Principles of Programming Languages: Erlang

Juho Snellman <juho.snellman@cs.helsinki.fi>

Helsinki 27. maaliskuuta 2004

Seminaariesitelmä

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

1 Johdanto

Erlang on Ericssonin Computer Science Laboratory-ryhmän kehittämä dynaamisesti tyypitetty funktionaalinen ohjelmointikieli, joka on suunniteltu isojen (miljoonien rivien) hajautettujen ja vikasietoisten järjestelmien toteuttamiseen. Kieli on suhteellisen yksinkertainen, ja kokemusten perusteella helposti opittavissa [Wig01].

Funktionaalisia ohjelmointikieliä on perinteisesti käytetty suhteellisen vähän kaupallisella alalla, ja vielä vähemmän teollisella sektorilla. On ehkä yllättävää, että dynaamista tyypitystä edes harkittaisiin suunniteltaessa kieltä kriittisiin sovelluksiin[1]. Erlang-ohjelmoijien kokemus on kuitenkin ollut, ettei tyyppivirheitä tapahdu kovinkaan usein, ja että dynaamisen tyypityksen hyödyt (esimerkiksi koodin päivittäminen järjestelmään ajonaikaisesti) painavat vaakakupissa mahdollisia virheitä enemmän.

Erlangin kiinnostavin piirre on sen hyvä tuki rinnakkaisohjelmoinnille (*concurrency oriented programming* on ilmeisesti lähinnä Erlang-yhteisön käyttämä termi tälle, jonka on tarkoitus korostaa rinnakkaisuuden merkitystä kielessä). Eräs esimerkki rinnakkaisuuden merkityksestä on käyttöliittymäkirjasto EX11, jossa jokaisella käyttöliittymäelementillä on oma prosessinsa [Arm04]. Erlang käyttää omia virtuaalikoneessa toteutettuja kevyitä prosesseja, jotka skaalautuvat useimpien käyttöjärjestelmien tarjoamia säikeitä paremmin. Lisäksi Erlangin standardikirjastoissa on määritelty valmiita ns. korkean kertaluokan prosesseja, jotka ovat tietynlaisiin tehtäviin tarkoitettuja parametrisoitavia prosessirunkoja [Arm97].

Tämän raportin luvussa 2 käsitellään pintapuolisesti Erlangin syntaksia, semantiikkaa ja tietorakenteita. Luvussa 3 käsitellään Erlangin rinnakkaisominaisuuksia.

2 Erlangin perusteet

Erlang on dynaamisesti tyypitetty funktionaalinen ohjelmointikieli, joka keskeisin piirre on sen laaja tuki rinnakkaisohjelmoinnille. Ensimmäiset kielen versiot olivat laajennoksia Prologiin, mistä Erlang on enimmäkseen perinyt syntaksinsa. Prologista ei kuitenkaan ole pe-

[1]Telekommunikaatiossa minimitavoitteena on 99.999% saatavuus, eli korkeintaan viisi minuuttia käyttökatkoksia vuodessa. Eräät Erlang-pohjaiset järjestelmät ovat päässeet keskimäärin 99.9999999% saatavuuteen, joka tarkoittaa noin 30 millisekuntia palvelukatkoksia vuodessa.[Wig01].

ritty mitään logiikkaohjelmointi-ominaisuuksia [Arm97].

Funktionaalisen ohjelmoinnin perinteinen esimerkkiohjelma on kertoman laskeminen:

```
-module(math).
-export([fac/1]).

fac(0)              -> 1;
fac(N) when N > 0   -> N * fac(N-1).
```

Ohjelmassa luodaan moduli `math`, ja viedään yhden parametrin funktio `fac/1` [2] julkiseksi. `fac/1`:lle määritellään kaksi erillistä osaa (`fac(0), fac(N)`), joiden suoritus määräytyy tapauskohtaisesti. Funktion osien nuolta edeltävä osuus määrää mallin, johon parametreja yritetään kutsun yhteydessä sovittaa (match). Osat käydään läpi niiden määrittelyjärjestyksessä, ja ensimmäinen osa jonka malli sopii parametreihin suoritetaan. Mikäli sovitus ei onnistu yhteenkään malleista, heittää ohjelma poikkeuksen.

Mallissa olevat literaalit sopivat ainoastaan, jos sovitettava arvo on sama. Tässä tapauksessa siis funktion ensimmäinen osa suoritetaan parametrin arvon ollessa `0`, jolloin funktio palauttaa arvon 1. Mallissa olevat muuttujat (kuten Prologissa, muuttujien ja parametrien nimet alkavat aina isolla kirjaimella) puolestaan sopivat oletusarvoisesti mihin tahansa arvoon, ja sitovat kyseisen arvon muuttujaan funktion suorituksen ajaksi. Jälkimmäisessä osassa on malliin kuitenkin määritelty muuttujan `N` lisäksi avainsanalla `when` lisärajoite `N > 0`, joten ainoastaan positiiviset numerot sopivat malliin. Rajoitteet voivat sisältää ainoastaan rajoitettuja predikaatteja; esimerkiksi funktiokutsut ovat kiellettyjä. Jälkimmäistä osaa suoritettaessa lasketaan `N`:n ja rekursiivisen kutsun `fac(N-1)` tulo, jonka funktio lopuksi palauttaa.

Malleissa voidaan käyttää myös monimutkaisempia tietorakenteita kuten listoja tai monikoita (tuples), jolloin näiden tietorakenteiden sisältämiä alkioita voidaan sitoa muuttujiin. Seuraavassa ohjelmassa määritellään funktio `reverse/1`, joka palauttaa saamansa listan käännettynä toisin päin. Koska Erlangin merkkijonot on (mielestäni hieman kyseenalaisesti) toteutettu kokonaislukulistoina, toimii funktio suoraan myös merkkijonoilla.

[2] Prologin tapaan funktion ariteetti (parametrien määrä) ilmoitetaan nimen yhteydessä kauttaviivalla eroteltuna. Ariteettia ei kuitenkaan tarvitse erikseen kirjoittaa funktiota kutsuttaessa, koska parametrien määrä on jo tunnettu.

```
-module(listutil).
-export([reverse/1]).

reverse([], List)   -> [];
reverse([ Head | Tail ], List) -> reverse(Tail, [ Head | List ]).

reverse(List) -> reverse(List, []).

%% listutil:reverse([alpha, beta, gamma]) => [gamma, beta, alpha]
%% listutil:reverse("123") => "321"
```

Poikkeuksena edelliseen ohjelmaan luotiin moduliin tällä kertaa useampia funktioita, joista `reverse/2`:ta ei asetettu `export`-määreellä julkiseksi. `reverse/1` toimii ainoastaan helppokäyttöisenä kääreenä `reverse/2`:lle. Sovitettaessa lista-arvoja, päästään listan elementteihin käsiksi `[ | ]`-syntaksilla. `reverse/2`:n ensimmäinen osa ajetaan ensimmäisen parametrin ollessa tyhjä lista, jolloin funktio palauttaa toisena parametrina saamansa listan. Toinen osa puolestaan ajetaan, jos ensimmäinen parametri on mikä tahansa muu lista. Tällöin listan ensimmäinen alkio sidotaan muuttujaan `Head`, ja loput alkiot sisältävä lista muuttujaan `Tail`. Toista osaa suoritettaessa kutsutaan rekursiivisesti `reverse/2`:ta, jossa jälkimmäiseksi parametriksi annetaan lista jonka ensimmäinen alkio on `Head` ja loput alkiot ovat samat kuin listassa `List`.

Erlangin toinen keskeinen tietorakenne on monikko. Monikot on tarkoitettu käytettäväksi tilanteissa joissa alkioiden määrä on tunnettu etukäteen. Tällaisia tilanteita ovat esimerkiksi prosessien välinen viestintä ja uusien tietorakenteiden luonti. Seuraava ohjelma toteuttaa yksinkertaisen ja tehottoman binääripuun. Puun jokainen solmu mallinnetaan joko tyhjällä monikolla {} tai monikolla `{Avain, Arvo, Pienemmät, Suuremmat}`, jossa `Pienemmät` ja `Suuremmat` ovat solmuja. Monikot, kuten muutkaan Erlangin tietorakenteet, eivät ole muokattavia. Siksi lisättäessä puuhun uutta alkiota funktio palauttaa uuden puun, eikä suinkaan muokkaa vanhaa.

```
-module(tree).
-export([insert/3]).

insert( Key, Value, {} ) ->
    {Key, Value, {}, {}};
insert( Key, Value, { Key, _, S, B } ) ->
    {Key, Value, S, B};
insert( Key, Value, { K, V, S, B } ) when Key < K ->
    {K, V, insert(Key, Value, S), B};
insert( Key, Value, { K, V, S, B } ) ->
    {K, V, S, insert(Key, Value, B)}.
```

`insert/3`:lle määritellään neljä osaa. Triviaalissa tapauksessa arvoa ollaan lisäämässä tyhjään alipuuhun `{}`, jolloin funktio voi vain luoda uuden solmu-monikon parametreina saamistaan avaimesta ja arvosta. Toinen tapaus on lähes yhtä yksinkertainen; jos `Key`:n arvo on sama kuin käsiteltävän solmun avainarvo, luodaan uusi solmu jossa vanhan solmun alipuista sekä parametreina saaduista arvosta ja avaimesta. Sovituksessa käytettiin nyt merkintää _ monikon alkiolle jonka arvosta emme funktiossa välitä. Lisäksi muuttujan `Key` käyttäminen mallissa kahteen kertaan tarkoittaa, että kyseisiin kohtiin sovitettavien arvojen on oltava yhtä suuria. Viimeiset kaksi osaa palauttavat alkuperäisen solmun, josta on korvattu joko `Pienemmät` tai `Suuremmat` alipuu rekursiivisen `insert/3`-kutsun tuloksella.

Hieman erikoinen Erlangin piirre on ns. bittisyntaksi `<< ... >>`, jolla voidaan helposti luoda binärityyppistä dataa ja sovittaa sitä muuttujiin. Ilmeisesti tämä ominaisuus on erityisen hyödyllinen telekommunikaatio- ja verkkoprotokollien käsittelyssä, jonne Erlang on alunperin suunnattu. Seuraavassa Armstrongin [Arm03] esimerkissä puretaan IP-protokollan mukaisen datagram-paketin bittikentät sovittamalla se rivien 6-9 malliin:

```
 1  -define(IP_VERSION, 4).
 2  -define(IP_MIN_HDR_LEN, 5).
 3  ...
 4  DgramSize = size(Dgram),
 5  case Dgram of
 6   <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
 7      ID:16, Flgs:3, FragOff:13, TTL:8, Proto:8,
 8      HdrChkSum:16, SrcIP:32, DestIP:32,
 9      RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
10        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
11        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
12        ...
```

`case`-lauseke toimii samaan tapaan kuin funktioiden parametrien sovitus, eli etsii mallin johon testattavat arvot (tässä tapauksessa `Dgram`-muuttuja) saadaan sovitettua, ja sitoo sovitetut arvot määriteltyihin muuttujiin.

Lisäksi Erlangilla on useita moderneille funktionaalisille kielille ominaisia piirteitä. Korkeamman kertaluokan funktiot, anonyymit funktiot ja leksikaaliset sulkeumat toimivat normaaliin tapaan[3]. Virhetilanteissa Erlang heittää poikkeuksen, joka yritetään sovittaa kutsuketjusta löytyvien `catch`-lausekkeiden malleihin. Erlang tukee myös Haskell-tyylisiä listageneraattoreita (list comprehensions), jotka ovat ytimekäs menetelmä listojen luomiseen:

```
[X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
=> [4,5,6]

[{X, Y} || X <- [1,2,3], Y <- [a,b]].
=> [{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

3 Rinnakkaisohjelmointi

Erlangin rinnakkaisuusmalli perustuu viestien lähettämiseen prosessien välillä. Prosessit eivät vastaa käyttöjärjestelmän prosesseja tai säikeitä, vaan ne on toteutettu virtuaalikoneessa. Erlangin prosessit ovat erittäin kevyitä; uusien prosessien käynnistäminen ja suoritettavan prosessin vaihto ovat kertaluokkaa nopeampia kuin käyttöjärjestelmän omia prosesseja käytettäessä. Lisäksi prosesseja voi olla kerralla ajossa kymmeniä tuhansia [Arm02].

Prosessit ovat toisistaan eristettyjä eivätkä jaa mitään tilaa keskenään, joten niiden välillä voi liikkua tietoa ainoastaan viestien yhteydessä. Näin vältetään rinnakkaisohjelman pahimmat sudenkuopat, jotka liittyvät jaetun tilatiedon hallitsemiseen [Arm03]. Viestit voivat olla mitä tahansa Erlang-tietorakenteita, joista viestin vastaanottaja saa oman kopionsa. Käytännössä prosessien sijaitessa samalla koneella voi virtuaalikone kuitenkin optimoida pois viestin kopioinnin, koska kieli ei salli tietorakenteiden sisällön muokkaamista.

Uusi prosessi luodaan `spawn`-primitiivillä, jolle annetaan parametriksi 0-ariteetin funktio, jota uusi prosessi ryhtyy suorittamaan. `spawn` palauttaa luodun prosessin tunnisteen,

[3]Toisin kuin useimmat epäpuhtaat funktionaaliset kielet, Erlang ei koskaan salli muuttujan arvon muuttamista. Tämä estää esimerkiksi laskurin toteuttamisen sulkeumilla, kuten Schemessä on joskus tapana. Laskuri toteutettaisiin Erlangissa prosessina.

jonka avulla prosessille voidaan lähettää viestejä. Hajautus onnistuu Erlangissa helposti, sillä `spawn`:lle voidaa ilmoittaa myös Erlang-solmu (käytännössä Erlang-instanssi, jota voidaa ajaa joko samalla tai eri koneella kuin `spawn`:ia suorittavaa solmua), jolloin prosessi avataan toisella solmulla. Lähetettäessä viestejä solmujen välillä Erlang huolehtii läpinäkyvästi viestin välittymisestä oikein[4].

Viestejä lähetetään `!`-operaattorilla, jolle annetaan vastaanottajan tunniste ja lähetettävä viesti (joka voi olla mikä tahansa Erlang-tietorakenne). Viestejä vastaanotetaan `receive`-lausekkeella, jossa määritellään funktiokutsuja vastaavalla tavalla malleja joihin viestiä yritetään sovittaa. Jos yksikään receive-lausekkeen malleista ei kelpaa, jätetään viesti prosessin viestijonoon, ja yritetään sen käsittelyä myöhemmin uudestaan.

Yksinkertainen `echo`-palvelin, joka vastaa sille lähetettyihin viesteihin voitaisiin toteuttaa seuraavalla tavalla:

```
-module(echo).
-export([make_server/0]).

make_server() -> spawn(fun echo/0).

echo() ->
  receive
    { echo, Sender, Data } ->
        Sender ! { reply, self(), "Hello, ", Data },
        echo();
    { quit, Sender } ->
        Sender ! { quitting, self() }
  end.
```

Uudessa prosessissa käynnistetty `echo`-funktio käsittelee kahdenlaisia monikkoja viesteinä. Molemmissa tapauksissa ensimmäisen alkion on oltava joko atomi `echo` tai `quit` ja toisen alkion pitäisi olla lähettäjän prosessitunniste (joskaan tätä ei tarkisteta). Jos ensimmäinen alkio oli `echo`, täytyy viestissä olla vielä kolmas alkio, jonka sisällöllä ei ole väliä. `echo`-viestit käsitellään lähettämällä vastaanottajalle viestinä neljän alkion monikko (jossa `self()` palauttaa prosessin oman tunnisteen), ja rekursoimalla takaisin `echo`-funktioon seuraavan viestin käsittelyä varten. Myös `quit`-viestiin vastataan lähettämällä viesti, mutta lähetyksen jälkeen palataan funktiosta, jolloin prosessin suoritus päättyy.

[4]Tietenkään läpinäkyvyys ei voi olla aivan täydellistä. Solmun sisäiset viestit menevät varmuudella perille, kun taas verkon yli lähetettäessä paketteja voi hyvinkin hukkua matkalla.

Palvelinprosessin tekeminen ja interaktio sen kanssa voisi sujua vaikkapa seuraavalla tavalla:

```
Pid = echo:make_server()
Pid2 = echo:make_server()
Pid ! { echo, self(), "world!"}
receive X -> X end;
  => {reply, Pid, "Hello, ", "world!"}
Pid ! { quit, self() }
  % process Pid1 is stopped
Pid2 ! { echo, self(), "hello"}
receive X -> X end;
  => {reply, Pid2, "Hello, ", "hello"}
  % process Pid2 still alive
```

Erlang on tarkoitettu vikasietoisten järjestelmien tekoon, jossa on oleellista virheiden oikeaoppinen käsittely. Edellisessä luvussa mainitut poikkeukset ja niiden käsittelyyn tarkoitetut primitiivit eivät yksinään riitä virheiden käsittelyyn rinnakkaisessa ohjelmistossa. Yhdessä prosessissa tapahtuva virhe kiinnostaa todennäköisesti ainakin joitakin muita prosesseja. Erlang-yhteisön motto aiheesta on varsin provokatiivinen "Let it crash" [Arm03]. Tällä tarkoitetaan, ettei prosessien kannattaisi yrittää väkisin palautua virheistä poikkeustilanteita peittämällä, vaan epävarmoissa tilanteissa on parempi lopettaa prosessin suoritus.

Prosessin suorituksen loppuessa poikkeukseen tästä tiedotetaan viesteillä prosesseille jotka on *linkitetty* lopetettuun prosessiin. Ajatuksena on, että prosessit järjestetään hierarkiaan, jossa varsinaiset työtä tekevät prosessit on linkitetty valvojaprosesseihin. Valvojat käsittelevät virheen tilanteeseen sopivalla tavalla, esimerkiksi käynnistämällä prosessin uudestaan tai pysäyttämällä muut kuolleesta prosessista riippuvaiset prosessit [Arm03].

Rinnakkaisessa koodissa prosessien käyttäytymismallit voidaan jaotella muutamiin yleisiin ryhmiin. Jokaisesta ryhmästä löytyy ominaispiirteitä, jotka on toteutettava kaikissa prosesseissa suunnilleen samalla tavalla, sekä erikostuneempia piirteitä. Tämän virhealttiin koodin kirjoittaminen joka kerta uudestaan ei kuulosta järkevältä, joten tarvitaan jokin menetelmä yhteisten piirteiden abstraktoimiseen pois.

Erlangissa tällaisia parametrisoitavia prosessitehtaita kutsutaan yleensä käyttäytymisiksi (behaviours), ja joskus harvemmin korkeamman kertaluokan prosesseiksi. Omien korkeamman kertaluokan prosessien kirjottaminen ei ole suhteettoman hankalaa kielen perus-

työkaluilla, mutta tuotantokäyttöön tarkoitetussa käyttäytymisessä on otettava huomioon paljon hienovaraisia yksityiskohtia. Onneksi Erlang sisältää valmiita käyttäytymisiä, joiden pitäisi kattaa suurin osa sovellusten tarpeista. Wiger arvioi [Wig01] 5-10 yleisimmän korkean kertaluokan prosessin kattavan noin 95% ohjelmistojen rinnakkaisuustarpeista. Armstrong puolestaan raportoi erittäin suuren ohjelmistoprojektin [Arm03] käyttäneen 63% tapauksista Erlangin vakiokirjaston client-server käyttäytymismallia.

Valmiit käyttäytymiset kattavat prosessien hallintaa (valvoja / aliprosessi-suhde), client-server sovelluksia, tilakoneita (jotka ovat ilmeisesti erityisen hyödyllisiä telekommunikaatio-ohjelmissa) ja tapahtumien hallintaa. Seuraavassa Ericssonin dokumentaatiosta [Eri04] peräisin olevassa esimerkissä luodaan gen_fsm-käyttäytymismallin avulla yksinkertainen tilakone, joka mallintaa koodilukollista ovea. Ovi aukeaa oikealla koodilla, ja sulkeutuu 30 sekunnin päästä aukeamisesta.

```
-module(code_lock).
-behaviour(gen_fsm).

-export([start_link/1]).
-export([button/1]).
-export([init/1, closed/2, open/2]).

start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, Code, []).

button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).

init(Code) ->
    {ok, closed, {[], Code}}.
closed({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_open(),
            {next_state, open, {[], Code}, 30000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, closed, {Incomplete, Code}};
        _Wrong ->
            {next_state, closed, {[], Code}};
    end.
open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

4 Yhteenveto

Erlang on dynaamisesti tyypitetty funktionaalinen ohjelmointikieli, joka on suunniteltu isojen vikasietoisten hajautettujen järjestelmien toteuttamiseen. Kielen ydin sisältää funktionaalisilta ohjelmointikieliltä odotetun perustoiminnallisuuden. Funktiot voidaan määritellä useassa osassa, joista valitaan osille annettuja malleja sovittamalla suoritettava koodi. Sovittaminen voi samalla sitoa joko tietorakenteita tai niiden osia muuttujiin funktion sisällä. Erlangin oleellisimmat tietorakenteet ovat lista (joita käytetään myös merkkijonojen esittämiseen) ja monikko.

Erlangin oleellisin ero muihin ohjelmointikieliin on sen rinnakkaisuusmalli, joka perustuu kevyisiin toisistaan eristettyihin prosesseihin. Prosessit voivat vaikuttaa toisiinsa ainoastaan lähettämällä toisilleen mielivaltaisista Erlang-tietorakenteista koostuvia viestejä. Virheiden hallintaan käytetään tavallisten poikkeusten lisäksi linkitystä, jolloin prosessit saavat selville milloin muut niitä kiinnostavat prosessit ovat kuolleet virhetilanteisiin. Rinnakkaisten sovellusten kirjoittamista voidaan yksinkertaistaa abstraktoimalla tiettyjen käyttäytymismallien yhteiset osat erillisiin korkeamman kertaluokan prosesseihin.

Lähteet

Arm97 Joe L. Armstrong. The development of erlang. In *International Conference on Functional Programming*, pages 196–203, 1997.

Arm02 Joe Armstrong. Concurrency oriented programming in erlang. In *Lightweight Languages Workshop 2002 (LL2)*, November 2002. `http://ll2.ai.mit.edu/talks/armstrong.pdf`.

Arm03 Joe Armstrong. *Making reliable systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, December 2003.

Arm04 Joe Armstrong. Ex11 notes, 2004. `http://www.sics.se/ joe/ex11/widgets/ex11.html`.

Eri04 Ericsson. Otp design principles, 2004. `http://www.erlang.org/doc/r9c/doc/design_principles/fsm.html`.

Wig01 Ulf Wiger. Four-fold increase in productivity and quality. In *Workshop on Formal Design of Safety Critical Embedded Systems*, March 2001.