Haskell-kielen I/O-järjestelmä

Mika Miettinen

Helsinki 3. huhtikuuta 2004

Ohjelmointikielten periaatteet, kevät 2004
seminaariesitelmä, 7.4.2004

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

I/O:n järjestäminen on eräs funktionaalisten ohjelmointikielten perinteisistä ongelmista. Eräs syy tähän on, että I/O rikkoo funktionaalisen ohjelmointimallin perusominaisuuden: riippumattomuuden ohjelman "ulkoisista" olosuhteista (siis sivuvaikutusten puuttumisen). Toisaalta ongelmana on myös suoritusjärjestys: I/O (kuten myös imperatiivinen ohjelmointi) on oleellisesti riippuvainen siitä järjestyksestä missä eri toiminnot suoritetaan. I/O ja imperatiivinen ohjelmointi yleensä onkin mahdollista nähdä jonona toimintoja, jotka suoritettuna saavat aikaan muutoksia järjestelmän tilassa. Laiskat funktionaaliset ohjelmointikielet sen sijaan ovat varsin sallivia järjestyksen suhteen; lausekkeet saa arvioida missä järjestyksessä hyvänsä kunhan niiden arvot ovat saatavilla silloin kun niitä tarvitaan. I/O-järjestelmän tulisikin kyetä ratkaisemaan kaksi erillistä ongelmaa:

- Toimintoja pitäisi pystyä ketjuttamaan, jotta ne myöhemmin voitaisiin "suorittaa järjestyksessä".
- Funktioiden tulisi säilyttää matemaattinen luonteensa: ne eivät saisi aiheuttaa sivuvaikutuksia ja niiden palauttaman arvon pitäisi olla ennustettavissa.

Epäpuhtaissa funktionaalisissa kielissä tätä ongelmaa ei ole. Esimerkiksi Scheme-kielessä [KCR98] I/O tapahtuu niinsanottujen *porttien* (engl. *ports*) avulla. Portti on Scheme-olio, jonka kautta voidaan lukea (input-portit) tai jonka kautta voidaan kirjoittaa (output-portit). Nyt esimerkiksi merkin lukeminen syötteestä tapahtuu (kuten voi arvatakin) funktiolla, jolle annetaan parametrina portti, ja joka palauttaa syötteen seuraavan merkin (Schemen funktio `read-char`). Funktio ei enää käyttäydy "matemaattisesti", mutta epäpuhtaissa kielissä tämä on sallittua. I/O-toiminnot voidaan myös suorittaa halutussa järjestyksessä `begin`-notaation avulla.

Puhtaasti funktionaalisissa kielissä (joihin Haskell kuuluu), eräs lähestymistapa ongelmaan ovat niinsanotut *dialogit* (engl. *dialogues*) [Don85]. Dialogi on kahden osapuolen vuoropuheluabstraktio. Tässä tapauksessa osapuolet

ovat ohjelma ja käyttöjärjestelmä. Varhaisimmissa Haskell-versioissa käytettiin juuri tätä tapaa. Kommunikointi ulkomaailman kanssa tapahtuu viestien avulla: ohjelma lähettää käyttöjärjestelmälle *pyyntöjä* ja saa takaisin *vastauksia*.

Haskell-kielessä ongelma on nykyään ratkaistu *monadien* avulla. Perusideana on erottaa *arvot*, joita funktiot palauttavat, *toiminnoista*, jotka suoritettuna saavat aikaan I/O:n. Tässä menetelmässä funktiot palauttavat (ja saavat parametriksi) arvoja, jotka edustavat toimintoja. Nämä toiminnot eivät kuitenkaan itsessään tee mitään; niitä täytyy eksplisiittisesti *kutsua* (engl. *invoke*), jotta toiminto suoritettaisiin. Toimintojen etu on siinä että ne ovat muiden arvojen kanssa täysin samassa asemassa: niitä voidaan esimerkiksi antaa parametreina ja tallettaa tietorakenteisiin. Toimintoja voi vaikkapa tallettaa listaksi, jolloin saadaan aikaan jono toimintoja, jotka myöhemmin voidaan suorittaa järjestyksessä.

2 Haskell

Haskell on puhdas ja laiska (engl. *non-strict*) funktionaalinen ohjelmointikieli, jonka kehitys aloitettiin vuonna 1987 korjaamaan "Baabelin sekaannusta": siihen aikaan oli olemassa useita puhtaita, laiskoja funktionaalisia — perusteiltaan melko samanlaisia — kieliä. Tavoitteena oli kehittää kieli, jota voisi paitsi käyttää realistisissa ohjelmointitehtävissä; ja joka myös toimisi perustana myöhemmälle tutkimukselle.

Haskell on vahvasti (ja staattisesti) tyypitetty kieli, joka käyttää niinsanottua Hindley-Milner -tyyppijärjestelmää. Tämä tarkoittaa erityisesti sitä, että jokaisen lausekkeen tyyppi voidaan automaattisesti päätellä. Kielen syntaksissa tämä näkyy niin, että ohjelmoijan ei ole pakko merkitä tyyppi-informaatiota näkyviin. Toinen syntaksin leimaa-antava piirre on koodin muotoilu: Haskell käyttää samanlaista järjestelmää kuin Python, jossa lähdekoodin ulkoasu määrää ohjelman lohkorakenteen.

Kuten Prologissa, myös Haskellissa tunnusten käyttötarkoitus määräytyy kir-

joitustavan perusteella. Muuttujien nimet alkavat pienellä alkukirjaimella, ja konstruktorien nimet suurella alkukirjaimella. Toinen Prologista tuttu piirre on hahmonsovitus. Tämä näkyy selvästi Haskell-ohjelmissa: funktiot määritellään usein "paloittain". Esimerkiksi funktionaalisten kielten perusfunktio `map` määritellään seuraavalla tavalla:

```
map :: [a] -> (a -> b) -> [b]
map [] f = []
map (x:xs) f = (f x) : map f xs
```

3 I/O Haskell-kielessä

Puhtaiden funktionaalisten kielten kompastuskivenä on aina ollut I/O. Tämä näkyy myös Haskellin historiassa: I/O:n toteutuksessa on käytetty kahta täysin erilaista tapaa. Alusta lähtien versioon 1.2 [HJW92] saakka totetus oli dialogipohjainen. Nykyisin käytettävä monadehin perustuva totetus otettiin mukaan kielen versiossa 1.3. Tässä luvussa esitellään lähemmin näitä kahta tapaa ja tutkitaan kuinka ne ratkaisevat johdannossa määritellyn "funktionaalisen I/O:n" ongelman.

3.1 I/O dialogien avulla

I/O:n dialogitoteutuksessa pääohjelma (joka Haskellissa on `main`-niminen funktio) kommunikoi käyttöjärjestelmän kanssa viestien välityksellä. Pääohjelma lähettää käyttöjärjestelmälle listan viestejä, ja saa vastaukset näihin listana. Tarkemmin sanoen `main` on tyyppiä `[Response] -> [Request]`. Edelleen `Request` on tietotyyppi, joka määrittelee millaisia käyttöjärjestelmälle lähetettävät pyynnöt ovat. `Response` taas on tietotyyppi, joka määrittelee käyttöjärjestelmän palvelupyyntöihin lähettämät vastaukset.

Tämä lähestymistapa selvästikin ratkaisee edellä esitetyn I/O-ongelman. Ensinnäkään varsinaisia I/O-funktioita ei tässä mallissa ole; on vain pyyntöjä ja vastauksia näihin. Siten epäpuhtaiden funktioiden ongelmaa ei edes esiinny.

```
main ~(Success : ~((Str userInput) : ~(r4 : _)))=
  [ AppendChan stdout "please type a filename\n",
    ReadChan stdin,
    ReadFile name,
    AppendChan stdout (case r4 of
                          Str contents     -> contents
                          Failure ioerror  -> "can't open file")
  ] where (name : _) = lines userInput
```

Kuva 1: Yksinkertainen I/O dialogien avulla

Toisaalta pyynnöt (ja vastaukset) ovat listassa ja listan alkioilla on luonnollinen järjestys.

Kuvassa 1 on ohjelma, joka havainnollistaa tätä vanhaa toteutusta. Ohjelma pyytää käyttäjältä tiedoston nimen, ja tulostaa tämän jälkeen tiedoston sisällön ruudulle. Ohjelmassa `main`-funktion parametrina oleva listahahmo esittää sitä listaa, joka tulee vastauksena ohjelman lähettämiin pyyntöihin. Tästä aiheutuva "muna-kana"-ongelma (vastaukset voivat olla olemassa vasta pyyntöjen *jälkeen*) estetään käyttämällä *laiskaa* hahmoa. Tämä tarkoittaa, että arvojen sidonta tapahtuu vasta silloin kun näin sidottua arvoa todella tarvitaan. Näin toimien ohjelma on aina hieman edellä vastauksia: esimerkiksi `ReadChan stdin` -pyyntö ehditään lähettää ennen kuin siihen tulevan vastauksen `userInput`-parametria tarvitaan.

Tässä menetelmässä on useita ongelmia. Selvin näistä on puhtaasti tyylillinen: ohjelma ei sisällä mitään näkyvää toiminnallista osaa. Toinen ongelma on pääohjelman parametrina olevan listahahmon koon kasvaminen jokaisen lisättävän pyynnön myötä. Tämä johtuu siitä, että parametrina saatavan vastauslistan on vastattava *täsmällisesti* varsinaisen toiminnan määrittelevää pyyntölistaa.

3.2 Monadinen I/O

Monadien käyttöä I/O-järjestelmän toteuttamisessa käsittelivät ensimmäisenä Wadler ja Peyton Jones [PJW93]. Tämä työ oli pohjana Haskellin nykyiselle monadeihin perustuvalle toteutukselle. Monadisen I/O:n ymmärtämisessä on oleellista tehdä ero *arvojen* ja *toimintojen* välille. Tyyppillisestihän funktionaalisessa ohjelmoinnissa ollaan kiinnostuneita nimenomaan arvoista: jokaisella lausekkeella on arvo, ja ohjelman suoritus on yksinkertaisimmillaan vain "pääohjelman" arvon laskemista. I/O:ssa ovat kuitenkin keskeisessä osassa toiminnot.

Toimintojen voidaan ajatella olevan omassa maailmassaan eläviä "epäfunktionaalisia" abstraktioita. Kaikki ovat pohjimmiltaan samanlaisia: kutsuttaessa ne "tekevät jotakin"[1] ja palauttavat lopulta jonkin arvon. Täsmällisemmin toiminnot ovat tyypin `IO a` arvoja, missä `a` on toiminnon palauttaman arvon tyyppi. Yksinkertainen esimerkki on syötteen seuraavan merkin palauttava funktio `getChar`: sen palauttama arvo on tyyppiä `IO Char`. Kyseessä on siis toiminto, joka suoritettaessa lukee seuraavan merkin syötteestä ja palauttaa tämän.

Tässä mallissa I/O-funktiot säilyttävät "puhtaan" luonteensa. Kielen näkökulmasta kaikki samantyyppiset toiminnot ovat samankaltaisia: ne tekevät jotakin ja palauttavat lopulta määrätyn tyyppisen arvon. Esimerkiksi kaikki `IO Char` -toiminnot ovat samanlaisia; niiden kutsuminen saa aikaan tyyppiä `Char` olevan arvon palauttamisen.

I/O-funktioiden puhtaudesta ei itsessään ole vielä paljon iloa. Tarvitsemme vielä keinon ketjuttaa toimintoja järjestyksessä suoritettaviksi. Tämä tapahtuu erityisen `do`-notaation avulla:

```
do putStr "Please type your name: "
   name <- getLine
   putStr ("Hello, " ++ name ++ "!\n")
```

[1] Käytännössä ne tekevät yhden tai useampia käyttöjärjestelmäkutsuja. Nämä puolestaan saavat esimerkiksi aikaan muutoksia jossakin puskurissa.

```
main = do
        putStr "please type a filename\n"
        name <- getLine
        catch (echoFile name)
              (\e -> putStr "can't open file\n")
        where
          echoFile fname = do
                           contents <- readFile name
                           putStr contents
```

Kuva 2: Yksinkertainen I/O monadien avulla

Syntaksi näyttää perinteiseltä imperatiiviselta kieleltä: "lauseet" suoritetaan järjestyksessä ylhäältä alas ja toisella rivillä käytetään "sijoitusoperaatiota" `<-`. Tämän perusteella voidaankin sanoa, että monadit luovat Haskelliin eräänlaisen imperatiivisen alikielen. Mutta kuten myöhemmin tulemme huomaamaan, se on vain näennäistä.

I/O-operaatiot voivat tietenkin aiheuttaa myös virhetilanteita: esimerkiksi haluttua tiedostoa ei ole olemassa tai oikeudet sen käsittelyyn puuttuvat. I/O:n mahdollisesti aiheuttamat virheet käsitellään poikkeusten avulla. Jokainen I/O-operaatio voi aiheuttaa poikkeuksen sen sijasta että palauttaisi tuloksen. Poikkeukset ovat abstraktia tyyppiä `IOError`: käyttäjälle on näkyvissä vain tyypin nimi, ei sen konstruktoreita. Poikkeuksen tyyppiä voi kuitenkin kysellä tarkemmin käyttäen `IO`-kirjaston funktioita. Käyttäjä voi määritellä `catch`-funktion avulla poikkeuskäsittelijän, joka käsittelee kaikki jossakin toiminnossa tai toimintojonossa tapahtuvat virheet. Poikkeuskäsittelijät eivät kuitenkaan ole millään tavalla selektiivisiä, vaan käsittelijän on käsiteltävä kaikentyyppiset virheet. Poikkeuksen propagointi on eksplisiittisesti ilmaistava.

Kuvan 2 ohjelma esittää kuinka edellisen kappaleen esimerkki voitaisiin nykyään toteuttaa. Tämä ohjelma on huomattavasti selkeämpi ja muistuttaa suuresti imperatiivista tyyliä. Koska Haskellissa listat ovat laiskoja, voidaan koko tiedoston sisältö lukea kerralla merkkijonoksi. Tämä on tietenkin vain

näennäistä, ja todellisuudessa tiedostoa luetaan sitä mukaa kun `putStr` sen sisältöä tarvitsee.

4 Monadit

Monadi-käsite on peräisin matematiikasta kategoriateorian alueelta, ja on eräänlainen monoidin yleistys. Kategoriateoriaa on perinteisesti sovellettu tyypitetyn lambdakalkyylin formaalin semantiikan määrittelyyn. Eugenio Moggi [Mog89] tutki perinteisten imperatiivisten piirteiden semanttista määrittelyä, ja huomasi monadien sopivan tähän tarkoitukseen. Philip Wadler [Wad90] sovelsi ensimmäisenä näitä ideoita käytännön ohjelmointiin.

Haskell-kielessä monadi on yksinkertaisesti tietotyyppi, jolle on määritelty joukko operaatioita. Haskellin standardikirjastossa on jo määriteltykin sopiva tyyppiluokka nimeltään `Monad` seuraavalla tavalla:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

Yleisestikin on hyödyllistä ajatella monadia toimintona, joka palauttaa tietyntyyppisen arvon. Siis `return` on funktio joka yksinkertaisesti muuttaa annetun arvon vastaavaksi toiminnoksi. Funktio `(>>=)` määrittelee toimintojen muuttamisen toisenlaisiksi toiminnoiksi. Sen ensimmäinen parametri on monadi, joka palauttaisi `a`-tyyppisen arvon. Toinen parametri on funktio, joka muuttaa `a`-tyypin arvot tyyppiä `b` olevan arvon palauttavaksi toiminnoksi. Lopputuloksena oleva toiminto palauttaa tyyppiä `b` olevan arvon.

Tämä tyyppiluokka määrittelee kaikille monadeille yhteisen toiminnallisuu-

den. Esimerkiksi tyyppi `IO` on standardikirjastossa määritelty luokan `Monad` ilmentymäksi. Siten toimintojen yhdistäminen tapahtuu nimenomaan tämän luokan funktioiden avulla. Edellä nähty `do`-notaatio on itse asiassa vain syntaktista sokeria, joka palautuu operaatioihin `(>>=)` ja `(>>)` [PJ03, s. 26–27]. Vaikka edellä nähdyt monadiset ohjelmat muistuttavat perinteisiä imperatiivisia ohjelmia, ei Haskellin puhdas funktionaalisuus kuitenkaan tuhoudu. Se on vain "piilossa" `do`-notaation takana.

5 Yhteenveto

I/O-järjestelmän toteutus puhtaissa funktionaalisissa kielissä on vaikeaa. Tämä johtuu pohjimmiltaan siitä, että I/O rikkoo funktionaalisuuden peruspilarit: sivivaikutusten puuttumisen ja vapaan suoritusjärjestyksen. Haskell kielen toteutuksessa käytettiin aluksi dialogeja, joka kuitenkin saa aikaan ongelmia.

Uudempi monadinen I/O perustuu abstraktioon, jolla erotetaan toisistaan funktionaalisille ohjelmointikielille ominaiset arvot ja I/O:ssa keskeiset toiminnot. Näimme että tämä käsite tuo mukanaan selviä etuja ja tekee ohjelmista hyvin selkeitä. Vaikka monadit tuovatkin Haskelliin eräänlaisen imperatiivisen alikielen, se tapahtuu silti puhtaille funktionaalisille kielille ominaisella tavalla.

Lähteet

Don85 O'Donnell, J., Dialogues: A Basis for Constructing Programming Environments. *ACM Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 1985.

HJW92 Hudak, P., Peyton Jones, S., Wadler, P. (editors), Report on the Programming Language Haskell, Version 1.2. *ACM SIGPLAN Notices* 27, 15 (1992).

KCR98 Kelsey, R., Clinger, W., Rees, J. (editors), Revised 5th Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 33, 9 (1998), 26–76.

Mog89 Moggi, E., Computational lambda-calculus and monads. *IEEE Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

PJ03 Peyton Jones, S. (editor), Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming* 13, 1 (2003).

PJW93 Peyton Jones, S., Wadler, P., Imperative functional programming. *20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

Wad90 Wadler, P., Comprehending Monads. *ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.