Funktionaalista grafiikkaa

Tommi Teistelä
totateis@jyu.fi

22.4.2008

Tiivistelmä

Haskellin kaltaisten funktionaalisten ohjelmointikielten ilmaisukyky mahdollistaa grafiikan koostamisen ”perinteistä” korkeammalla tasolla. Tässä paperissa pyritään esittelemään ”legacy-free” tapoja rakentaa Haskell-kielellä tietokonegrafiikkaa alunperin kirjassa *The Fun of Programming* ja Haskell-grafiikkakirjastossa *Pan* [1] esitettyjen ideoiden pohjalta.

1 Johdanto

Grafiikkaoperaatioiden toteuttaminen ”perinteisellä” ohjelmointikielellä (lue: C) johtaa usein lähdekoodiin, jonka uudelleenkäytettävyys on heikkoa. Sopivien ”korkeamman tason” ilmaisujen puutteessa samat operaatiot voi joutua pienien erojen takia toteuttamaan useita kertoja, ellei koodin selkeydestä, yksinkertaisuudesta ja/tai tehokkuudesta olla valmiita tinkimään.

Avoimen lähdekoodin grafiikkaeditori GIMP on kärsinyt tästä jo pitkään: alkuperäinen GIMP tuki kunnolla ainoastaan 32-bittisiä RGBA-bittikarttoja, ja projektin koodin määrän kasvaessa pitkään kaivattu tuki 48-bittisille RGBA-bittikartoille on yhä vaikeampi toteuttaa, puhumattakaan painotuotannossa käytettävistä CMYK-väreistä (Cyan, Magenta, Yellow, Key – joka tässä tarkoittaa mustaa) ja vielä harvinaisemmista värien esitystavoista. Grafiikkakoodin korvaajaksi on kehitetty yleisempää, graafipohjaista kirjastoa jo vuodesta 2000, joskin tätä kirjoittaessa kirjaston liittäminen GIMP:iin on vasta alullaan. Kehittäjien haluttomuus ”sotkea” vielä kehitteillä olevaa ohjelmaa tuella muille formaateille johti lopulta mm. CinePaint-projektin forkkaamiseen GIMP:n koodista [3].

Haskell-kieli tarjoaa joitakin ominaisuuksia, joiden avulla voidaan kirjoittaa huomattavasti selkeämpää ja uudelleenkäytettävämpää grafiikkakoodia. Näistä tärkeimpiä ovat korkeamman kertaluokan funktiot ja tyyppiluokat.

2 Kuvatiedon esittäminen

Kuva täytyy pystyä esittämään jotenkin ohjelmassa. Ensimmäisenä kuvan esitysmuotona saattaa mieleen tulla mieleen lista kuvan muodostavien pikselien väriarvoista:

```
type Color = (Float, Float, Float, Float)
type Image = [[Color]]
```

Värin esitys on yksinkertainen ja todettu toimivaksi: *Color*:in komponentit ovat punainen, vihreä, sininen ja *alpha*, joka määrittää värin "läpikuultavuuden" (0 = läpinäkyvä, .. , 1 = ei läpinäkyvä).

Toisaalta kuvan esitys näin vastaa melko hyvin sitä, miten kuva oikeasti säilytetään bittikarttamuodossa ja miten sitä on yleensä käsiteltykin ohjelmointikielissä. Tällaisen kuvatiedon käsittely on kuitenkin osa edellä mainittua ongelmaa, sillä se ei abstrahoi hyvin kuvan rakennetta ja tuottaa hankaluuksia useampien kuvien koostamisessa; listarakenne on otettava huomioon monessa vaiheessa. Kokeillaan jotakin *funktionaalisempaa*:

```
type Point = (Float, Float)
type Image = Point -> Color
```

Kuva on nyt funktio, joka kuvaa tietyn kuvapisteen sen väriksi. Tämä poistaa "pikseli"-käsitteen kokonaan tästä vaiheesta. Piste-tyypin käyttöä voidaan vielä helpottaa toteuttamalla sille joitakin alkeellisia vektorioperaatioita.

Muutetaan vielä *Image* tyyppikonstruktoriksi niin, että voimme käyttää erilaisia värityyppejä:

```
type Image t = Point -> t
type ImageC = Image Color
```

Nyt kuva voi koostua mistä tahansa tyypistä t, kunhan se toteuttaa kuvaoperaatioiden tarpeisiin riittävästi perusoperaatioita, kuten esim. Haskellin Fractional-luokkaan kuuluvat tyypit, ja *ImageC* on nyt yllä esiteltyä *Color*-tyyppiä käyttävä kuva. Erilaisten värityyppien lisäksi ainakin *Boolean* voi olla hyödyllinen kuvamaskien muodostamisessa (True = osa kuvaa, False = ei osa kuvaa) [2, s. 132].

3 Kuvien koostaminen

Yksi funktionaalisen kielen etu on kyky käyttää funktioita kunnollisina objekteina. Tässä esitettävä tapa koostaa kuvia perustuukin funktioiden yhdistämiseen korkeamman tason funktioilla, jotka ottavat parametriksi toisia funktioita (ja tässä tapauksessa myös palauttavat niitä).

3.1 Esimerkki koostamisesta: Alpha blending

Alpha blending kuvaa kaksi väriarvoa ja nk. alpha-arvon [0..1] yhdeksi väriarvoksi seuraavasti:

$$tulos = arvo1 * (1 - alpha) + arvo2 * alpha \quad (1)$$

Usein alpha:n arvo saadaan toisesta väriarvosta. Tämä voidaan yllä määritellyn kuvatyypin rajoissa ilmaista korkeamman tason funktiona, joka ottaa parametreikseen kaksi kuvaa (funktiota) sekä alpha-arvon ja palauttaa alpha blendingin toteuttavan funktion:

```
simpleAlphaCompose :: Float -> ImageC -> ImageC -> ImageC
simpleAlphaCompose alpha image1 image2 p
    = ((image1 p) `mulC` (1-alpha)) `addC` ((image2 p) `mulC` alpha)
    where
    addC :: Color -> Color -> Color
    addC (r,g,b,a) (r2,g2,b2,a2)
        = (r+r2,g+g2,b+b2,a+a2)
    mulC :: Color -> Float -> Color
    mulC (r,g,b,a) x = (r*x, g*x, b*x, a*x)
```

Kaksi ”kuvaa” voidaan nyt yhdistää *simpleAlphaCompose*:lla:

```
imageRed :: ImageC
imageRed _ = (1,0,0,1)
imageBlue :: ImageC
imageBlue _ = (0,1,0,1)
imageYellow = simpleAlphaCompose 0.5 imageRed imageBlue
```

Alpha:n arvolla 0.5 syntyvän kuvan väri on lähdekuvien värien keskiarvo, joten imageYellow on keltainen (0.5, 0.5, 0.0, 1.0) kuva.

4 Vektorigrafiikkaa

Yksi ylläolevan hienous on sen riippumattomuus perinteisestä kuvan ”pikseliesityksestä”; kuvaan voidaan sisällyttää helposti myös vektorigrafiikkaa. Vektorigrafiikkakuva voi yksinkertaisesti koostua funktiosta (kuvasta), joka tuottaa läpinäkyvän (alpha = 0) väriarvon pisteelle, joka ei ole osa vektorikappaletta. Monipuolisempia kappaleita voidaan toteuttaa esim. tekemällä kuvafunktio, joka käy läpi listan vektorikuvan alkeisosia ja palauttaa oikean väriarvon näiden pohjalta. Tämä lähestymistapa muistuttaa 3d-grafiikassa käytettyä säteen jäljitystä (*raytracing*).

Esimerkiksi punainen 1-säteinen kiekko voidaan määritellä näin:

```
imageRed :: ImageC
imageRed (x,y) | sqrt(x+y) <= 1 = (1,0,0,1)
imageRed _ = (0,0,0,0)
```

Tällaisista yksinkertaisista kappaleista voidaan muodostaa monimutkaisempia kuvioita koostamalla ja *transformaatioilla*.

4.1 Transformaatiot

Erilaiset transformaatiot, kuten kuvan skaalaus, rotaatio ja erikoisemmat venytykset eivät tarvitse erikseen toteutusta joka kuvafunktiossa; koska kuvaa ”luetaan” vain yksi piste kerrallaan, transformaatiot voidaan toteuttaa yksinkertaisesti toteuttamalla funktioita, jotka *siirtävät* kuvafunktiolle annettavia pisteitä [2, s. 139], esim. kuvan siirto (translaatio):

```
imageTranslate :: Point -> Image a -> Image a
imageTranslate translation image p = image (p - translation)
```

Kuvan skaalaus:

```
imageScale :: Point -> Image a -> Image a
imageScale 0 _ _ = const (0,0,0,0)
imageScale scale image p = image (p / scale)
```

Skaalatessa *jaetaan* annettu piste skaalausarvolla, koska tarkoitus on edelleenkin siirtää kohtaa, josta kuvaa ”luetaan”. Yrittäessä skaalata nollalla palautetaan const-funktionaalilla muodostettu funktio, joka palauttaa millä tahansa syötteellä täysin läpinäkyvän pikselin – äärettömän pientä kuvaa ei tarvitse näyttää. Rotaatiot ja erikoisemmat transformaatiot voidaan toteuttaa samalla tavalla.

5 Bittikartat

5.1 Kuvan muuttaminen bittikartaksi

Jos kuvaa on tarkoitus käyttää myös ohjelman ulkopuolella, se täytyy pystyä muuttamaan bittikartaksi. Triviaalisti bittikartan pikseleille saadaan laskettua väriarvot laskemalla kuvafunktiollamme leveys * korkeus kappaletta väriarvoja sopivista kuvan pisteistä ja muuntamalla ne bittikartassa käytettävään muotoon (tässä voidaan toteuttaa muunnokset eri väriprofiilien välillä).

Tämä voi kuitenkin tuottaa melko karkean lopputuloksen aliasoinnin takia, joten parempi tapa on soveltaa *supersämpläystä*: jokaista tuotettavaa pikseliä kohden lasketaan useampia ”näytteitä” ja lasketaan niistä painotettu keskiarvo niin, että pikselin ”oikeasta” paikasta saatu väri saa korkeimman painoarvon, ja muutama sen läheltä laskettu väriarvo pienempiä painoarvoja. Painotetun keskiarvon laskeminen voidaan toteuttaa esim. seuraavasti:

```
superSample :: (Fractional a) => Float -> Image a -> Image a
superSample offset image (x,y)
    = image (x,y) * 0.4
      + image(x + offset, y + offset) * 0.15
      + image(x + offset, y - offset) * 0.15
      + image(x - offset, y + offset) * 0.15
      + image(x - offset, y - offset) * 0.15
```

Esimerkkikoodissa lasketaan lopullinen väriarvo pisteen ”oikean” sijainnin (painoarvo 0.4) lisäksi neljästä muusta sijainnista (painoarvo 0.15). Näytteiden sijaintia voidaan skaalata sopivalle etäisyydelle pisteestä *offset*:illä. Paras tulos syntynee pitämällä offset pienempänä kuin kuvapisteiden etäisyys toisistaan. Funktio(naali) *superSample* on myös kuva-tyyppiä, joten sitä voidaan tarvittaessa soveltaa vain osaan kuvarakenteesta. Näytteidenottokohtien sijoittelu voi vaikuttaa paljon algoritmin tehokkuuteen, ja esimerkissä olevat eivät välttämättä ole optimaalisia.

5.2 Bittikarttojen tuonti

Bittikarttoja voidaan tuoda kuvarakenteeseen toteuttamalla funktio, joka lukee bittikarttakuvasta pikselien väriarvoja. Koska bittikartan koko tuskin on ääretön, eikä sen pikselitarkkuus ole todennäköisesti aivan sama kuin Point-tyypin (*Float*), pisteiden värien laskemisessa tarvitaan jonkinlainen interpolaatioalgoritmi, kun kuvasta luettava piste ei vastaa täysin minkään bittikarttakuvan pikselin sijaintia. Functional Images -tekstissä käytetty yksinkertainen box-filter -algoritmi [2, s. 149], joka tunnetaan myös nimellä *bilinear filtering*, antaa siedettäviä tuloksia kun kuvalle ei tehdä merkittäviä transformaatioita. Erikoisemmissa tilanteissa triviaali algoritmi voi johtaa näkyviin moiré-kuvioihin ja muihin epäkohtiin.

6 Yhteenveto

Haskell-kielen korkeamman tason funktioiden päälle voidaan rakentaa helposti ilman muuta pohjakoodia yksinkertainen grafiikkakirjasto, joka vielä säilyttää ”perinteisiä” imperatiivisiä ratkaisuja paremmin eron kuvan rakenteen ja sen esityksen välillä. Funktionaalisella kielellä voidaan myös pysyä hyvin lähellä erilaisten grafiikkaoperaatioiden *matemaattisia* määritelmiä.

Ratkaisu ei sellaisenaan edellytä Haskellin laiskaa evaluaatiota [2, s. 132], joten se on mahdollinen toteuttaa monilla ei-laiskoilla funktionaalisilla kielillä. Laiskasta evaluaatiosta voi kuitenkin olla apua monimutkaisempien kuvien generoinnissa ohjelmallisesti, tai käsitellessä isoa kuvaa, josta tuotetusta bittikartasta tarvitsee näyttää kerrallaan vain pieni alue; ei-laiskalla kielellä tilanteen optimointi edellyttäisi ylimääräistä koodia.

Lähteet

[1] Conal Elliott, *The Pan Home Page* <URL: `http://conal.net/pan/`>, viitattu 21.4.2008.

[2] Conal Elliott, *Functional images*, kirjassa The Fun of Programming (ISBN 0333992857). Functional Images-osa saatavilla WWW:stä: <URL: `http://conal.net/papers/functional-images`

[3] The CinePaint Project, *CinePaint FAQ*, <URL: `http://www.cinepaint.org/faq.html`>, viitattu 22.4.2008.