatsioonis.

- `TT` ja `FF` seatakse vastavusse tõese ja vääraga.
- `ZZ` vastab nullile. `SS t` on `t` interpretatsiooni järgarv.
- `⟨ t , t' ⟩` tõlgendatakse kui `t` ja `t'` interpretatsioonide paari.
- `FST t` ja `SND t` projekteerivad esimese ja teise komponendi `t` interpretatsioonist, mis on paar.
- `VAR x` projekteerib konteksti interpretatsioonist ρ tõestusele `x` vastava (n-ö `x`-nda) väärtuse.
- `LAM σ t` interpreteeritakse kui Agda lambda-abstraktsiooni, mille seotud muutuja `x` lisatakse arvutustermi `t` interpreteerimise konteksti.
- `VCAST t p` puhul interpreteeritakse väärtusterm `t` ja konverteeritakse see vastavalt alamtüüpimise tõestusele `p`.

```
mutual
  ⟪_⟫V : VType → Set
  ⟪ nat ⟫V = ℕ
  ⟪ bool ⟫V = Bool
  ⟪ σ • σ' ⟫V = ⟪ σ ⟫V × ⟪ σ' ⟫V
  ⟪ σ ⇒ τ ⟫V = ⟪ σ ⟫V → ⟪ τ ⟫c

  ⟪_⟫c : CType → Set
  ⟪ e / σ ⟫c = T e ⟪ σ ⟫V

⟪_⟫X : Ctx → Set
⟪ [] ⟫X = ⊤
⟪ σ :: Γ ⟫X = ⟪ σ ⟫V × ⟪ Γ ⟫X
```

Joonis 18: Väärtus-, arvutustüüpide ja konteksti semantika.

```
⟦_⟧V : {Γ : Ctx} {σ : VType} → VTerm Γ σ → ⟪ Γ ⟫X → ⟪ σ ⟫V
⟦ TT ⟧V ρ = true
⟦ FF ⟧V ρ = false
⟦ ZZ ⟧V ρ = zero
⟦ SS t ⟧V ρ = suc (⟦ t ⟧V ρ)
⟦ ⟨ t , t' ⟩ ⟧V ρ = ⟦ t ⟧V ρ , ⟦ t' ⟧V ρ
⟦ FST t ⟧V ρ = proj₁ (⟦ t ⟧V ρ)
⟦ SND t ⟧V ρ = proj₂ (⟦ t ⟧V ρ)
⟦ VAR x ⟧V ρ = proj x ρ
⟦ LAM σ t ⟧V ρ = λ x → ⟦ t ⟧C (x , ρ)
⟦ VCAST t p ⟧V ρ = vcast p (⟦ t ⟧V ρ)
```

Joonis 19: Eranditega keele väärtustermide semantika.

Rafineeritud arvutustermi semantiline interpretatsioon etteantud konteksti interpretatsioonis on toodud joonisel 20.

- `VAL x` interpreteerib väärtustermi `x` antud kontekstis ja rakendab sellele gradeeritud monaadi ühikut η.
- Kuna arvutustüübi, mille efekt on `err`, interpretatsioon erandite gradeeritud monaadis on üheelemendiline tüüp $\top$, siis `FAIL σ` koostab selle ainsa elemendi `tt`.
- `TRY_WITH_ e e' t t'` kombineerib osaarvutuste `t` ja `t'` interpretatsioonid vastavalt arvutuste efektidele. Semantiline erandikäsitlus `or-else` käitub järgnevalt. Kui esimese osaarvutuse efektiks on ebaõnnestumine `err`, siis kogu arvutus on määratud erandikäsitlejaga. Kui esimene arvutus õnnestub efektiga `ok`, siis kogu arvutuseks ongi esimene arvutus. Kui esimese arvutuse õnnestumine pole teada, st efektiks on `errok`, siis analüüsitakse ka erandikäsitleja efekti. Kui erandikäsitleja efekt on `err`, siis on kogu arvutus määratud põhiarvutusega. Ülejäänud juhtudel analüüsitakse esimese arvutuse tulemuse kuju: kui esimene arvutus ikkagi õnnestus (konstruktor `just`), siis saab sealt ka kogu arvutuse tulemuse; vastasel korral on kogu arvutuse tulemuseks erandikäsitleja tulemus.
- `IF_THEN_ELSE_` korral interpreteeritakse tingimus ja harud tingimuslauses, kusjuures kummagi haru efekt neeldub efektide ülemises rajas.
- `PREC x t t' p` interpretatsioon vastab primitiivsele rekursioonile, mille sammude arv on on väärtustermi `x` interpretatsioon, baas on arvutustermi `t` interpretatsioon ja sammuks on arvutustermi `t'` interpretatsioon kontekstis, kuhu on lisatud sammuloendur ja vahetulemuse akumulaator. Semantiline primitiivne rekursioon `primrecT` on defineeritud induktsiooniga sammude arvul. Nulli korral on tulemuseks baasile vastav arvutus `z`. Sammu korral rakendatakse sammule vastavat

funktsiooni `s` sammuloendurile `n` ja saadud funktsioon seotakse rekursiivse väljakutsega gradeeritud monaadi `bind`-tehte abil. Tulemuse efekt neeldub efektide eelpüsipunkti tõestuse `p` tõttu baasarvutuse efektis.

- `f $ x` korral rakendatakse väärtustermi `f` interpretatsiooni väärtustermi `x` interpretatsioonile.
- `LET_IN_` seob arvutuste interpretatsioonid, kasutades gradeeritud monaadi `bind`-tehet.
- `CCAST t p` puhul interpreteeritakse arvutusterm `t` ja konverteeritakse see vastavalt alamtüüpimise tõestusele `p`.

2.5 Optimisatsioonid

Etteantud kontekstist saab jätta välja selle mingis kohas oleva tüübi, eeldusel, et sellele vastavat muutujat pole mingis termis tarvis. Seda nimetatakse konteksti lühendamiseks `dropX` (jn 21). Vastavalt saab lõdvendada rafineeritud väärtusterme `wkV` ja arvutusterme `wkC`, nihutades vajadusel sobivalt muutujaid. Teades konteksti interpretatsiooni ja väljajäetavat muutujat, saab koostada lühendatud konteksti interpretatsiooni `drop`. Lemmad `lemma-wkV` ja `lemma-wkC` näitavad, et termi interpretatsioon lühendatud kontekstis on sama, mis lõdvendatud termi interpretatsioon algses kontekstis.

Etteantud konteksti saab laiendada dubleerides `dupX` selle mingit elementi (jn 22). Termi kontraheerimine, funktsioonid `ctrV` ja `ctrC`, seisneb selle kontekstis olevate muutujate koondamises, eeldusel, et koondatavad on võrdsed (teisisõnu: dubleeritud). Vajadusel tuleb selleks nihutada muutujaid ühe elemendi võrra. Konteksti interpretatsioonis saab dubleerida mingile muutujale vastava väärtuse funktsiooniga `dup`. Lemmad `lemma-ctrV` ja `lemma-ctrC` näitavad, et termi interpretatsioon dubleeritud kontekstis on sama, mis kontraheeritud termi interpretatsioon algses kontekstis.

Lihtsuse huvides pole mainitud lõdvendamise ja kontraheerimise definitsioone ja tõestusi siinkohal toodud.

Mõned erandite monaadi jaoks spetsiifilised, efektide suhtes geneerilised optimisatsioonid on toodud joonisel 23. `handler-idemp` näitab, et arvutust `m` ei saa parandada, lisades sellele erandikäsitlejana sama arvutuse. Erandikäsitlejate assotsiatiivsus on näidatud teisendusega `handler-ass`. Selle tõestus matkib arvutuse parandusoperaatori assotsiatiivsuse `◇-ass` tõestust, milles efektide juhte läbi vaadatakse.

```
or-else : (e e' : E) {X : Set} → T e X → T e' X → T (e ◇ e') X
or-else err _ _ x' = x'
or-else ok _ x _ = x
or-else errok err x _ = x
or-else errok ok (just x) _ = x
or-else errok ok nothing x' = x'
or-else errok errok (just x) x' = just x
or-else errok errok nothing x' = x'

primrecT : {e e' : E} {X : Set} →
           ℕ → T e X → (ℕ → X → T e' X) → e · e' ⊑ e → T e X
primrecT zero z s p = z
primrecT {e} {e'} (suc n) z s p =
    sub p (bind {e} {e'} (s n) (primrecT n z s p))

⟦_⟧C : {Γ : Ctx} {τ : CType} → CTerm Γ τ → ⟪ Γ ⟫X → ⟪ τ ⟫c
⟦ VAL x ⟧C ρ = η (⟦ x ⟧V ρ)
⟦ FAIL σ ⟧C ρ = tt
⟦ TRY_WITH_ {e} {e'} t t' ⟧C ρ = or-else e e' (⟦ t ⟧C ρ) ( (⟦ t' ⟧C ρ))
⟦ IF_THEN_ELSE_ {e} {e'} x t t' ⟧C ρ =
    if ⟦ x ⟧V ρ
    then (sub (lub e e') (⟦ t ⟧C ρ))
    else (sub (lub-sym e' e) (⟦ t' ⟧C ρ))
⟦ PREC x t t' p ⟧C ρ = primrecT (⟦ x ⟧V ρ) (⟦ t ⟧C ρ)
                                ((λ i acc → ⟦ t' ⟧C (acc , i , ρ))) p
⟦ f $ x ⟧C ρ = ⟦ f ⟧V ρ (⟦ x ⟧V ρ)
⟦ LET_IN_ {e} {e'} t t' ⟧C ρ =
    bind {e} {e'} (λ x → ⟦ t' ⟧C (x , ρ)) (⟦ t ⟧C ρ)
⟦ CCAST t o ⟧C ρ = ccast o (⟦ t ⟧C ρ)
```

Joonis 20: Eranditega keele arvutustermide semantika.

```
dropX : (Γ : Ctx) {σ : VType} (x : σ ∈ Γ) → Ctx
-- proof omitted
mutual
  wkV : {Γ : Ctx} {σ σ' : VType} (x : σ ∈ Γ) →
        VTerm (dropX Γ x) σ' → VTerm Γ σ'
  -- proof omitted
  wkC : {Γ : Ctx} {σ : VType} {τ : CType} (x : σ ∈ Γ) →
        CTerm (dropX Γ x) τ → CTerm Γ τ
  -- proof omitted
drop : {Γ : Ctx} → ⟪ Γ ⟫X → {σ : VType} → (x : σ ∈ Γ) → ⟪ dropX Γ x ⟫X
-- proof omitted
mutual
  lemma-wkV : {Γ : Ctx} (ρ : ⟪ Γ ⟫X) →
              {σ : VType} (x : σ ∈ Γ) →
              {σ' : VType} (t : VTerm (dropX Γ x) σ') →
              ⟦ wkV x t ⟧V ρ ≡ ⟦ t ⟧V (drop ρ x)
  -- proof omitted
  lemma-wkC : {Γ : Ctx} (ρ : ⟪ Γ ⟫X) →
              {σ : VType} (x : σ ∈ Γ) →
              {τ : CType} (t : CTerm (dropX Γ x) τ) →
              ⟦ wkC x t ⟧C ρ ≡ ⟦ t ⟧C (drop ρ x)
  -- proof omitted
```

Joonis 21: Konteksti lühendamine ja termide lõdvendamine.

```
dupX : {Γ : Ctx} {σ : VType} → σ ∈ Γ → Ctx
-- proof omitted
mutual
  ctrV : {Γ : Ctx} {σ σ' : VType} (p : σ ∈ Γ) →
         VTerm (dupX p) σ' → VTerm Γ σ'
  -- proof omitted
  ctrC : {Γ : Ctx} {σ : VType} {τ : CType} (p : σ ∈ Γ) →
         CTerm (dupX p) τ → CTerm Γ τ
  -- proof omitted
dup : {Γ : Ctx} → ⟪ Γ ⟫X → {σ : VType} → (p : σ ∈ Γ) → ⟪ dupX p ⟫X
-- proof omitted
mutual
  lemma-ctrV : {Γ : Ctx} (ρ : ⟪ Γ ⟫X) →
               {σ : VType} (p : σ ∈ Γ) →
               {σ' : VType} (t : VTerm (dupX p) σ') →
               ⟦ t ⟧V (ctr ρ p) ≡ ⟦ ctrV p t ⟧V ρ
  -- proof omitted
  lemma-ctrC : {Γ : Ctx} (ρ : ⟪ Γ ⟫X) →
               {σ : VType} (p : σ ∈ Γ) →
               {τ : CType} (t : CTerm (dupX p) τ) →
               ⟦ t ⟧C (ctr ρ p) ≡ ⟦ ctrC p t ⟧C ρ
  -- proof omitted
```

Joonis 22: Konteksti dubleerimine ja termide kontraheerimine.

```
◇-idemp : (e : Exc) → e ◇ e ≡ e
◇-idemp err = refl
◇-idemp ok = refl
◇-idemp errok = refl

handler-idemp : {e : Exc} {Γ : Ctx} {ρ : ⟪ Γ ⟫X} {σ : VType}
                (m : CTerm Γ (e / σ)) →
                sub-eq (◇-idemp e) (⟦ TRY m WITH m ⟧C ρ) ≡ ⟦ m ⟧C ρ
handler-idemp {err} m = refl
handler-idemp {ok} m = refl
handler-idemp {errok} {ρ = ρ} m with ⟦ m ⟧C ρ
... | just _ = refl
... | nothing = refl


◇-ass : (e e' e'' : Exc) → e ◇ (e' ◇ e'') ≡ (e ◇ e') ◇ e''
◇-ass err e' e'' = refl
◇-ass ok e' e'' = refl
◇-ass errok err e'' = refl
◇-ass errok ok e'' = refl
◇-ass errok errok err = refl
◇-ass errok errok ok = refl
◇-ass errok errok errok = refl

handler-ass : {e₁ e₂ e₃ : Exc} {Γ : Ctx} {ρ : ⟪ Γ ⟫X} {σ : VType}
              (m₁ : CTerm Γ (e₁ / σ)) (m₂ : CTerm Γ (e₂ / σ))
              (m₃ : CTerm Γ (e₃ / σ)) →
              sub-eq (◇-ass e₁ e₂ e₃)
                     (⟦ TRY m₁ WITH (TRY m₂ WITH m₃) ⟧C ρ)
              ≡ ⟦ TRY (TRY m₁ WITH m₂) WITH m₃ ⟧C ρ
handler-ass {err} m₁ m₂ m₃ = refl
handler-ass {ok} m₁ m₂ m₃ = refl
handler-ass {errok} {err} m₁ m₂ m₃ = refl
handler-ass {errok} {ok} m₁ m₂ m₃ = refl
handler-ass {errok} {errok} {err} m₁ m₂ m₃ = refl
handler-ass {errok} {errok} {ok} {ρ = ρ} m₁ m₂ m₃ with ⟦ m₁ ⟧C ρ
... | just _ = refl
... | nothing = refl
handler-ass {errok} {errok} {errok} {ρ = ρ} m₁ m₂ m₃ with ⟦ m₁ ⟧C ρ
... | just x = refl
... | nothing = refl
```

Joonis 23: Erandite monaadi spetsiifilised, efekti suhtes geneerilised teisendused.

Lihtsustus `dup-comp` (jn 24) võimaldab arvutuse `m` topelt arvutamise asendada ühekordse arvutamisega. Tõestuses analüüsitakse kõigepealt arvutuse `m` efekti kuju.

- Kui see arvutus ebaõnnestub, siis kogu arvutuse interpretatsioon on paratamatult `tt` ja seega tõestus on triviaalne.
- Kui arvutuse `m` efektiks on `ok`, siis analüüsitakse arvutuse `m` interpretatsiooni. Õnnestunud arvutuse `just x` korral näidatakse ülesande tüüpi nõrgendamise `lemma-wkC` ja `m`-i uuritud interpretatsiooni `eq`-ga ümberkirjutades, et tulemus järeldub lemmast `lemma-ctrC`. Ebaõnnestunud arvutuse korral pole arvutusse `n` ühtegi väärtust siduda ja kogu arvutuse interpretatsiooniks on `nothing`.
- Kui efektiks on `errok`, siis vaadatakse läbi arvutuse `n` efekti kuju:
  - Kui arvutus `n` ebaõnnestub, siis kogu arvutuse interpretatsioon on paratamatult `tt` ja seega tõestus on triviaalne.
  - Kui arvutuse `n` efektiks on `ok`, siis arutelu on sarnane nagu juhtumis, kus arvutuse `m` efekt oli `ok`.
  - Kui efektiks on `errok`, siis on tõestus analoogne efekti `ok` juhtumiga, v.a. asjaolu, et arvutuse `n` interpretatsioon on `Maybe` tüüpi.

Mõned erandite monaadi spetsiifilised, efekti-spetsiifilised optimisatsioonid on toodud joonisel 25. Iga arvutuse `m`, mille efekt on `err`, saab samaväärselt asendada arvutusega `FAIL` σ. Samaväärsus `failure m` põhineb asjaolul, et ebaõnnestunud arvutuse semantiline interpretatsioon erandite gradeeritud monaadis on tüüp $\top$, milles ongi ainult üks element ja seetõttu on tõestus triviaalne.

Kuna tingimuslause `IF_THEN_ELSE_` arvutuse efektiks on harude efektide ülemine raja, siis juhul kui mõlemad harud ebaõnnestuvad, on ka kogu arvutuse efektiks `err` ja seega saab seda lihtsustada analoogselt `failure` arutelule. Samaväärsus on näidatud tõestusega `both-fail` (jn 25).

Lihtsustus `dead-comp` (jn 25) näitab, et kui kindlasti õnnestuvat osaarvutust `m` ei pruugita osaarvutuses `n`, siis nende sidumisel pole mõtet ja võib kasutada lihtsalt osaarvutust `n`. Tõestus on eespool antud arvutustermi lõdvenduse `lemma-wkC` rakendus.

On ka monaadist sõltumatuid optimisatsioone, mille korrektsus järeldub juba üldistest monaadi seadustest ning mis seetõttu kehtivad mitte ainult erandite vaid ka iga teise monaadi jaoks.

Alapeatükis 2.1 toodud näitetermidele `CMPLX` ja `SMPL` (jn 3) tuletati tüübid alapeatükis 2.3.3. Kuna tüübituletus õnnestus kontekstis $\Gamma_\emptyset$, siis saab neid arvutusterme selles sa-

```
·-idemp : (e : Exc) → e · e ≡ e
·-idemp err = refl
·-idemp ok = refl
·-idemp errok = refl

lemma : (e e' : Exc) → e · (e · e') ≡ e · e'
lemma e e' = begin
               e · (e · e')
             ≡⟨ sym (ass {e}) ⟩
               (e · e) · e'
             ≡⟨ cong (λ e → e · e') (·-idemp e) ⟩
               e · e'
             ∎

dup-comp : {e e' : Exc} {Γ : Ctx} {σ σ' : VType}
           (m : CTerm Γ (e / σ)) (n : CTerm (dupX here) (e' / σ')) →
           (ρ : ⟪ Γ ⟫X) →
           sub-eq (lemma e e')
                  (⟦ LET m IN LET wkC here m IN n ⟧C ρ)
           ≡ ⟦ LET m IN ctrC here n ⟧C ρ
dup-comp {err} m n ρ = refl
dup-comp {ok} m n ρ with ⟦ m ⟧C ρ | inspect ⟦ m ⟧C ρ
... | x | [ eq ] rewrite lemma-wkC (x , ρ) here m | eq
                        = lemma-ctrC (x , ρ) here n
dup-comp {errok} {err} m n ρ = refl
dup-comp {errok} {ok} m n ρ with ⟦ m ⟧C ρ | inspect ⟦ m ⟧C ρ
... | just x  | [ eq ] rewrite lemma-wkC (x , ρ) here m | eq
                   = cong just (lemma-ctrC (x , ρ) here n)
... | nothing | _ = refl
dup-comp {errok} {errok} m n ρ with ⟦ m ⟧C ρ | inspect (⟦ m ⟧C) ρ
... | just x  | [ eq ] rewrite lemma-wkC (x , ρ) here m | eq
                   = lemma-ctrC (x , ρ) here n
... | nothing | _ = refl
```

Joonis 24: Erandite monaadi spetsiifiline liiase arvutuse eemaldamise lihtsustus.

```
failure : {Γ : Ctx} {σ : VType} (m : CTerm Γ (err / σ)) →
          ⟦ m ⟧C ≡ ⟦ FAIL σ ⟧C
failure m = refl

both-fail : {Γ : Ctx} {σ : VType}
            (m : VTerm Γ bool) (n n' : CTerm Γ (err / σ)) →
            (ρ : ⟪ Γ ⟫X) →
            ⟦ IF m THEN n ELSE n' ⟧C ρ ≡ ⟦ FAIL σ ⟧C ρ
both-fail m n n' ρ = refl

dead-comp : {Γ : Ctx} {σ σ' : VType} {e : Exc}
            (m : CTerm Γ (ok / σ)) (n : CTerm Γ (e / σ')) →
            (ρ : ⟪ Γ ⟫X) →
            ⟦ LET m IN (wkC here n) ⟧C ρ ≡ ⟦ n ⟧C ρ
dead-comp m n ρ = lemma-wkC (⟦ m ⟧C ρ , ρ) here n

cmplx-smpl : {ρ : ⟪ Γ∅ ⟫X} →
            ⟦ refine-cterm Γ∅ CMPLX ⟧C ρ ≡ ⟦ refine-cterm Γ∅ SMPL ⟧C ρ
cmplx-smpl = refl
```

Joonis 25: Erandite monaadi efekti-spetsiifilised optimisatsioonid.

mas kontekstis rafineerida. Lihtsustus `cmplx-smpl` (jn 25) väidab, et saadud rafineeritud termide interpretatsioonid sellises kontekstis on ekvivalentsed. Tõestus tuleneb triviaalselt arvutustermide interpreteerimise definitsioonist. Selgitusena tasub märkida, et tegu on degenereerunud näitena lihtsustusest `dead-comp`, kuna tingimuslause ja erandikäsitlejaga arvutuse tulemust ei pruugita. Siiski, eelnevalt peab efektisüsteem veenduma, et see arvutus kindlasti õnnestub, kuna vastasel juhul poleks lihtsustus korrektne.

3 Mittedeterminism

Selles peatükis vaadeldakse keele laiendust mittedeterministliku valikuga. Keele efekt seisneb selles, et arvutuse tulemuseks võib olla null või rohkem väärtust. Staatilise hinnanguna tõkestatakse väärtuste arvu ülevalt.

Baaskeeleks on tüübitud lambdaarvutus koos tõeväärtuste, naturaalarvude ja paaridega. Kuna baaskeel on sama, mis peatükis 2, siis järgnevates alapeatükkides on toodud välja ainult olulisemad muudatused keele laienduse, tüübituletuse, semantika ja efektianalüüsi osas.

3.1 Mittedeterministlik keel

Järgnev BNF esitab mittedeterministliku keele grammatika.

```
t ::= nat | bool | t • t | t ⇒ e / t                          (e ∈ E)

v ::= TT | FF | ZZ | SS v | ⟨ v , v ⟩ | FST v | SND v
   |  VAR n | LAM t c                                         (n ∈ ℕ)

c ::= VAL v | FAIL t | CHOOSE c c
   |  IF v THEN c ELSE c | v $ v | PREC v c c | LET c IN c
```

Võrreldes eranditega keelega (ptk 2) on erandikäsitlusega arvutus `TRY_WITH_` asendunud arvutusega `CHOOSE`, mis valib mittedeterministlikult, kumba osaarvutust täita.

Sellise keele rafineeritud ja rafineerimata arvutustermid on toodud joonisel 26. Väärtustermid on mõlemal keelel defineeritud samamoodi. Muutunud on arvutuste efektide tüüp `E`, mis defineeritakse alapeatükis 3.2. Deterministliku rafineeritud arvutustermi `VAL v`, millel on täpselt üks tulemus, efekti hinnanguks on `1` ja tulemuseta arvutustermi `FAIL` hinnanguks on `0`. Tasub märkida, et `1` on ülehinnang, kuna lubab nii null kui ka täpselt üks tulemust.

```
data cTerm : Set where
  VAL : vTerm → cTerm
  FAIL : VType → cTerm
  CHOOSE : cTerm → cTerm → cTerm
  IF_THEN_ELSE_ : vTerm → cTerm → cTerm → cTerm
  _$_ : vTerm → vTerm → cTerm
  PREC : vTerm → cTerm → cTerm → cTerm
  LET_IN_ : cTerm → cTerm → cTerm

data CTerm (Γ : Ctx) : CType → Set where
  VAL : {σ : VType} → VTerm Γ σ → CTerm Γ (1 / σ)
  FAIL : (σ : VType) → CTerm Γ (0 / σ)
  CHOOSE : {e e' : E} {σ : VType} → CTerm Γ (e / σ) →
           CTerm Γ (e' / σ) → CTerm Γ ((e ◇ e') / σ)
  IF_THEN_ELSE_ : {e e' : E} {σ : VType} → VTerm Γ bool →
            CTerm Γ (e / σ) → CTerm Γ (e' / σ) → CTerm Γ ((e ⊔ e') / σ)
  _$_ : {σ : VType} {τ : CType} →
        VTerm Γ (σ ⇒ τ) → VTerm Γ σ → CTerm Γ τ
  PREC : {e e' : E} {σ : VType} → VTerm Γ nat →
         CTerm Γ (e / σ) → CTerm (σ :: nat :: Γ) (e' / σ) →
         e · e' ⊑ e → CTerm Γ (e / σ)
  LET_IN_ : {e e' : E} {σ σ' : VType} → CTerm Γ (e / σ) →
            CTerm (σ :: Γ) (e' / σ') → CTerm Γ (e · e' / σ')
  CCAST : {e e' : E} {σ σ' : VType} → CTerm Γ (e / σ) →
          e / σ ≤C e' / σ' → CTerm Γ (e' / σ')
```

Joonis 26: Mittedeterministliku keele arvutustermid.

3.2 Mittedeterminismi gradeering

Naturaalarvud ℕ, nende korrutamine `_*_` ja ühik 1 moodustavad monoidi. Naturaalarvude järjestusseos `_≤_` on refleksiivne `refl≤`, transitiivne `trans≤`. Korrutamine on selle seose suhtes monotoonne `mon*`. Korrutamise vasakühiku `lu*`, paremühiku `ru*` ja assotsiatiivsuse `ass*` ning monotoonsuse tõestused on toodud töö lähtekoodis. Sellega rahuldatakse alaptk 2.2.2 toodud tingimusi ja saab moodustada eeljärjestatud monoidi ℕ* (jn 27).

Ülalt tõkestatud pikkusega vektorite tüüp `BVec X` (jn 28) mingi hulga `X` jaoks on indekseeritud naturaalarvuga `n`, mis näitab vektoris olevate elementide suurimat võimalikku arvu. Ainsaks konstruktoris on `bv`, mis moodustab vektorist täpse pikkusega `m` ja `n`-ö "lõtku" tõestusest, et `m ≤ n`, uue ülalt `n`-iga tõkestatud vektori. Ülalt tõkestatud vektori päisesse elemendi lisamine `_::bv_` lisab selle elemendi täpse pikkusega vektori päisesse. Uue lõtku tõestus saadakse vanast kasutades asjaolu, et võrratus jääb kehtima, kui mõlemale poole liita 1. Vektorite liitmisel `_++bv_` liidetakse täpse pikkusega vektorid omavahel ja elementide lõtku tõestus koostatakse liitmise monotoonsusega kummagi vektori lõtkude tõestusest.

Eeljärjestatud monoid ℕ* ja parametriseeritud tüübikonstruktor `TBV`, mis annab vastava ülalt tõkestatud vektori tüübi, rahuldavad gradeeritud monaadi omadusi (alaptk 2.2.3). Tagastamine η`BV` koostab üheelemendilise ülalt tõkestatud ja ilma lõtkuta vektori. Sidumine `bindBV` rakendab antud funktsiooni igale vektori elemendile ja liidab saadud ülalt tõkestatud vektorid. Vastav gradeeritud monaadi definitsioon `NDBV` on toodud joonisel 29.

```
ℕ* : OrderedMonoid
ℕ* = record { E = ℕ
            ; _·_ = _*_
            ; i = 1
            ; lu = lu*
            ; ru = ru*
            ; ass = λ {m n o} → ass* {m} {n} {o}
            ; _⊑_ = _≤_
            ; ⊑-refl = refl≤
            ; ⊑-trans = trans≤
            ; mon = mon*
            }
```

Joonis 27: Mittedeterminismi eeljärjestatud monoid.

```
data BVec (X : Set) : (n : ℕ) → Set where
  bv : {m n : ℕ} → Vec X m → m ≤ n → BVec X n

_::bv_ : {X : Set} {n : ℕ} → X → BVec X n → BVec X (suc n)
x ::bv (bv xs p) = bv (x :: xs) (s≤s p)

_++bv_ : {X : Set} {m n : ℕ} → BVec X m → BVec X n → BVec X (m + n)
bv xs p ++bv bv xs' q = bv (xs ++ xs') (mon+ p q)
```

Joonis 28: Ülalt tõkestatud pikkusega vektor.

```
TBV = λ e X → BVec X e

ηBV : {X : Set} → X → BVec X i
ηBV x = bv (x :: []) (s≤s z≤n)

bindBV :  {m n : ℕ} {X Y : Set} →
        (X → BVec Y n) → BVec X m → BVec Y (m · n)
bindBV f (bv [] z≤n) = bv [] z≤n
bindBV f (bv (x :: xs) (s≤s p)) = (f x) ++bv bindBV f (bv xs p)

NDBV : GradedMonad
NDBV = record { OM = ℕ*
              ; T = TBV
              ; η = ηBV
              ; bind = λ {e} {e'} → bindBV {e} {e'}
              ; sub = subBV
              ; sub-mon = subBV-mon
              ; sub-refl = subBV-refl
              ; sub-trans = subBV-trans
              ; mlaw1 = mlaw1BV
              ; mlaw2 = mlaw2BV
              ; mlaw3 = mlaw3BV
              }
```

Joonis 29: Mittedeterminismi gradeeritud monaad.

3.3 Termide tüübituletus ja rafineerimine

Efektide järjestus võimaldab defineerida alamtüübid. Kuna see definitsioon on sama, mis eranditega keele puhul (alaptk 2.3.1), siis pole seda siinkohal toodud mittedeterministliku keele jaoks.

Osa arvutustermide tüübituletusest on esitatud joonisel 30.

- `VAL x` on hästi tüübitud, kui väärtusterm `x` on antud kontekstis tüübitud. Arvutuse efekt `1` tähistab ühte tulemust, mille tüüp σ vastab väärtustermile tuletatud tüübile. See on ülehinnang, kuna hinnang `1` lubab ka 0 tulemust.
- `FAIL` σ korral on efektiks `0`, kuna ühtki σ tüüpi tulemust ei teki.
- `CHOOSE t t'` on hästi tüübitud, kui mõlemad arvutustermid `t` ja `t'` on hästi tüübitud. Kogu arvutuse tüüp on määratud vastavalt tuletatud tüüpide τ ja τ`'` kombinatsiooniga τ `◇C` τ`'`: efektid liidetakse `_+_`-ga, sest kogu arvutusel on nii palju tulemusi, kui arvutustel `t` ja `t'` kokku. Väärtustüübiks on väärtustüüpide ülemine raja. Kui ülemine raja puudub, siis pole arvutus hästi tüübitud.

Rafineeritud arvutustermid on toodud joonisel 26. "Toorete" arvutustermide rafineerimine on esitatud joonisel 30.

- `VAL t` korral kontrollitakse, et väärtusterm `t` on hästi tüübitud, ja rafineeritud arvutusterm koostatakse vastavast rafineeritud väärtustermist `u`.
- `FAIL` σ korral näidatakse, et selle arvutustermi tüübituletus õnnestub, ning koostatakse samasugune rafineeritud arvutusterm.
- `CHOOSE t t'` puhul peavad mõlemad osaarvutused `t` ja `t'` olema hästi tüübitud. Kui neile tuletatud arvutustüüpide väärtustüüpidel on ülemine raja, siis rafineeritud arvutus koostatakse vastavate rafineeringutest `u` ja `u'`, suurendades neid vastavalt ülemise raja tõestusele `p`.

3.4 Semantika

Väärtustermide semantika on antud samamoodi nagu eranditega keeles (alaptk 2.4). Joonisel 31 on toodud osa arvutustermide semantikast.

```
infer-ctype : (Γ : Ctx) → cTerm → Maybe CType
infer-ctype Γ (VAL x) with infer-vtype Γ x
... | just σ = just (1 / σ)
... | _      = nothing
infer-ctype Γ (FAIL σ) = just (0 / σ)
infer-ctype Γ (CHOOSE t t') with infer-ctype Γ t | infer-ctype Γ t'
... | just τ | just τ' = τ ◇C τ'
... | _      | _       = nothing
-- rest of definition omitted

refine-cterm : (Γ : Ctx) (t : cTerm) → refined-cterm Γ t
refine-cterm Γ (VAL t) with infer-vtype Γ t | refine-vterm Γ t
... | just _ | u = VAL u
... | nothing | u = tt
refine-cterm Γ (FAIL σ) with infer-ctype Γ (FAIL σ)
... | _ = FAIL σ
refine-cterm Γ (CHOOSE t t')
    with infer-ctype Γ t | refine-cterm Γ t |
         infer-ctype Γ t' | refine-cterm Γ t'
... | nothing | _ | _ | _ = tt
... | just _ | _ | nothing | _ = tt
... | just (e / σ) | u | just (e' / σ') | u'
        with σ ⊔V σ' | inspect (_⊔V_ σ) σ'
...     | nothing | _ = tt
...     | just _ | [ p ] =
  CHOOSE (CCAST u (⊔V-subtype p))
         (CCAST u' (⊔V-subtype-sym {σ} p))
-- rest of definition omitted
```

Joonis 30: Mittedeterministliku keele tüübituletus ja rafineerimine.

- `VAL x` korral rakendatakse väärtustermi `x` interpretatsioonile ühikut η ehk moodustatakse temast lõtkuta vektor pikkusega 1.
- `FAIL` σ korral koostatakse tühi ülalt tõkestatud vektor funktsiooniga `sfail`. Selle vektori elementide tüüp on määratud väärtustüübi σ interpretatsiooniga.
- `CHOOSE t t'` interpretatsioon vastab mittedeterministlikule valikule arvutuste `t` ja `t'` vahel. See on realiseeritud vastavate arvutustermide interpreteerimisel saadud vektorite liitmisega.
- Ülejäänud arvutustermi konstruktorite semantika on nii nagu eranditega keeles.

3.5 Optimisatsioonid

Struktuursed teisendused – lõdvendamine ja kontraheerimine – toimivad mittedeterministliku keele puhul analoogselt eranditega keelega. Vastavad tüübisignatuurid on samad, mis alapeatükis 2.5 joonistel 21 ja 22 esitatud.

Näited mittedeterminismi monaadi jaoks spetsiifilistest, kuid efekti-geneerilistest optimisatsioonidest on toodud joonisel 32. Lihtsustus `choose-lu` näitab, et valides mittedeterministlikult arvutuste `FAIL` σ ja `m` vahel on tulemus sama nagu ainult `m` arvutamisel. Kuna konstruktori `CHOOSE` interpretatsioonile vastab osaarvutuste interpreteerimisel saadud tõkestatud vektorite liitmine ja konstruktori `FAIL` interpretatsioon on lihtsalt tühi vektor, siis ekvivalentsi tõestus taandub ülalt tõkestatud vektorite liitmise definitsioonile.

Mittedeterministlik valik on assotsiatiivne. Selline teisendus `choose-ass` on näidatud joonisel 32. Tõestus tugineb ülalt tõkestatud vektorite liitmise assotsiatiivsusel, mis on tõestatud töö lähtekoodis.

```
sfail : {X : Set} → T 0 X
sfail = bv []V z≤n

sor : (e e' : ) {X : Set} → T e X → T e' X → T (e + e') X
sor e e' = _++bv_

⟦_⟧C : {Γ : Ctx} {τ : CType} → CTerm Γ τ → ⟪ Γ ⟫X → ⟪ τ ⟫C
⟦ VAL x ⟧C ρ = η (⟦ x ⟧V ρ)
⟦ FAIL σ ⟧C ρ = sfail {⟪ σ ⟫V}
⟦ CHOOSE {e} {e'} t t' ⟧C ρ = sor e e' (⟦ t ⟧C ρ) (⟦ t' ⟧C ρ)
-- rest of definition omitted
```

Joonis 31: Mittedeterministliku keele semantika.

Lihtsustus `fails-earlier` (jn 32) näitab, et kui siduda ebaõnnestuv arvutus mingi arvutusega `m`, siis tulemus on sama kui kogu arvutus ebaõnnestuks. Sidumise konstruktori `LET_IN_` interpretatsioon seob kõik väärtused esimese osaarvutuse interpretatsioonist, milleks arvutuse `FAIL` σ korral on tühi vektor, teise osaarvutusega. Kuna esimesest osaarvutusest ei tekkinud ühtegi väärtust, siis sidumisel ei saa ka ühtegi väärtust tekkida. Seega on samaväärsus triviaalne.

Joonisel 33 on toodud mõned mittedeterminismi efekti-spetsiifilised teisendused. Lihtsustus `failure` näitab, et iga arvutuse `m`, mille arvutuse tulemusena ei teki mitte ühtegi väärtust (teisisõnu: arvutusel on ülimalt `0` väärtust), võib samaväärsena asendada arvutuse ebaõnnestumise konstruktsiooniga `FAIL`. Kuna arvutustermi `m` interpretatsioon antud konteksti interpretatsioonis ρ on tühi ülalt `0`-ga tõkestatud vektor, siis tõestus on triviaalne.

Samaväärsus `dup-comp` (jn 33) näitab, et iga arvutust `m`, mille efekt on ülimalt `1`, pole vaja topelt arvutada. Põhjendus on järgnev: kui `m` tulemuseks on täpselt üks väärtus, siis `LET_IN_` sidumisel `m`-iga ei teki väärtuseid juurde ja võib kohe selle väärtuse siduda `n`-iga; kui `m` arvutuse tulemusel ühtegi väärtust ei teki, siis pole ka järgnevatesse arvutustesse midagi siduda. Tõestus on antud töö lähtekoodis.

Antud töös on mittedeterministliku keele semantika antud ülalt tõkestatud vektoriga, kuid seda võib teha ka multihulkadel, kus pole tulemuste järjekord oluline. Lisada võib ka nt alumised tõkked, st hinnata listi või multihulka intervalliga. Minnes pisut ebatäpsemaks, võib multihulgad asendada ka hulkadega (nt `0` – ebaõnnestumine, `1` – deterministlik, `01` – pooldeterministlik, `1+` – mitmikdeterministlik, `N` – mittedeterministlik), siis saab tõestada surnud arvutuse eemaldamise (*dead computation*) ja arvutuse väljatõstmise (*pure lambda hoist*) lihtsustused [6].

```
choose-lu : {Γ : Ctx} {σ : VType} {e : ℕ} (m : CTerm Γ (e / σ)) →
            (ρ : ⟪ Γ ⟫X) →
            ⟦ CHOOSE (FAIL σ) m ⟧C ρ ≡ ⟦ m ⟧C ρ
choose-lu m ρ with ⟦ m ⟧C ρ
... | bv xs p = refl

choose-ass : {e₁ e₂ e₃ : ℕ} {Γ : Ctx} {σ : VType}
             (m₁ : CTerm Γ (e₁ / σ)) (m₂ : CTerm Γ (e₂ / σ))
             (m₃ : CTerm Γ (e₃ / σ)) (ρ : ⟪ Γ ⟫X) →
             sub-eq (+ass {e₁} {e₂} {e₃})
                    (⟦ CHOOSE m₁ (CHOOSE m₂ m₃) ⟧C ρ)
             ≡ ⟦ CHOOSE (CHOOSE m₁ m₂) m₃ ⟧C ρ
choose-ass m₁ m₂ m₃ ρ with ⟦ m₁ ⟧C ρ | ⟦ m₂ ⟧C ρ | ⟦ m₃ ⟧C ρ
... | bv₁ | bv₂ | bv₃ = lemma-ass++ bv₁ bv₂ bv₃

fails-earlier : {e : ℕ} {Γ : Ctx} {ρ : ⟪ Γ ⟫X} {σ σ' : VType}
                (m : CTerm (σ ::l Γ) (e / σ')) →
                ⟦ LET FAIL σ IN m ⟧C ρ ≡ ⟦ FAIL σ' ⟧C ρ
fails-earlier m = refl
```

Joonis 32: Mittedeterminismi spetsiifilised, efekti suhtes geneerilised teisendused.

```
failure : {Γ : Ctx} {σ : VType} (m : CTerm Γ (0 / σ)) →
          (ρ : ⟪ Γ ⟫X) →
          ⟦ m ⟧C ρ ≡ ⟦ FAIL σ ⟧C ρ
failure m ρ with ⟦ m ⟧C ρ
... | bv [] z≤n = refl

dup-comp : {e : ℕ} {Γ : Ctx} {σ σ' : VType}
           (m : CTerm Γ (1 / σ)) (n : CTerm (dupX here) (e / σ')) →
           (ρ : ⟪ Γ ⟫X) →
           sub-eq (errok-seq e)
                  (⟦ LET m IN LET wkC here m IN n ⟧C ρ)
           ≡ ⟦ LET m IN ctrC here n ⟧C ρ
-- proof omitted
```

Joonis 33: Mittedeterminismi monaadi efekti-spetsiifilised optimisatsioonid.

4 Kokkuvõte

Käesoleva töö eesmärgiks oli realiseerida sõltuvate tüüpidega programmeerimiskeeles Agda efektianalüüside ja neil põhinevate programmiteisenduste raamistu.

Esitati erandeid toetav näitekeel. Seejärel defineeriti erandite efektide hindamine, tuues sisse gradeeritud monaadi mõiste. Gradeeringu abil määrati alamtüübid ja -efektid, millele tugines keele rafineerimine. Keele semantika defineeriti juba rafineeritud keelele. Töö käigus valmisid programmiteisendused, mh "surnud" arvutuse ja korduva arvutuse eemaldamise optimisatsioonid. Ühtlasi näidati, et need teisendused on korrektsed.

Töö teises pooles kasutati mittedeterminismi toetavat näitekeelt. Keele semantika andmiseks loodi ülalt tõkestatud vektori andmestruktuur. Sellega koos anti naturaalarvude korrutamise jaoks sobiv gradeeritud monaad. Defineeriti termide tüübituletus ja keele rafineerimine. Esitati ebaõnnestunud arvutuse ja korduva arvutuse eemaldamise optimisatsioonid ning näidati selliste teisenduste korrektsust.

Sertifitseeritud programmeerimine on mahukas ettevõtmine, kuna arutelu isegi intuitiivselt õige aritmeetika üle võib osutuda ajakulukaks. Antud töö käigus ei jõutud tõestada tüübituletuste ja rafineerimiste korrektsust. Kuna optimeerimine eeldas, et semantika on antud rafineeritud keelele, siis on kriitilise tähtsusega kontrollida, et rafineerimine säilitab "toores" keeles kirjutatud programmi, mida tahetakse optimeerida.

Lisaks tüübituletuste ja rafineerimiste korrektsuse tõestamisele saab tööd jätkata, lisades monaadi seadustele tuginevad optimeerimised või hinnata mittedeterministliku keele tulemuste arvu nii ülevalt kui ka alt. Erandite ja mittedeterminismi näidetele sarnaselt on võimalik baaskeelt laiendada muteeritava olekuga, mille efektiks on oleku lugemine ja kirjutamine, ja tõestada sellele keelele spetsiifilised optimisatsioonid.

Kõigele vaatamata õnnestus efektianalüüside ja programmiteisenduste raamistu realiseerimine Agdas andmetüüpe toetavatele keeltele ja seega saab idee tõestuse lugeda edukaks.

Kasutatud kirjandus

[1] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[2] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. *Reading, Writing and Relations*, pages 114–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[3] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[4] Philip Wadler. The marriage of effects and monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 63–74, New York, NY, USA, 1998. ACM.

[5] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. *SIGPLAN Not.*, 49(1):633–645, January 2014.

[6] Nick Benton, Andrew Kennedy, Martin Hofmann, and Vivek Nigam. *Counting Successes: Effects and Transformations for Non-deterministic Programs*, pages 56–72. Springer International Publishing, Cham, 2016.