View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Helsingin yliopiston digitaalinen arkisto

Puhdas imperatiivinen uniikkityypitetty ohjelmointikieli

Martin Pärtel

Helsinki 23.3.2014
Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Martin Pärtel

Työn nimi — Arbetets titel — Title

Puhdas imperatiivinen uniikkityypitetty ohjelmointikieli

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	23.3.2014	77 sivua

Tiivistelmä — Referat — Abstract

Ohjelmoinnissa puhdas funktio ei vaikuta ohjelman tilaan muuten kuin palauttamalla paluuarvon, joka riippuu ainoastaan annetuista parametreista. Jotkut ohjelmointikielet sallivat lähtökohtaisesti vain puhtaiden funktioiden kirjoittamisen. Nämä kielet ovat yleensä tyyliltään 'funktionaalisia', eli ne suosivat funktion toteutuksen esittämistä yhtenä lausekkeena tavanomaisemman 'imperatiivisen' lausejonon sijaan.

Tutkielmassa esitellään uusi ohjelmointikieli *Uniic*, joka on tyyliltään imperatiivinen, mutta sallii vain puhtaiden funktioiden kirjoittamisen. Uniic käännetään puhtaasti funktionaaliselle kohdekielelle, mikä estää suunnittelemasta kieleen epäpuhtaita ominaisuuksia. Kääntäjästä on toteutettu prototyyppi.

Funktionaalisessa kohdekielessä on Clean-kieleen perustuva uniikkityypitysjärjestelmä. Uniikkityypitys mahdollistaa epäpuhtaiden operaatioiden turvallisen mallintamisen rajoittamalla uniikiksi merkityt arvot yhteen käyttökertaan. Uniicin lähdekielessä on rakenne nimeltä *lainaus*, jolla ohjelmoija voi näennäisesti käyttää uniikkia arvoa monta kertaa. Lainaus on toteutettu siten, ettei kohdekielen uniikkityypitysjärjestelmää tarvitse muuttaa.

Uniicin imperatiiviset piirteet sekä lainaus mahdollistavat koodin kirjoittamisen imperatiivisella tyylillä monessa tilanteessa, jossa olemassa olevat puhtaat kielet pakottavat funktionaaliseen tyyliin. Yritykset laajentaa Uniicia kielen puhtautta menettämättä auttavat hahmottamaan funktionaalisen ja imperatiivisen ohjelmoinnin eroavaisuuksia ja toisaalta myös potentiaalisia samankaltaisuuksia.

ACM Computing Classification System (CCS):
D.3 [Programming languages],
D.3.2 [Language Classifications] F.3.3 [Studies of Program Constructs]

Avainsanat — Nyckelord — Keywords

ohjelmointikieli, imperatiivinen, funktionaalinen, kääntäjä, tyyppijärjestelmä, uniikkityypitys

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto 1

2 Funktionaalinen ohjelmointi 3

2.1 Muuttujat 5

2.2 Funktiot 5

2.3 Ulkoinen tila, syöte ja tulostus 6

2.4 Uniikkityypitys I/O-ongelman ratkaisuna 7

2.5 I/O:n suoritusjärjestys 8

2.6 Puiden ja verkkojen tekstimuotoinen esitys 9

2.7 Yksinkertaisen funktionaalisen kielen määritelmä 10

2.8 Funktionaalisen kielen verkkoesitys 12

2.9 Verkkomuotoisen ohjelman suorittaminen 16

3 Imperatiivisen ohjelman analysointi 19

3.1 Yksinkertaisen imperatiivisen kielen määritelmä 19

3.2 Vuoverkko 20

3.3 Vuoanalyysit 23

3.4 Vuoanalyysirunko 24

3.5 Dominointipuu 26

3.6 SSA-muoto 28

4 Tyyppijärjestelmät ja uniikkityypitys 31

4.1 Hindley–Milner-tyypinpäättely 31

4.1.1 Tyyppiyhtälöt 33

4.1.2 Tyyppiyhtälöiden tuottaminen 34

4.1.3 Tyyppiyhtälöiden ratkaiseminen 36

4.1.4 Yleistetyt tyypit 37

4.1.5 Yleistettyjen tyyppien käyttö 38

4.2 Uniikkityypitys 39
4.2.1 Uniikkityypitysjärjestelmien perusteet 39
4.2.2 Viitteiden laskenta funktionaalisessa verkossa 40
4.2.3 De Vriesin uniikkityypitysjärjestelmä 43
4.2.4 Uniikkiusattribuutit propositiologiikan lausekkeina 43
4.2.5 Funktioiden sulkeumien uniikkiusattribuutit 45

5 Uniic-ohjelmointikieli 47

5.1 Imperatiivinen lähdekieli 47
5.1.1 Lähdekielen viitelaskuri 47
5.1.2 Lähdekielen tyypityssäännöt 49
5.1.3 Lainaus ja sen poistaminen 51
5.1.4 Lainauksen lisääminen tyyppijärjestelmään 52
5.2 Funktionaalinen kohdekieli 53
5.2.1 Kohdekielen tyyppijärjestelmä 53
5.2.2 Huomioita kohdekielen tyyppijärjestelmän toteutuksesta . . . 55
5.3 Lähdekielen muuntaminen kohdekieleksi 56
5.4 Uniic-ohjelmaesimerkki 60

6 Uniic-ohjelmointikielen analyysi 62

6.1 Uniic funktionaalisiin kieliin verrattuna 63
6.2 Uniicin jatkokehitysmahdollisuuksia 65
6.3 Uniicin rajat imperatiivisessa ohjelmoinnissa 70
6.4 Lievempiä versioita puhtaudesta 72

7 Yhteenveto 74

Lähteet 75

Liitteet

1 Imperatiivisen ohjelman muuttaminen vuoverkoksi

2 Uniicin prototyyppitoteutus

1 Johdanto

Ohjelmoinnissa funktio on *puhdas*, jos se tuottaa samoilla parametreilla aina saman paluuarvon eikä aiheuta *sivuvaikutuksia*. Sivuvaikutus on mikä tahansa funktion ulkopuolelle näkyvä muutos ohjelman tai järjestelmän tilaan paluuarvon tuottamista lukuun ottamatta. Puhtaus tekee ohjelmasta ymmärrettävämmän sekä helpommin optimoitavan ja rinnakkaistettavan.

Funktionaaliset ohjelmointikielet pakottavat tai ainakin johdattelevat ohjelmoijaa käyttämään vain sellaisia kielen rakenteita, jotka takaavat koodin puhtauden. Näihin rakenteisiin ei tavallisesti kuulu mahdollisuutta sijoittaa kerran alustettuun muuttujaan tai tietorakenteeseen uutta arvoa. Näin ollen myös perinteiset laskurimuuttujan päivittämiseen perustuvat silmukkarakenteet puuttuvat. Ohjelman tilan muuttamiseen perustuvia kielen rakenteita sanotaan *imperatiivisiksi* rakenteiksi.

Tässä työssä tutkitaan puhtaita imperatiivisia rakenteita rakentamalla imperatiivinen kieli nimeltä *Uniic*, joka käännetään puhtaalle funktionaaliselle kielelle. Käännös on ”yksi yhteen” siinä mielessä, että jokainen kohdekielelle käännetty funktio saa täsmälleen samat parametrit kuin vastaava lähdekielinen funktio. Tämä osoittaa imperatiivisen lähdekielen puhtauden ja asettaa mielenkiintoisen rajoitteen imperatiivisten ominaisuuksien suunnittelulle.

Uniicin käytettävyys perustuu osittain *uniikkityypitykseen* ja *lainaukseen*. Uniikkityypitys valvoo, että joitakin arvoja käytetään ohjelmassa vain kerran ja mahdollistaa muuttuvan tilan mallintamisen puhtaassa kielessä. Lainaus puolestaan virtaviivaistaa uniikkityypityksen käyttöä imperatiivisessa kielessä siten, että yhä useammat ohjelmat saadaan näyttämään aidosti imperatiivisilta.

Eräs mahdollinen käytännönläheinen käyttötapaus Uniicin kaltaiselle kielelle on funktionaalisen kielen alikielenä toimiminen. Esimerkiksi funktionaalisen Haskell-kielen **do**-syntaksi [Has10, luku 3.14] voidaan nähdä hyvin rajoitettuna puhtaana imperatiivisena alikielenä, johon voisi helposti lisätä imperatiivisia rakenteita Uniicissa käytettävillä tekniikoilla.

Uniic toimii myös eräänlaisena mittarina imperatiivisten rakenteiden puhtaudelle. Jos imperatiivisen rakenteen voi lisätä Uniiciin helposti, rakenne on pohjimmiltaan puhdas. Jos lisäys ei onnistu suoraan, voidaan rakennetta ehkä muokata tai rajoittaa sopivalla tavalla, jolloin rakenteesta saadaan ainakin puhtauden näkökulmasta parempi versio.

Tutkielman luvut 2-4 esittävät tarvittavat taustatiedot funktionaalisten ja impera-

tiivisten kielten sekä tyyppijärjestelmien formaalista käsittelystä. Uniicin lähde- ja kohdekieli sekä muunnos lähdekieleltä kohdekielelle esitetään luvussa 5. Luku 6 vertaa Uniicia muihin kieliin ja hakee Uniicin käännöstekniikan rajoja pohtimalla kielen erilaisia laajennusmahdollisuuksia. Luku 7 tekee yhteenvedon kielen toteutuksen mielenkiintoisista puolista sekä kielen laajennusmahdollisuuksista.

2 Funktionaalinen ohjelmointi

Perinteisessä imperatiivisessa ohjelmoinnissa ohjelma käsitetään sarjana komentoja, jotka muuttavat ohjelman tilaa muistissa. Muuttujia ja tietorakenteiden kenttiä voi pääsääntöisesti muuttaa sijoituskomennoilla. Funktionaalisessa ohjelmoinnissa muuttujat ja arvot ovat muuttumattomia (*immutable*). Muuttujan tai tietorakenteen arvo asetetaan pysyvästi sen luonnin aikana, eikä *puhtaasti funktionaalisessa* kielessä ole mitään tapaa arvon muuttamiseen. Funktionaalista ohjelmaa ei yleensä ole edes mielekästä ajatella sarjana muistia käsitteleviä komentoja. Sen sijaan ohjelma on vain lauseke ja ohjelman suorittaminen on vain lausekkeen arvon laskentaa.

Seuraava esimerkki näyttää imperatiivisen Java-aliohjelman kertoman laskemiseksi. Aliohjelmassa on silmukka, joka päivittää kierroslaskuria i ja tulosmuuttujaa x.

Esimerkki 1. Kertoman laskenta imperatiivisesti (Java)

```
public static int factorial(int n) {
    int x = 1;
    for (int i = 2; i <= n; ++i) {
        x = x * i;
    }
    return x;
}
```

□

Funktionaalisessa ohjelmoinnissa ei voi päivittää muuttujia tällä tavalla, eikä funktionaalisissa kielissä siksi ole samanlaisia silmukkarakenteita kuin imperatiivisissa kielissä. Sen sijaan funktionaalisessa ohjelmoinnissa perustoistorakenne on rekursio. Seuraava esimerkki näyttää kertomafunktion toteutuksen rekursiolla. Tämä toteutus on funktionaalinen, koska siinä ei ole ainuttakaan arvoa muuttavaa komentoa.

Esimerkki 2. Kertoman laskenta funktionaalisesti (Java)

```
public static int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}
```

□

Seuraava esimerkki näyttää saman rekursiivisen kertomafunktion määritelmän Haskell-ohjelmointikielellä. Esimerkistä nähdään funktionaalinen ajattelutapa: aliohjelmaa pidetään matemaattisena funktiona, joka määritellään yhtälöllä. Tässä tapauksessa funktio on määritelty paloittain tavalla, joka muistuttaa matemaattista määrittelytapaa.

Esimerkki 3. Kertoman laskenta funktionaalisesti (Haskell)

```
factorial n | (n <= 1)  = 1
            | otherwise = n * factorial (n - 1)
```

Vertaa matemaattiseen määrittelytapaan:

$$f(n) = \begin{cases} 1 & \text{jos } n \leq 1 \\ n \cdot f(n-1) & \text{muuten} \end{cases}$$

□

Funktionaalisessa ohjelmoinnissa lausekkeen tulos on aina sama riippumatta siitä, milloin ja montako kertaa se lasketaan. Tätä sanotaan *viitteiden läpinäkyvyydeksi* (*referential transparency*). Esimerkiksi sijoitusten $y = f(x)$ ja $z = f(x)$ jälkeen y:llä on varmasti sama arvo kuin z:lla. Tämä johtuu siitä, ettei x:n arvo voi muuttua kutsujen aikana tai niiden välissä, ja toisaalta siitä, ettei f voi lukea ulkoista tilaa, kuten ohjelman syötettä tai järjestelmän kelloa.

Funktionaaliselle ohjelmoinnille on ominaista funktioiden käyttäminen arvoina. Tämä mahdollistaa muun muassa hyödyllisten kontrolliabstraktioiden luomisen, koska tavanomaisia silmukoita ei kielessä ole ja rekursion käyttäminen aina, kun tarvitaan toistoa, olisi työlästä.

Esimerkiksi funktio `map` ottaa parametrikseen funktion f sekä listan $[x_0, x_1, \ldots]$ ja palauttaa uuden listan $[f(x_0), f(x_1), \ldots]$. Seuraava Haskellin määritys palauttaa listan kaikista parillisista luvuista välillä 2..200 kaksinkertaistamalla listan [1..100] luvut.

```
parillisiaLukuja = map f [1..100]
  where f x = 2 * x
```

Viitteiden läpinäkyvyyden nojalla `map` voi laskea uuden listan alkioiden arvot missä järjestyksessä hyvänsä, sillä laskennat eivät voi vaikuttaa toisiinsa. Kääntäjä, tai ohjelmoija, voi siis halutessaan huoletta rinnakkaistaa `map`:n suorituksen.

Viitteiden läpinäkyvyydestä nauttivassa kielessä ei myöskään ole merkitystä, sisäl-

tävätkö muuttujat arvoja vai viitteitä arvoihin. Jos ohjelma ei voi muuttaa mitään olemassa olevaa arvoa, ei ole väliä, kopioidaanko esimerkiksi parametrinvälityksessä koko arvo vai vain viite siihen.

2.1 Muuttujat

Funktionaalisessa ohjelmoinnissa apumuuttujia sidotaan yleensä `let`-rakenteella. Tarkastellaan seuraavaa esimerkkiä.

```
let x = 2
and y = x * x
and z = y * 10 + x
in z + 2
```

`let ... in ...`-rakenne on yksi lauseke, joka luo annetut muuttujat yhteiseen näkyvyysalueeseen, jolloin muuttujien määritelmät voivat viitata toisiinsa. `let`-rakenteen arvo lasketaan `in`-osassa olevasta lausekkeesta. `let`-rakenteella ei voi muuttaa olemassa olevan muuttujan arvoa, kuten seuraavassa esimerkissä yritetään tehdä.

```
let x = 1
and y = (let x = 100 in x)
in x + y
```

Tässä sisempi `let` vain *peittää* (*shadows*) ulomman `x`:n omassa näkyvyysalueessaan, joten ulommassa `in`-osassa `x`:n arvo on yhä `1`. Tämän `let`-lausekkeen tulos on siis `101`, eikä esimerkiksi `200`.

2.2 Funktiot

Funktionaalisessa ohjelmoinnissa funktiot ovat tavallisia arvoja, joita voi välittää parametreina ja palauttaa paluuarvoina. Funktionaalisen kielen syntaksi tavallisesti sallii funktion määrittelemisen kaikkialla, missä sallitaan mielivaltaisia lausekkeita.

Funktion määrittelyn syntaksi on jatkossa `\(`x_1`, `x_2`, ..., `x_n`).` e, missä x_i:t ovat parametreja ja e on funktion runko. Seuraavassa esimerkissä määritellään muuttujan `f` arvoksi kaksiparametrinen funktio, jota sitten kutsutaan.

```
let f = \(x, y). x + y
in f(3, 4)
```

Funktioita voi määritellä missä tahansa lausekkeessa, siis myös muiden funktioiden sisällä. Funktiot saavat vapaasti viitata kaikkiin näkyvyysalueessaan oleviin muuttujiin, kuten seuraavassa esimerkissä.

```
let f = \(x).
  let g = \(y). x + y
  in g
and h = f(3)
in h(4) + h(5)
```

Tässä lausekkeessa funktio `h` kutsuu funktiota `f`, joka palauttaa uuden funktion `g`, joka puolestaan käyttää `f`:lle annettua parametria `3` oman parametrinsa `y` lisäksi. Lausekkeen `h(4) + h(5)` tulos on siis $(3 + 4) + (3 + 5) = 15$.

Niiden muuttujien joukko, jota funktio käyttää ulkopuoleltaan, on nimeltään funktion *sulkeuma* (*closure*). Sulkeumalliset funktiot ovat funktionaalisessa ohjelmoinnissa tärkeä abstraktion väline. Ilman niitä, ohjelmoijan täytyisi välittää erittäin paljon parametreja pienille apufunktioilleen. Asia käy ilmi myös Uniicin käännöksessä luvussa 5.3.

2.3 Ulkoinen tila, syöte ja tulostus

Puhtaasti funktionaalisessa ohjelmoinnissa ei lähtökohtaisesti voi olla funktiota, joka lukisi käyttäjän syötettä. Tällainen funktio rikkoisi viitteiden läpinäkyvyyden, sillä käyttäjän syöte voi olla erilainen eri kutsukerroilla. Olisi kuitenkin hyödyllistä, että ohjelmat voisivat keskustella suoritusympäristön ja käyttäjän kanssa. Tätä sanotaan funktionaalisten kielten syöte/tulostus-ongelmaksi eli *I/O-ongelmaksi*. Suosituksi tullut ratkaisu, joka ei riko viitteiden läpinäkyvyyttä ja jota Haskellkin käyttää, on ns. monadinen (*monadic*) I/O.

Monadisessa I/O:ssa pääohjelma on tyypiltään *toiminto*. Valmiiksi annettuja kirjastotoimintoja voi olla esimerkiksi ”lue rivi syötettä” tai ”tulosta luku n”, ja toiminnolla voi olla jokin tulos, kuten juuri luettu syöterivi. Toimintoja voi yhdistää toisiinsa ”*jatkefunktioilla*”.

Jatkefunktio on tavallinen puhtaasti funktionaalinen funktio, joka ottaa parametrikseen edellisen toiminnon tuloksen ja palauttaa sen perusteella seuraavaksi suoritettavan toiminnon. Ohjelman suoritusympäristö siis laskee ensiksi ohjelman ensimmäisen toiminnon, suorittaa sen ja kutsuu tuloksella jatkefunktiota, jolta se saa seuraavan suoritettavan toiminnon.

Seuraava esimerkki näyttää monadista I/O:ta käyttävän Haskell-ohjelman, joka lukee yhden syöterivin ja tulostaa sen isoilla kirjaimilla. `getLine` on toiminto rivin lukemiseksi. Operaattori `>>=` rakentaa `getLine`:sta jatkefunktioon `f` yhdistetyn toiminnon. Jatkefunktio `f` saa luetun rivin parametrikseen ja muuttaa rivin kunkin kirjaimen isoksi (`map toUpper`). Funktiolta `putStrLn` saadaan toiminto isokirjaimisen rivin tulostamiseksi.

Esimerkki 4. Monadisen I/O:n käyttö Haskellissa

```
main = getLine >>= f
  where f rivi = putStrLn (map toUpper rivi)
```

□

I/O-ongelmaan on muitakin ratkaisuja [HHP07, luku 7]. Näistä eräs on uniikkityypitys.

2.4 Uniikkityypitys I/O-ongelman ratkaisuna

Clean-ohjelmointikielessä [PvE02] I/O-operaatiot ovat tavallisia funktioita, jotka kohdistuvat johonkin tilaa mallintavaan olioon. Esimerkiksi tiedostoa käsittelevät funktiot ottavat parametrikseen `File`-tyyppisen olion. Nämä tilalliset oliot ovat uniikkeja, eli tyyppijärjestelmä varmistaa, että niihin on enintään yksi viite kerrallaan. Uniikin olion muuttaminen ei riko viitteiden läpinäkyvyyttä, koska uniikkia oliota ei voi käyttää kahdessa paikassa.

Funktio, joka muuttaa uniikkia oliota, joutuu palauttamaan uuden viitteen muuttamaansa olioon. Esimerkiksi Cleanin tiedostosta rivin lukeva `freadline`-funktio ottaa parametrikseen uniikin `File`-olion ja palauttaa (merkkijono, `File`-olio) -parin. Jos `File`-oliota ei palautettaisi, kutsuva funktio ei voisi enää mitenkään jatkaa tiedoston käsittelyä, sillä parametrina välitettyä `File`-oliota ei voi uniikkiuden takia käyttää uudelleen.

Seuraava esimerkki näyttää Clean-ohjelman, joka lukee syöterivin ja tulostaa sen

isoilla kirjaimilla. Funktio `stdio` avaa syöttö- ja tulostusvirran tiedostona, `freadline` lukee tiedostosta yhden rivin, `fwrites` kirjoittaa tiedostoon rivin ja `fclose` synkronoi muutokset tiedostoon ja sulkee sen. Pääohjelma `Start` on tyypiltään funktio, joka ottaa ulkoista tilaa esittävän uniikin `world`-olion ja palauttaa muuttuneen `world`-olion.

Esimerkki 5. Uniikkityypitetty I/O Cleanissa

```
Start world
  # (tiedosto, world) = stdio world
  # (rivi, tiedosto) = freadline tiedosto
  # tiedosto = fwrites {toUpper kirjain \\ kirjain <-: rivi} tiedosto
  # (ok, world) = fclose tiedosto world
  = world
```

□

Merkintä '`#`' tarkoittaa muuttujien esittelyä. Esimerkiksi `tiedosto`-muuttuja täytyy aina esitellä ja sitoa uudelleen käytön jälkeen, sillä muuten kääntäjä ei anna käyttää sitä toista kertaa. Huomaa, että kyseessä ei ole muuttujan arvon uudelleenasettaminen vaan vanhan muuttujan peittäminen. Näin ollen '`#`' ei mahdollista esimerkiksi imperatiivisten silmukoiden kirjoittamista.

Uniikkiuteen perustuva I/O ja monadinen I/O ovat yhtä voimakkaita, ja molemmat voivat sisältyä samaan ohjelmointikieleen. Ohjelma, joka käyttää niistä yhtä voidaan suoraviivaisesti muuttaa käyttämään toista [AP01] (myös [deV08, s. 66]).

2.5 I/O:n suoritusjärjestys

Funktionaaliset ohjelmointikielet eivät aina takaa lausekkeiden tarkkaa laskentajärjestystä. Tilan mallintaminen monadisella I/O:lla tai uniikeilla olioilla kuitenkin pakottaa tilaa muuttavat operaatiot oikeaan järjestykseen.

Kahdesta peräkkäisestä oliota muuttavista funktiokutsuista jälkimmäistä ei voida suorittaa (laskea loppuun) ennen ensimmäistä, koska jälkimmäinen tarvitsee ensimmäisen paluuarvon. Vastaavasti monadisen I/O:n jatkefunktiolle ei ole antaa parametria, jolla laskea seuraava toiminto, ennen kuin ensimmäinen I/O-operaatio on suoritettu.

Suoritusjärjestys voi olla jopa liian tiukasti rajoitettu. Esimerkiksi lukuoperaatiot

uniikista tietorakenteesta voitaisiin hyvin suorittaa rinnakkain, kunhan välissä ei tapahdu kirjoitusoperaatioita. Tämän ilmaiseminen pelkällä uniikkityypityksellä ei onnistu, ja monadinen I/O kärsii samasta ongelmasta.

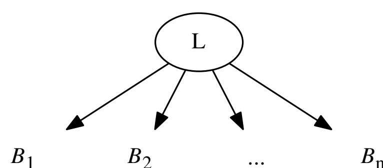
Ongelma voidaan ratkaista pitämällä luku- ja kirjoitusoperaatioita sopivalla tavalla eriarvoisina tyyppijärjestelmässä. Esimerkiksi eräässä uniikkityypitystä muistuttavassa ratkaisussa [TeA05] sekä luku- että kirjoitusoperaatiot ottavat ja palauttavat "todistajia" (*witness*) toistensa suorituksesta. Tyyppijärjestelmä tarkistaa, että todistajat liikkuvat ohjelmassa siten, että luku- ja kirjoitusoperaatioiden keskinäinen suoritusjärjestys on riittävän hyvin määritelty, vaikka lukuoperaatiot saatetaankin suorittaa rinnakkain. Tässäkin järjestelmässä täytyy tyyppiturvallisuuden takia huomioida, mitkä muuttujat saattavat osoittaa samaan olioon.

2.6 Puiden ja verkkojen tekstimuotoinen esitys

Ohjelmointikielten rakenne on luontevaa määritellä abstrakteina syntaksipuina. Jatkossa käsitellään myös ohjelmien verkkomuotoisia esityksiä. On siis hyödyllistä olla tekstimuotoinen merkintätapa puille ja verkoille.

Olkoon puiden ja verkkojen tekstimuotoinen merkintätapa seuraava[BaS93]. Tekstissä solmu A, jonka sisältö on `L` ja joka osoittaa solmuihin $B_1, B_2, \ldots, B_n$, merkitään seuraavasti.

$$A = \mathtt{L}(B_1, B_2, \ldots, B_n)$$

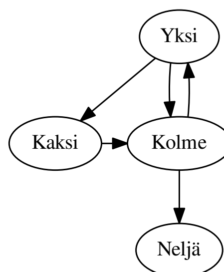


Solmu, jolla ei ole alisolmuja, saa tyhjät sulut (esimerkiksi `L()`).

Seuraava esimerkki määrittelee pienen syklisen verkon tällä merkintätavalla.

Esimerkki 6. Verkon merkintätapa

$$A = \texttt{Yksi}(\texttt{Kaksi}(B), B)$$
$$B = \texttt{Kolme}(A, \texttt{Neljä}())$$



□

2.7 Yksinkertaisen funktionaalisen kielen määritelmä

Seuraavaksi määritellään yksinkertaisen ML:ää muistuttavan funktionaalisen kielen rakenne. Luvussa 4.1 tähän kieleen lisätään tyyppijärjestelmä, ja luvussa 5.2 se laajennetaan Uniicin kohdekieleksi. Määritelmä kattaa funktionaalisten kielten tärkeimmät ominaisuudet ja jättää tämän työn kannalta epäoleelliset ominaisuudet pois. Tässä sivuutettuja funktionaalisten kielten ominaisuuksia käsittelee formaalisti muun muassa [Pie02].

Seuraava määritelmä listaa funktionaalisen esimerkkikielen abstraktin syntaksipuun mahdolliset solmut. Kukin solmutyyppi selitetään alla. Kielen kaikki rakenteet ovat joko vakioita tai lausekkeita.

Määritelmä 1. Yksinkertaisen funktionaalisen kielen rakenteet

Vakiot:

$$c \texttt{ ::= } n \in \mathbb{Z} \mid \texttt{true} \mid \texttt{false}$$

Lausekkeet:

$$
\begin{aligned}
e \; \texttt{::=} \; & \texttt{Const[}c\texttt{]}(), \text{ missä } c \text{ on vakio} \\
\mid \; & \texttt{Var[}v\texttt{]}(), \text{ missä } v \text{ on muuttuja} \\
\mid \; & \texttt{Tuple(}e_1, \ldots e_n\texttt{)} \\
\mid \; & \texttt{Let}(v_1, e_1, \ldots, v_n, e_n, e) \\
\mid \; & \texttt{Lambda}(\texttt{ParamList}(v_1, \ldots, v_n), e) \\
\mid \; & \texttt{Apply}(e, e_1, \ldots, e_n) \\
\mid \; & \texttt{If(}e_1\texttt{,}e_2\texttt{,}e_3) \\
\mid \; & \texttt{Match}(e, \texttt{Case}(p_1, e_1), \ldots, \texttt{Case}(p_n, e_n)) \qquad \square
\end{aligned}
$$

Monikko (`Tuple`) on järjestetty kokoelma, jonka arvojen määrä ja tyypit tiedetään käännösaikana. Monikkoa voi käyttää esimerkiksi usean arvon palauttamiseen funktiosta yhtenä arvona, eikä kieleen tarvitse näin lisätä erityistä tukea useammalle kuin yhdelle paluuarvolle.

`Let`-solmu määrittelee joukon muuttujia. Sen alisolmut ovat vuorotellen muuttujanimiä ja lausekkeita, ja muuttuja v_i sidotaan saa lausekkeen e_i arvoon. `Let`-solmun arvoksi lasketaan viimeisen alisolmun e arvo. Määritellyt muuttujat ovat jokaisen alilausekkeen e_i sekä e näkyvyydessä. `Let`-solmujen täsmällinen käyttäytyminen määriteltävässä kielessä käy ilmi seuraavissa luvuissa.

`Lambda`-solmu on funktiomääritelmä. Sen ensimmäinen alisolmu listaa parametrina otettavat muuttujat, ja toinen alisolmu on funktion runko. Funktiokutsussa funktion rungosta tehdään kopio, jossa parametrimuuttujat korvataan annetuilla parametriarvoilla.

`Apply`-solmu on funktiokutsu. Sen ensimmäinen alisolmu on kutsuttava funktio, ja loput alisolmut ovat funktiolle annettavia parametreja.

`If`-solmu on ehtolauseke. Sen ensimmäinen alisolmu on ehto, toinen alisolmu `true`-tapauksessa suoritettava lauseke ja kolmas alisolmu `false`-tapauksessa suoritettava lauseke.

`Match`-solmu on hahmonsovituslauseke. Sen ensimmäinen alisolmu on mielivaltainen lauseke, jonka arvoa vertailaan hahmoihin. Loput solmut ovat hahmo–lauseke -pareja.

Hahmo on lauseke, jossa on vain `Const`-, `Var`- ja `Tuple`-solmuja. Hahmo–lauseke

-pareista valitaan suoritettavaksi se lauseke, jonka parina olevaan hahmoon vertailtavan lausekkeen arvo sopii. Arvo sopii hahmoon, kun se on muuttujia lukuunottamatta sama kuin vertailtava arvo. Esimerkiksi arvo `Tuple(1, Tuple(2, 3))` sopii hahmoihin `Tuple(1, x)` ja `Tuple(1, Tuple(x, 3))` mutta ei hahmoon `Tuple(1, 2)`.

Huomaa, että ehtolauseke on oikeastaan vain erikoistapaus hahmonsovituslausekkeesta. Lauseke `If(`a`, `b`, `c`)` voidaan aina korvata lausekkeella `Match(`a`, Case(`**`true`**`, `b`), Case(`**`false`**`, `c`))` Ehtolauseke otetaan kuitenkin kieleen mukaan, koska sitä käytetään todellisessa koodissa usein, ja sen käyttäytyminen eri määritelmissä on hieman helpompi ymmärtää.

Kielen suoritustapa voitaisiin määritellä suoraan abstrakteille syntaksipuille, mutta koska kielelle halutaan määritellä myöhemmin uniikkityypitys, on hyödyllisempää muuntaa abstrakti syntaksipuu ensin verkoksi.

2.8 Funktionaalisen kielen verkkoesitys

Funktionaalinen ohjelma voidaan esittää abstraktista syntaksipuusta johdettuna *funktionaalisena verkkona*. Funktionaalisia verkkoja kutsutaan usein *verkon uudelleenkirjoitusjärjestelmiksi* (*graph rewrite system*), koska kielen suoritussäännöt määritellään monesti verkkomuotoiselle ohjelmalle puumuotoisen sijaan. Verkot ovat hyödyllisiä myös uniikkityypityksessä, koska niistä on helpompaa löytää useaan kertaan käytetyt arvot kuin syntaksipuista, kuten luvussa 4.2.2 nähdään.

Tässä esitettävä verkkoesitys on saman kaltainen kuin Barendsenin ja Smetsersin uniikkityypitystä varten määrittelemä verkon uudelleenkirjoitusjärjestelmä [BaS93]. Barendsenin ja Smetsersin järjestelmä ei määrittele mekanismia uusien funktioiden luomiseksi vaan jättää funktioiden määrittelytavan auki. Funktioiden määrittelytavaksi valitaan tässä luvun 2.7 lambda-solmut yleistettynä verkkoihin.

Esimerkki muunnoksesta

Tarkastellaan seuraavaa funktionaalista ohjelmaa. Huomaa, että ohjelman funktio `f` on rekursiivinen.

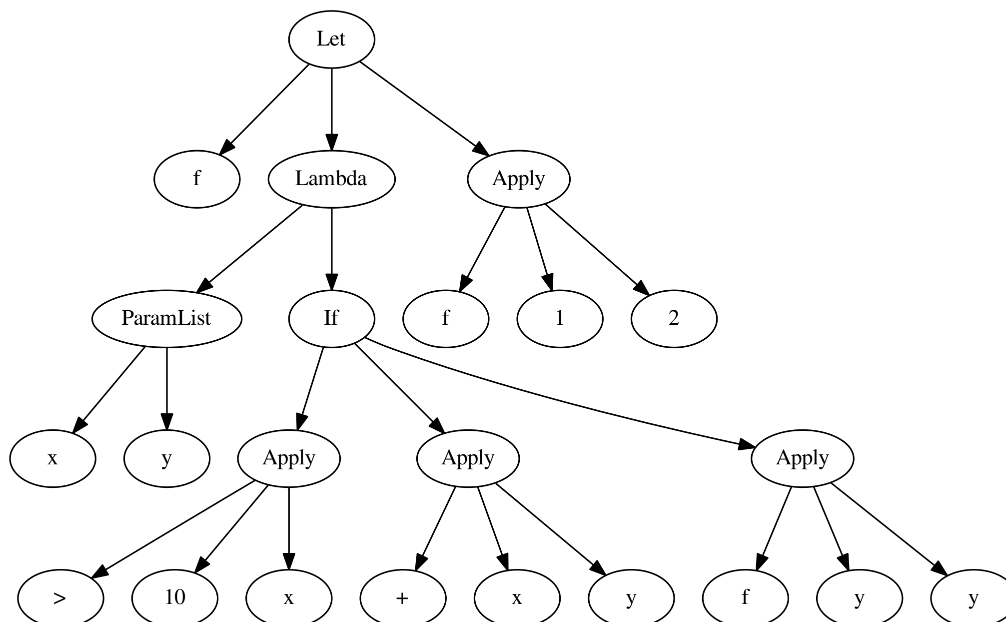
Esimerkki 7

```
let f = \(x, y). if 10 > x then x + y else f(y, y)
in f(1, 2)
```

□

Ohjelman abstrakti syntaksipuu on seuraava.

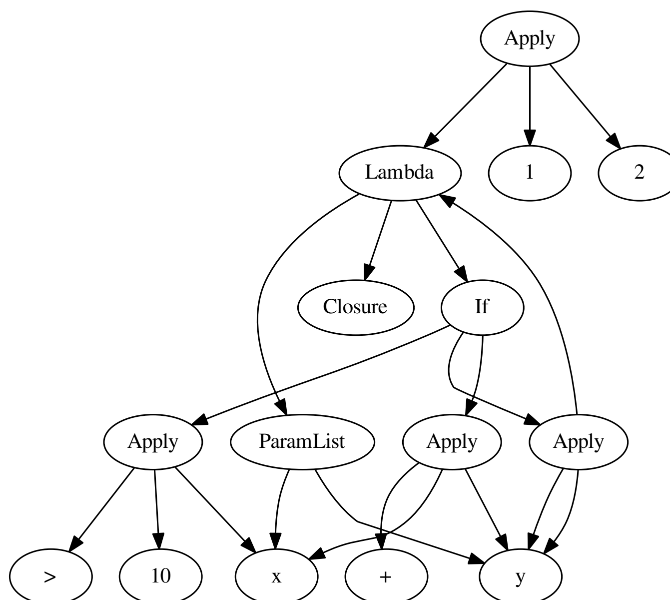
Esimerkki 8



□

Verkkoesitys on syntaksipuuhun nähden erilainen kahdella tavalla: samaa vakiota tai muuttujaa esittävät solmut on yhdistetty yhdeksi solmuksi, ja kaikki viittaukset `let`-rakenteella sidottuihin muuttujiin on muutettu viittaamaan vastaavaan arvoon. Funktion parametreja esittäviä muuttujia ei yleisessä tapauksessa voida korvata, sillä niiden arvoja ei (yleensä) tiedetä käännösaikana.

Esimerkki 9



□

Erilliset muuttujasolmut on nyt yhdistetty, ja verkosta näkee selvästi esimerkiksi, että muuttujaa `y` käytetään `f`:n rungon sisältä kolme kertaa. Muuttujan käyttökertojen laskeminen ei ole kuitenkaan aivan näin yksinkertaista, sillä funktio `f` on rekursiivinen, mikä ilmenee verkossa syklinä. Muuttujien käyttökertojen laskentaan palataan uniikkityypityksen yhteydessä luvussa 4.2.2.

Verkon solmutyypit ovat samat kuin abstraktissa syntaksipuussa sillä erotuksella, että `Let`-solmuja ei ole ja `Lambda`-solmuilla on yksi alisolmu lisää edustamassa funktion sulkeumaa. Sulkeuman esittäminen on oleellista, kun uniikkityypityksessä halutaan tarkistaa, viittaako funktio ulkoisiin uniikkeihin muuttujiin.

Muunnoksen määritelmä

Seuraavaksi määritellään muunnos abstraktilta syntaksipuulta verkoksi täsmällisesti. Muunnos tehdään seuraavissa vaiheissa, joiden yksityiskohdat selitetään alla.

1. Abstrakti syntaksipuu muunnetaan verkoksi, jossa on ylimääräisiä apusolmuja `Let`-solmujen poistamisen jäljiltä. Myös sulkeumasolmut puuttuvat.

2. Apusolmut poistetaan.
3. Identtiset vakiosolmut yhdistetään.
4. Sulkeumasolmut täydennetään.

Ensimmäinen vaihe määritellään rekursiivisena funktiona $G(S, T)$, missä T on muunnettava abstrakti syntaksipuu ja S on symbolitaulu eli kuvaus muuttujilta solmuille. Algoritmin jokainen askel käsittelee puun juurta tapauskohtaisesti.

Algoritmi 1. Funktionaalisen ohjelman muuntaminen verkoksi

$$
\begin{aligned}
&G(S, \texttt{Const[}c\texttt{]}()) = \texttt{Const[}c\texttt{]}() \\
&G(S, \texttt{Var[}v\texttt{]}()) = S(v) \text{ jos määritelty, } \texttt{Var[}v\texttt{]}() \text{ muuten} \\
&G(S, \texttt{Lambda}(\texttt{ParamList}(v_1, \ldots, v_n), T) = \\
&\quad \texttt{Lambda}(\texttt{ParamList}(G(S', v_1), \ldots, G(S', v_n)), \texttt{Closure}(), G(S', T)) \\
&\qquad \text{missä } S' = S \cup \{v_1 \to \texttt{Var[}v_1\texttt{]}, \ldots, v_n \to \texttt{Var[}v_n\texttt{]}\} \\
&G(S, \texttt{Let}(v_1, T_1, \ldots, v_n, T_n, T_e)) = G(S', T_e) \\
&\quad \text{missä } S' = S \cup \{v_1 \to \texttt{Thunk}(S', T_1), \ldots, v_n \to \texttt{Thunk}(S', T_n)\} \\
&G(S, X(T_1, \ldots, T_n)) = X(G(S, T_1), \ldots, G(S, T_n)) \text{ muille solmutyypeille } X
\end{aligned}
$$

□

`Lambda`-solmujen parametrimuuttujat sidotaan symbolitaulussa uusiin muuttujasolmuihin peittäen mahdolliset vanhat sidonnat. Näin `Lambda`-solmujen alla olevat parametrimuuttujien käyttökohdat tulevat osoittamaan samaan muuttujasolmuun kuin parametrilistasolmu `ParamList` viittaa. `Lambda`-solmuille lisätään myös toistaiseksi tyhjäksi jäävä `Closure`()-solmu.

`Let`-solmujen muunnoksessa halutaan, että jokaisen let-sidonnan v_i = T_i oikeaa puolta käsiteltäessä olisi symbolitaulussa käytettävissä sidonnat $v_j \to G(S', T_j)$ jokaiselle j. Tämä ei kuitenkaan onnistu suoraan, koska tarvittava symbolitaulu S' ei ole vielä valmis, kun let-sidontoja muunnetaan. Sopivaa symbolitaulua ei voi rakentaa yksi let-sidonta kerrallaan, koska sidonnat voivat vapaasti viitata toisiinsa.

Ratkaisu on luoda uusi symbolitaulu, jossa on sidonnat $v_i \to \texttt{Thunk}(S', T_i)$. Tällä merkinnällä tarkoitetaan, että muuttuja v_i sidotaan apusolmuun, jossa on puolestaan viittaus takaisin luotavaan symbolitauluun.

Muunnoksen viimeisteleminen

Muunnoksen seuraava vaihe on apusolmujen poistaminen. Apusolmut voidaan poistaa missä tahansa järjestyksessä. Apusolmu A = `Thunk`(S, T) poistetaan korvaamalla kaikki viittaukset solmuun A viittauksella solmuun $G(S, T)$. Ratkaisu toimii, koska $G(S, T)$ toimii, vaikka osa solmuista, joihin S:ssä on sidonnat, olisivat vielä `Thunk`-apusolmuja.

Ratkaisu joutuu kuitenkin ikuiseen silmukkaan, jos ohjelmassa on kehämääritelmä `let a1 = a2 and a2 = ... and an = a1 in ...`. Jokaisen kehässä olevan muuttujan uudeksi sidonnaksi tulee apusolmun eliminoinnin jälkeen uudestaan apusolmu. Toteutuksen täytyy antaa tällaisesta kehämääritelmästä virheilmoitus. Huomaa, että let-sidonta `a1 = f(a2) ...` ei voi olla osa kehämääritelmää, koska `a1`:n sidonnaksi tulee apusolmun poistamisen seurauksena `Apply`-solmu eikä uusi `Thunk`-solmu.

Muunnoksesta ovat jäljellä enää vakiosolmujen yhdistäminen ja sulkeumasolmujen lisääminen. Vakiosolmujen yhdistämisessä jokaiselle vakiolle c valitaan yksi edustajasolmu `Const[`c`]`, jolla kaikki muut samanlaiset solmut korvataan.

Sulkeumasolmut

Funktion `Lambda`(P, C, B) sisäpuolella katsotaan olevan niiden solmujen, joihin on polku funktion rungosta B. Kaikki muut solmut ovat funktion ulkopuolella.

Huomaa, että rekursiivinen funktio on itsensä sisäpuolella, mutta ei-rekursiivinen funktio ei ole. Funktion sulkeumaan kuuluvat ne solmut, joihin on polku sekä funktion ulkopuolelta että sisäpuolelta.

Kun jokaisen `Closure`()-solmun alisolmuiksi on liitetty kyseisen funktion sulkeumaan kuuluvat solmut, funktionaalinen verkko on valmis. Funktionaalista verkkoa käytetään seuraavassa luvussa, kun kohdekieltä suoritetaan, ja luvussa 4, kun kohdekieltä tyyppitarkistetaan.

2.9 Verkkomuotoisen ohjelman suorittaminen

Seuraavaksi selitetään funktionaalisen kielen suoritustapa. Kielen suoritus muotoillaan funktionaalisen verkon muuntamisena. Funktionaalisten kielten laskentasäännöt voitaisiin määritellä myös abstraktille syntaksipuulle [Pie02], mutta koska uniikkityypitys tehdään jatkoassa verkomuotoiselle ohjelmalle, on mielekästä myös suorittaa ohjelmat verkkomuotoisena. Kielen suoritustapa on samankaltainen kuin Clean-kielelle suunniteltu suoritustapa [BaS93].

Verkkomuotoisen ohjelman yksi suoritusaskel etsii verkosta kohdan, jossa voidaan suorittaa laskentaa ja muuttaa verkkoa tästä kohdasta laskentasääntöjen määräämällä tavalla. Tätä toistetaan, kunnes verkosta ei enää löydy mahdollisuuksia suorittaa laskentaa.

Laskettavan kohdan valintaan eli *laskentastrategiaan* (*evaluation strategy*) on useita vaihtoehtoja [Pie02, s. 56]. Laskentastrategia vaikuttaa useimmiten lähinnä ohjelman tehokkuuteen, mutta se voi joissakin tilanteissa vaikuttaa myös ohjelman pysähtyvyyteen.

Esimerkiksi ahkera (*eager*) laskentastrategia, joka laskee funktion parametrit auki ennen funktion kutsumista, tekee turhaa työtä, jos funktio ei tarvitsekaan kyseistä parametria. Jos parametrin laskenta johtaa päättymättömään laskentaan, niin *laiska* (*lazy*) laskentastrategia mahdollistaa ohjelman pysähtymisen, koska se jättää parametrin laskematta kunnes sen arvoa todella tarvitaan funktion rungossa.

Haskell on suosittu laiska funktionaalinen kieli, ja esimerkiksi ML ja Scala ovat suosittuja ahkeria funktionaalisia kieliä.

Laskentastrategialla ei ole jatkon kannalta merkitystä, mutta koska jatkossa aiotaan tulkita imperatiivisia ohjelmia funktionaalisina ohjelmina, valitaan selkeyden vuoksi ahkera laskentastrategia, joka laskee solmun aliverkosta kaiken mahdollisen ennen solmun itsensä laskemista. Tämä strategia on nimeltään *call-by-value* [Pie02, s. 56], koska se välittää aina funktioiden parametreiksi valmiiksi laskettuja arvoja eikä keskeneräisiä lausekkeita.

Määritelmä 2. Call-by-value-laskentastrategia

- Solmu on *arvo*, jos se on `Lambda`-solmu tai sen aliverkko koostuu vain vakio- ja monikkosolmuista.
- Call-by-value-strategian mukainen seuraava laskettava solmu on ensimmäinen syvyyssuuntaisella haulla löydettävä solmu, johon kulkevalla polulla ei ole arvoa, `Case`-solmua eikä `If`-solmua ja jonka kaikki alisolmut ovat arvoja.
- Jos verkon juuri on arvo, niin laskenta on valmis.
- Jos laskettavaa solmua ei löydy, niin ohjelma on virheellinen. □

Kielen laskentasäännöt määrittelevät, miten laskettavaksi valittu solmu korvataan laskennan tuloksella. Solmun A korvaaminen solmulla B tarkoittaa, että kaikki sol-

muun A osoittavat kaaret vaihdetaan osoittamaan solmuun B, ja solmut, joihin ei enää ole polkua juuresta, poistetaan. Esimerkkikielen laskentasäännöt ovat seuraavat.

Määritelmä 3. Yksinkertaisen funktionaalisen kielen laskentasäännöt

Olkoon A suoritettavaksi valittu solmu.

1. Jos $A = \texttt{If}(\textbf{true}, B, C)$, niin korvataan A solmulla B.

2. Jos $A = \texttt{If}(\textbf{false}, B, C)$, niin korvataan A solmulla C.

3. Jos $A = \texttt{Match}(B, \texttt{Case}(P_1, E_1), \ldots, \texttt{Case}(P_n, E_n))$ ja B sopii johonkin hahmoon P_i, niin valitaan pienin tällainen i ja korvataan A solmulla E_i.

4. Jos $A = \texttt{Apply}(\texttt{Lambda}(\texttt{ParamList}(\texttt{Var[}v_1\texttt{]}, \ldots, \texttt{Var[}v_n\texttt{]}), C, B), A_1, \ldots, A_n)$, niin korvataan A solmulla B', joka saadaan seuraavasti. B' on sellaisen verkon juuri, joka on B:stä alkavan aliverkon kopio, missä kukin solmu $\texttt{Var[}v_i\texttt{]}$ on korvattu A_i:llä ja mahdolliset viittaukset A:n kopioon viittaavat alkuperäiseen solmuun A.

   Selitys: *Funktiokutsussa funktion runko kopioidaan ja parametrimuuttujat korvataan annetuilla todellisilla parametreilla. Jos funktion runko viittaa rekursiivisesti itseensä, niin suoraviivainen kopiointi kuitenkin korvaisi parametrit pysyvästi myös rekursiivisesti kutsuttavan funktion rungosta, joten rekursiivista viittausta ei saa kopioida.* □

Kielen puhtaus voidaan todeta huomaamalla, että laskentasäännöillä ei ole mitään tapaa muuttaa kerran valmiiksi laskettua arvoa toiseksi. Tarkemmin sanottuna, yksikään arvoon viittaava kaari ei voi laskentasäännön seurauksena ohjautua osoittamaan muualle. `If`- ja `Match`-sääntöjen osalta asia on selvä, sillä vain `If`- ja `Match`-solmuihin viittaavia kaaria muutetaan. `Apply`-säännössä taas varovainen funktion rungon kopiointi pitää huolen siitä, ettei alkuperäinen funktio muutu, kun parametrit sijoitetaan runkoon.

Seuraavan luvun tavoite on määritellä imperatiivinen kieli, joka voidaan kääntää edellä määritellylle funktionaaliselle kielelle. Käännöksen onnistuminen osoittaa imperatiivisen kielen olevan myös puhdas.

3 Imperatiivisen ohjelman analysointi

Tässä luvussa palataan imperatiiviseen ajattelutapaan, joissa ohjelman suoritus nähdään peräkkäisten komentojen suorittamisena. Tavoite on pohjustaa Uniicin lähdekielen määritelmää sekä Uniicin lähdekielestä kohdekieleen kääntämisessä tarvittavia väliesitysmuotoja ja algoritmeja.

3.1 Yksinkertaisen imperatiivisen kielen määritelmä

Seuraava määritelmä esittää yksinkertaisen imperatiivisen kielen rakenteet listaamalla kielen abstraktin syntaksipuun mahdolliset solmut. Määritelmä laajennetaan luvussa 5.1 Uniicin lähdekieleksi.

Määritelmä 4. Yksinkertaisen imperatiivisen kielen rakenteet

Vakiot:

$$\begin{aligned} c \; \texttt{::=} \; & n \in \mathbb{Z} \\ & \mid \textbf{true} \\ & \mid \textbf{false} \end{aligned}$$

Lausekkeet:

$$\begin{aligned} E \; \texttt{::=} \; & \texttt{Const[}c\texttt{]}(), \text{ missä } c \text{ on vakio} \\ & \mid \texttt{Var[}v\texttt{]}(), \text{ missä } v \text{ on muuttuja} \\ & \mid \texttt{Call}(E, E_1, \ldots, E_n) \\ & \mid \texttt{MakeTuple(}E_1, \ldots E_n\texttt{)} \end{aligned}$$

Lauseet:

$$\begin{aligned} S \; \texttt{::=} \; & E \\ & \mid \texttt{Set}(\texttt{Var[}v\texttt{]}(), E) \\ & \mid \texttt{Return}(E) \\ & \mid \texttt{SplitTuple}(\texttt{Var[}v_1\texttt{]}, \ldots, \texttt{Var[}v_n\texttt{]}, E) \\ & \mid \texttt{If}(E_1, S_2, S_3) \\ & \mid \texttt{While}(E, S) \\ & \mid \texttt{Block}(S_1, \ldots, S_n) \end{aligned}$$

□

Kielen lauseet ja lausekkeet toimivat kuten useimmissa imperatiivisissa kielissä. Ainoastaan luvun 2.7 kaltaiset monikot ovat imperatiivisille kielille hieman epätyypillisiä. Lause `MakeTuple` luo monikon aivan kuten luvun 2.7 `Tuple`, ja `SplitTuple`-lause sijoittaa monikon kunkin arvon muuttujaan v_i.

Seuraavassa on esimerkkejä kielen lauseista ja niiden syntaksipuuesityksistä.

Esimerkki 10. Imperatiivisen kielen lauseita ja lausekkeita

```
{ a := 3; return a; }
```

$\Rightarrow$ `Block`(`Set`(`Var`[a], `Const`[3]), `Return`(`Var`[a]))

```
if a > 5 then {
  return 0;
} else {
  return 1;
}
```

$\Rightarrow$ `If`(`Call`(`Var`[>], `Var`[a](), `Const`[5]()), `Return`(`Const`[0]), `Return`(`Const`[0]))

```
t := (1, true, 3);
```

$\Rightarrow$ `Set`(`Var`[t](), `MakeTuple`(`Const`[1](), `Const`[true](), `Const`[3]()))

```
(a, b, c) := t;
```

$\Rightarrow$ `SplitTuple`(`Var`[a](), `Var`[b](), `Var`[c](), `Var`[t]()) □

Imperatiivisen kielen suoritussääntöjä ei määritellä tässa täsmällisesti, koska niitä ei tarvita jatkossa. Kielen suorituksen voi katsoa etenevän saman kaltaisesti kuin esimerkiksi Javassa.

3.2 Vuoverkko

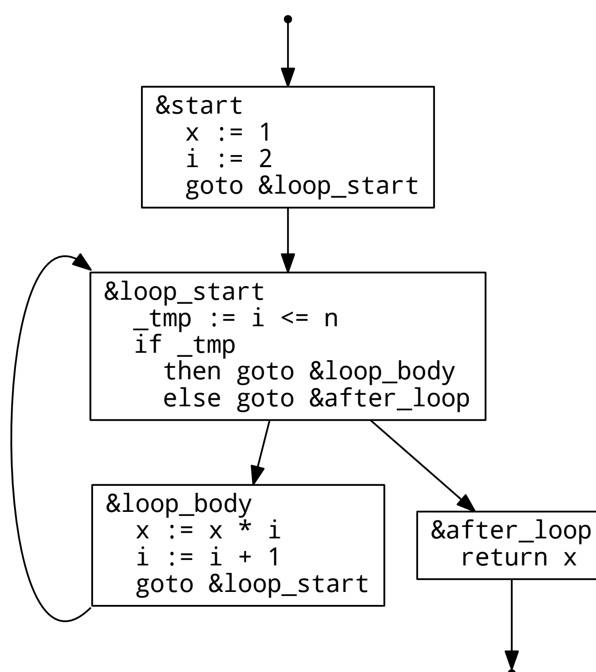
Imperatiivinen ohjelma on hyödyllistä kääntää *vuoverkoksi* (*flow graph*), koska vuoverkoille tehtävillä *vuoanalyyseilla* (*flow analysis*) voidaan tutkia ohjelman käyttäytymistä varsin monipuolisesti. Vuoverkot ovat myös luonteva ohjelman väliesitysmuoto imperatiivisten kielten kääntäjissä.

Vuoverkossa kaikki lausekkeet on jaettu välivaiheisiinsa siten, että jokainen vuoverkon komento tekee vain yhden yksinkertaisen operaation. Esimerkiksi lauseke `a := b * c + d` on jaettava komennoiksi `_tmp := b * c` ja `a := _tmp + d`.

Komennot on ryhmitelty nimettyihin *(perus)lohkoihin* (*basic block*). Lohkon viimeinen ja vain viimeinen komento on hyppykomento, joka joko vie suorituksen toiseen lohkoon tai palaa funktiosta. Hyppykomennon kohde voi riippua jonkin muuttujan arvosta. Lähdekielen ehto- ja silmukkarakenteet käännetään hyppykomennoiksi.

Seuraavassa esimerkissä näkyy vuokaavio, joka toteuttaa imperatiivisen kertomafunktion (ks. esimerkki 1). Ensimmäinen lohko `start` alustaa muuttujat ja hyppää lohkoon `loop_start`, jonka tehtävä on tarkistaa silmukan toistoehto. Jos ehto on tosi, hypätään silmukan runkoon `loop_body`, joka puolestaan hyppää aina takaisin `loop_start`:iin. Jos toistoehto on epätosi, hypätään lohkoon, joka palauttaa funktion paluuarvon.

Esimerkki 11. Imperatiivisen kertomafunktion vuoverkko



□

Määritellään seuraavaksi esimerkkikielen vuoverkot täsmällisesti. Vuoverkossa peruslohkot koostuvat peruskomennoista ja yhdestä hyppykomennosta, joka määrää lohkosta lähtevät kaaret. Määritelmää täydennetään hieman luvussa 3.6 SSA-muotoa varten, sekä luvussa 5.1 Uniicin lainausta varten.

Määritelmä 5. Vuoverkko

1. *Otsake* (*label*) on `&`-merkillä alkava nimi.

2. Seuraavat komennot ovat *peruskomentoja.*

   (a) v `:=` c *(vakion asetus)*

   (b) v `:=` v' *(muuttujan kopiointi)*

   (c) v `:=` v_f`(`v_1`, ..., `v_n`)` *(funktiokutsu)*

   (d) v `:=` `(`v_1`, ..., `v_n`)` *(monikon luominen)*

   (e) `(`v_1`, ..., `v_n`)` `:=` v *(monikon purkaminen)*

3. Seuraavat komennot ovat *hyppykomentoja.*

   (a) `return` v *(arvon palautus)*

   (b) `goto` L, missä L on otsake *(ehdoton hyppy)*

   (c) `if` v `then goto` L_1 `else goto` L_2, missä L_1 ja L_2 ovat otsakkeita *(ehdollinen hyppy)*

4. *Peruslohko* on otsakkeella varustettu lista komentoja, joista viimeinen on hyppykomento, ja muut ovat peruskomentoja.

5. *Vuoverkko* on joukko yksikäsittellä otsakkeella nimettyjä peruslohkoja, joista täsmälleen yksi on valittu *alkulohkoksi.*

6. Jokaisen otsakkeen, joka on mainittu peruslohkon B hyppykomennossa, tulee nimetä vuoverkossa jokin peruslohko C. Tällöin sanotaan, että C on B:n *seuraaja* ja B on C:n *edeltäjä.*

7. Alkulohkoa lukuun ottamatta jokaisen vuoverkon peruslohkon täytyy olla jonkin peruslohkon seuraaja. □

Vuoverkkojen komennot ovat niin yksinkertaisia, että ne on yleensä suoraviivaista muuttaa konekielelle, koska yksi komento vastaa yleensä yhtä konekielistä komentoa tai selkeää komentosarjaa. Imperatiivisten kielten kääntäjillä on yleensä jokin koodin väliesitysmuoto, joka muistuttaa tässä määriteltyjä komentoja [LLV13, GCC13].

Abstraktin syntaksipuun muuntamisesta vuoverkoksi kertoo muun muassa [ALS07, s. 357]. Uniicin käyttämä algoritmi selitetään yksityiskohtaisesti liitteessä 1. Seuraavaksi näytetään yleinen vuoverkkojen analysointimenetelmä, jota käytetään vuoverkon oikeellisuustarkistuksissa sekä SSA-muodon laskennassa.

3.3 Vuoanalyysit

Monet vuoverkolle tehtävät algoritmit voidaan muotoilla *vuoanalyyseiksi* (*dataflow analysis*) [ALS07, s. 597]. Vuoanalyysi liittää kunkin peruslohkon alkuun ja loppuun tiedon jostain kääntäjälle mielenkiintoisesta asiasta.

Algoritmi 2 esittää vuoanalyysin, joka selvittää kullekin ohjelman kohdalle, missä kohdissa muuttujaan v saattaa olla viimeksi kirjoitettu. Analyysi laskee siis jokaiseen ohjelman kohtaan *näkyvät kirjoituspaikat* (*reaching definitions*) [ALS07, s. 601].

Muuttujan x *kirjoituspaikka* on kolmikko (x, B, i), missä B on lohko ja i on kokonaisluku, joka tarkoittaa indeksissä i olevaa B:n komentoa. *Näkyvä kirjoituspaikka* tarkoittaa kirjoituspaikkaa, joka saattoi olla ohjelman suorituspolulla edellinen paikka, jossa x sai uuden arvon.

Algoritmi laskee jokaiselle lohkolle B joukon $In(B)$ ja joukon $Out(B)$ muuttujien kirjoituspaikkoja. Joukko $In(B)$ sisältää kaikki B:n alussa näkyvät kirjoituspaikat, ja joukko $Out(B)$ sisältää kaikki B:n jälkeen näkyvät kirjoituspaikat.

$Out(B)$-joukot lasketaan seuraavalla säännöllä. Jos B sisältää x:n kirjoittavan komennon indeksissä i, niin $In(B)$:ssä olleet x:n kirjoituspaikat korvataan $Out(B)$:ssä kirjoituspaikalla (x, B, i).

Ohjelman suoritus saattaa kulkeutua B:hen mistä tahansa B:n edeltäjästä P_i, joten kaikki P_i:n lopussa näkyvät kirjoituspaikat näkyvät myös B:n alussa. $In(B)$-joukko on siis kaikkien B:n edeltäjien P_i joukkojen $Out(P_i)$ yhdiste. Intuitiivisesti voidaan ajatella, että kirjoituspaikat valuvat vuoverkossa ylhäältä alas ohjelman mahdollisia suorituspolkuja pitkin.

Algoritmi soveltaa sääntöjä $In(B)$- ja $Out(B)$-joukkojen laskemiseksi, niin kauan kuin säännöt muuttavat jotain joukkoa. Algoritmin pysähtyminen voidaan nähdä huomaamalla, ettei mikään $In(B)$ tai $Out(B)$ koskaan pienene ja että mahdollisia kirjoituskohtia on ohjelmassa aina äärellinen määrä.

Algoritmi 2. Näkyvät kirjoituspaikat

```
ReachingDefs(vuoverkko G):
  B1 = G:n alkulohko
  In(B1) = ∅
  jokaiselle G:n lohkolle B:
    Out(B) = ∅
```

```
toista, kunnes In ja Out eivät enää muutu:
  jokaiselle G:n lohkolle B:
    Preds = kaikki B:n edeltäjät
    In(B) = ⋃P∈Preds Out(P)

    Out(B) = In(B)
    jokaiselle B:n komennolle c indeksissä i:
      jokaiselle c:n kirjoittamalle muuttujalle x:
        Out(B) = Out(B) \ {(x, B', i') kaikilla (B', i')}
        Out(B) = Out(B) ∪ {(x, (B, i))}

palauta joukot In ja Out
```

□

Näkyvien kirjoituspaikkojen analyysiä voidaan käyttää myös löytämään komennot, jotka lukevat muuttujan, jota ei välttämättä ole alustettu. Tämä tehdään alustamalla In(`B1`) tyhjän joukon sijaan joukolla, jossa on valekirjoituspaikka `(x, B1, -1)` jokaiselle muuttujalle `x` [ALS07, s. 602]. Jos valekirjoituspaikka näkyy johonkin `x`:n lukevaan komentoon, niin kyseinen komento saattaa käyttää `x`:ää alustamattomana.

3.4 Vuoanalyysirunko

Vuoanalyyseille voidaan määritellä yleinen runko [ALS07, s. 618], joka soveltuu useiden vuoverkolle suoritettavien algoritmien pohjaksi. Runko on hyödyllinen sekä teoriatasolla ajattelun välineenä että toteutustasolla apukirjastona.

Algoritmi 3 toteuttaa vuoanalyysirungon. Runko voidaan nähdä näkyvien kirjoituspaikkojen analyysialgoritmin yleistyksenä, jossa on abstrahoitu seuraavat asiat:

1. $In(B)$- ja $Out(B)$-joukkojen arvojen tyyppi
2. joukon In(`B1`) alkuarvo alkulohkolle `B1` (parametri `In1Init`)
3. joukon $Out(B)$ alkuarvo jokaiselle B (parametri `OutInit`)
4. yhdistämisfunktio, joka yhdistää B:n edeltäjälohkojen loppuarvot $Out(P_i)$ uudeksi alkuarvoksi $In(B)$ (parametri `meet`)

5. siirtymäfunktio, joka laskee $In(B)$:stä $Out(B)$:n (parametri `transfer`)

Algoritmi 3. Vuoanalyysirunko

```
FlowAnalysis(vuoverkko G, In1Init, OutInit, meet, transfer):
  B1 = G:n alkulohko
  In(B1) = In1Init
  jokaiselle G:n lohkolle B:
    Out(B) = OutInit

  toista, kunnes In ja Out eivät enää muutu:
    jokaiselle G:n lohkolle B:
      Preds = kaikki B:n edeltäjät
      In(B) = meet({Out(P) | P ∈ Preds})
      Out(B) = transfer(B, In(B))

  palauta In ja Out
```

□

Nyt näkyvien kirjoituspaikkojen analyysi voidaan muotoilla uudelleen vuoanalyysirungon avulla seuraavasti.

Algoritmi 4. Näkyvät kirjoituspaikat vuoanalyysirungon avulla

Vuoanalyysirunko parametrisoidaan seuraavasti.

1. $In(B)$ ja $Out(B)$ ovat kirjoituspaikkakolmikkojen (x, B, i) joukkoja.
2. `In1Init = ∅`
3. `OutInit = ∅`
4. `meet(Out1, ..., OutN) = Out1 ∪ ... ∪ OutN`
5. `transfer(B, in)` määritelty kuten $Out(B)$:n laskenta algoritmissa 2. □

Tässä esitetty vuoanalyysirunko kulkee ohjelman suorituspolun suuntaisesti ”eteenpäin”. Vastaavan vuoanalyysirungon voi rakentaa kulkemaan ”taaksepäin”. Jatkossa tarvitaan myös taaksepäin kulkevaa vuoanalyysirunkoa.

Intuitiivisesti taaksepäin kulkeva runko on kaikilta osin eteenpäin kulkevan rungon peilikuva. Siirtymäfunktio `transfer` laskee $Out(B)$:stä $In(B)$:n ja yhdistämisfunktio laskee lohkon seuraajien In-joukoista lohkolle Out-joukon.

Vielä ei kuitenkaan ole määritelty mitään, mikä voisi vastata taaksepäin kulkevassa analyysissa eteenpäin kulkevan analyysin alkulohkoa. Olkoon vuoverkon *lopetuslohko* erityinen lohko, jonka edeltäjiksi katsotaan kaikki `return`-komentoon päättyvät lohkot.

Taaksepäin kulkeva vuoanalyysi on nyt helppo määritellä.

Algoritmi 5. Taaksepäin kulkeva vuoanalyysirunko

```
BackwardsFlowAnalysis(vuoverkko G, OutRInit, InInit, meet, transfer):
  Br = G:n lopetuslohko
  Out(Br) = OutRInit
  jokaiselle G:n lohkolle B:
    In(B) = InInit

  toista, kunnes In ja Out eivät enää muutu:
    jokaiselle G:n lohkolle B:
      Succs = kaikki B:n seuraajat
      Out(B) = meet({In(S) | S ∈ Succs})
      In(B) = transfer(B, Out(B))

  palauta In ja Out
```

□

3.5 Dominointipuu

Jos kaikki polut alkulohkosta lohkoon B kulkevat lohkon A kautta, niin sanotaan, että A *dominoi* B:tä. Dominointirelaatio on refleksiivinen ja transitiivinen: jokainen lohko dominoi itseään, ja jos A dominoi B:tä ja B dominoi C:tä, niin A dominoi myös C:tä.

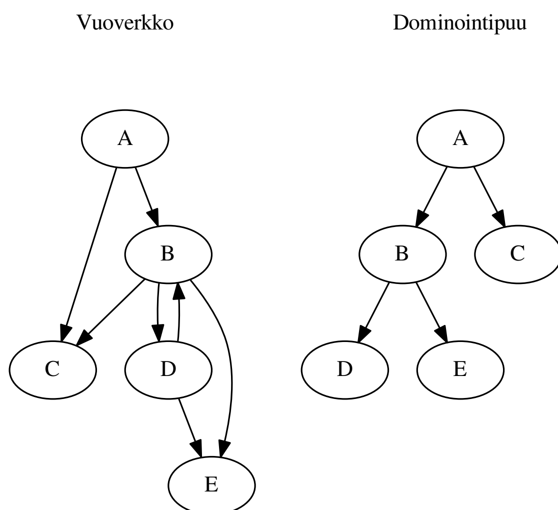
Jatkossa tarvitaan dominointisuhteesta myös ei-transitiivista versiota. Sanotaan, että lohko A dominoi lohkoa B *suoraan*, jos A dominoi B:tä eikä mikään $C \notin \{A, B\}$ dominoi B:tä dominoimatta myös A:ta.

Jokaisella lohkolla alkulohkoa lukuun ottamatta on yksikäsitteinen suora dominoija. Tämä voidaan näyttää esimerkiksi seuraavasti. Olkoon lohkolla A (ainakin) dominoijat B ja C. Kaikki polut A:han kulkevat siis aina sekä B:n että C:n kautta. Tämä on mahdollista vain, jos B dominoi C:tä tai päinvastoin, joten vain toinen B:stä ja C:stä voi olla A:n suora dominoija.

Lohkojen suora dominointisuhde määrittää verkolle yksikäsitteisen *dominointipuun*, jossa lohkosta A on kaari lohkoon B joss A dominoi B:tä suoraan. Dominointipuuta käytetään luvussa 3.6 SSA-muodon laskemiseen.

Seuraava esimerkki näyttää erään vuoverkon ja sen dominointipuun.

Esimerkki 12. Vuoverkko ja dominointipuu



□

Esimerkistä voi tehdä muun muassa seuraavat havainnot.

- Alkulohko A on myös dominointipuun juuri, sillä kaikki polut juuresta mihin tahansa lohkoon kulkevat triviaalisti juuren läpi.
- B on D:n suora dominoija, koska D:n ainoa muu dominoija on A, joka on myös B:n dominoija.

- B ei voi dominoida C:tä, koska sinne pääsee suoraan A:sta käymättä B:ssä.
- Vastaavasti D ei voi dominoida E:tä, koska sinne pääsee suoraan B:stä käymättä D:ssä.

Dominointipuun rakentamiseksi täytyy laskea jokaiselle lohkolle kaikkien sen dominoijien joukko. Lohkojen dominoijien joukot voidaan laskea seuraavalla eteenpäin kulkevalla vuoanalyysilla [ALS07, s. 656]. Vuoanalyysin pysähdyttyä kunkin lohkon B dominoijien joukko voidaan lukea analyysin laskemasta lohkon lopun arvosta $Out(B)$. Analyysin pysähtyminen voidaan todeta huomaamalla, että $Out(B)$-joukot eivät voi kasvaa.

Algoritmi 6. Lohkojen dominoijat

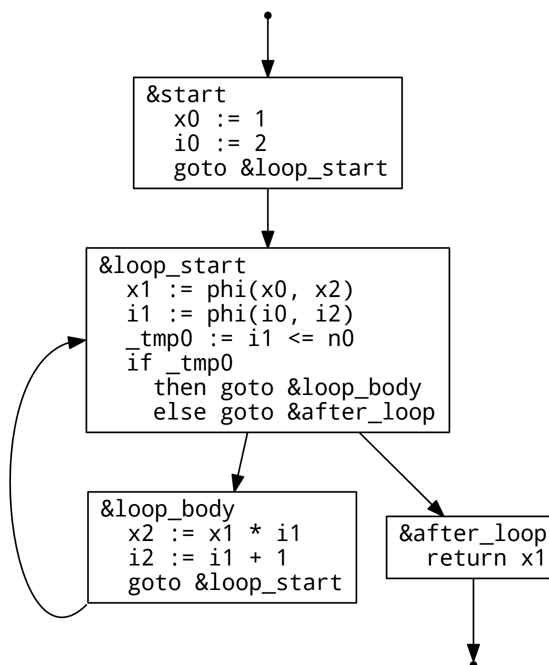
1. Joukot $In(B)$ ja $Out(B)$ sisältävät lohkojen joukkoja.
2. `In1Init = ∅`
3. `OutInit` on vuoverkon kaikkien lohkojen joukko
4. `meet(Out1, ..., OutN) = Out1 ∩ ... ∩ OutN` on joukkojen leikkaus (sillä kahden lohkon A ja B yhteistä seuraajaa dominoivat vain ne lohkot, jotka dominoivat sekä A:ta että B:tä).
5. `transfer(B, in) = in ∪ {B}` (sillä lohko dominoi aina itseään). □

Lohkon A suora dominoija löydetään etsimällä vuoverkon edeltäjiä seuraamalla lähin lohko, joka on A:n dominoija. Dominointipuu rakennetaan asettamalla kunkin lohkon vanhemmaksi lohkon suora dominoija.

3.6 SSA-muoto

Vuoverkko on *SSA-muodossa* (*Single Static Assignment form*), jos jokaiseen muuttujaan sijoitetaan vain kerran. Vuoverkko muunnetaan SSA-muotoon versionumeroimalla muuttujat. Seuraava esimerkki näyttää esimerkin 11 vuoverkon SSA-muodon.

Esimerkki 13. Imperatiivisen kertomafunktion SSA-vuoverkko



Huomaa esimerkissä olevat `phi`-komennot. Lause `x1 := phi(x0, x2)` sijoittaa muuttujaa `x1` joko muuttujan `x0` tai muuttujan `x2` arvon riippuen siitä, kumpaa kaarta pitkin ohjelman suoritus kulki.

Muunnos SSA-muotoon alkaa antamalla jokaiselle sijoitetulle muuttujalle uniikki versionumero. Tämän jälkeen kaikki muuttujia lukevat komennot lohkossa muutetaan lukemaan muuttujan viimeisintä versiota. Muuttujan viimeisimmän version sijoittava komento löydetään laskemalla lukukohtaan näkyvät kirjoituspaikat (algoritmi 2). Jos muuttujasta löytyy enemmän kuin yksi viimeisin versio, täytyy sitä lukevan lohkon alkuun lisätä `phi`-komento.

Olisi helppoa lisätä jokaisen lohkon alkuun yksi `phi`-komento jokaista sitä tarvitsevaa muuttujaa kohden. Dominointipuun avulla voidaan kuitenkin lisätä vain välttämättömät `phi`-komennot [App98].

Dominointipuusta voidaan laskea kunkin lohkon A *dominointirintama* (*dominance frontier*), joka tarkoittaa lohkojen joukkoa, jota A ei dominoi mutta jota dominoi

suoraan jokin A:n dominoima lohko. Dominointirintama on siis A:n dominoimien lohkojen ympärillä oleva ”raja”. Dominointirintamalla olevat lohkot ovat toisin sanoen ensimmäisiä lohkoja, joihin suoritus voi päästä kulkematta A:n kautta.

Olkoon B jokin A:n dominointirintamassa oleva lohko. Jos lohkossa A sijoitetaan muuttuja x ja muuttujaa x luetaan joko B:ssä tai jossain B:n seuraajassa, niin lohkon B alkuun lisätään x:n sijoittava `phi`-komento.

Tätä menetelmää toistetaan, niin kauan kuin se löytää uusia kohtia `phi`-komennoille. Komentojen lisäämisen jälkeen täytyy muistaa päivittää myös muuttujia lukevat komennot lukemaan uusien `phi`-komentojen sijoittamia muuttujia siellä missä tarpeen.

`phi`-komennot voi aluksi lisätä tyhjinä ($\texttt{x}_\texttt{i}$ `:= phi()`). Kun kaikki tarvittavat `phi`-komennot on lisätty, voidaan ne täydentää lukemaan lohkon edeltäjien lopussa voimassa olevat muuttujat.

Uniic muunnetaan funktionaaliseen muotoon SSA-muodon kautta luvussa 5.3. SSA-muodon `phi`-komennot vastaavat suoraviivaisesti funktionaaliseen muotoon syntyvien funktioiden parametrilistoja. Luvussa 5.1.3 esitetty Uniicin lainauksen poisto tehdään myös SSA-muotoiselle ohjelmalle, koska lainauksen tarkka käyttäytyminen voidaan esittää selkeästi versioiduilla muuttujilla.

4 Tyyppijärjestelmät ja uniikkityypitys

Uniicissa sekä lähde- että kohdekieli tyypitetään Hindley–Milner -tyypinpäättelijällä. Tyyppijärjestelmä esitetään ensin ilman uniikkityypitystä sellaisena kuin se esiintyy yleensä kirjallisuudessa. Tämän jälkeen esitellään de Vriesin uniikkityypitysjärjestelmä, joka on Hindley–Milnerin melko suoraviivainen laajennos.

Uniikkityypittämätön tyyppijärjestelmä tyypittää Uniicin kohdekielen kaltaista funktionaalista kieltä. Kielen arvot ovat joko kokonaislukuja, totuusarvoja, monikkoja, yksikköarvoja tai funktioita. Käytössä olevien (uniikkityypittämättömien) tyyppien joukko voidaan siis määritellä seuraavasti.

Määritelmä 6. Tavalliset tyypit

1. Primitiivityypit `Int`, `Bool` ja `Unit`.

2. Monikkotyypit $(T_1, \ldots, T_n)$, missä $n \geq 2$ ja $T_1, \ldots, T_n$ ovat tyyppejä.

3. Funktiotyypit $(T_1, \ldots, T_n) \to T_r$, missä $T_1, \ldots, T_n$ ja T_r ovat tyyppejä. □

Monissa funktionaalisissa kielissä sallitaan vain yhden parametrin funktiot. Näissä kielissä monta parametria tarvitseva funktio voidaan kirjoittaa vaatimaan monikko tai vaihtoehtoisesti ottamaan vain ensimmäisen parametrinsa ja palauttamaan uusi funktio, joka odottaa toista parametria jne.

Uniicissa toteutetaan lainaus kuitenkin sellaisella tavalla, että monipaikkaiset funktiot ovat kielelle sopivampi valinta. Näin ollen funktiotyyppi rakennetaan listasta parametrityyppejä ja yhdestä paluutyypistä.

4.1 Hindley–Milner-tyypinpäättely

Hindley–Milner-tyypinpäättely on suosittu tyypinpäättelymenetelmä funktionaalisissa ohjelmointikielissä [Pie02, s. 326]. Menetelmän vahvuuksiin kuuluu sen yksinkertaisuus ja se, että ohjelmoijan on hyvin harvoin pakko määritellä tyyppejä itse. Toisaalta menetelmän eräs heikkous on virheviestien vaikeaselkoisuus.

Menetelmä toimii kolmessa vaiheessa:

1. Annetaan jokaiselle ohjelman solmulle tyyppi. Jos tyyppiä ei tiedetä suoraan, tyypiksi asetetaan aluksi uusi tyyppimuuttuja.

2. Tuotetaan ohjelmasta joukko tyyppiyhtälöitä.

3. Yritetään ratkaista tyyppiyhtälöt.

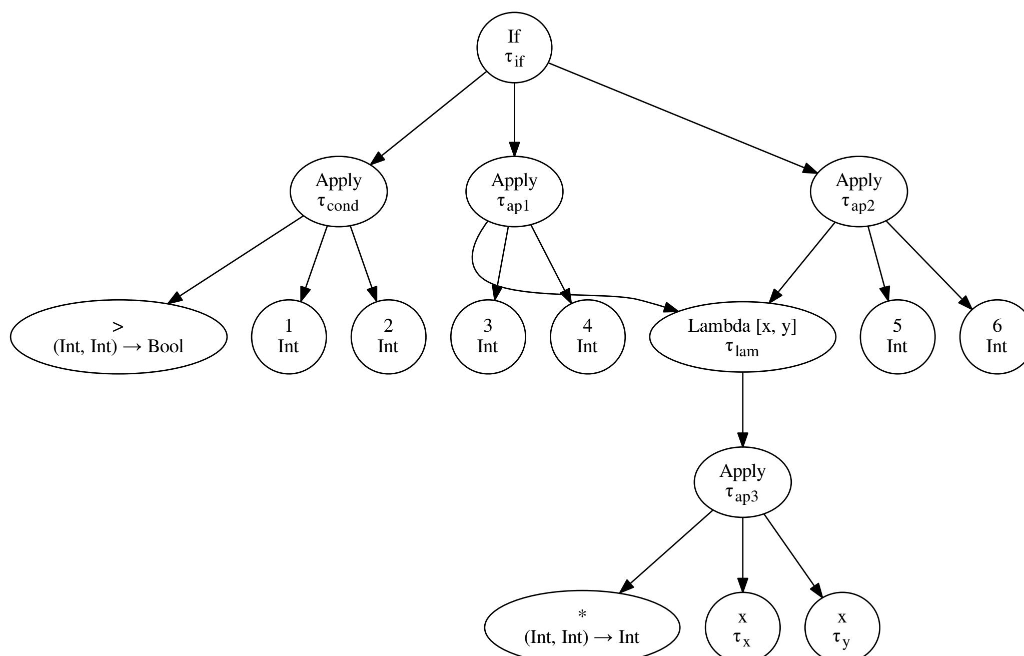
Tarkastellaan tässä luvussa seuraavaa funktionaalista lauseketta.

Esimerkki 14

```
let f = \(x, y) -> x * y
in if 1 > 2 then f(3, 4) else f(5, 6)
```

□

Lauseketta vastaava verkko on seuraava. Verkkoon on merkitty näkyviin kullekin solmulle annettu tyyppi tai tyyppimuuttuja. Esimerkiksi kertomerkkifunktiolla `*` on tunnettu tyyppi (`Int`, `Int`) $\rightarrow$ `Int`, mutta funktion `f` tyyppi halutaan päätellä, joten se saa tyypikseen tyyppimuuttujan τ_{lam}. Vastaavasti funktion `f` sovellusten (Apply-solmut) tulosten tyypit ovat tuntemattomat τ_{ap1} ja τ_{ap2}. Tyyppimuuttujat on nimetty mielivaltaisesti.

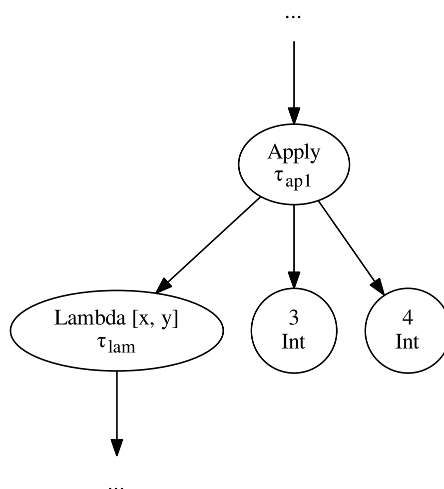


Tyypinpäättelyalgoritmin tehtävä on löytää tyyppimuuttujille sellaiset arvot, jotka noudattavat kielen tyypityssääntöjä. Algoritmin pitäisi siis pystyä muun muassa

päättelemään, että $\tau_{\text{if}} = \texttt{Int}$ ja että $\tau_{\text{lam}} = (\texttt{Int}, \texttt{Int}) \rightarrow \texttt{Int}$. Hindley–Milner-tyypinpäättelyssä tyyppimuuttujien arvot löydetään ratkaisemalla tyypityssääntöjen mukaisesti saatu joukko tyyppiyhtälöitä.

4.1.1 Tyyppiyhtälöt

Tarkastellaan esimerkin 14 funktiokutsua `f(3, 4)`, jonka verkkoesitys on seuraava.



Funktiokutsujen oikeellinen tyypitys vaatii, että kutsuttu arvo on tyypiltään funktio, jonka ottamien parametrien tyypit vastaavat annettujen parametrien tyyppejä. Tästä säännöstä voidaan johtaa tämän esimerkin kohdalla seuraava tyyppiyhtälö.

$$\tau_{\text{lam}} \approx (\texttt{Int}, \texttt{Int}) \rightarrow \tau_{\text{ap1}}$$

Yhtälö ilmaisee, että funktion on hyväksyttävä annetuntyyppiset parametrit $(\texttt{Int}, \texttt{Int})$ ja palautettava kutsukohdassa vaadittua tyyppiä τ_{ap1} oleva arvo. Paluuarvon tyypin on oltava sama kuin mitä `Apply`-solmuun osoittavat verkon osat vaativat.

Toisesta samanlaisesta funktiokutsusta `f(5, 6)` syntyy samankaltainen yhtälö.

$$\tau_{\text{lam}} \approx (\texttt{Int}, \texttt{Int}) \rightarrow \tau_{\text{ap2}}$$

Yhtälönratkaisija, jonka yksityiskohtiin palataan alempana, voi nyt eliminoida muuttujan τ_{lam} ja saada tulokseksi seuraavan yhtälön.

$$(\texttt{Int}, \texttt{Int}) \to \tau_{\text{ap1}} \approx (\texttt{Int}, \texttt{Int}) \to \tau_{\text{ap2}}$$

Tämän uuden yhtälön molemmat puolet ovat rakenteellisesti samanlaisia, sillä molemmat ovat kaksipaikkaisia funktioita. Yhtälönratkaisija voi näin ollen päätellä, että yhtälö on voimassa, joss molempien puolten parametrityypit ovat samoja ja molempien puolten paluutyypit ovat samoja. Yhtälö voidaan siis jakaa kahdeksi pienemmäksi yhtälöksi seuraavasti.

$$\begin{aligned}(\texttt{Int}, \texttt{Int}) &\approx (\texttt{Int}, \texttt{Int})\\ \tau_{\text{ap1}} &\approx \tau_{\text{ap2}}\end{aligned}$$

Näistä ensimmäinen yhtälö on valmiiksi ratkaistu, sillä sen molemmat puolet ovat samat. Toiseen voidaan soveltaa jälleen muuttujan eliminointia: kaikki τ_{ap1}:n esiintymät voidaan korvata τ_{ap2}:lla (tai päinvastoin). Tämän jälkeen toisenkin yhtälön molemmat puolet ovat samat ($\tau_{\text{ap2}} \approx \tau_{\text{ap2}}$), joten toinenkin yhtälö on ratkaistu.

4.1.2 Tyyppiyhtälöiden tuottaminen

Seuraavassa määritellään tyypityssäännöt luvussa 2.7 määritellylle yksinkertaiselle funktionaaliselle kielelle. Säännöt ovat samanlaiset kuin Hindley–Milner -tyyppijärjestelmissä yleensä [Pie02, s. 322]. Luvussa 4.2 säännöt täydennetään uniikkityyppityssäännöiksi.

Tyypityssäännöt antavat ensin jokaiselle solmulle E tyypin $\Gamma(E)$. Useimpien solmujen tyypiksi asetetaan *tuore tyyppimuuttuja* eli tyyppimuuttuja, jota ei käytetä missään muualla kyseisen tyypintarkistuksen aikana.

Määritelmä 7. Funktionaalisen kielen tyyppien alustus

1. Jos n on kokonaisluku, niin $\Gamma(\texttt{Const[}n\texttt{]}) = \texttt{Int}$.
2. $\Gamma(\texttt{true}) = \Gamma(\texttt{false}) = \texttt{Bool}$.
3. Jos x on symbolitaulussa, niin $\Gamma(\texttt{Var[}x\texttt{]})$ on symbolitaulusta luettu tyyppi.
4. Muissa tapauksissa $\Gamma(E)$ on tuore tyyppimuuttuja. □

Kun jokaiselle solmulle on näin annettu tyyppi, voidaan tyyppiyhtälöt tuottaa soveltamalla jokaiselle solmulle seuraavia tyyppiyhtälösääntöjä.

Määritelmä 8. Funktionaalisen kielen tyyppiyhtälösäännöt

1. Solmu $E = \texttt{Apply}(F, E_1, E_2, \ldots, E_n)$ tuottaa seuraavan tyyppiyhtälön:

   (a) $\Gamma(F) \approx (\Gamma(E_1), \Gamma(E_2), \ldots, \Gamma(E_n)) \to \Gamma(E)$

   Selitys: *Yhtälö vaatii kutsuttavan funktiosolmun olevan funktio, jonka parametrien tyypit ovat samat kuin mitä funktiokutsussa annetaan. Funktiokutsun tyyppi on sama kuin funktion paluutyyppi.*

2. Solmu $E = \texttt{Lambda}(\texttt{ParamList}(x_1, x_2, \ldots, x_n), C, B)$ tuottaa seuraavan tyyppiyhtälön:

   (a) $\Gamma(E) \approx (\Gamma(x_1), \Gamma(x_2), \ldots, \Gamma(x_n)) \to \Gamma(B)$

   Selitys: *Yhtälö vaatii solmun tyypin olevan funktiotyyppi parametrien tyypeiltä funktion rungon tyypille.*

3. Solmu $E = \texttt{If}(A, B, C)$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma(A) \approx \texttt{Bool}$

   (b) $\Gamma(E) \approx \Gamma(B)$

   (c) $\Gamma(E) \approx \Gamma(C)$

   Selitys: *Ehdon tyypin on aina oltava totuusarvo, ja kummankin haaran tyypin on oltava keskenään samoja.* `If`*-solmun tyypiksi tulee haarojen tyyppi.*

4. Solmu $E = \texttt{Tuple}(E_1, E_2, \ldots, E_n)$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma(E) \approx (\Gamma(E_1), \Gamma(E_2), \ldots, \Gamma(E_n))$

   Selitys: *Monikon tyyppi on monikon alkioiden tyypeistä koostuva monikkotyyppi.*

5. Solmu $E = \texttt{Match}(E_0, \texttt{Case}(P_1, E_1), \texttt{Case}(P_2, E_2), \ldots, \texttt{Case}(P_n, E_n))$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma(E_0) \approx P_i$ jokaiselle $i \in \{1, \ldots, n\}$.

   (b) $\Gamma(E) \approx \Gamma(E_i)$ jokaiselle $i \in \{1, \ldots, n\}$

Selitys: *Tietorakenteiden hahmonsovituksessa jokaisen hahmon on oltava samaa tyyppiä kuin purettavan arvon E_0. Jokaisen haaran on tuotettava sama palautettava tyyppi, kuten yllä `If`-solmun tapauksessa.*

6. Muut solmut eivät tuota tyyppiyhtälöitä. □

Esimerkistä 14 tuotetaan näillä säännöillä seuraavat tyyppiyhtälöt. Kolme ensimmäistä yhtälöä syntyvät `If`-solmua koskevasta säännöstä. Neljäs yhtälö syntyy `Lambda`-solmusta ja viimeiset kolme yhtälöä puolestaan `Apply`-solmuista.

Esimerkki 15

$$\begin{aligned}
\tau_{\text{cond}} &\approx \texttt{Bool} \\
\tau_{\text{if}} &\approx \tau_{\text{ap1}} \\
\tau_{\text{if}} &\approx \tau_{\text{ap2}} \\
\tau_{\text{lam}} &\approx (\tau_x, \tau_y) \to \tau_{\text{ap3}} \\
(\texttt{Int}, \texttt{Int}) \to \texttt{Bool} &\approx (\texttt{Int}, \texttt{Int}) \to \tau_{\text{cond}} \\
\tau_{\text{lam}} &\approx (\texttt{Int}, \texttt{Int}) \to \tau_{\text{ap1}} \\
\tau_{\text{lam}} &\approx (\texttt{Int}, \texttt{Int}) \to \tau_{\text{ap2}} \\
(\texttt{Int}, \texttt{Int}) \to \texttt{Int} &\approx (\tau_x, \tau_y) \to \tau_{\text{ap3}}
\end{aligned}$$

□

4.1.3 Tyyppiyhtälöiden ratkaiseminen

Saatu yhtälöjoukko ratkaistaan soveltamalla siihen aiemmin nähtyjä sääntöjä: tyyppimuuttujan eliminointia ja rakenteellista purkamista. Järjestyksellä, jossa sääntöjä sovelletaan yhtälöihin, ei ole merkitystä, mutta jos kumpaakaan sääntöä ei voida enää soveltaa yhteenkään yhtälöön, on löydetty ristiriita ja ohjelmassa on tyyppivirhe. Yhtälönratkaisusäännöt voidaan määritellä seuraavasti.

Algoritmi 7. Tyyppiyhtälöiden ratkaiseminen

1. Yhtälö, joka on muotoa $\tau \approx T$ tai $T \approx \tau$, voidaan poistaa, jos jäljelle jäävissä yhtälöissä kaikki tyyppimuuttujan τ:n esiintymät korvataan tyypillä T:llä.

2. Yhtälö, jonka molemmat puolet ovat identtisiä, voidaan poistaa.

3. Yhtälö, joka on muotoa $(T_1, T_2, \ldots, T_n) \to T_r \approx (S_1, S_2, \ldots, S_n) \to S_r$, voidaan poistaa, jos yhtälöjoukkoon lisätään yhtälöt $T_1 \approx S_1$, $T_2 \approx S_2$, ..., $T_n \approx S_n$ ja $T_r \approx S_r$.

4. Yhtälö, joka on muotoa $(T_1, T_2, \ldots, T_n) \approx (S_1, S_2, \ldots, S_n)$, voidaan poistaa, jos yhtälöjoukkoon lisätään yhtälöt $T_1 \approx S_1$, $T_2 \approx S_2$, ..., $T_n \approx S_n$. □

Esimerkin 15 yhtälöt voitaisiin ratkaista näillä säännöillä monella tavalla. Aloittaa voi vaikka eliminoimalla ensin säännön 1 nojalla muuttujan τ_{cond} sijoittamalla $\tau_{\text{cond}} \approx \texttt{Bool}$. Tämä tuottaa yhtälön $(\texttt{Int}, \texttt{Int}) \to \texttt{Bool} \approx (\texttt{Int}, \texttt{Int}) \to \texttt{Bool}$, jonka voi saman tien poistaa säännön 2 nojalla. Seuraavaksi voi esimerkiksi soveltaa sääntöä 3 yhtälön $(\texttt{Int}, \texttt{Int}) \to \texttt{Int} \approx (\tau_x, \tau_y) \to \tau_{\text{ap3}}$ purkamiseksi yhtälöiksi $\tau_x \approx \texttt{Int}$, $\tau_y \approx \texttt{Int}$ ja $\tau_{\text{ap3}} \approx \texttt{Int}$.

4.1.4 Yleistetyt tyypit

Yhtälönratkaisu voi jättää joitakin tyyppimuuttujia eliminoimatta. Tämä tarkoittaa, että kyseinen kohta ohjelmasta toimii millä tahansa tyypillä. Esimerkiksi funktiolle `\(x, y). x` voidaan yllä kuvatulla menetelmällä päätellä yleistetty (*generic*) tyyppi $\forall t_1\ t_2.(t_1, t_2) \to t_1$, missä merkintä $\forall t_1\ t_2. \ldots$ tarkoittaa, että tyyppi toimii millä tahansa tyypeillä t_1 ja t_2. Tyyppimuuttujat t_1 ja t_2 ovat tässä yleistetyn tyypin *tyyppiparametreja*.

Yleistetyn tyypin päättely toimii ongelmitta, kun yleistettävä tyyppi päätellään kokonaan ennen kuin sitä käyttävää koodia yritetään tyypittää. Yllä kuvattu tyypinpäättelijä ei nimittäin osaa yleistää tyyppejä kesken tyypinpäättelyn. Tilannetta havainnollistaa seuraava esimerkkiohjelma.

```
let f = \(x, y). x
in (f(1, 2), f(3, false))
```

Ohjelma on järkevä, sillä funktion `f` pitäisi voida toimia sekä parametreilla `(1, 2)` että parametreilla `(3, false)`. Yllä esitetyn kaltainen yksinkertainen tyypinpäättelijä joutuu yhtälönratkaisussa kuitenkin seuraavanlaiseen tilanteeseen.

$$\tau_f \approx (Int, Int) \to Int$$
$$\tau_f \approx (Int, Bool) \to Int$$

Tästä seuraa ennen pitkää ratkeamaton yhtälö $Int \approx Bool$. Ongelma on, ettei tyypinpäättelijä tiedä, mitä muuttujia sen pitäisi yrittää yleistää kesken tyypinpäättelyn.

Ohjelmoija voi kiertää ongelman kahdella tavalla: joko kirjoittamalla `f`:n kaltaiset yleiset funktiot päätasolle, jolloin ne tyypitetään erikseen, tai määrittelemällä niille yleistetyn tyypin käsin.

Tyypinpäättelijää voidaan myös kehittää yleistämään `let`-rakenteella sidotut lausekkeet automaattisesti, jolloin saavutetaan *let-polymorfismiksi* kutsuttu ominaisuus [Pie02, s.331]. Ominaisuus kuitenkin vaikeuttaa tyypinpäättelijän kehittämistä muilla tavoilla, joten ei ole selvää, että se on hyvä ominaisuus [VPJ10]. Let-polymorfismia ei oteta tässä käyttöön, koska sitä ei jatkossa tarvita. Yleistetyt tyypit ovat tästä huolimatta Uniicin uniikkityypitysjärjestelmän kannalta tärkeitä, kuten luvussa 4.2 nähdään.

4.1.5 Yleistettyjen tyyppien käyttö

Vielä on määriteltävä, miten tyypinpäättelijä käyttäytyy, kun osalle solmuista on määrätty symbolitaulussa yleistetty tyyppi. Riittää täydentää merkinnän $\Gamma(E)$ käyttäytymistä seuraavasti.

Olkoon solmun E tyypiksi on asetettu yleistetty tyyppi $\forall t_1\ t_2\ \ldots\ t_n.\ T$. Joka kerta kun algoritmi tarvitsee $\Gamma(E)$:n arvoa, täytyy $\Gamma(E)$:n palauttaa T, jossa jokainen t_i korvataan uudella tuoreella muuttujalla[1].

Jos symbolitaulussa on tyyppi $\mathtt{f} : \forall t_1\ t_2.(t_1, t_2) \to t_1$, niin tyypinpäättelijä tuottaa nyt lausekkeesta `(f(1, 2), f(3, false))` seuraavat yhtälöt.

$$\tau_1 \approx (Int, Int) \to Int$$
$$\tau_2 \approx (Int, Bool) \to Int$$

Näistä yhtälöistä ei enää voi johtaa ratkeamatonta yhtälöä $Int \approx Bool$, sillä `f` sai ensimmäisellä käyttökerralla tyypin τ_1 ja toisella käyttökerralla eri tyypin τ_2. Lauseke on siis oikein tyypitetty.

[1] Algoritmin kuvaus muuttuu tässä luonteeltaan imperatiiviseksi, koska $\Gamma(E)$ ei ole enää puhdas funktio. Algoritmin voisi kuvata myös puhtaalla tyylillä, mutta kuvauksesta tulisi tällöin vaikeaselkoisempi, eikä puhtaasta tyylistä ole tässä tapauksessa hyötyä.

4.2 Uniikkityypitys

Tässä luvussa esitellään uniikkityypitys kuvailemalla de Vriesin kehittämän uniikkityypitysjärjestelmän yksinkertainen versio [deV08, s. 129], joka puolestaan on jatkokehitetty [deV08, s. 78] Cleanin uniikkityypitysjärjestelmästä [BaS93]. De Vries määrittelee järjestelmänsä syntaksipuiden suhteen, muttei määrittele niille viitelaskurialgoritmia. Tässä luvussa otetaan pohjaksi syntaksipuiden sijaan funktionaaliset verkot (luku 2.8) mukaillen tapaa, jolla Cleanin järjestelmä on kuvailtu. Näin voidaan luontevasti käyttää Cleanin viitelaskuria, joka toimii vain funktionaalisille verkoille.

4.2.1 Uniikkityypitysjärjestelmien perusteet

Uniikkityypityksessä jokaiseen tavalliseen tyyppiin (eli *perustyyppiin*) liitetään *uniikkiusattribuutti* (*uniqueness attribute*) [BaS93]. Uniikkiusattribuutti voi olla arvoltaan joko uniikki ($\bullet$) tai ei-uniikki ($\times$). Esimerkiksi $\texttt{IntArray}^\bullet$ tarkoittaa uniikkia kokonaislukutaulukkoa, ja $(\texttt{Int}^\times, \texttt{Int}^\times)^\bullet$ tarkoittaa uniikkia paria, joka sisältää kaksi ei-uniikkia kokonaislukua. Ei-uniikki attribuutti saatetaan jättää joskus selkeyden vuoksi merkitsemättä, joten ei-uniikkien kokonaislukujen uniikki pari voidaan kirjoittaa myös $(\texttt{Int}, \texttt{Int})^\bullet$.

Uniikkityypitys perustuu *viitteiden laskentaan* (*reference counting*). Funktionaalisen ohjelman verkkoesityksessä solmun viitteiden määrä kertoo, saatetaanko solmua käyttää enemmän kuin kerran laskennassa. Uniikkityypityksen tarkoitus on taata, että jos arvon tyyppi on uniikki ($T^\bullet$) tai mahdollisesti uniikki ($\forall u.\ T^u$), niin viitelaskuri on merkinnyt arvon kertakäyttöiseksi [BaS93, s. 18]. Lisäksi jokaisen arvon tyypin, mukaan lukien uniikkiusattribuutin, täytyy säilyä jokaisen suoritusaskeleen jälkeen samana. Tämä ”tyypin säilyvyys” on itse asiassa tärkeä oikeellisuusominaisuus missä tahansa tyyppijärjestelmässä [Pie02, s. 95].

Luvussa 4.1.4 esitellyt yleistetyt tyypit ovat uniikkityypityksessä keskeisiä, sillä uniikkityypitetyssä kielessä ohjelmoija tulee kirjoittaneeksi paljon funktioita, jotka toimivat sekä uniikeilla että ei-uniikeilla parametreilla. Tällaisten funktioiden tyypittämisessä kannattaa käyttää uniikkiusattribuutteina tyyppimuuttujia, jottei funktiosta tarvitse kirjoittaa uniikkia ja ei-uniikkia versiota. Esimerkiksi tyyppiä $\forall u.\ T^u \to T^u$ oleva funktio palauttaa paluuarvonsa samalla uniikkiudella u kuin se saa parametrinsa, ja tyyppiä $\forall u.\ (T_1^u, T_2^u) \to S$ oleva funktio vaatii parametriensa olevan keskenään yhtä uniikkeja.

4.2.2 Viitteiden laskenta funktionaalisessa verkossa

Viitelaskuri on funktionaaliselle verkolle (luku 2.8) suoritettava algoritmi, joka merkitsee jokaisen solmun joko *kertakäyttöiseksi* tai *kierrätetyksi*. Viitelaskurin toteutuksen idea on tarkistaa, onko samaan solmuun kaksi tai useampaa syklitöntä polkua jostain muusta solmusta. Jos on, niin solmu merkitään kierrätetyksi, koska sitä saatetaan käyttää monta kertaa.

On kuitenkin tilanteita, joissa samaan solmuun voi olla monta polkua, mutta solmua käytetään kuitenkin vain kerran. Eräs tällainen tilanne on `If`-solmujen kohdalla. Tarkastellaan seuraavaa lauseketta, ja oletetaan, että `f` ja `g` tarvitsevat uniikin parametrin.

```
if p then f(x) else g(x)
```

Lausekkeen verkkoesityksessä on selvästi kaksi viittausta muuttujaan `x`. Kuitenkin `x` saa olla uniikki, koska vain toinen `if`-lausekkeen haaroista suoritetaan. Todetaan, että funktionaalisessa verkossa jotkut kaaret ovat *keskenään vaihtoehtoiset*.

Määritelmä 9. Vaihtoehtoiset kaaret

1. Jokaisen `If`-solmun kaaret 1 ja 2 ovat keskenään vaihtoehtoiset.
2. Jokaisen `Case`-solmun kaaret $n \geq 1$ ovat keskenään vaihtoehtoiset. □

Verkoissa on lisäksi *metatietokaaria*. Nämä ovat kaaria, joita suoritusympäristö ei käsittele arvoina, kuten useimpia kaaria, vaan suoritusta ohjaavana metatietona. Metatietokaaret eivät osallistu viitteiden laskentaan.

Määritelmä 10. Metatietokaaret

1. Jokaisen `Lambda`-solmun kaaret 1 ja 3 ovat metatietokaaria.
   Perustelu: *Ensimmäinen kaari viittaa parametrilistaan, ja kolmas kaari viittaa funktion runkoon, joka kopioidaan, kun funktio suoritetaan (suoritussäännöt, luku 2.9).*
2. Jokaisen `Case`-solmun ensimmäiset kaaret ovat metatietokaaria.
   Perustelu: *Kaari viittaa sovitettavaan hahmoon.* □

Nyt viitelaskurialgoritmille saadaan seuraava spesifikaatio, joka vastaa Barendsenin ja Smetsersin Cleania varten määrittelemää viitelaskurispesifikaatiota [BaS93, s. 23]. Spesifikaatio jättää pois metatietokaaret, huomioi vaihtoehtoiset kaaret ja määrittelee tarkasti, milloin kahta polkua voi pitää erillisinä.

Määritelmä 11. Viitelaskurin spesifikaatio funktionaaliselle verkolle

1. Polku on *metatiedoton*, jos se ei kulje yhdenkään metatietokaaren kautta.

2. Kaksi polkua ovat *keskenään vaihtoehtoiset*, jos niiden ensimmäiset kaaret ovat keskenään vaihtoehtoiset.

3. Kaksi polkua ovat *erilliset*, jos niillä ei ole alkusolmuja ja loppusolmuja lukuun ottamatta yhtään yhteistä solmua eikä kaarta[2].

4. Jos solmuun N on olemassa kaksi *syklitöntä*, *metatiedotonta*, *erillistä* ja *keskenään ei-vaihtoehtoista* polkua jostain solmusta A, niin solmu N on kierrätetty, muuten solmu N on kertakäyttöinen. □

Spesifikaatio voidaan toteuttaa seuraavalla algoritmilla. Algoritmi yrittää etsiä jokaiselle solmulle `N` kaksi erillistä polkua `P1` ja `P2` kaikista mahdollisista solmuista `A`. Ensimmäinen polku `P1` pakotetaan kulkemaan jonkin `N`:n vanhemman `M` kautta, jottei siitä tulisi triviaali nollan askeleen polku `N`:stä itseensä. Näin voidaan havaita myös syklit `N` $\rightsquigarrow$ `M` $\rightarrow$ `N`.

Algoritmi 8. Viitelaskuri funktionaaliselle verkolle

```
Jokaiselle kaarelle M -> N
{
  Etsi metatiedottomat syklitömät polut
    P1 = (A -> ... -> M -> N)  (mukaan lukien A = M)  ja
    P2 = (A -> ... -> N)
    missä
      P1 ja P2 eivät ole keskenään vaihtoehtoiset ja
      P1 ja P2 ovat erilliset.
  Jos tällaiset P1 ja P2 löytyivät,
    niin merkitään N kierrätetyksi.
}
```

[2] Samat kaaret kielletään, jottei yhden askeleen polku olisi erillinen itsensä kanssa.

```
Kaikki solmut, joita ei merkitty kierrätetyksi,
  merkitään kertakäyttöiseksi.
```

□

Olkoon E verkon kaarien määrä. Algoritmin pääsilmukka suoritetaan jokaiselle kaarelle. Pääsilmukan sisällä tehdään enintään kaksi polun hakua. Polun etsiminen kahden solmun välillä vie ajan $O(E)$. Kaaren metatiedottomuus, vaihtoehtoisuus ja erillisyys voidaan tarkistaa vakioajassa jokaisen läpi käytävän kaaren kohdalla. Algoritmin aikavaativuus on siis $O(E^2)$.

Algoritmi suoritetaan jokaiselle päätason funktiolle erikseen. Funktionaaliset verkot ovat harvoja verkkoja, joten kaarien määrä on samassa suuruusluokassa kuin solmujen määrä. Solmujen määrä taas on samassa suuruusluokassa kuin funktion ohjelmakoodissa esiintyvien sanojen ja erikoismerkkien määrä. Voidaan siis arvioida, että käytännön ohjelmoinnissa syntyvät verkot ovat riittävän pieniä $O(E^2)$ algoritmille. Algoritmin tehostaminen esimerkiksi dynaamisella ohjelmoinnilla lienee mahdollista, mutta sitä ei tutkittu tässä tarkemmin.

Barendsenin ja Smetsersin viitelaskuri eroaa tässä esitellystä viitelaskurista siten, että se merkitsee solmujen sijaan kaaret joko kierrätetyiksi tai kertakäyttöisiksi. Tämä mahdollistaa täsmällisemmän viitteiden laskennan [BS96] esimerkiksi seuraavassa tilanteessa.

```
if check(x) then change(x) else 0
```

Clean sallii `check`in käyttää `x`:ää tässä tapauksessa uniikkina, vaikka siihen on toinen viite, koska `if`-lausekkeen ehdon laskennasta ei voi jäädä muuta tulosta kuin yksi totuusarvo. Näin ollen `check` ei voi mitenkään piilottaa paluuarvoonsa uutta viittausta `x`:ään. Cleanissa kaikki sivuvaikutukselliset operaatiot palauttavat jonkin uuden uniikin ulkoista tilaa esittävän arvon, joten `check` ei voi olla sivuvaikutuksellinenkaan.

Tämän ominaisuuden toteutus sivuutettiin tässä yksinkertaisuuden vuoksi, koska ominaisuutta ei tarvita Uniicissa. Uniicissa `check` voidaan kirjoittaa lainaamaan parametrinsa, jolloin se jää uniikiksi `change`a varten.

4.2.3 De Vriesin uniikkityypitysjärjestelmä

De Vriesin uniikkityypitysjärjestelmä [deV08, s. 129] on melko suoraviivainen laajennos tavanomaiseen Hindley–Milner-tyyppijärjestelmään, sillä siinäkin tuotetaan ensin joukko tyyppiyhtälöitä, jotka sitten ratkaistaan. Osa tyyppiyhtälöistä koskee uniikkiusattribuutteja, ja näille on määriteltävä oma ratkaisumenetelmänsä, joka ei kuitenkaan vaikuta perustyyppien yhtälönratkaisuun.

Eräs oleellinen ero Cleanin ja de Vriesin järjestelmän välillä on seuraava. Clean esittää tiedon ”jos u on uniikki, niin v on myös uniikki” liittämällä symbolitauluun tallennettuun tyyppiin epäyhtälön $[v \leq u]$. De Vries ei tarvitse tällaisia epäyhtälöitä, joten tavallista Hindley–Milner-tyypinpäättelijää ei tarvitse laajentaa ymmärtämään niitä. Tämä on merkittävä yksinkertaistus.

De Vriesin toinen merkittävä parannus Cleaniin nähden on uniikkityypityksen yleistäminen toimimaan *korkeamman kertaluvun tyypeillä* (*arbitrary rank types*) [deV08, s. 115 ja 135]. Korkeamman kertaluvun tyypit sallivat luvussa 4.1.4 nähdyn universaalikvanttorin $\forall$ esiintymisen missä tahansa osassa tyyppiä, esimerkiksi funktion parametrin tyypissä tai paluutyypissä. Korkeamman kertaluvun tyypeistä kertoo tarkemmin esimerkiksi [Pie02, s. 339]. Yksinkertaisuuden vuoksi tässä käsitellään vain ensimmäisen kertaluvun tyyppejä.

4.2.4 Uniikkiusattribuutit propositiologiikan lausekkeina

Tarkastellaan seuraavaa funktiota, joka palauttaa parametrikseen saamansa parin ensimmäisen alkion.

```
\(p). p match { case (a, b) => a }
```

Funktio ottaa parametrikseen parin ja palauttaa sen ensimmäisen alkion. Funktion tyyppi ilman uniikkityypitystä olisi seuraava.

$$\forall t_1\ t_2.\ ((t_1, t_2)) \to t_1$$

Uniikkityypitettynä tyyppi voisi melkein olla seuraava.

$$\forall t_1\ t_2\ u_1\ u_2\ u_3.\ ((t_1^{u_1}, t_2^{u_2})^{u_3}) \to t_1^{u_1}$$

Tämä tyyppi on kuitenkin virheellinen, sillä se sallisi sijoituksen $u_1 = u_2 = \bullet$ ja $u_3 = \times$. Tällöin funktion tyyppi olisi $(t_1^\bullet, t_2^\bullet)^\times \to t_1^\bullet$, eli sillä voisi lukea saman parin ensimmäisen uniikin alkion kahdesti ja palauttaa sen muka uniikkina.

Eräs ratkaisu olisi korvata $u_3 = \bullet$, mutta näin funktio ei olisi yhtä yleinen. Se ei enää toimisi lainkaan ei-uniikeille pareille.

Cleanin ratkaisu on liittää tyyppiin rajoitteet $[u_3 \leq u_2, u_3 \leq u_1]$, jotka pakottavat u_3:n olemaan uniikki, jos joko u_1 tai u_2 on uniikki. Ratkaisu sallisi yhä ei-uniikkien parien välittämisen funktiolle, kunhan parin jäsenetkin ovat ei-uniikkeja. Ratkaisun käyttämien rajoitteiden lisääminen järjestelmään kuitenkin monimutkaistaisi sitä.

De Vriesin ratkaisu on korvata parin attribuutti u_3 attribuutilla $u_1 \vee u_2$, joka tarkoittaa oleellisesti samaa kuin yllä mainittu Cleanin epäyhtälö: pari on uniikki, jos joko u_1 tai u_2 on uniikki. Yleisesti, de Vries rinnastaa uniikkiusattribuutit propositiologiikan lausekkeisiin määrittelemällä uniikkiusattribuuttien joukon $\mathcal{U}$ seuraavasti.

1. $\bullet \in \mathcal{U}$ ja $\times \in \mathcal{U}$.
2. $v \in \mathcal{U}$ kaikille tyyppimuuttujille v.
3. Jos $a \in \mathcal{U}$, niin $(\neg a) \in \mathcal{U}$.
4. Jos $\{a_1, a_2\} \subset \mathcal{U}$, niin $\{(a_1 \vee a_2), (a_1 \wedge a_2)\} \subset \mathcal{U}$.

Uniikkiusattribuutteja käsitellään propositiologiikan lausekkeina pitämällä uniikkiutta $\bullet$ totena (1) ja ei-uniikkiutta $\times$ epätotena (0).

Hindley–Milner-tyypinpäättely saattaa johtaa kahden uniikkiusattribuutin väliseen yhtälöön, esimerkiksi yhtälöön $u_1 \vee u_2 \approx u_3 \wedge \bullet$. Tällaisen yhtälön ratkaisemiseksi eivät riitä samat keinot kuin muille tyyppiyhtälöille, sillä tällaisella uniikkiusyhtälöllä saattaa olla ratkaisu, vaikka yhtälön molempien puolten rakenne ei olekaan sama.

De Vries selittää ja perustelee sopivan yhtälönratkaisumenetelmän [deV08, s. 25]. Menetelmän yksityiskohtia ei kuvailla uudestaan tässä, mutta siitä huomioidaan seuraavaksi muutama seikka.

Menetelmä saattaa tuottaa ratkaisun, joka korvaa muuttujan u lausekkeella, joka sisältää korvattavan muuttujan u. Tässä ei ole mitään outoa, sillä ratkaisulle riittää saada yhtälöiden molemmat puolet tavalla tai toisella samoiksi.

Tyyppiyhtälön *yleisin ratkaisu* (*most general unifier*) on sellainen, josta voidaan johtaa kaikki muut ratkaisut sopivilla muuttujien korvauksilla. Yleisimpien ratkaisujen teoria sivuutetaan tässä toteamalla vain, että tavallinen Hindley–Milner-yhtälönratkaisu tuottaa yleisimmän ratkaisun [Pie02, s. 326], eikä uniikkiusattribuuttien ratkaiseminen de Vriesin valitsemalla menetelmällä hävitä tätä ominaisuutta [deV08, s. 27].

4.2.5 Funktioiden sulkeumien uniikkiusattribuutit

Funktiot, jotka viittaavaat ulkopuolisiin muuttujiin, eli funktiot, joilla on sulkeuma (ks. luku 2.2), vaativat uniikkityypityksessä erityistä varovaisuutta [deV08, s. 82]. Tarkastellaan seuraavaa koodia olettaen, että `a` on jokin uniikki arvo.

```
let f = \x. \y. x
and g = f(a)
in ...
```

Koodissa määritelty `g` on parametriton funktio, joka palauttaa kutsuttaessa sulkeumassa olevan `a`:n. Jos funktiota kutsutaan kahdesti, on `a` palautettu kahdesti, ja siihen saattaa olla voimassa kaksi viitettä. Näin ollen vaikuttaa siltä, että sulkeumaan menevät arvot tulisi merkitä varmuuden vuoksi ei-uniikeiksi.

Sekä Cleanissa että de Vriesin järjestelmissä on kuitenkin mahdollistettu uniikkien arvojen vieminen sulkeumaan. Tämä toimii, kunhan uniikkiin sulkeumaan viittaavaa funktiota voidaan kutsua enintään kerran.

Funktion kertakäyttöisyys voidaan periaatteessa toteuttaa merkitsemällä funktio uniikiksi. Ratkaisu ei kuitenkaan sellaisenaan toimi, jos kieli sallii uniikin arvon käyttämisen ei-uniikkina, koska ohjelmoija voisi ensin tyyppimuuntaa funktion ei-uniikiksi ja sitten kutsua sitä kahdesti [deV08, s. 80]. Clean sallii tällaisen tyyppimuunnoksen kaikille arvoille paitsi funktioille. De Vriesin järjestelmä taas kieltää tyyppimuunnoksen. Samoin tehdään tässä työssä.

Kiellon eräs haittapuoli on, että kielessä täytyy nyt olla aiempaa tarkempi sen suhteen, merkitäänkö arvo uniikiksi ($\bullet$) vai uniikkiudeltaan yleiseksi (u) [deV08, s. 133, 149]. Cleanissa uniikkeja parametreja ottava funktio on varsin joustava, koska sille kelpaavat myös ei-uniikit arvot, mutta de Vriesin järjestelmässä uniikin parametrin vaativaa funktiota voi kutsua vain uniikilla arvolla. Cleanin ratkaisu myös monimutkaistaa tyyppijärjestelmää.

De Vries esittää kaksi tapaa varmistaa uniikkisulkeumallisten funktioiden kertakäyttöisyys ja vertailee niitä [deV08, s. 134, 147]. Toisessa niistä funktion sulkeuma saa oman tyyppiattribuuttinsa funktion tavallisen tyyppiattribuutin lisäksi. Funktiolla on tällöin kaksi attribuuttia, mikä merkitään $(T_1 \underset{u_s}{\longrightarrow} T_2)^{u_f}$ tai lyhyemmin $T_1 \overset{u_f}{\underset{u_s}{\longrightarrow}} T_2$, missä u_s on sulkeuman attribuutti ja u_f on funktion tavallinen attribuutti. Ajatus on, että uniikkisulkeumaista funktiota saa kutsua vain kerran.

De Vriesn toinen ratkaisu ei anna sulkeumalle omaa attribuuttiaan. vaan uniikkisulkeumallinen funktio tyypitetään yksinkertaisesti $T_1 \xrightarrow{\bullet} T_2$. Tällöin tyyppijärjestelmän määrittelyssä täytyy varoa tilanteita, joissa uniikkisulkeumallinen funktio kulkee uniikista muuttujasta ei-uniikkiin muuttujaan, minkä kautta sitä voisi kutsua kahdesti. Myös tyyppimuuttuja voi saada arvokseen funktiotyypin, joten mikään tyyppiä $\forall t.\ t^{\bullet}$ tai $\forall t\ u.\ t^{u}$ oleva arvo ei myöskään saa koskaan saada tyyppiä $\forall t.\ t^{\times}$. Sulkeuma-attribuutti poistaa tämän rajoituksen, koska riittää, ettei funktio voi kulkea sulkeuma-attribuutiltaan uniikista muuttujasta sulkeuma-attribuutiltaan ei-uniikkiin muuttujaan. Näin on helpompi nähdä, että tyyppijärjestelmä toimii oikein sulkeumien osalta.

5 Uniic-ohjelmointikieli

Tässä luvussa esitellään Uniic-ohjelmointikielen toteutus. Kieli käännetään seuraavissa vaiheessa.

1. Imperatiivisesta ohjelmasta tehdään vuoverkko.
2. Vuoverkko muunnetaan SSA-muotoon.
3. SSA-muotoinen ohjelma tyyppitarkistetaan.
4. SSA-muotoisesta ohjelmasta poistetaan lainaus.
5. SSA-muotoinen ohjelma käännetään funktionaaliseksi ohjelmaksi.
6. Funktionaalinen ohjelma muunnetaan funktionaaliseksi verkoksi.
7. Funktionaalinen ohjelma tyyppitarkistetaan.

Tässä luvussa määritellään ensin Uniicin lähde- ja kohdekielet aiemmissa luvussa määriteltyjen kielten laajennoksina. Tämän jälkeen esitetään muunnos lähdekieleltä kohdekielelle. Lainauksen poisto tapahtuu ennen varsinaista muunnosta, joten se esitetään heti lähdekielen määritelmän jälkeen.

5.1 Imperatiivinen lähdekieli

Uniicin lähdekieli laajentaa luvun 3.1 yksinkertaista imperatiivista kieltä. Kieleen lisätään ensin uniikkityypitysjärjestelmä ja tämän jälkeen lainaus helpottamaan uniikkityypitettyä ohjelmointia.

Hindley–Milner-tyypinpäättelyä käytetään yleensä funktionaalisten kielten kanssa, mutta mikään ei estä sen käyttämistä imperatiivisessa kielessä. Imperatiivisesta ohjelmasta voi tuottaa tyyppiyhtälöitä melkein samalla tavalla kuin funktionaalisestakin ohjelmasta, eikä yhtälönratkaisijaa tarvitse muuttaa. Uniikkityypityksen osalta on huomioitava myös viitteiden laskenta.

5.1.1 Lähdekielen viitelaskuri

Vuoverkossa jokainen komento lukee ensin joidenkin muuttujien arvot, suorittaa sitten toimenpiteen ja kirjoittaa lopuksi tuloksen joihinkin muuttujiin. Jokainen

lähdekielen komento noudattaa seuraavaa sääntöä: jos komento lukee muuttujasta v arvon x yhden kerran, niin komennon suorituksen jälkeen x:ään on enintään yksi uusi viittaus. Näin ollen lähdekielen viitelaskuri voi merkitä muuttujan kertakäyttöiseksi, jos se voidaan jokaisella mahdollisella suorituspolulla lukea enintään kerran ennen kuin siihen sijoitetaan uusi arvo. Tyyppijärjestelmä huolehtii siitä, ettei komennon suorittama toimenpide jätä jälkeensä kahta viittausta luettuun arvoon.

Viitelaskuri toteutetaan algoritmissa 9 taaksepäin kulkevana vuoanalyysina. Analyysi käsittelee kuvauksia muuttujilta kokonaisluvuille, missä luvut ilmaisevat, montako kertaa muuttujaa saatetaan kyseisen kohdan jälkeen vielä lukea ennen seuraavaa ylikirjoitusta.

Analyysi etenee peruslohkon lopusta alkuun yksi komento kerrallaan. Jos komento c kirjoittaa muuttujaan x, niin x:n lukukerroiksi merkitään nolla, koska ollaan laskemassa lukukertoja ennen ylikirjoitusta. Tämän jälkeen x:n lukukertoja kasvatetaan niin monta kertaa kuin c lukee x:n. Lukukertojen määrä rajoitetaan kahteen, jotta algoritmi pysähtyisi, kun ohjelmassa on x:ää lukeva silmukka.

SSA-verkon `phi`-komennot ovat algoritmille erikoistapaus. Oletetaan, että algoritmi käsittelisi `phi`-komennon lukuoperaatioita kuten muita lukuoperaatioita. Nyt silmukka, jossa olisi esimerkiksi komento `a1 = phi(a2, ...)`, muttei mitään muuttujaa `a2` lukevaa tai kirjoittavaa komentoa, saattaisi algoritmin mielestä lukea `a2`:ta toistuvasti. Algoritmi siis toimisi `phi`-komentojen osalta liian konservatiivisesti.

Luvussa 3.6 esitetty tapa muuntaa vuoverkko SSA-muotoon lisää vain tarpeellisia `phi`-komentoja. Tarpeellinen `phi`-komento lukee aina vain sellaisia muuttujia, jotka eivät muuten olisi lohkon näkyvyydessä, koska niiden kirjoituskohta ei sijaitse lohkon dominoijassa. Viitelaskuri voi näin ollen turvallisesti käsitellä `phi`-komennon lukemia muuttujia ikään kuin ne poistettaisiin heti `phi`-komennon jälkeen. Niiden lukukerrat voi siis nollata, koska poistettua muuttujaa ei varmasti lueta.

Algoritmi 9. Lähdekielen viitelaskuri

Taaksepäin kulkeva vuoanalyysirunko parametrisoidaan seuraavasti.

1. $In(B)$ ja $Out(B)$ ovat kuvauksia muuttujilta kokonaisluvuille.
2. `OutRInit =` $x \mapsto 0$ `kaikille` x.
3. `InInit =` $x \mapsto 0$ `kaikille` x.
4. `meet(In1, ..., InN) =` $x \mapsto max($`In1`$(x), \ldots,$ `InN`$(x))$.

5. Siirtymäfunktio toteutetaan seuraavasti.

```
transfer(B, out):
  A = out
  jokaiselle B:n komennolle c viimeisestä ensimmäiseen:
    transferCmd(B, c, A)
  palauta A

transferCmd(c, A):
  jokaiselle muuttujalle x, johon c kirjoittaa:
    A[x] = 0

  jos c = phi(x1, ..., xn):
    jokaiselle muuttujalle x, jonka c lukee:
      A[x] = 0
  muuten:
    jokaiselle muuttujalle x, jonka c lukee:
      k = montako kertaa c lukee x:n
      A[x] = min(A[x] + k, 2)

  palauta A
```

□

SSA-verkossa jokaisella muuttujalla, joka ei ole globaali eikä parametri, on yksikäsitteinen sijoituskomento c_i jossain lohkossa B. Muuttuja x on kertakäyttöinen, jos se luetaan komennon c_i jälkeen enintään kerran ennen uudelleensijoitusta, eli formaalisti, jos `transfer(`B`[i + 1 .. length(`B`)],` $Out(B)$`)` ≤ 1. Globaali muuttuja tai parametrimuuttuja on kertakäyttöinen, jos $In(\texttt{B1})(x) \leq 1$, eli se luetaan koko vuoverkossa enintään kerran ennen uudelleensijoitusta.

5.1.2 Lähdekielen tyypityssäännöt

Lähdekielen tyypityksessä tyypit annetaan muuttujille eikä verkon solmuille ja tyypityssäännöt kohdistuvat komentoihin eivätkä solmuihin. Tyypityssäännöt tuottavat tyyppiyhtälöitä, jotka ratkaistaan kuten luvussa 4.1.3 ja uniikkiusattribuutten

osalta kuten luvun 4.2.4 lyhyesti esittelemässä de Vriesin järjestelmässä [deV08, s. 25].

Jokaisen kertakäyttöisen muuttujan v tyyppi $\Gamma(v)$ alustetaan viitelaskurin tuloksen perusteella. Jos v on kertakäyttöinen, niin $\Gamma(v) = t^u$, missä t ja u ovat tuoreita tyyppimuuttujia. Jos v ei ole kertakäyttöinen, niin $\Gamma(v) = t^\times$. Globaalien muuttujien ja parametrien tyypit luetaan symbolitaulusta.

Tyyppiyhtälöt tuotetaan SSA-verkon komennoista seuraavasti. Merkintä $\Gamma_B(v)$ tarkoittaa v:n perustyyppiä ja merkintä $\Gamma_A(v)$ tarkoittaa v:n uniikkiusattribuuttia. Huomaa, että perustyyppien osalta yhtälöt muistuttavat luvussa 4.1.2 nähtyjä yhtälöitä.

Määritelmä 12. Lähdekielen tyyppiyhtälösäännöt (ilman lainausta)

1. Komento v `:= phi(`v_1`, ..., `v_n`)` tuottaa seuraavat tyyppiyhtälöt:
   (a) $\Gamma(v) \approx \Gamma(v_i)$ jokaiselle $i \in \{1, \ldots, n\}$
2. Komento v `:=` c, missä c on vakio, jonka tyyppi on t_c, tuottaa seuraavan tyyppiyhtälön:
   (a) $\Gamma(v) \approx t_c^\times$
3. Komento v `:=` v' tuottaa seuraavan tyyppiyhtälön:
   (a) $\Gamma(v) \approx \Gamma(v')$
4. Komento v `:=` v_f`(`a_1`, ..., `a_n`)` tuottaa seuraavan tyyppiyhtälön, missä u_c on tuore:
   (a) $\Gamma(v_f) \approx (\Gamma(a_1), \Gamma(a_2), \ldots, \Gamma(a_n)) \xrightarrow[u_c]{u_c} \Gamma(v)$

   Selitys: *Funktion attribuutin ja sulkeuma-attribuutin pakottaminen samaksi takaa, että kutsuttava funktio on uniikki, jos sen sulkeuma on merkitty uniikiksi*[3].
5. Komento v `:= (`v_1`, ..., `v_n`)` tuottaa seuraavat tyyppiyhtälöt:
   (a) $\Gamma_B(v) \approx (\Gamma(v_1), \Gamma(v_2), \ldots, \Gamma(v_n))$

[3] Tämän takaamiseksi riittäisi $\Gamma_A(F) \approx u_c \vee u_f$, missä u_f on tuore. Se sallisi tilanteen, jossa funktio on uniikki vaikka sulkeuma ei ole, mutta tällaista tilannetta ei käytännössä tule, joten ei ole tarpeen tuottaa monimutkaisempaa tyyppiyhtälöä [deV08, s. 133].

(b) $\Gamma_A(v) \approx u \vee \Gamma_A(v_1) \vee \Gamma_A(v_2) \vee \cdots \vee \Gamma_A(v_n)$, missä u on tuore

Selitys: *Monikon on oltava uniikki, jos jokin sen alkioista on uniikki. Jos mikään alkioista ei ole uniikki, niin monikon uniikkius jää rajoittamattomaksi tyyppimuuttujaksi u.*

6. Komento `(`v_1`, ..., `v_n`) :=` v tuottaa seuraavan tyyppiyhtälön:

   (a) $\Gamma_B(v) \approx (\Gamma(v_1), \ldots, \Gamma(v_n))$

7. Komento `goto` L ei tuota tyyppiyhtälöitä.

8. Komento `if` v `then goto` L_1 `else goto` L_2 tuottaa seuraavan tyyppiyhtälön:

   (a) $\Gamma_B(v) \approx Bool$

9. Olkoot t ja u tuoreita. Verkon kaikkien `return` v_i -komentojen joukko tuottaa yhdessä seuraavat tyyppiyhtälöt:

   (a) $\Gamma(v_i) \approx t^u$ jokaiselle i □

Nämä säännöt eivät vielä tue lainausta. Lainaus lisätään kieleen ja tyypityssääntöihin seuraavissa luvuissa.

5.1.3 Lainaus ja sen poistaminen

Yksinkertaisessa uniikkityypitetyssä kielessä joudutaan usein kirjoittamaan seuraavanlaista koodia, jossa uniikin arvon `a` uudet versiot on aina sijoitettava uusiin muuttujiin, koska uniikkiin muuttujaan ei saa viitata kahdesti.

```
let a2 = f(a)
and a3 = g(a2)
...
```

Muuttujien peittäminen auttaa hieman, ja Clean salliikin tämän esimerkin kirjoittamisen seuraavasti.

```
# a = f(a)
# a = g(a)
...
```

Seuraava askel on pyrkiä seuraavanlaiseen täysin imperatiiviseen tyyliin, jossa `a` ohjelmoijan näkökulmasta ”muuttuu”.

```
f(a);
g(a);
...
```

Tämä tyyli onnistuu laajentamalla uniikkityypitystä *lainauksella* (*borrowing*) [Boy01]. Ajatus on, että jos funktio `f(@a)` lainaa parametrinsa (’`@`’-merkki), niin kutsuja voi käyttää `a`:ta kutsun jälkeen uudelleen.

Uniic toteuttaa lainauksen muuntamalla lainausta käyttävän ohjelman vastaavaksi lainausta käyttämättömäksi ohjelmaksi. Kutsu `x := f(@a);` muutetaan muotoon `tmp := f(a); (x, a2) := tmp;` ja kutsua seuraavat `a`:n käyttökohdat muutetaan käyttämään `a2`:ta.

Funktion `f(@a)` toteutuksessa jokainen palautuslause täytyy muuttaa vastaavasti palauttamaan monikko, jossa on varsinaisen paluuarvon lisäksi lainattujen muuttujien viimeisimmät versiot.

```
return x;   ⇒   return (x, a2);
```

Lainauksen poisto onnistuu periaatteessa aina, mutta ohjelmoija voi silti käyttää lainausta väärin esimerkiksi antamalla parametrin lainaan kun funktio odottaa tavallista parametria tai lainaamalla ei-uniikin arvon kun funktio odottaa uniikkia arvoa. Tyyppijärjestelmä havaitsee tällaiset virheet lainauksen poiston jälkeen. Tyyppijärjestelmää kannattaa silti laajentaa ymmärtämään lainausta, koska lainauksen poiston jälkeen suoritettavasta tyyppitarkistuksesta saatavat virheilmoitukset voivat olla epäselvempiä.

5.1.4 Lainauksen lisääminen tyyppijärjestelmään

Funktioiden tyypeissä osa parametrien tyypeistä $T_i^{u_i}$ voidaan merkitä lainatuiksi: $@T_i^{u_i}$. Lainaava funktiokutsu `f(@a)` esitetään SSA-muodossa `f(a@a2)`, missä `a2` on muuttujan `a` seuraava versio. Tällaista funktiokutsua pidetään viitteiden laskennan kannalta `a`:n lukevana ja `a2`:n sijoittavana lauseena. Tyypityssääntöjä täytyy muuttaa vain funktiokutsun osalta.

Määritelmä 13. Lainaus Uniicin tyypityssäännöissä

Määritelmää 12 muutetaan funktiokutsun osalta seuraavasti.

Komento v `:=` v_f`(`a_1`, ..., `a_n`)` tuottaa seuraavat tyyppiyhtälöt, missä u_c on tuore:

1. $\Gamma(v_f) \approx (T_1, T_2, \ldots, T_n) \xrightarrow[u_c]{u_c} \Gamma(v)$, missä
   (a) $T_i = \Gamma(a_i)$, jos a_i on tavallinen muuttuja
   (b) $T_i = @\Gamma(x_i)$, jos $a_i = x_i\texttt{@}y_i$
2. $\Gamma(x_i) \approx \Gamma(y_i)$ jokaiselle lainaukselle $a_i = x_i\texttt{@}y_i$ □

Kun ohjelma muutetaan ei-lainaavaan muotoon, parametrien lainausmerkinnät poistetaan ja lainattujen parametrien tyypit lisätään paluuarvon tyyppiin, vastaavasti kuin `return`-lauseiden muunnoksessa, seuraavan esimerkin mukaisesti.

$$(@T_1^{u_1}, T_2^{u_2}, @T_3^{u_3}) \to T_4^{u_4} \quad \Rightarrow \quad (T_1^{u_1}, T_2^{u_2}, T_3^{u_3}) \to (T_4^{u_4}, T_1^{u_1}, T_3^{u_3})$$

Huomaa, että sama tyyppijärjestelmä toimii nyt sekä ennen lainauksen poistamista että sen jälkeen. Uniic suorittaa tyypintarkistuksen molemmissa vaiheissa sisäisenä virheentarkistuksena.

5.2 Funktionaalinen kohdekieli

Uniicin kohdekielenä toimii luvuissa 2.7-2.9 esitetty yksinkertainen funktionaalinen kieli. Tyyppijärjestelmä on luvussa 4.2 esitetyn de Vriesin uniikkityypitysjärjestelmän eräs variantti. Tässä luvussa esitetään tyyppijärjestelmän tyypityssäännöt formaalisti ja käsitellään muutama toteutustekninen havainto.

5.2.1 Kohdekielen tyyppijärjestelmä

De Vries esittää kolme uniikkityypitysjärjestelmää, ja kustakin järjestelmästä on sekä yksinkertainen versio että korkeamman kertaluvun tyypeillä toimiva versio. Uniicin kohdekielen tyyppijärjestelmä perustuu de Vriesin toisen järjestelmän yksinkertaiseen versioon [deV08, s. 129].

Ensin määritellään, miten tyyppikokoelma Γ alustetaan. Ainoa oleellinen ero luvun 4.1.2 uniikkityypittämättömään versioon on, että tässä täytyy huomioida viitelaskurin tulos.

Määritelmä 14. Kohdekielen tyyppien alustus

1. Jos n on kokonaisluku, niin $\Gamma_0(\texttt{Const[}n\texttt{]}) = \texttt{Int}^\times$.
2. $\Gamma_0(\texttt{true}) = \Gamma_0(\texttt{false}) = \texttt{Bool}^\times$.
3. Jos x:lle on symbolitaulussa tyyppi, niin $\Gamma_0(\texttt{Var[}x\texttt{]})$ on symbolitaulusta luettu tyyppi.
4. Muissa tapauksissa $\Gamma_0(E) = t^u$, missä t ja u ovat tuoreita tyyppimuuttujia.
5. Olkoon $t^u = \Gamma_0(A)$. Jos solmu A on viitelaskennan perusteella kertakäyttöinen, niin $\Gamma(A) = t^u$, muuten $\Gamma(A) = t^\times$ □

Kohdekielen tyypityssäännöt ovat perustyyppien osalta samanlaiset kuin luvussa 4.1.2 ja uniikkiusattribuuttien osalta samankaltaiset kuin lähdekielessä. Tyyppiyhtälöiden määrittelyssä käytetään taas apumerkintää $\Gamma_B(E)$ tarkoittamaan E:n perustyyppiä ja $\Gamma_A(E)$ tarkoittamaan E:n uniikkiusattribuuttia.

Määritelmä 15. Kohdekielen tyyppiyhtälösäännöt

1. Solmu $E = \texttt{Apply}(F, E_1, E_2, \ldots, E_n)$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma(F) \approx (\Gamma(E_1), \Gamma(E_2), \ldots, \Gamma(E_n)) \xrightarrow[u_c]{u_c} \Gamma(E)$

   Selitys: *Funktion attribuutti pakotetaan samaksi kuin sen sulkeuman attribuutti, kuten lähdekielessä.*

2. Solmu $E = \texttt{Lambda}(\texttt{ParamList}(x_1, x_2, \ldots, x_n), \texttt{Closure}(C_1, C_2, \ldots, C_m), B)$ tuottaa seuraavan tyyppiyhtälön:

   (a) $\Gamma_B(E) \approx (\Gamma(x_1), \Gamma(x_2), \ldots, \Gamma(x_n)) \xrightarrow[u_c]{} \Gamma(B)$,
   missä $u_c = \Gamma_A(C_1) \vee \Gamma_A(C_2) \vee \cdots \vee \Gamma_A(C_m)$ jos $m > 0$, muuten $u_c = \times$.

   Selitys: *Funktion sulkeuman tyypin u_c vaaditaan olevan uniikki, jos jonkin sulkeumassa olevan solmun tyyppi on uniikki. Muuten yhtälö on oleellisesti sama kuin luvussa 4.1.2.*

3. Solmu $E = \texttt{If}(A, B, C)$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma_B(A) \approx \texttt{Bool}$

   (b) $\Gamma(E) \approx \Gamma(B)$

(c) $\Gamma(E) \approx \Gamma(C)$

4. Solmu $E = \texttt{Tuple}(E_1, E_2, \ldots, E_n)$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma_B(E) \approx (\Gamma(E_1), \Gamma(E_2), \ldots, \Gamma(E_n))$

   (b) $\Gamma_A(E) \approx u \vee \Gamma_A(E_1) \vee \Gamma_A(E_2) \vee \cdots \vee \Gamma_A(E_n)$, missä u on tuore.

   Selitys: *Kuten lähdekielessäkin, monikon on oltava uniikki, jos jokin sen alkioista on uniikki.*

5. Solmu $E = \texttt{Match}(E_0, \texttt{Case}(P_1, E_1), \texttt{Case}(P_2, E_2), \ldots, \texttt{Case}(P_n, E_n))$ tuottaa seuraavat tyyppiyhtälöt:

   (a) $\Gamma(E_0) \approx P_i$ jokaiselle $i \in \{1, \ldots, n\}$.

   (b) $\Gamma(E) \approx \Gamma(E_i)$ jokaiselle $i \in \{1, \ldots, n\}$

6. Muut solmut eivät tuota tyyppiyhtälöitä. □

Tyyppiyhtälöt ratkaistaan luvun 4.1.3 menetelmällä, johon on lisätty luvussa 4.2.4 kuvattu yhtälönratkaisija uniikkiusattribuuttien välisille yhtälöille.

5.2.2 Huomioita kohdekielen tyyppijärjestelmän toteutuksesta

Luvun luvussa 4.2.4 attribuuttien ratkaisumenetelmä tuottaa jokaista alkuperäisessä yhtälössä esiintyvää muuttujaa kohti välivaiheen. Välivaiheen koko on noin kaksi kertaa suurempi kuin sitä edeltävän välivaiheen koko, joten menetelmän aika- ja tilavaativuus on pahimmassa eksponentiaalinen. De Vries on havainnut, että käytännössä algoritmi kuitenkin näyttää toimivan nopeasti, jos jokaisen välivaiheen sieventää, ja Uniicin automatisoidut testit ovat tukeneet tätä havaintoa.

De Vries ei määrittele käyttämäänsä sievennysalgoritmia. Uniic käyttää Quine–McCluskey-algoritmin erästä heuristista varianttia [Qui12], joka saavuttaa yleensä paremman nopeuden mutta ei aina tuota täysin optimaalista sievennystä. Heuristiikka ei kuitenkaan korjaa algoritmin pahimmassa tapauksessa eksponentiaalista suoritusaikaa lausekkeen kokoon nähden.

Satunnaisesti luoduissa testitapauksissa algoritmi osui erittäin hitaisiin tapauksiin hyvin usein, joten sitä tehostettiin käsin kirjoitetulla sieventäjällä, joka soveltaa

viittätoista käsin kirjoitettua sievennyssääntöä rekursiivisesti. Esimerkiksi seuraavia sievennyssääntöjä käytettiin.

$$
\begin{aligned}
\neg 1 &= 0 \\
a \wedge 0 &= 0 \\
a \wedge \neg a &= 0 \\
a \vee 1 &= 1 \\
a \vee a &= a
\end{aligned}
$$

Tämän tehostuksen jälkeen yhteenkään hitaaseen tapaukseen ei enää osuttu, vaikka testejä toistettiin tuhansia kertoja. De Vries myöntää, että tyyppijärjestelmää pitäisi tutkia enemmän, ennen kuin voidaan todeta kohtuullisella varmuudella, että parhaat sievennysalgoritmit ovat luotettavasti riittävän nopeita käytännön ohjelmointia varten [deV08, s. 150].

Uniicin testauksessa ei havaittu yhtään eksponentiaalista räjähdystä tyyppijärjestelmän testeissä. Lisäksi kaikki testauksessa havaitut yhtälönratkaisijan tuottamat ratkaisut olivat sieventyneet helposti ymmärrettävään muotoon, joitakin tyyppivirheen sisältäviä ohjelmia lukuun ottamatta. Menetelmä, ainakin sellaisena kuin se on toteutettu Uniicissa, saattaa siis joskus johtaa tavallista vaikeammin ymmärrettäviin tyyppivirheisiin.

5.3 Lähdekielen muuntaminen kohdekieleksi

Tässä luvussa esitetään Appelin kuvaama muunnosalgoritmi [App98] joka muuttaa SSA-muotoisen imperatiivisen ohjelman funktionaaliseksi. Appelin menetelmä on variaatio Kelseyn ja Richardin muunnosalgoritmista [Kel95]. Algoritmien tuottamat ohjelmat eroavat hieman toisistaan, mutta ero on tämän työn kannalta epäoleellinen.

Muunnoksen idea on, että jokainen SSA-muotoisen vuoverkon lohko muutetaan funktioksi ja jokainen sijoituskomento `let`-sidonnaksi. Hyppykomennot muunnetaan funktiokutsuiksi ja `phi`-rakenne parametrinvälitykseksi.

Toimiakseen oikein, muunnoksen täytyy lisäksi asetella lohkoista saadut funktiot sisäkkäin siten, että jokaisella funktiolla on tarvitsemansa muuttujat näkyvyydessään. Tähän palataan alempana. Yksittäisen lohkon muunnos tehdään täsmällisemmin ilmaistuna seuraavasti.

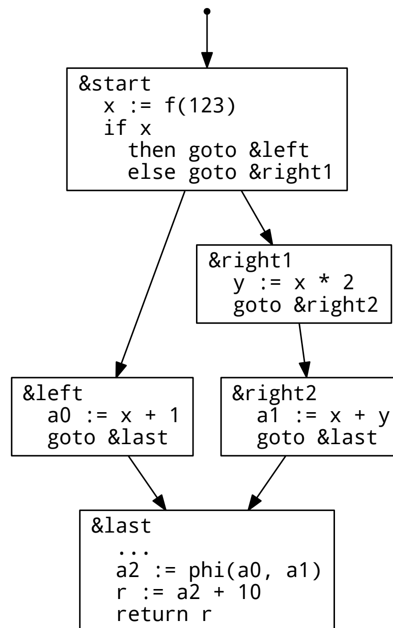
Algoritmi 10. SSA-vuoverkon muunnos funktionaaliseksi ohjelmaksi

1. Jokainen `phi`-komento `x := phi(...)` muutetaan lohkoa vastaavan funktion parametriksi `x`.
2. Jokainen tavallinen komento `x := ...` monikon purkamista lukuun ottamatta muutetaan `let`-sidonnaksi `let x = ... in` X, missä X on seuraavista komennoista rakennettava lauseke.
3. Jokainen monikon purkamiskomento `(x1, ..., xn) := y` muutetaan hahmonsovitusrakenteeksi `y match { case (x1, ..., xn) =>` X `}`, missä X on seuraavista tällä algoritmilla rakennettu lauseke.
4. Ehdoton hyppykomento `goto` B, muunnetaan funktiokutsuksi F_B`(a0, a1, ..., an)`, missä parametrit ovat ne näkyvyydessä olevat muuttujat, joita vastaavat B:n `phi`-komennot lukevat.
5. Ehdollinen hyppykomento `if` p `then goto` B `else goto` C muunnetaan lausekkeeksi `if` p `then` F_B`(...)` `else` F_C`(...)`, missä funktiokutsujen parametrit asetetaan kuten yllä.
6. Paluukomento `return` x muunnetaan lausekkeeksi x. □

Seuraava esimerkki havainnollistaa muunnosta. Esimerkin lohkot `left` ja `right2` päättyvät ehdottomaan hyppykäskyyn `goto &last`, joten vastaavat funktiot kutsuvat funktiota `last` ja välittävät sen tarvitseman parametrin. Lohko `start` hyppää joko lohkoon `left` tai `right1`, joten vastaava `start`-funktio kutsuu jompaa kumpaa funktiota, joista kumpikaan ei tarvitse parametreja, koska kummassakaan lohkossa ei ole `phi`-komentoja. Selittämättä on enää luotujen funktioiden sisäkkäisyys.

Esimerkki 16. Vuokaavioiden ja funktioiden vastaavuus

SSA-versio:



Funktionaalinen versio:

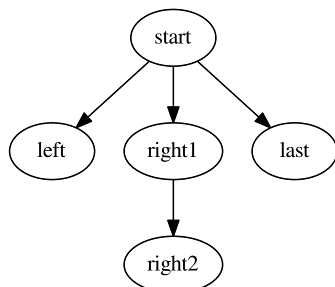
```
let start = \().

  let left = \().
    let a0 = x + 1
    in last(a0)

  and right1 = \().
    let right2 = \().
      let a1 = x + y
      in last(a1)
    and y = x * 2
    in right2()

  and last = \(a2).
    let r = a2 + 10
```

```
    in r

  and x = f(123)
  in if x
       then left()
       else right()

in start
```

□

SSA-muotoinen lohko A voi lukea muuttujia, joita se ei itse määrittele, vain, jos muuttujat on määritelty jossain sen dominoijassa (tai globaalisti). Näin ollen F_A kannattaa sijoittaa kaikkien dominoijiensa sisään, jottei sille tarvitse turhaan välittää parametreja, jotka se voisi saada näkyvyydestään. Funktioiden sisäkkäisyys voidaan siis asettaa vastaamaan lohkojen sijaintia dominointipuussa. Jos lohko A on lohkon B (suora) dominoija, niin F_B asetetaan (suoraan) F_A:n sisään.

Yllä olevan esimerkin dominointipuu on seuraava.



Esimerkin funktioiden sisäkkäisyys vastaa suoraan dominointipuun rakennetta: `right2` on `right1`:n sisällä, ja `left`, `right1` sekä `last` ovat `start`:n sisällä. Esimerkin tapauksessa `last` voisi myös olla ylimmällä tasolla `start`:n rinnalla, koska `last` ei viittaa mihinkään dominoijassa määriteltyyn muuttujaan.

5.4 Uniic-ohjelmaesimerkki

Tarkastellaan seuraavaa Uniic-lähdekoodiesimerkkiä, joka täyttää 15-alkioisen taulukon Fibonaccin luvuilla. Lainausmerkintöjä '`@`' ja kankeita taulukonkäsittelyfunktioita lukuun ottamatta esimerkki näyttää tavanomaiselta imperatiiviselta koodilta.

Esimerkki 17. Fibonaccin lukujen taulukointi Uniicilla

```
a := mkIntArray(15);
intArraySet(@a, 0, 0);
intArraySet(@a, 1, 1);

i := 2;
while i < intArrayLength(@a) do {
  next := intArrayGet(@a, i - 1) + intArrayGet(@a, i - 2);
  intArraySet(@a, i, next);
  i := i + 1;
}

return intArrayGet(a, 14);
```

□

Ohjelma käyttää taulukon käsittelyssä seuraavia kirjastofunktioita:

- `mkIntArray` : $\forall u.$(`Int`) $\rightarrow$ `IntArray`^{u} (taulukon luonti)
- `intArrayGet` : $\forall u.$(`@IntArray`^{u}, `Int`) $\rightarrow$ `Int` (taulukon alkion luku)
- `intArraySet` : (`@IntArray`$^\bullet$, `Int`, `Int`) $\rightarrow$ `Unit` (taulukon alkion kirjoitus)

Nämä ovat funktioita vain yksinkertaisuuden vuoksi. Mikään ei estäisi tavanomaisten taulukonkäsittelyrakenteiden `a[i] = ...` lisäämistä kieleen.

Seuraava on siistitty versio yllä olevasta esimerkistä käännetystä kohdekielisestä ohjelmasta. Esimerkistä nähdään muun muassa, että silmukka on muuttunut rekursiiviseksi funktioksi `while_start` ja että taulukon `a` toistuva lainaus on muuttunut toistuvaksi uuden version laskennaksi. Kutakin muuttujan versiota käytetään vain kerran, niin kuin pitääkin. Versio `a5` esiintyy koodissa kahdesti, mutta sen esiintymät sijaitsevat eri `if`-haaroissa.

Esimerkki 18. Fibonaccin lukujen taulukointi Uniicin kohdekielellä

```
let a = mkIntArray(15)
and a2 = intArraySet(a, 0, 0)
and a3 = intArraySet(a2, 1, 1)
and i = 2
and while_start = \a4, i2.
  intArrayLength(a4) match {
    case (len, a5) =>
      let while_body = \.
        let (tmp, a6) = intArrayGet(a5, i2 - 1)
        and (tmp2, a7) = intArrayGet(a6, i2 - 2)
        and next = tmp + tmp2
        and a8 = intArraySet(a7, i2, next)
        and i3 = i2 + 1
        in while_start(a8, i3)
      and while_end = \.
        intArrayGet(a5, 14)
      in
        if i2 < len then while_body() else while_end()
  }
in while_start(a3, i)
```

□

Esimerkki 17 on testattu Uniicin prototyyppitoteutuksella, johon on toteutettu kokonaislukutaulukot apukirjastona. Jatkossa nähdään myös esimerkkejä, jotka käyttävät hypoteettisia apukirjastoja, joita Uniicin prototyyppiin ei ole vielä toteutettu.

6 Uniic-ohjelmointikielen analyysi

Mikä tahansa imperatiivinen kieli voidaan helposti kääntää funktionaaliseksi toteuttamalla imperatiivisen kielen tulkki funktionaalisella kielellä. Tämä ei kerro mitään mielenkiintoista imperatiivisen ja funktionaalisen ohjelmoinnin suhteesta, eikä ole kovin hyödyllistä muutenkaan. On parmpi vaatia, että jokainen imperatiivinen funktio voidaan kääntää erikseen.

Kuitenkin erikseen kääntäminenkin voidaan tehdä melko hyödyttömästi sallimalla jonkinlainen ”imperatiivista tilaa” edustava lisäparametri käännetyissä funktioissa. Tähän tilaparametriin voitaisiin nimittäin piilottaa imperatiivisen koodin käsittelemän muokattavan muistin esitys. Kaikki imperatiiviset rakenteet voitaisiin näin kääntää funktionaaliseksi koodiksi, joka laskee uuden version tuosta muistista. Tilaparametrin vaatiminen tekisi myös imperatiivisen koodin kutsumisen funktionaalisesta koodista hankalaksi.

Uniic käännetään funktio kerrallaan ilman mitään lisäparametreja. Sanotaan tätä *yksi yhteen* -käännökseksi.

Yksi yhteen -käännöksen onnistuminen kertoo, että lähdekieli on modulaarisella tavalla ”vain” vaihtoehtoinen syntaksi kohdekielelle. Minkä tahansa kohdekielisen funktion voi kirjoittaa uudelleen lähdekielellä, ellei lähdekieli ole jollain tavalla vähemmän voimakas. Uniicin käännös siis osoittaa, että sopivasti rajoitettu imperatiivinen ohjelmointi on pohjimmiltaan ”vain” funktionaalista ohjelmointia.

Uniicin suoraviivaisin käytännön sovellus on funktionaalisen kielen alikielenä toimiminen. Uniicin kaltainen käännös on myös eräs tapa osoittaa imperatiivisen funktion puhtaus funktionaalisen kielen puhtautta hyväksi käyttäen. Puhtauden voi osoittaa muillakin tavoilla, mutta puhtauden määrittelemisessä on tällöin syytä olla varovainen. Puhtaudelle on imperatiivisessa ohjelmoinnissa monta mielekästä määritelmää, kuten luvussa 6.4 nähdään.

Seuraavassa luvussa perustellaan Uniicin imperatiivisten rakenteiden hyödyllisyyttä vertaamalla Uniicia muihin funktionaalisiin kieliin. Tämän jälkeen koetellaan Uniicin käännöstekniikan rajoja tarkastelemalla Uniicille hankalia ohjelmointitilanteita ja pohtimalla, voisiko Uniicia kehittää pärjäämään kyseisessä tilanteessa paremmin yksi yhteen -käännöksestä luopumatta.

6.1 Uniic funktionaalisiin kieliin verrattuna

Uniicissa voidaan katsoa olevan kolme tärkeää ominaisuutta: sijoituslause, silmukkarakenne ja lainaus. Sijoituslause ja silmukkarakenne tukevat toisiaan, sillä silmukka olisi täysin hyödytön ilman sijoitusta ja sijoituslauseet voisi melkein aina korvata suoraviivaisesti `let`-rakenteella koodissa, joka ei käytä silmukoita. Sijoituslausetta ja silmukkarakennetta kannattaa siis tarkastella yhdessä, mutta lainausta voidaan pitää näistä erillisenä.

Sijoituslause ja silmukkarakenne

Funktionaalisissa kielissä kaikki toisto perustuu rekursioon, mutta tyypillisiä toiston käyttötapauksia, kuten joitakin listojen läpikäyntejä, on helpotettu standardikirjastofunktioilla. Vertailukelpoisuuden vuoksi oletetaan, että käsillä on toistoa vaativa ohjelmointitilanne, johon mikään standardikirjastofunktio ei suoraan sovi, vaan ainoa luonteva vaihtoehto silmukalle on rekursio.

Seuraavassa esimerkissä esitetään silmukka ensin Uniicin `while`:llä, ja sitten kahdella erilaisella rekursiota käyttävällä tavalla.

Esimerkki 19. Silmukka ja rekursio

```
// Imperatiivinen versio
while f(a, b) do {
  b := g(a, b);
}
after_loop(b);

// Rekursiivinen versio 1
let loop = \(a, b).
  if f(a, b)
    then loop(a, g(a, b))
    else b
in after_loop(loop(a, b))

// Rekursiivinen versio 2
let loop = \(a, b).
  if f(a, b)
    then loop(a, g(a, b))
    else after_loop(b)
```

```
in loop(a, b)
```

□

Imperatiivisen version selkeyden puolesta voidaan esittää esimerkiksi seruaavia (hyvin subjektiivisia) argumentteja:

- Tilan päivittäminen sijoituslauseilla on selkeämpää kuin parametrinvälitys.
- Silmukan jälkeen suoritettava koodi kirjoitetaan imperatiivisessa versiossa suoraan silmukan perään, kun taas rekursiivisessa koodissa se voi sijaita silmukkaa kutsuvassa koodissa (yllä versio 1) tai vaihtoehtoisesti rekursion perustapauksessa (yllä versio 2).
- Silmukasta voidaan tarvittaessa poistua `return`, `break`-, `continue`- tai `throw`-rakenteilla. Funktionaalisissa kielissä rekursion jättäminen kesken mielivaltaisissa kohdissa ei aina onnistu luontevasti, sillä laskenta on ilmaistava yhtenä lausekkeena usean lauseen sijaan eikä lausekkeen laskennan ”keskeyttämiselle” ole funktionaalisissa kielissä yleensä yhtä joustavia mekanismeja kuin imperatiivisille silmukoille.

Lainaus

Lainauksen tuoma hyöty lainauksettomaan uniikkityypitettyyn kieleen on periaatteessa triviaali, sillä lainaus säästää ohjelmoijaa vain joidenkin paluuarvojen toistuvalta kirjoittamiselta. Vaikutus ei ole edes rakenteellinen vaan lausekohtainen. Saavutettu hyöty ei kuitenkaan ole aivan vähäpätöinen riittävän imperatiivisessa koodissa.

Clean tunnustaa uniikkityypitetyn paluuarvon toistamisen rasittavuuden tarjoamalla standardikirjastossaan kaksi versiota useasta uniikkia arvoa lukevasta funktiosta [Ach11]. Tarkastellaan esimerkiksi taulukon koon palauttavia Clean-funktioita.

- $\texttt{size} : \forall t\ u.\ \texttt{Array}(t)^u \to \texttt{Int}$
- $\texttt{usize} : \forall t\ u.\ \texttt{Array}(t)^u \to (\texttt{Int}, \texttt{Array}(t)^u)$

Selvästi `size`-funktiota on mukavampi käyttää, mutta sille annettua taulukkoa ei voi käyttää uniikkina muualla. Uniicin vastaava funktio lainaa taulukkoparametrinsa ja näyttää täten ohjelmoijalle samanlaisena kuin Cleanin `size`, mutta kääntyy

Cleanin `usize`:a vastaavaan muotoon. Uniic siis automatisoi asian, joka on Cleanin kirjastossa tehty käsin.

Luvussa 4.2.2 huomautettiin, että Cleanissa voidaan joissakin tilanteissa lukea uniikkia arvoa kahdesti sijoittamatta sitä uudelleen ja myöskään menettämättä sen uniikkiutta. Tämä vaatii, että viitelaskuri ymmärtää ylimääräisten käyttökohtien olevan turvallisia, koska ne ovat esimerkiksi `if`-lauseen ehdossa. Lainauksen ansiosta Uniic välttää tämän monimutkaistuksen viitelaskurissaan. Luultavasti lainaus on ohjelmoijallekin selkeämpi mekanismi kuin edistynyt viitelaskuri.

6.2 Uniicin jatkokehitysmahdollisuuksia

Uniic on nykymuodossaan vielä melko vaatimaton, ja mielenkiintoisia jatkokehitysmahdollisuuksia on paljon. Seuraavassa hahmotellaan joitakin mahdollisia laajennoksia Uniiciin sekä niiden kääntämistä kohdekielelle.

Rekursiiviset tietorakenteet

Uniicin ainoa ohjelmoijan määriteltävissä oleva tietorakenne on monikko. Tyyppijärjestelmä ei mahdollista esimerkiksi linkitetyn listan rakentamista monikoista, koska monikon alkion tyyppi ei voi olla sama kuin monikon itsensä tyyppi. Uniicin monikot eivät siis ole *rekursiivisia tyyppejä* [Pie02, s. 267].

Rekursiiviset tyypit eivät ole ongelma uniikkityypityksen kannalta [BS96], ja Uniicin imperatiiviset rakenteet soveltuisivat myös rekursiivisten arvojen läpikäyntiin. Esimerkiksi listan läpikäynti Uniicilla voisi näyttää seuraavalta.

Esimerkki 20. Listan läpikäynti Uniicilla

```
while !isEmpty(list) do {
  process(head(list));
  list := tail(list);
}
```

□

Koodi toimii myös uniikille listalle, jos funktiot `isEmpty`, `head` ja `tail` muutetaan lainaamaan listan. Koodi ei kuitenkaan toimi, jos lista sisältää uniikkeja arvoja, koska `head` ei voi sekä palauttaa uniikkia arvoa että jättää sitä listaan. Tähän uniikkityypitykselle ominaiseen ongelmaan palataan luvussa 6.3.

Olio-ohjelmointi

Edellisessä esimerkissä voisi olla luontevampaa kutsua listan käsittelyfunktioita olio-ohjelmointityylisellä metodisyntaksilla `list.isEmpty()`, `list.head()` ja `list.tail()`. Helppo tapa mahdollistaa tämä on tehdä `a.f(...)`:stä yksinkertaisesti vaihtoehtoinen syntaksi `f(a, ...)`:lle, ja näin Uniicin prototyyppitoteutus tekeekin.

Tämä ratkaisu ei kuitenkaan mahdollista saman metodin eri versioita eri tyypeille. Kutsuttavan funktion valitseminen tyypin perusteella ei ole helppoa Hindley–Milner-tyypinpäättelyssä, sillä tyypinpäättelijä ei yleisessä tapauksessa voi etukäteen tietää, minkä luokan symbolitaulusta funktion tyyppi kuuluu lukea.

Myös perintä on Hindley–Milner-tyypinpäättelyn kannalta tunnetusti ongelmallista, sillä perintä johtaisi tyyppiyhtälöiden sijaan tyyppiepäyhtälöihin, jotka ovat muotoa $A \leq B$ ("tyyppi A on tyypin B alityyppi"). Näitä ei voi enää ratkaista samoilla yhtälönratkaisumenetelmillä, mikä vaikeuttaa tyypinpäättelyä merkittävästi [Pie02, s. 355]. Tyyppiluokat ovat tosin eräs alityypityksen välttävä vaihtoehto perinnälle [HHP07, luku 6].

Hindley–Milner -tyypinpäättelyn menettämistä lukuun ottamatta olio-ohjelmoinnin ominaisuuksien lisäämiselle Uniiciin tapaiseen kieleen tuskin on merkittäviä esteitä. Muutos olisi kuitenkin niin suuri, ettei sen yksityiskohtia lähdetä tässä selvittämään.

Lineaariset tyypit

Uniikkityypitys vaatii, että arvoa käytetään *enintään* kerran. Tätä rajoitusta sanotaan joskus myös *affiiniksi tyypitykseksi* [Pie05, s. 5]. Joskus on hyödyllistä vaatia, että arvoa käytetäänkin *täsmälleen* kerran. Näin saadaan *lineaarinen tyypitys*. Lineaarisella tyypityksellä voidaan esimerkiksi varmistaa, että ohjelmoija muistaa sulkea avaamansa tiedoston, koska muuten tyyppijärjestelmä varoittaa käyttämättömäksi jääneestä tiedosto-oliosta.

Lineaariset tyypit voitaisiin toteuttaa uudella uniikkiusattribuutilla, mutta tämä olisi ongelmallista, koska nyt kaikkia uniikkiudeltaan yleistettyjä tyyppejä ($\forall u.\ T^u$) täytyisi pitää myös mahdollisesti lineaarisina. Parempi ratkaisu olisi luultavasti tehdä joistakin perustyypeistä lineaarisia, ja vaatia, että jos lineaarinen perustyyppi esiintyy tietorakenteen, kuten monikon, osana, niin tietorakenne on myös lineaarinen. Tällöin kielessä ei kuitenkaan saisi olla tapaa "käyttää" tällaista tietorakennetta käyttämättä sen sisältämiä lineaarisia tyyppejä.

Tuhoamisfunktiot

Lineaarisia tyyppejä yksinkertaisempi ratkaisu resurssien sulkemiselle voisi olla sallia tyyppeihin liitettävä tuhoamisfunktio (*destructor*)[4], jota kutsutaan, kun tämän tyyppinen uniikki olio poistuu näkyvyydestä. Nyt tiedoston tyyppi voitaisiin merkitä sellaiseksi, että sitä voi käyttää vain uniikkina, ja siihen voitaisiin liittää tiedoston sulkeva destruktori.

Uniikkityypitettyä I/O:ta käyttävässä kielessä tiedoston sulkeminen vaatii kuitenkin `world`-parametrin, jota destruktorilla ei lähtökohtaisesti ole. Ongelman voi ratkaista kielen puhtautta menettämättä lisäämällä kieleen samanlaiset *implisiittiset arvot* kuin Scalassa [OAC04, s. 16], jotka mahdollistavat `world`-parametrin välittämisen (ja lainaamisen) funktioille (ja tuhoamisfunktioille) automaattisesti.

Silmukoiden keskeyttäminen

Imperatiivisissa kielissä silmukat voi yleensä keskeyttää `break`-lauseella, ja silmukan rungosta voi hypätä silmukan alkuun `continue`-lauseella. Nämä lauseet kääntyvät hyppykomennoiksi joko silmukan alkuun tai loppuun. Niiden toteuttaminen ei siis vaadi mitään muutosta Uniicin muunnosalgoritmiin, sillä se toimii jo kaikille valideille vuokaavioille. Yhtä helposti voitaisiin myös toteuttaa `goto`-lause.

Monadit

Haskellin monadinen I/O on vaihtoehto uniikkityypitetylle I/O:lle (ks. luku 2.3). Monadeilla voidaan ilmaista I/O:n lisäksi muitakin ohjelmointimalleja. Esimerkiksi monadisessa tilan käsittelyssä ketjutetaan I/O-komentojen sijaan jonkinlaista tilaa käsitteleviä komentoja. Tämä tila voi olla esimerkiksi satunnaisgeneraattorin tila. Monadisessa virheen käsittelyssä puolestaan ketjutetaan funktioita, jotka saattavat palauttaa virheen.

Haskellin `do`-syntaksi [Has10, luku 3.14] on eräänlainen imperatiivinen alikieli, joka helpottaa monadisten ohjelmien kirjoittamista. Syntaksin ajatus on kääntää peräkkäin kirjoitetut monadiset ”komennot” koodiksi, joka liittää komennot yhteen `>>=`-operaatiolla. Seuraava esimerkki näyttää tämän muunnoksen erittäin yksinkertaisessa tapauksessa.

[4]Tässä termiä *destructor* käytetään samassa merkityksessä kuin esimerkiksi C++-kielessä. Funktionaalisissa kielissä sanaa käytetään joskus tarkoittamaan tietorakenteen purkavaa funktiota tai rakennetta.

Esimerkki 21. Do-syntaksin kääntäminen

```
do x <- getLine
   print x
```

Käännös:

```
(getLine) >>= (\x -> print x)
```

□

Uniicin silmukkarakenteiden lisääminen `do`-syntaksiin olisi luultavasti melko helppoa. Seuraavassa on hahmotelma siitä, miltä silmukan käännös voisi näyttää.

Esimerkki 22. Monadisen Uniicin kääntäminen

Imperatiivinen koodi:

```
x := getLine;
while x != "" do {
  y := x + x;
  print(y);
  x := getLine;
}
print("bye");
```

Käännös:

```
let while_start =
  \x. if x != "" then while_body(x) else while_end()
and while_body =
  \x. let y = x + x
      in print(y) >>=
        \unused. getLine >>=
          \x2. while_start(x2)
and while_end =
  \. print("bye")
in getLine >>= (\x. while_start(x))
```

□

Silmukat kääntyisivät monadisessa koodissa keskinäisesti rekursiivisiksi funktioiksi aivan kuten uniikkityypitetyssä Uniicissa. Ainoa oleellinen ero on, että mutta monadisten funktioiden ”paluuarvot” täytyy ottaa vastaan `>>=`:n oikealla puolella olevan funktion parametreina. Peräkkäiset monadisten funktioiden kutsut muuttuvat siis sisäkkäisiksi `>>=`-operaattorin sovelluksiksi, kuten esimerkin `while_body`:ssa.

Monadit ovat hyvä korvike uniikkityypitetylle I/O:lle, mutta muuttuvan tilan esittäminen monadisessa koodissa ei ole aivan yhtä sujuvaa. Kun Cleanin käyttöliittymäkirjastoa kirjoitettiin uudelleen Haskellilla, käytettiin uniikkityypitetyn tilan ilmaisemiseen Haskellin `MVar`-muuttujia [AJ01]. `MVar` on viittaus muuttujaan, jota voi lukea ja kirjoittaa monadisilla I/O-komennolla.

Esimerkki 23. `MVar`-muuttujat Haskellissa

```
do muuttuja <- newMVar 0
   arvo <- takeMVar muuttuja
   putMVar muuttuja (arvo + 1)
```

□

Uniicista voitaisiin luultavasti tehdä myös versio, joka tukee muuttuvan tilan ilmaisemista `MVar`-muuttujilla. Lainaus ilmaistaisiin tällöin antamalla parametrina viittaus `MVar`-muuttujaan, jonka arvoa funktio saa vapaasti muokata.

Implisiittinen lainaus

Uniicissa lainaksi annettavat parametrit merkitään kutsukohdassa eksplisiittisesti `@`-merkillä. Tämä eksplisiittisyys saattaa olla hyvä ohjelman ymmärrettävyyden kannalta, mutta on silti mielenkiintoista pohtia, voisiko tyyppijärjestelmä asettaa tarvittavat ’`@`’-merkit funktiokutsuihin automaattisesti.

Tämä on lähtökohtaisesti hankalaa Hindley–Milner-tyypinpäättelyä käyttävässä kielessä, koska kutsuttavan funktion tyyppiä ei yleisessä tapauksessa tunneta tyyppiyhtälöiden muodostusvaiheessa. Eräs ratkaisu on siirtyä tavanomaisempaan tyyppijärjestelmään, jossa ohjelma tyyppitarkastetaan ”ylhäältä alas”. Nyt jokaisessa funktiokutsussa tiedetään funktion tyyppi, johon kuuluu tieto lainattavista parametreista, ja lainaus ilman ’`@`’-merkkejä onnistuu.

6.3 Uniicin rajat imperatiivisessa ohjelmoinnissa

Tässä luvussa esitetään joitakin perinteisen imperatiivisen ohjelmoinnin tilanteita, joita on vaikeaa tukea Uniicissa ilman kielen puhtauden menettämistä tai käännöstavan merkittävää muuttamista. Toisaalta nähdään, että monesti näihin tilanteisiin on silti olemassa tyydyttävä ratkaisu.

Jaettu muisti

Uniikkityypitys lähtökohtaisesti estää samaan muutettavaan olioon viittaamisen monesta paikasta. Tämä on yleensä hyvä asia ohjelman selkeyden kannalta, mutta on silti tilanteita, joissa tätä rajoitusta voidaan perustellusti haluta kiertää.

Jaettu muisti voidaan toteuttaa sopivalla kirjastolla. Esimerkiksi seuraava kirjasto toteuttaa jaetun muistin käsittelyn tavalla, jossa rinnakkaisuutta hallitaan lukolla. Toteutustapa on samanlainen kuin uniikkityypitetyssä I/O:ssa ja se muistuttaakin tiedoston avaamista ja sulkemista.

Esimerkki 24. Lukolla suojattu jaettu muisti

- `guard` : $T^\bullet \to$ `Guarded`$(T)^\bullet$ – käärii uniikin olion lukolla vahdittuun olioon.
- `lockGuarded` : `@Guarded`$(T)^\bullet \to T^\bullet$ – lukitsee uniikin olion ja ottaa sen käyttöön.
- `unlockGuarded` : (`@Guarded`$(T)^\bullet, T^\bullet) \to$ `Unit` – palauttaa uniikin olion takaisin lukolla vahdittuun olioon ja vapauttaa lukon. □

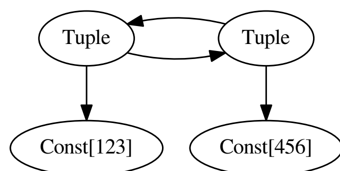
Toinen tapa, joka jättää kaiken vastuun turvallisuudesta ohjelmoijalle, on tarjota kirjastofunktio, joka yksinkertaisesti kopioi uniikin viitteen ja valehtelee kopionkin olevan uniikki. Tällaisen funktion tyyppi olisi $t^\bullet \to (t^\bullet, t^\bullet)$ ja se rikkoisi kielen puhtaustakeet suunnilleen yhtä rajusti kuin esimerkiksi Haskellin `unsafePerformIO`, joka sallii I/O monadin suorittamisen kesken tavallisen laskennan.

Sykliset tietorakenteet

Joissakin tietorakenteissa, kuten verkoissa ja kahteen suuntaan linkitetyissä listoissa, on luonnollisesti syklisiä viittauksia. Funktionaalisissa kielissä `let`-rakenne useimmiten kyllä sallii syklisen tietorakenteen luomisen seuraavasti.

Esimerkki 25. Syklinen tietorakenne funktionaalisessa kielessä

```
let a = (123, b)
and b = (456, a)
in ...
```



□

Uniikkityypitys ei voi hyväksyä uniikkeja alkioita sisältäviä syklisiä rakenteita. Sykliset tietorakenteet voidaan onneksi yleensä esittää myös syklittömästi ilman kohtuutonta vaivaa. Esimerkiksi verkon voi esittää siten, että solmut pidetään taulukossa, ja kaaret esitetään solmujen indeksien pareina.

Tietorakenteiden osien lukeminen

Oletetaan, että koodi saa parametrikseen uniikin tietorakenteen `x`, jossa on kaksi uniikkia kenttää `a` ja `b`, joiden perusteella koodin pitäisi muuttaa tietorakenteen kolmatta kenttää `c`. Koodina tämä voisi näyttää seuraavalta.

```
x.c := laske(x.a, x.b);
```

Uniikkityypitys ei kuitenkaan voi sallia, että `x.a`:n arvoon syntyy uusi viittaus parametrin välitystä varten, koska alkuperäinen viittaus `x`:stä `x.a`:han jäisi kutsun jälkeen yhä voimaan. Edistynyt uniikkityypitys tosin saattaisi nähdä, ettei `x.a`:ta lueta toista kertaa, vaikka `x.b`:tä luetaankin.

Uniic voisi yrittää selviytyä tällaisista tilanteista kääntämällä yllä olevan kaltaiset lauseet kohdekielessä koodiksi, joka purkaa tietorakenteen, käyttää sen osia ja kokoaa tietorakenteen lopulta uudestaan. Esimerkkilauseen käännös voisi näyttää esimerkiksi seuraavalta.

```
x match {
  case (a, b, c, d, e) =>
    let (c2, a2, b2) = laske(a, b)
    and x2 = (a2, b2, c2, d, e)
```

```
    in ...
}
```

Tällaisen käännöksen toteuttaminen saattaa olla monimutkaista, mutta suurempi ongelma on, että se vaikeuttaa abstraktioita. Tietorakenteen kenttien lukemista ja kirjoittamista ei enää voisi piilottaa apufunktioihin esimerkiksi seuraavasti.

```
setC(x, laske(getA(x), getB(x))
```

Ongelmaa voi nyt tarkastella näiden ”aksessorifunktioiden” tyypittämisen kannalta. Ainoa tapa antaa `getA`:n ja `getB`:n parametriksi uniikki `x` näyttää olevan `x`:n lainaaminen, mutta sitten ongelma on vain siirretty `getA`:n ja `getB`:n toteutukseen. Näitä funktioita ei voi toteuttaa poistamatta niiden palauttamaa oliota jollain tavalla `x`:stä.

Palautettavan olion poistaminen tietorakenteesta on mahdollista yleisessä tapauksessa, jos tietorakenteeseen jäävä ”aukko” voidaan täyttää joko jollain ohjelmoijan antamalla toisella arvolla tai yleisellä `null`-arvolla [Hog91]. Kumpikaan vaihtoehto ei vaikuta houkuttelevalta.

Tietorakenteen osan lukemisen ongelma on esitetty tässä monikkojen tai tietueiden avulla, mutta täsmälleen sama ongelma on uniikkialkioisten taulukoiden ja muiden suurten tietorakenteiden kanssa. Yleinen `null`-arvoihin tai arvon vaihtoihin perustuva ratkaisu pätee näihinkin.

Taulukoiden osalta ongelma voitaisiin ratkaista joissakin tapauksissa sopivanlaisella `for each` -silmukkarakenteella. Rakenne sallisi vain jokaisen alkion käsittelyn vuorollaan. Tämä lienee yleisin taulukon läpikäyntitapa, mutta se ei riitä, kun taulukon arvoja halutaan lukea tai kirjoittaa mielivaltaisessa järjestyksessä.

Täysin yleinen ratkaisu, joka ei turvautuisi `null`-arvoihin eikä vaihtoihin, vaatisi staattisen analyysin, joka voisi osoittaa, että kaikki samaan aikaan voimassa olevat viitteet taulukkoon ovat tulleet eri indeksistä. Indeksien erillisyyden osoittaminen on sukua osoittimien erillisyyden eli aliaksettomuuden osoittamiselle, joka on laajasti tutkittu ja vaikea ongelma [Hin01].

6.4 Lievempiä versioita puhtaudesta

Funktionaalisissa kielissä puhtaus on luontevaa määritellä siten, ettei mitään kerran luotua arvoa voida muuttaa. Uniikkiuden mahdollistamia optimointeja lukuun

ottamatta Uniicin kohdekieli noudattaa tätä määritelmää, ja täten Uniicin lähdekielenkin on sitä noudatettava. Määritelmää voidaan kuitenkin pitää liian rajoittavana, sillä kaikki mahdolliset funktion sivuvaikutukset eivät ole yhtä haitallisia.

Uniikkityypityksestä voidaan luopua menettämättä funktion puhtautta *kutsuvan koodin näkökulmasta*, jos kääntäjä voi osoittaa, että uniikkiussääntöjä rikkovat viitteet eivät koskaan näy abstraktiokerroksen, kuten luokan tai moduulin, ulkopuolelle. Tähän on pyritty esimerkiksi Joline-kielessä heikentämällä uniikkiuden käsite *ulkoiseksi uniikkiudeksi* [CW03] ja hyödyntämällä omistustyyppejä [CPN98] olioiden sisäkkäisyyden esittämiseen.

Arvon omistustyyppi kertoo, minkä olion sisään arvo kuuluu. Olion metodit voivat muuuttaa vapaasti vain olion omistamia tietorakenteita. Ulkoinen uniikkius taas sallii yhden ulkoisen viitteen lisäksi mielivaltaisen määrän sisäisiä (syklisiä) viitteitä samaan olioon, kunhan syklit kulkevat vain olion omistamien arvojen kautta.

Uniikkius voidaan myös hylätä kokonaan sallimalla kaikkien funktiolle paikallisten tietorakenteiden muokkaus. *Deterministinen* funktio saa muokata itse luomiensa olioiden lisäksi vain parametriensa kautta löydettävissä olevia olioita, vaikka näihin olisi viitteitä muualtakin. Deterministisen funktion tulos ei siis yhä saa riippua esimerkiksi kellosta, satunnaisgeneraattorista tai globaaleista muuttujista. Huomaa, että deterministinen funktio, joka suorittaa kirjoitusoperaatioita vain itse luomiinsa olioihin, on myös puhdas.

Vaikka ulkoinen uniikkius, omistustyypit ja determinismi ovatkin tavallista puhtautta heikompia, niiden lisääminen Uniiciin voisi silti olla mielekästä, jos niiden käyttö sallitaan vain moduulin sisäisissä funktioissa. Moduuli, jonka ulkoinen rajapinta ei vaadi Jolinen omistustyyppejä eikä halua muillakaan tavoilla muokata ei-uniikkeja parametrejaan, näyttäytyy ulospäin puhtaana, vaikka sen toteutuksessa saatetaankin käyttää epäpuhtaita ominaisuuksia.

Tällaisen moduulin voisi periaatteessa kääntää puhtaalle kohdekielelle yhtenä kokonaisuutena, vaikka sen sisäisiä funktioita ei voisikaan kääntää ”yksi yhteen”. Tämän voisi toteuttaa esimerkiksi lisäämällä moduulin ”sisäistä tilaa” esittävän erikoisparametrin moduulin sisäisiin funktioihin luvun 6 alussa mainitulla tavalla.

7 Yhteenveto

Uniic muuntaa imperatiivisesti kirjoitetut funktiot funktionaaliselle kohdekielelle ”yksi yhteen”, eli parametrilistaa muuttamatta ja funktion puhtautta menettämättä. Muunnos, joka perustuu Appelin [App98] sekä Kelseyn ja Richardin [Kel95] esittämiin tekniikoihin, analysoi imperatiivisen ohjelman vuokaaviota ja muuttaa silmukat rekursioksi. Näennäiset muuttujan ylikirjoitukset tulkitaan muuttujan uuden version laskemiseksi.

Uniicin tyyppijärjestelmä on sekä lähde- että kohdekielessä oleellisesti sama Hindley–Milner-tyypinpäättelyyn perustuva uniikkityypitysjärjestelmä. Järjestelmän uniikkityypitys perustuu de Vriesin jatkokehittämään versioon [deV08] Cleanin tyyppijärjestelmästä [BaS93]. Uniiciin lisättiin lainaus, joka tekee uniikkityypityksen käytöstä mukavampaa automatisoimalla uniikin arvon muuttuneen version eksplisiittisen vastaanottamisen jokaisen funktiokutsun jälkeen.

Uniic todettiin mielekkääksi laajennokseksi funktionaaliseen ohjelmointikieleen, vaikka jatkokehitettävääkin on paljon. Keskeiset jatkokehityksen kohteet ja nähtävissä olevat vaikeudet liittyvät muuttuvien ja varsinkin syklisten tietorakenteiden käsittelyyn kohdekielen uniikkityypityksen puitteissa. Lähdekielen puhtaudesta voitaisiin tinkiä hallitusti moduulien sisäisissä funktioissa [CW03], mutta muunnos ei olisi tämän jälkeen enää ”yksi yhteen” näiden sisäisien funktioiden osalta. Uniikkityypityksen korvaaminen monadeilla [AJ01] on myös eräs mahdollinen kehityssuunta.

Uniicista on prototyyppitoteutus, joka esitellään lyhyesti liitteessä 2.

Lähteet

Ach11 Achten, P., A Concise Guide to Clean StdEnv. Tekninen raportti, University of Nijmegen, 2011. URL `http://www.mbsd.cs.ru.nl/publications/papers/2010/CleanStdEnvAPI.pdf`.

AJ01 Achten, P. ja Jones, S., Porting the Clean Object I/O Library to Haskell. Teoksessa *Implementation of Functional Languages*, Mohnen, M. ja Koopman, P., toimittajat, osa 2011 sarjasta *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2001, sivut 194–213, URL `http://dx.doi.org/10.1007/3-540-45361-X_12`.

ALS07 Aho, A., Lam, M., Sethi, R. ja Ullman, J., *Compilers: principles, techniques, & tools*. Pearson international edition. Pearson/Addison Wesley, 2007.

AP01 Achten, P. ja Peyton-Jones, S. L., Porting the Clean Object I/O Library to Haskell. *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, IFL '00, London, UK, 2001, Springer-Verlag, sivut 194–213.

App98 Appel, A. W., SSA is functional programming. *SIGPLAN Not.*, 33,4(1998), sivut 17–20.

Boy01 Boyland, J., Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31,6(2001), sivut 533–553.

BaS93 Barendsen, E. ja Smetsers, S., Conventional and uniqueness typing in graph rewrite systems. *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, London, UK, 1993, Springer-Verlag, sivut 41–51, URL `http://dl.acm.org/citation.cfm?id=646831.707715`.

BS96 Barendsen, E. ja Smetsers, S., Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 1996, sivut 579–612.

CPN98 Clarke, D. G., Potter, J. M. ja Noble, J., Ownership types for flexible alias protection. *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, New York, USA, 1998, ACM, sivut 48–64.

CW03 Clarke, D. ja Wrigstad, T., External uniqueness is unique enough. *European Conference for Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2003, sivut 176–200.

deV08 de Vries, E., *Making Uniqueness Typing Less Unique*. Väitöskirja, Trinity College Dublin, Ireland, 2008.

GCC13 GNU Compiler Collection (GCC) Internals, 2013. `http://gcc.gnu.org/onlinedocs/gccint/`

Has10 Haskell 2010 Language Report, 2010. `http://www.haskell.org/onlinereport/haskell2010/`

HHP07 Hudak, P., Hughes, J., Peyton-Jones, S. ja Wadler, P., A history of Haskell: being lazy with class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, USA, 2007, ACM, sivut 12–1–12–55.

Hin01 Hind, M., Pointer analysis: haven't we solved this problem yet? *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, New York, USA, 2001, ACM, sivut 54–61.

Hog91 Hogg, J., Islands: aliasing protection in object-oriented languages. *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, New York, USA, 1991, ACM, sivut 271–285.

Kel95 Kelsey, R. A., A correspondence between continuation passing style and static single assignment form. *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, New York, USA, 1995, ACM, sivut 13–22.

LHJ95 Liang, S., Hudak, P. ja Jones, M., Monad transformers and modular interpreters. *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, New York, NY, USA, 1995, ACM, sivut 333–343, URL `http://doi.acm.org/10.1145/199448.199528`.

LLV13 LLVM Language Reference Manual, 2013. `http://llvm.org/docs/LangRef.html`

OAC04 Odersky, M., Altherr, P., Cremet, V., Dragos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihalyov, N., Schinz, M., Stenman, E., Spoon, L. ja Zenger, M., An overview of the scala programming language. Tekninen raportti IC/2004/64, EPFL Lausanne, Switzerland, 2004.

Oka99 Okasaki, C., *Purely functional data structures.* Cambridge University Press, 1999.

Pie02 Pierce, B. C., *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

Pie05 Pierce, B. C., toimittaja, *Advanced Topics in Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2005.

PvE02 Plasmeijer, R. ja van Eekelen, M., Clean Language Report Version 2.1. Tekninen raportti, University of Nijmegen, 2002.

Qui12 Quine-McCluskey algorithm (Java), 2012. Lähde on esimerkkitoteutus, joka on sittemmin poistettu verkosta. Linkki johtaa arkistoituun versioon. `https://web.archive.org/web/20120204174421/http://en.literateprograms.org/Quine-McCluskey_algorithm_(Java)`

TeA05 Terauchi, T. ja Aiken, A., Witnessing side-effects. *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming. ACM*, 2005, sivu 115.

VPJ10 Vytiniotis, D., Peyton Jones, S. ja Schrijvers, T., Let should not be generalized. *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI ’10, New York, NY, USA, 2010, ACM, sivut 39–50.

Liite 1. Imperatiivisen ohjelman muuttaminen vuoverkoksi

Seuraavassa esitellään tarkasti luvussa 3.2 mainittu algoritmi [ALS07, s. 357] abstraktin syntaksipuun muuntamiseksi vuoverkoksi.

Määritellään rekursiivinen aliohjelma `TrExprInto(rv, E)`, joka muuntaa lähdekielen abstraktin syntaksipuun lausekkeen `E` listaksi peruskomentoja, joiden vaikutus on sijoittaa muuttujaan `rv` lausekkeen `E` tulos. Koska vuoverkon komentojen parametreina voi olla vain muuttujia, algoritmi tuottaa lausekkeen jokaisen välituloksen tallentamiseksi tuoreen muuttujan apufunktiossa `TrExprOrVar`.

Algoritmi 11. Lausekkeiden muuntaminen vuoverkkokomennoiksi

```
TrExprInto(rv, E):
  jos E = Const[c]:
    palauta [rv := c]

  jos E = Var[v]:
    palauta [rv := v]

  jos E = MakeTuple(E[1], ..., E[n]):
    jokaiselle i välillä 1..n:
      (v[i], C[i]) = TrExprOrVar(E[i])
    palauta C[1] + ... + C[n] + [rv := (v[1], ..., v[n])]

  jos E = Call(F, A[1], ..., A[n]):
    (vf, Cf) = TrExprOrVar(F)
    jokaiselle i välillä 1..n:
      (a[i], C[i]) = TrExprOrVar(A[i])
    palauta Cf + C[1] + ... + C[n] + [rv := vf(a[1], ..., a[n])]

TrExprOrVar(E):
  jos E = Var[v]:
    palauta (v, [])
  muuten:
    rv = tuore muuttuja
    palauta (rv, TrExprInto(rv, E))
```

□

Lauseiden kääntämisessä on samankaltainen ajatus kuin lausekkeiden kääntämisessä. Tuoreiden apumuuttujien lisäksi lauseiden kääntämisessä tarvitaan tuoreita otsakkeita hyppykomentojen kohteiksi. Lauseiden kääntämiseksi määritellään rekursiivinen algoritmi `TrStmt(S)`, joka tuottaa abstraktin syntaksipuun lauseesta `S` listan, jonka alkiot voivat olla otsakkeita, peruskomentoja ja hyppykomentoja. Apufunktio `TrJump(C, L)` tuottaa hyppykomennon otsakkeeseen `L`, ellei komentojono `C` jo pääty hyppykomentoon.

Algoritmi 12. Lauseiden muuntaminen vuoverkkokomennoiksi

```
TrStmt(S):
  jos S = Set(v, E):
    palauta [TrExprInto(v, E)]

  jos S = Return(E):
    (rv, C) = TrExpr(rv, E)
    palauta C + [return rv]

  jos S = SplitTuple(v1, ..., vn, E):
    (rv, C) = TrExpr(rv, E)
    palauta C + [(v1, ..., vn) = rv]

  jos S = If(E1, S2, S3):
    L2, L3, L4 = tuoreita otsakkeita
    v1 = tuore muuttuja
    (v1, C1) = TrExpr(E1)
    C2 = TrStmt(S2)
    C3 = TrStmt(S3)
    J2 = TrJump(C2, L4)
    palauta C1 +                              (ehdon laskenta)
            [if v1 goto L2 else goto L3] +
            [L2] + C2 + J2 +                  (then-osa)
            [L3] + C3 +                       (else-osa)
            [L4]
```

```
  jos S = While(E1, S2):
    L1, L2, L3 = tuoreita otsakkeita
    (v1, C1) = TrExpr(E1)
    C2 = TrStmt(S2)
    J2 = TrJump(C2, L3)
    palauta [L1] + C1 +                         (toistoehdon laskenta)
            [if v1 goto L2 else goto L3] +
            [L2] + C2 + J2 +                    (silmukan runko)
            [L3]                                (toiston lopetus)

  jos S = Block(S1, ..., Sn):
    palauta [TrStmt(S1), ..., TrStmt(Sn)]

  jos S = E:
    rv = tuore muuttuja
    palauta [TrExprInto(rv, E)]

TrJump(C, L):
  jos C:n viimeinen komento on hyppykomento:
    palauta []
  muuten:
    palauta [goto L]
```

□

Algoritmin `TrStmt` tuottama komentojono ei vielä sellaisenaan kelpaa peruslohkoihin pilkottavaksi, vaan se tarvitsee vielä muutaman siivoustoimenpiteen

Algoritmi 13. Lähdekielen muuttaminen vuoverkoksi

1. Aloitetaan algoritmin `TrStmt` tuloksesta.
2. Ohjelman alkuun lisätään tuore otsake.
   Perustelu: `TrStmt` ei tiedä, milloin se käsittelee ohjelman alkua, joten se ei välttämättä ole tuottanut siihen otsaketta.
3. Ohjelman loppuun lisätään komennot `[v := (), return v]`, missä `v` on tuore muuttuja.
   Perustelu: näin ohjelmoija voi kirjoittaa funktion, jolla ei ole mielekästä pa-

luuarvoa, ilman `return`-komentoa.

4. Lisätään komento `goto L` jokaisen sellaisen otsakkeen `L` eteen, jonka edessä ei jo ole jotain hyppykomentoa.
   Perustelu: tämä takaa, että jokainen lohko päättyy hyppykomentoon. `TrStmt` ei vielä lisää esimerkiksi `while`:n alkuun hyppäävää komentoa `while`ä edeltävään lohkoon.

5. Poistetaan iteratiivisesti kaikki lohkot, joihin ei ole hyppykomentoa.
   Perustelu: jos esimerkiksi kesken silmukan on `return`-lause, `TrStmt` tuottaa sen jälkeen turhia hyppykomentoja.

6. Ohjelma jaetaan osiin jokaisen otsakkeen kohdalta. Osat ovat vuoverkon peruslohkoja. Ensimmäinen peruslohko on vuoverkon alkulohko. □

Liite 2. Uniicin prototyyppitoteutus

Uniicin prototyypin lähdekoodin voi ladata seuraavasta osoitteesta.

`https://github.com/mpartel/uniic-proto`

Toteutus on kirjoitettu Scalalla [OAC04], ja se on noin 8500 riviä pitkä (kommentit ja tyhjät rivit mukaan lukien). Testitapauksia on 272, ja testikoodi on noin 4200 riviä pitkä.

Toteutuksen koodi on jaettu pakkauksiin, joista tärkeimmät ovat seuraavat.

- `boolalg` – De Vriesin uniikkityypitysjärjestelmän tarvitseman Boolen algebran toteutus.
- `flow` – Vuokaavioiden ja SSA-muodon esitys ja käsittely.
- `fun` – Funktionaalisen kohdekielen puuesitys ja muunnos verkoksi.
- `grs` – Funktionaalisen kohdekielen verkkoesitys ja sen käsittely.
- `imp` – Imperatiivisen lähdekielen esitys ja muunnos vuoverkoksi.
- `parsers` – Lähde- ja kohdekielen syntaksin jäsentäjät.
- `stdlib` – Prototyypin pieni standardikirjasto.
- `types` – Tyyppijärjestelmän yhteiset osat.