Sovellusaluekohtaisen kielen kehitys funktionaalisella kielellä

Teemu Kurppa

Helsinki 10.5.2000

Funktionaalisen ohjelmoinnin periaatteet - seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto **1**

2 Esimerkki: Animaatiokielen kehittäminen **2**

2.1 Grafiikkafunktioita 2

2.2 Animaation käsite 3

2.3 Funktioiden kohottaminen 4

2.4 Koostettuja animaatioita 6

2.5 Lisää työkaluja 7

2.6 Reaktiivisia animaatioita 8

2.7 Yhteenveto 9

3 Sovellusaluekohtaisista kielistä **10**

4 Sovellusaluekohtaiset upotetut kielet **10**

5 Sovellusaluekohtaisen kielen kehitys modulaarisesti **11**

Lähteet **11**

1 Johdanto

"In fact, a language is any mechanism to express intent." [Ben86]

Monet ohjelmistojen sovellusalueet ovat vaikeasti hallittavia kokonaisuuksia, joissa ohjelmiston toiminta jakautuu usealle tasolle. Matalan tason toiminta on usein sovellusriippumatonta, ja sovelluksen luonteen määrää korkena tason logiikka.

Yleiskäyttöiset ohjelmointikielet eivät tarjoa mahdollisuutta ohjelmoida käyttäen sovellusalueen korkean tason käsitteistöä. Tämän takia ne ovatkin yleiskäyttöisiä. Yleiskäyttöisellä kielellä on vaikea nopeasti kehittää ohjelmistoja sovellusalueelle. Käytännössä eri sovellusalueiden ohjelmoijat kehittävät käyttöönsä esimerkiksi aliohjelmakirjastoja, joissa on modularisoitu sovellusalueelle keskeisiä operaatioita, ja jotka operoivat sovellusalueelle luonteenomaisilla tietotyypeillä.

Aliohjelmakirjasto voidaan mieltää pienenä kielenä. Se abstrahoi joitakin sovellusalueen käsitteitä, ja antaa täten ohjelmoijalle mahdollisuuden ajatella korkeamman tason käsitteillä myös ohjelman tasolla, ei vain paperilla suunniteltaessa.

Usein aliohjelmakirjastojen abstrahoinnit jäävät sovellusalueen operaatioiden tasolle. Ohjelmoijan tehtävä helpottuisi edelleen, jos hänelle tarjottaisiin operaatioiden lisäksi käyttöön sovellusalueen käsitteitä ja kontrollirakenteita ohjelman tasolla. Sovellusaluekohtaiset kielet pyrkivät abstrahoimaan ja modularisoimaan juuri nämä keskeiset käsitteet, operaatiot ja kontrollirakenteet.

Sovellusaluekohtaisten kielten kehittäminen on kuitenkin vaikeaa, varsinkin sovellusalueilla, joita ei vielä tunneta hyvin. On mahdotonta abstrahoida "keskeisiä käsitteitä", jos niitä ei vielä tunneta. Tämän takia sovellusaluekohtaisen kielen kehittäminen pitäisi nähdä pitkänä prosessina alkaen pienten operaatioiden modularisoinnista. Tällöin kielen, jolla sovellusaluekohtaista kieltä kehitetään, tulee tarjota hyvät mahdollisuudet modularisointiin.

Funktionaaliset kielet tarjoavat modularisointivälineitä, joita ei perinteisistä imperatiivisista kielistä löydy. *Korkeamman kertaluvun funktiot* ja *laiska laskenta* antavat monia modularisointimahdollisuuksia. [Hug89].

Seuraavassa kappaleessa esitellään pitkän esimerkin avulla sovellusaluekohtaisen kielen kehitysprosessia funktionaalisella kielellä.

2 Esimerkki: Animaatiokielen kehittäminen

Sovellusaluekohtaisen kielen kehittymistä pienistä osasista esitellään seuraavassa animaatiokielen kehittämisen avulla. Esimerkki on pitkähkö johdattelu, jonka tarkoituksena on valaista sovellusaluekohtaisen kielen kehittämistä, joten esityksen yksityiskohtiin ei ole syytä takertua.

Kehiteltävä animaatiokieli on yksinkertaistettu versio Fran-kielestä. Fran (*F*unctional *R*eactive *An*imation) on Haskellin päälle kehitetty sovellusaluekohtainen kieli [Fran]. Se on vapaassa levityksessä, ja nykyisin osana Hugs98-jakelua [Hugs]. Tarkempi esittely Franista on esimerkiksi [ElH97].

2.1 Grafiikkafunktioita

Lähdetään liikkelle tilanteesta, jossa on toteutettuna muutamia yksinkertaisia operaatioita graafisten objektien käsittelyyn. Graafiset objektit voivat olla yksinkertaisia geometrisia muotoja, kuten palloja tai neliöitä, mutta myös monimutkaisempia bittikarttakuvia. Kutsutaan näitä graafisia objekteja lyhyyden vuoksi “kuviksi”. Alla on esitelty muutamia operaatioita kuville:

```
-- Atomisia objekteja:
circle                -- yksikköympyrä
square                -- yksikköneliö
import ``p.gif''      -- bittikarttakuva

-- Koostettuja objekteja (koosteoperaatiot):
scale v p             -- skaalaa kuvaa p vektorilla v
```

```
color c p          -- värittää kuvan p värillä c
trans v p          -- siirtää kuvaa p vektorilla v
p1 'over' p2       -- laittaa kuvan p1 kuvan p2 ''päälle''
p1 'above' p2      -- laittaa kuvan p1 kuvan p2 yläpuolelle
p1 'beside' p2     -- laittaa kuvan p1 kuvan p2 viereen
```

2.2 Animaation käsite

Tavoitteena on tehdä animaatioita. Pohtimalla staattisten kuvien ja animaation suhdetta, huomaamme, että animaatio on ajasta riippuva kuva. Toisin sanoen, animaatio on funktio ajalta kuville. Haskell-syntaksilla ilmaistuna

```
type Animation = Time -> Picture
```

Sinänsä tylsä animaatio staattisesta ympyrästä saadaan seuraavasti

```
circleAnim t = circle
```

Eli animaatiofunktion `circleAnim` arvo olisi ajasta riippumatta vakio `circle`. Kuville määritellyt operaatiot ovat hyödyllisiä myös animaatioissa. Esimerkiksi operaatiota

```
scale v (a t)
```

voidaan käyttää animaation skaalaamiseen.

Edelliset "animaatiot" ovat kuitenkin vain staattisia kuvia. Animaatio, muutos ja liike, on tavoitteena. Jos `scale v p` -operaation skaalausparametri `v` riippuisi myös ajasta, saataisiin aikaan yksinkertainen kokoa muuttava animaatio. Tarvitaan siis ajassa muuttuva vektori,

```
type VectorT = Time -> Vector
```

Kun tarkemmin asiaa mietitään, huomataan, että animaatioiden teossa kaikenlaiset ajasta riippuvat arvot ovat hyödyllisiä. Esimerkiksi `color c p` -operaation väriparametri `c` voisi myös muuttua ajan mukaan. Ajasta riippuvat arvot ovat siis keskeisiä animaatioiden kannalta, joten määritellään käsite *käyttäytyminen* (engl. behavior). Tyyppia a oleva käyttäytyminen on funktio ajalta arvolle a:

```
type Behavior a = Time -> a
```

Animaation määritelmä voidaan siten ilmaista uuden käsitteen avulla, samoin kuin ajasta riippuva väri ja vektori,

```
type Animation = Behavior Picture
type ColorB    = Behavior Color
type VectorB   = Behavior Vector
```

Käytetään jatkossa käyttäytymisfunktioiden perässä `B` kirjainta erottamaan ne vastaavista staattisista funktioista.

2.3 Funktioiden kohottaminen

Kun ajasta riippuvat arvot, käyttäytymiset, on otettu käyttöön voidaan skaalausfunktiota `scale` voidaan käyttää seuraavasti:

```
scale (v t) (b t)
```

Ongelmana on aikaparametri `t`, joka joudutaan antamaan kaikille parametreille erikseen. Kätevämmäksi tämä saadaan *kohottamalla* (engl. lift) kuville ja muille staattisille arvoille toteutetut funktiot toimimaan ajalla parametrisoiduilla arvoilla:

```
(scaleB v b) t = scale (v t) (b t)
(colorB c b) t = color (c t) (b t)
(transB v b) t = trans (v t) (b t)
```

```
circleB t = circle
squareB t = square
```

Korkeamman kertaluvun funktioita käyttäen voidaan kohottaminen abstrahoida omaksi funktiokseen. Tosin joudutaan tekemään kohotusfunktiot eri määrän parametreja omaaville funktioille:

```
(lift0 v)       t = v
(lift1 f b1)    t = f (b1 t)
(lift2 f b1 b2) t = f (b1 t) (b2 t)
```

Nyt edelliset funktioiden kohotukset voidaan ilmaista yksinkertaisemmin:

```
scaleB = lift2 scale
colorB = lift2 color
transB = lift2 trans

circleB = lift0 circle
squareB = lift0 square
```

Yllä `circleB` ja `squareB` ovat vakiofunktioita. Samoin voidaan muitakin vakioita ja funktioita kohottaa:

```
piB  = lift0 3.14159
42B  = lift0 42
sinB = lift1 sin
```

Tämä on kuitenkin epäkäytännöllistä. Huomattavasti selkeämpää olisi vakiofunktion 42B sijasta käyttää literaalivakiota 42 tai sinB-funktion sijasta käyttää pelkkää sin-funktiota. Haskellin tyyppiluokkien avulla voidaan mahdollistaa literaalivakioiden ja monien muiden funktioiden implisiittinen kohotus [ElH97]. Jatkossa vakioarvot ja muutamat numero-operaattorit (*, +, -, /) oletetaan implisiittisesti kohotetuiksi, mutta muuten pitäydytään ekspliisittesti kohotetuissa funktioissa yksinkertaisuuden vuoksi.

2.4 Koostettuja animaatioita

Animaatioiden koostamista toisista animaatioista esitellään esimerkkien avulla seuraavassa. Määritellään sitä ennen käytännöllinen käyttäytymisfunktio, ajan identiteettifunktio:

```
time t = t
```

Kun peruskäsitteet ovat kasassa, voidaan yksinkertaisia animaatioita tehdä helposti. Koska animaatioiden perusfunktioiden, esimerkiksi `scaleB`, `colorB`, arvo on uusi animaatio, niin näistä voidaan koostaa helposti erilaisia animaatioita.

Määritellään siniaaltofunktio, jonka arvo muuttuu ajan mukana pehmeästi ja jaksollisesti välillä [-1,1] :

```
wiggle = sinB (pi*time)
```

Perusaaltofunktiota käyttäen voidaan määritellä toinen funktio, joka skaalaa ja siirtää aaltofunktion tietylle reaalilukuvälille:

```
wiggleRange lo hi = lo + (hi-lo) * (wiggle+1)/2
```

Animaatio, jossa punainen pallo sykkii kokoaan muuttamalla, voidaan koostaa aikaisemmin esitellyistä funktioista:

```
pulsatingBall = colorB red
                       (scaleB (wiggleRange 0.5 1) circle)
```

Jos meillä on käytössä värifunktio `rgb r g b`, saadaan sykkivään palloon mukaan vielä värin sykintä:

```
pulsatingBall2 = colorB (rgb (wiggleRange 0 1) 0 0)
                        (scaleB (wiggleRange 0.5 1) circle)
```

Sykkivä pallo voidaan laittaa liikkumaan sivuttain:

```
movingPulseBall = transB (vectorB wiggle 0) pulsatingBall
```

Määritellään toinen animaatio, jossa neliö liikkuu ylös-alas ja koostetaan siitä ja `movingPulseBall`-animaatiosta uusi animaatio:

```
movingSquare = transB (vectorB 0 wiggle) (colorB blue square)
dancingBallAndSquare = movingSquare 'overB' movingPulseBall
```

2.5 Lisää työkaluja

Koska aika on arvo muiden joukossa, saadaan animaation kannalta mielenkiintoisia työkaluja, kun määritellään operaatioita ajalle. Esimerkiksi:

```
timeTransform dt b t = b (t+dt)
```

aikaistaa tai myöhäistää käyttäytymistä ajassa. Liikkuvan sykkivän pallon yläpuolelle voidaan laittaa toinen, jäljessä tuleva pallo:

```
movingPulseBall 'aboveB' (timeTransform (-1) movingPulseBall)
```

Animaatiot ovat hyviä työkaluja esimerkiksi fysiikan ilmiöiden kuvaamiseen, ja toisaalta viihteellisissäkin animaatioissa on hyötyä yksinkertaisista fysiikan laeista. Aikaisemmissa esimerkeissä kuvien paikka on määritelty suoraan. Newtonin fysiikan kannalta liike aiheutuu voimien antamasta kiihtyvyydestä, joka antaa kappaleille nopeuden, josta edelleen saadaan kappaleen paikka. On siis luontevaa määrittää kappaleiden liikkeet kiihtyvyyksiä tai nopeuksina, joista kappaleen paikat määräytyvät. Tunnetusti paikka ja nopeus saadaan integroimalla kiihtyvyydestä. Määritellään integrointifunktio `integral`, jonka avulla voidaan määritellä animaatio putoavalle pallolle:

```
fallingBall x0 y0 v0 = transB (vectorB x0 y) ball
  where y = y0 + integral v
        v = v0 + integral g
        g = 9.81
```

Edellinen esimerkki kuvaa hyvin, kuinka kielen funktionaalisuus antaa mahdollisuuden *määritellä* animaatioita. Riittää määritellä liike taulukkokirjasta kopioitujen fysiikan yhtälöiden avulla. Kaikki arvot on parametrisoitu ajalla, ja joten kunhan ulkopuolelta annetaan aina kulloinenkin ajanhetki, saadaan jatkuvaa animaatiota. Animaatio-ohjelmoijan ei tarvitse huolehtia ajan pilkkomisesta tai arvojen laskemisesta.

2.6 Reaktiivisia animaatioita

Edellisten esimerkkien animaatiot koostuivat käyttäytymisistä, jotka toistivat itseään loputtomiin. Sykkivä pallo sykkii ja putoava pallo putoaa, kunnes animaatio katkaistaan. Mielenkiintoisempia animaatioita saataisiin, kun animaatiot, tai yleisemmin käyttäytymiset, reagoisivat tapahtumiin. Esimerkiksi sykkivä pallo lopettaisi sykkimisensä, kun käyttäjä painaisi hiiren nappia, tai törmätessään maahan putoava pallo pomppaisi ylöspäin.

Saadaksemme tapahtumiin reagoivia animaatioita, *reaktiivisia animaatioita*, otetaan käyttään uusi käsite, *tapahtuma* (engl. event). Oleellista on sitoa tapahtumat ja käyttäytymiset jollain tapaa yhteen. Yleisesti voidaan ajatella, että on voimassa jokin käyttäytyminen `b1`, *kunnes* tapahtuu tapahtuma `e`, jonka jälkeen on voimassa käyttäytyminen `b2`. Tarvitaan siis seuraavan kaltaista funktiota:

```
b1 'untilB' e => b2
```

Tässä sekä `untilB` että `=>` ovat infix-operaattoreita, joten sama voitaisiin kirjoittaa

```
untilB b1 (=> e b2)
```

Hiiren painalluksesta väriä sinisen ja punaisen välillä vaihtava pallo voidaan nyt kuvata seuraavasti:

```
colorChanger = colorB (cycle red blue) circle
  where cycle c1 c2 =
```

```
        c1 'untilB' lbp => cycle c2 c1
```

Tässä `lbp` on hiiren vasemman napin painalluksesta laukeava tapahtuma. Esimerkissä tapahtumien ja käytösten vuorovaikutus näkyy siten, että tapahtuma `lbp` antaa `cycle`-funktion arvoksi uuden käyttäytymisen (`cycle` argumenteilla `c2 c1`), joka taas on ehdollistettu uudelle tapahtumalle, eli seuraavalle napin painallukselle. Esimerkki edellyttää myös laiskaa laskentaa, jotta uutta käyttäytymistä ei lasketa ennen kuin sitä tarvitaan.

Myös animaation sisäisiä tapahtumia voidaan kuvata, esimerkiksi putoava pallo voidaan pysäyttää sen osuessa maahan. (Mielenkiintoisempi esimerkki pomppivasta pallosta vaatisi muutaman uuden työkalufunktion esittelemistä.)

```
fallingBall2 groundVal x0 y0 v0 = transB (VectorB x y) circle
  where x = x0
        y = y0 + integral v
        v = (v0 + integral g) 'untilB' bounce => 0
        g = 9.81
        bounce = predicate( y <= groundVal )
```

2.7 Yhteenveto

Edellä esiteltiin todellisuudessa pitkähköä prosessia, jossa modularisoitiin sovellusaluekohtaisia operaatioita, esimerkiksi `scale, trans, color`. Samalla hahmotettiin sovellusalueen keskeisiä käsitteitä, käyttäytymiset ja tapahtumat. Lopussa abstrahointiin ja modularisoitiin sovellusalueelle tyypillinen kontrollirakenne, `untilB`. Voidaan sanoa, että nämä uudet käsitteet ja niihin liittyvät operaatiot ja kontrollirakenteet muodostavat uuden, pienen animaatiokielen.

Animaatiokielen kehitysprosessin aikana saatiin monia käsitteitä ja operaatioita modularisoitua siten, ettei myöhemmin, kieltä laajennettaessa, niihin tarvinnut enää koskea. Aluksi kehitettiin yksinkertaisia, loputtomiin jatkuvia animaatioita, "käyttäytymisiä". Myöhemmin lisättiin mahdollisuus reaktiivisuuteen uuden käsitteen "tapahtuma" avulla. Alla oleva funktionaalinen kieli Haskell tarjosi hyvät mahdollisuudet modularisointiin korkeamman kertaluvun funktioiden ja laiskan laskennan avulla, eikä käsitteen "tapahtuma" lisääminen aiheuttanut muutoksia aikaisemmin tehtyihin funktioihin.

3 Sovellusaluekohtaisista kielistä

Sovellusaluekohtainen kieli on tietylle sovellusalueelle räätälöity kieli. Se mahdollistaa nopean ohjelmistonkehityksen sovellusalueella. Sovellusaluekohtaisen kielen avulla ohjelmoijat voivat käyttää suoraan ohjelmakoodissa solvellusalueen korkeamman tason käsitteitä. Tämän vuoksi sovellusaluekohtaisilla kielillä kirjoitetut ohjelmat ovat yleisesti helpompia tuottaa ja ymmärtää verrattuna vastaaviin yleiskäyttöisillä ohjelmointikielillä kirjoitettuihin ohjelmiin.

Esimerkkejä yleisesti tunnetuista sovellusaluekohtaisista kielistä ovat esimerkiksi Lex ja Yacc jäsentäjien tuottamiseen tai Mathematica symboliseen laskentaan.

4 Sovellusaluekohtaiset upotetut kielet

Uuden ohjelmointikielen suunnittelu ja toteutus on kuitenkin työlästä. Kielen suunnittelussa joudutaan tekemään useita syntaktisia ja rakenteellisia valintoja, joilla ei kuitenkaan ole keskeistä merkitystä sovellusalueen kannalta. Koska kehitetään erikoistettua kieltä, ei kaikkia sovellusalueen ulkopuolisia tarpeita voida tyydyttää. Usein ohjelmistot koostuvat muustakin kuin sovellusaluekohtaisesta ohjelmiston osasta. Kieleen voidaan toki lisätä sovellusalueen ulkopuolisiakin käsitteitä, mutta tällöin päädytään helposti yleiskäyttöiseen kieleen, joka vain on monimutkaisempi ja vaikeampi ymmärtää kuin yksinkertaisempi yleiskäyttöinen kieli. [Hud98]

Näistä syistä johtuen on usein järkevämpää kehittää sovellusaluekohtainen kieli jonkin yleiskäyttöisen kielen päälle. Tällöin voidaan lainata tiettyjä rakenteellisia ja syntaktisia ratkaisuja suoraan alla olevasta kielestä, ja lisäksi mahdollistetaan sovellusalueen ulkopuolisten tarpeiden toteutus alla olevalla kielellä. Tällaista kieltä kutsutaan *sovellusaluekohtaiseksi upotetuksi kieleksi* (engl. Domain Specific Embedded Language, DSEL).

5 Sovellusaluekohtaisen kielen kehitys modulaarisesti

Ohjelmistonkehityksestä on erityisesti hyötyä vähemmän tunnettuilla sovellusalueilla, joissa joudutaan kokeillamaan prototyyppiohjelmistojen avulla erilaisia menetelmiä. On kuitenkin mahdotonta kehittää uusi sovellusaluekohtainen ohjelmointikieli oikeanlaiseksi, jos ei vielä tunnetta hyvin sovellusaluetta ja sen keskeisiä piirteitä ja käsitteitä.

Sovellusaluekohtainen kieli voidaan kehittää vaiheittain, lisäämällä siihen uusia abstraktioita, kun ymmärrys sovellusalueesta kasvaa. Vaiheittainen kehitys edellyttää kuitenkin sovelluskohtaisen kielen alla olevalta kieleltä hyviä mahdollisuuksia modularisoida abstraktioita ja yhdistellä näitä.

Hughes [Hug89] esittää, kuinka funktionaaliset kielet antavat voimakkaampia työkaluja modularisointiin kuin perinteiset imperatiiviset kielet. Erityisesti korkeamman kertaluvun yhtälöt ja laiska laskenta mahdollistavat sellaista modularisointia, joka imperatiivisissa kielissä on vaikeaa.

Haskellin päälle on viime vuosina kehitetty useita sovellusaluekohtaisia kieliä. Yhtenä haarana ovat *funktionaalisen reaktiivisen ohjelmoinnin* perustuvat kielet, kuten tässä esitelmässä esitelty Fran, ja robottien ohjaukseen käytetty Frob [PHE98]. Molemmissa käsitteet "käyttäytyminen" ja "tapahtuma" ovat keskeisessä asemassa.

Lähteet

Ben86 Bentley, John, *Little languages.* CACM 29(8) (1986), ss. 711–721.

ElH97 Elliot, Conal & Hudak, Paul, *Functional Reactive Animation.* Proc. International Conference on Functional Programming, Amsterdam, Netherlands, June 1997, ss. 164–173. [Myös http://research.microsoft.com/ conal/papers/icfp97, 8.5.2000]

Fran Fran-järjestelmän kotisivu. http://www.research.microsoft.com/čonal/fran/ . [8.5.2000]

Hud98 Hudak, Paul, *Modular Domain Specific Languages and Tools.* Proc. Fifth International Conference on Software Reuse, Victoria, Canada, June 1998, ss. 134–142. [Myös http://haskell.cs.yale.edu/frob/dsl.ps, 8.5.2000]

Hug89 Hughes, John, *Why Functional Programming Matters.* The Computer Journal, Vol. 32 (1989), No. 2. [Myös http://www.md.chalmers.se/ rjmh/Papers/whyfp.html, 8.5.2000]

Hugs Hugs Haskell tulkin kotisivu. http://haskell.cs.yale.edu/hugs/ . [8.5.2000]

PHE98 Peterson, John & Hudak, Paul & Elliot, Conal, *Lambda in Motion: Controlling Robots with Haskell.* Practical Aspects of Declarative Languages (PADL'99) 1998, ss. 91-105. [Myös http://haskell.cs.yale.edu/frob/padl99/padl99.ps, 8.5.2000]