Haskell ohjelmointikielen tyyppijärjestelmä

Sakari Jokinen

Helsinki 19. huhtikuuta 2004

Ohjelmointikielten perusteet - seminaarityö

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

1 Johdanto

Tyyppien avulla voidaan rajoittaa datalle suoritettavia operaatioita semanttisesti oikeisiin. Staattisesti tyypitetty ohjelma suorittaa tyyppien semanttisen tarkistuksen ennen ajoa. Staattinen tyypitys voi kuitenkin haitata koodin uudelleenkäyttöä koska lisäyksien joukossa voi olla uusia datatyyppejä ja valmiissa koodissa tyypit on jo määritelty [Car89].

Haskell ohjelmontikielen käyttämä tyyppijärjestelmä pyrkii mahdollistamaan koodin uudelleenkäytön sekä samalla säilyttämään staattisen tyypityksen. Haskell on puhtaasti funktionaalinen laiskasti evaluoiva ohjelmointikieli. Haskell 98 on stabiili versio haskelista. Stabiili tarkoittaa tässä sitä, että toteuttajat lupautuvat tukemaan kieltä tulevaisuudessa kuten se on määritelty [Jon98]. Haskell kielestä on tehty useita toteutuksia kuten GHC sekä Hugs, jotka kumpikin tarjoavat lisäyksiä Haskell 98 määrittelyyn. Tämä artikkeli viittaa jatkossa Haskell 98 kieleen.

Kappaleessa 2 käydään lävitse Haskell kielen syntaksia. Kappaleessa 3 käsitellään tyyppiluokkia. Kappaleessa 4 esitellään algebralliset tyypit. Kappale 5 käsittelee Haskell tyyppijärjestelmään kuuluvaa tyyppienpäättelyä.

2 Haskell

2.1 Yleinen syntaksi sekä semantiikka

Tämän kappaleen tarkoitus on esitellä artikkelin esimerkeissä käytetty Haskell syntaksi ja semantiikka. Haskell on funktionaalinen ohjelmointikieli. Tämä tarkoittaa sitä, että funktiot ovat ensimmäisen luokan olioita eikä sivuefektejä ole, eli kielessä ei voida asettaa muuttujaa tuhoavasti(destructive update). Funktioita

voidaan osittain applikoida (partial application).

```
x = f
```

Asettaa x:n arvoksi f.

```
(a,b) = (1,2)
c = (1,2)
```

Ensimmäinen lause asettaa a:n arvoksi 1 sekä b:n arvoksi 2 käyttäen hahmonsovitusta. Toinen lause asettaa c:n arvoksi kaksikon (1,2). (,) on kaksikkotyypin konstruktori, Konstruktoreilla voidaan rakentaa uusia arvoja, sekä käyttää purkamaan konstruktorilla luotuja arvoja.

```
a:st = [1,2,3]
```

: on listakonstruktori. [1,2,3] on syntaktista sokeria määritelmälle 1:(2:(3:[])). Lause asettaa a:lle arvon 1 sekä st:lle arvon 2:(3:[]).

```
let g = a in z
```

let lauseen oikea puoli g = a määrittelee funktion g = a let lauseen loppuosassa z.

Erikoismerkeillä +,-,*, ... nimetyt funktiot ovat infix muodossa, prefix muotoon ne saa ympäröimällä ne suluilla.

```
a * b == (*) a b
```

2.2 Tyypit

Kaikilla lauseilla on tyyppi Haskell kielessä. Lauseen tyyppi ilmaistaan tyyppinotaation avulla.

```
q :: Int
```

q on tyyppiä Int

```
f :: Int -> Char->Int
```

f on tyyppiä funktio Int tyypistä funktioon Char tyypistä Int tyyppiin, eli ekvivalentti tyypin Int -> (Char -> Int) kanssa.

```
l :: [Int]
```

l on tyyppiä lista joka koostuu Int tyypeistä.

```
t :: (Int,Int,Char)
```

t on tyyppiä monikko jonka ensimmäinen jäsen on tyyppiä Int, toinen jäsen tyyppiä Int, sekä kolmas jäsen tyyppiä Char. Yllä Int ja Char tyypit ovat tyyppivakioita.

```
v :: a -> a
```

v:n tyyppi on funktio joltain tyypiltä a samalle tyypille a olevia arvoja. a v:n tyypissä on tyyppimuuttuja se voi viitata mihin tahansa tyyppiin.

```
c :: (Ctx a) => a->a
```

c on tyyppiä funktio tyypistä a tyyppiin a siten että tyyppi a kuuluu tyyppiluokkaan Ctx (kts kappale 3). a:n sanotaan olevan rajoitettu tyyppimuuttuja ja Ctx on a:n konteksti [Jon98].

3 Tyyppiluokat

Parametrinen polymorfismi tarkoittaa sitä että funktio on määritelty usealla eri tyypille ja toimii samalla tavalla kaikille. Ad-hoc polymorfismi tarkoittaa sitä että funktio on määritelty usealla eri tyypille ja toimii kaikille eri tavoin [WaB88].

```
square x = x * x
square 3
square 3.14
```

square funktio on esimerkki parametrisestä polymorfismista. Sen odotetaan toimivan niin kokonaisluvuilla kuin reaaliluvuilla. * funktio on esimerkki ad-hoc polymorfismista. Sen toteutus on todennäköisesti erilainen kokonais -sekä rationaaliluvuilla.

Tyyppiluokat ovat Haskell kielen menetelmä toteuttaa ad-hoc polymorfismi [WaB88]. Samalla saadaan menetelmä parametrisen polymorfismin toteuttamiseksi. Tyyppiluokka määritteelee joukon funktioita jotka luokkaan kuuluvien tyyppien tulee toteuttaa. Jos jotain luokan funktiota ei toteuteta niin toteuttamaton funktio sidotaan undefined funktioon. Tästä ei anneta käännösvirhettä [Jon98].

```
class Num a where
(+), (*):: a->a->a
negate :: a->a
instance Num Int where
(+) = addInt
(*) = mullInt
negate = negInt
```

class Num määrittelee luokan Num jolle kuuluu (+),(-) sekä negate funktiot. instance Num Int määrittelee tyypin Int kuuluvaksi Num luokkaan ja samalla antaa

luokan funktioiden toteutukset.

```
square :: Num a => a->a
square x = x * x
```

square funktio voidaan määritellä tyyppiluokkien avulla siten että sen saamille argumenteille on toteutettu * funktio.

4 Algebralliset datatyypit

Algebralliset datatyypit (Algebraic Data Types) luodaan data avainsanan avulla.

```
data Tree = Nil |
Node Int Tree Tree
```

Komento data luo uuden algebrallisen datatyypin Tree. Esimerkissä Nil sekä Node ovat konstruktoreita.

```
Nil :: Tree
Node :: Int->Tree ->Tree -> Tree
```

Algebralliset tyypit voivat olla rekursiivisesti määriteltyjä kuten esimerkissä. Node konstruktori ottaa Tree tyyppisen argumentin sekä palauttaa Tree arvon.

Algebrallisen datatyypin määrittelyyn voidaan myös antaa tyyppimuuttujia, jolloin saadaan polymorfisia algebrallisia datatyypeejä. Tyyppimuuttujia voi rajoittaa kontekstilla samalla tavalla kuin polymorfisilla funktioilla.

```
data (Eq a) => Tree a = Nil |
Node a (Tree a) (Tree a)
```

Konstruktorien tyypit ovat tällöin

```
Nil :: (Eq a) => Tree a
Node :: (Eq a ) => a -> Tree a -> Tree a
```

Algebrallisia datatyyppejä voidaan käyttää hahmon sovitukseen kuten muitakin tyyppejä.

```
a = Node 1 Nil Nil
(Node b _ _) = a
```

b saa arvon 1.

5 Tyyppien päättely

5.1 Hindley-Milner tyyppijärjestelmä

Haskell käyttää tyyppien päättelyyn tyyppiluokilla laajennettua Hindley-Milner algoritmia [Jon98]. Hindley-Milner algoritmi on ratkeava algoritmi, jolla pystytään päättelemään ohjelmalle yleisin mahdollinen tyyppi käyttämättä mitään explisiittistä tyyppinotaatiota [Car89]. Hindley-Milner tyyppijärjestelmässä tyyppimerkinnöillä voidaan vain rajoittaa löydettyä yleisintä mahdollista tyyppiä.

Kun Hindley-Milner tyyppijärjestelmässä annetaan tyyppimuuttujalle tyyppi A niin kaikki tyyppimuuttujan esiintymät tyyppiympäristössä saavat saman tyypin. Tätä kutsutaan unifikaatioksi. Unifikaatio epäonnistuu kun yriteään asettaa samaa tyyppimuuttujaa kahdelle eri tyypille, tai yritetään asettaa tyyppimuuttujaan a tyyppi joka sisältää tyyppimuuttujan a [Car89]. Tyyppien päättely etenee unifikaation myötä.

Hindley-Milner tyyppijärjestelmä määrittää let lauseen tyypin seuraavasti; ensiksi let lauseen oikealle puolella annetaan tyyppi q. Seuraavaksi kaikki tyypissä q esiintyvät tyyppimuuttujat generalisoidaan. Generalisoinnissa kaikille tyypissä esiintyville tyyppimuuttujille annetaan universaalikvanttoroitu tyyppi, ellei niitä ole sidottu tyyppiympäristössä. Viimeiseksi let lauseen loppuosalle määritetään tyyppi.

f x = let g y = (y, y) in ...

Edellisessä let lauseessa ensiksi määritellään g:n tyyppi. a: a -> (a,a), jossa a on universaalikvanttoroitu, eli voi saada tyypikseen kaikkia tyyppejä [Jon98]. Jos lauseen g tyypissä q oleva tyyppimuuttuja a esiintyy tyyppiympäristössä joudutaan rajoittaamaan g:n polymorfisuutta.

f x = let g y z = ([x, y], z) in ...

g:n tyyppi ylläolevassa lauseessa, jos x:n tyyppi on a, on a->b->([a], b). g on monomorfinen tyyppimuuttujassa a. Monomorfisuudesta johtuu että kaikissa g:n applikaatioissa y:n tulee olla samaa tyyppiä [Jon98].

5.2 Haskell tyyppijärjestelmä

Haskell tyyppijärjestelmä poikkeaa paikoin tavallisesta Hindley-Milner tyyppijärjestelmästä: tyyppimuuttujien generalisointia on rajoitettu vahvemmin sekä tyyppiluokista johtuen joskus tarvitaan explisiittistä tyyppien merkintää.

Haskell tyyppijärjestelmässä on mahdollista että jollakin lauseella on moniselitteinen tyyppi. Haskell kieli kieltää moniselitteiset tyypit joten tällöin tarvitaan tyyppien explisiittistä merkintää.

let x = read “...” in show x

jossa

```
show :: Show a => a -> String
read :: Read a => String -> a
```

read saa argumenttina merkkijonon ja palauttaa sitä vastaavan arvon tyyppiä a. show ottaa arvon tyyppiä a ja palauttaa sitä vastaavan merkkijonon. let lauseen tyyppi on (Read a, Show a) => String. Sisäänrakennetut tyypit kuten Int sekä Bool kuuluvat kumpikin kumpaankin luokkaan joten lauseen tyyppi on moniselitteinen [Jon98].Tilanne voidaan korjata antamalla explisiittisesti x:lle tyyppi. Toinen tapa jolla Haskell voi selvitä moniselitteisistä tyypeistä on käyttää oletusarvoisia tyyppejä.

Lähteet

Jon98 Jones S. P.(editor), Haskell 98 Language and Libraries the Revised Report.

WaB88 Wadler P., Blott S., How to make ad-hoc polymorphism less ad-hoc, Proceedings of the 16th ACM SIGPLAN-SIGACTsymposium on Principles of programming languages, Pages: 60 - 76, 1989.

Car89 Cardelli L., Basic Polymorphic Type-checking, Science of computer programming 8/2 (april 1987) revised 6/21/88.