II.6.5 Lambdakalkyylistä

Loogikko Alonzo *Church* julkaisi vuonna 1936 λ-*kalkyylin* (calculus) seuraavien asioiden formalisointiin:

- Funktiot sääntöinä ("lauseke $f(x)$") joukko-opin ("parien $\langle x, f(x)\rangle$ joukko") sijasta.

- Mekaaninen laskeminen lausekkeita sieventämällä (eli $(2+3)/5$ on $5/5$ on 1).

  Samana vuonna (hieman aiemmin) Alan Turing oli julkaissut oman *laite*mallinsa mekaanisen laskettavuuden pohjaksi.

  Church-Turingin teesi: näin saadut (ekvivalentit) systeemit ovat oikea formaali vastine intuitiiviselle mekaanisen laskettavuuden käsitteelle.

- Leibnizin operaatio "korvaa tämä ilmaus tuolla ilmaukselle siinä yhteydessä".

Lisp-kielten suunnittelijat ottivat nämä ideat omiin tarkoituksiinsa. Myöhemmin niitä on alallamme hyödynnetty erityisesti ohjelmointikielten teoriassa.

Peruskäsitteet:

- "Riittävä" määrä *termimuuttujia* $x_0, x_1, x_2, .$

  Termimuuttujan esiintymä yksinään on yksinkertaisin *termi*, ja *vapaa* esiintymä.

- Jos t_1 ja t_2 ovat termejä, niin myös $(t_1\ t_2)$ o termi.

  Intuitiivisesti siinä *sovelletaan* funktiota t_1 argumenttiin t_2.

  Vapaat esiintymät ovat samat kuin termeiss t_1 ja t_2.

- Jos t on termi ja x termimuuttuja, niin $\lambda x.t$ termi.

  Intuitiivisesti siinä *abstrahoidaan* termistä t funktio jonka parametriposition nimi on x.

  Vapaat esiintymät ovat kuten termissä t pai että kaikki termimuuttujan x vapaat esiinty tulevat nyt *sidotuiksi* koska ne esittävät koh johon parametria vastaava argumentti kutsu tulee.

Termien sievennykseen on *β-reduktio:*
lauseke $((\lambda x.t)\ t')$ voidaan korvata
lausekkeella $t[x \leftarrow t']$ eli "korvaa kaikki
termimuuttujan x vapaat esiintymät termissä t
termillä t'".

- Termimuuttujille

$$x_j[x_i \leftarrow t] = \begin{cases} t & \text{kun } i = j \\ x_j & \text{kun } i \neq j. \end{cases}$$

- Sovelluksille $(t\ t')[x \leftarrow t''] = (t[x \leftarrow t''])\ t'[x \leftarrow t'']).$

- Abstraktioille

$$(\lambda x_j.t)[x_i \leftarrow t'] = \begin{cases} \lambda x_j.t & \text{kun } i = j \\ \lambda x_j.(t[x_i \leftarrow t']) & \text{kun } i \neq j. \end{cases}$$

Operaatiolta $t[x \leftarrow t']$ edellytetään kuitenkin lisäksi,
että jos jokin x' esiintyy vapaana termissä t', niin
mikään muuttujan x vapaa esiintymä termissä t ei
sijaitse minkään alitermin $\lambda x'.t''$ sisällä.

Muutenhan rekursiivinen korvaus $t''[x \leftarrow t']$ tekisi
ennen vapaasta esiintymästä x' sidotun esiintymän!

Lisäedellytys voidaan taata *α-konversiolla:*
alitermi $\lambda x'.t''$ voidaan tarvittaessa korvata
alitermillä $\lambda x''.t''[x' \leftarrow x'']$ kunhan korvaava
muuttuja x'' ei esiinny vapaana alitermissä t''.

Konversion intuitio: $\lambda x'$ esittelee *parametriposit*
eli "aukon" johon β-reduktio voi korvata
argumenttiterminsä, mutta tälle positiolle annet
nimi x' ei ole merkityksellinen
sisälausekkeen t'' ulkopuolella.

Vastaavasti β-reduktion intuitio on *proseduuriku*
kalvojen II.5 korvausmallissa: $((\lambda x.t)\ t')$ "suorite
kopioimalla argumentti t' parametripositioon x
kutsuttavassa rungossa t.

Useampiparametriset funktiot sisäkkäisin sovellu

$(\underline{(\lambda x_1.\lambda x_2.((f\ x_1)\ x_2)\ p_1)}\ p_2)$ on $\underline{(\lambda x_2.((f\ p_1)\ x_2}$
on $((f\ p_1)\ p_2)$

eli 2-parametrisen funktion f arvo argumenteilla
ja p_2. Välivaiheena on 1-parametrinen "osittain
kutsuttu" lauseke.

Menetelmän nimi on *kuritus* (Currying) loogikk
Haskell B. Curryn kunniaksi (vaikka sen keksikir
toinen loogikko Harold Schönfinkel).

Kurituksen vuoksi λ-kalkyylissä vähennetäänkin pakollisia sulkuja sopimalla, että termi $(t_1\ t_2\ t_3)$ tarkoittaa termiä $((t_1\ t_2)\ t_3)$ jne.

[Tämä muistuttaa kalvojen II.5.3 funktionkutsua, mutta Scheme-kielessä *ei* ole suoraan mahdollista kutsua funktiota vähemmillä parametreilla kuin se vaatii ja saada funktio joka odottaa loppuja. Monissa funktionaalisissa kielissä (kuten ML, Haskell,. . .) on. Scheme-parametrinvälitykseen palataan kalvoilla II.8.8.]

Haluaisimme funktioiden samuudelle *ekstensionaalisuusperiaatteen:* termit t ja t' kuvaavat samaa funktiota täsmälleen silloin kun $(t\ t'')$ ja $(t'\ t'')$ päätyvät (α-konversioita vailla) yhteiseen muotoon kun β-reduktiota sovelletaan niihin toistuvasti, olipa syöte t'' mikä tahansa.

On kuitenkin termejä, joihin β-redusoinnin toistaminen ei pääty: Olkoon termi $Y = \lambda x_1.(x_1\ x_1)$. Silloin sen sovellus itseensä $(Y\ Y)$ antaa tulokseksi jälleen $(Y\ Y)$ (muuttujat muuttaen).

Intuitiivisesti tällaiset termit vastaavat sellaista ohjelman suoritusta, jotka jäävät ikuiseen silmukkaan.

Mikä on silloin termin

$$((\lambda x_1.\lambda x_2.x_2)\ (Y\ Y))$$

kuvaama funktio?

1. Ikuinen silmukka jos tarpeettoman parametr termi $(Y\ Y)$ yritetään sieventää ennen sovel

2. Identiteettifunktio $\lambda x_2.x_2$ jos ei yritetäkään.

Ekstensionaalisuusperiaatetta on siis tarkennett kertomalla myös *missä järjestyksessä* β-redusoid

Scheme ja monet muut (funktionaalisetkin kute ML) ohjelmointikielet ovat valinneet "sisältä ulo -vaihtoehdon 1: Kutsuja laskee argumenttien ar ennen kutsun suoritusta.

Matemaattisena teoriana λ-laskenta onkin valin "ulkoa sisään" -vaihtoehdon 2: Valitaan kirjoitusjärjestyksessä vasemmanpuoleisin λ joh soveltaa β-reduktiota.

Kutsun suoritus laskee argumenttien arvot vast silloin *jos* niitä tarvitsee. Tämän ns. laiskan suorituksen Scheme-toteutukseen palataan kalvoilla II.8.10. (Haskell-ohjelmat suoritetaan laiskasti.)

Tyyppiteoria matemaattisen logiikan haarana [H97] saikin alkunsa λ-laskennan termien syntaktisena luokitteluna

hyviin joilla β-redusoinnit aikanaan pysähtyvät (tehtiinpä ne missä järjestyksessä tahansa)

huonoihin joilla voidaan joutua ikuiseen silmukkaan.

Esimerkkitermimme $Y = \lambda x_1.(x_1\ x_1)$ on huono:

- Olkoon parametri x_1 tyyppiä τ_1.
- Koska x_1 on kutsuttavan funktion paikalla lausekkeen Y sisällä, on τ_1 tyyppiä "funktio tyypin τ_2 syötearvoilta tyypin τ_3 tulosarvoille".

  Merkitään tätä funktiotyyppiä $\tau_2 \to \tau_3$.
- Koska x_1 on myös oma syötteensä, pitää tyyppien τ_1 ja τ_2 olla samat.
- Silloin myös tyyppien τ_2 ja $\tau_2 \to \tau_3$ pitää olla samat.
- Silloin tyypin τ_2 pitäisi olla *ääretön* $((\ldots \to \tau_3) \to \tau_3) \to \tau_3$ — ristiriita!

Tämä luokittelu — tyypinpäättely — on jälleen esimerkki symbolisesta laskutoimituksesta...

Kalvoilla II.5.5 esitellyssä Scheme-ydinkielen suoritusmekanismissa käytettiin muuttujien korvaamiseen arvoillaan sanakirjoja eikä β-reduk mekanismia $t[x \leftarrow t']$.

Näillä menetelmillä on kuitenkin sama intuitio:

- λ-laskennassa formaali parametri x korvataa *heti* arvollaan t' termissä t
- Suoritusmekanismissa merkitään sanakirjaan jos termin t parametriin x viitataan *myöhem* niin silloin pitääkin toimia ikään kuin sen pa olisi jo korvattu arvo t'.

Sanakirja on siis se *korvaushistoria*, joka olisi syntynyt tekemällä vastaavat β-reduktiot.

Merkintää $\lambda x.\ldots$ käytetään usein myös muiden funktiomerkintöjen kuten $x \mapsto \ldots$ tai $f(x) = \ldots$ sijasta.

Se esittää silloin yleistä parametrinvälitysmekani

II.7 Rekursio

Useimmissa ohjelmointikielissä esitellään *iteraatio- eli toistorakenteet* (`for`, `while`, `repeat until`,...) ja mainitaan että aliohjelmat voivat kutsua itseään *rekursiivisesti*.

Scheme-kielessä ei ole lainkaan toistorakenteita, vaan ainoa tapa tehdä toistoa on juuri rekursiivisilla aliohjelmilla [AS96,§1.3].

Tai pikemminkin: Scheme-kielessä on vain aliohjelmakutsu, eikä rekursiivisten ja muiden kutsujen välillä ole eroa.

Iterare (lat.) toistaa.

Recurrere (lat.) juosta takaisin.

Toistaminen on juoksemista takaisin *alkuun*.

Rekursiolla on helppo ilmaista *induktiivisesti* määriteltyjä operaatioita. Esimerkiksi kertomafunktion määritelmä

$$n! = \begin{cases} 1 & \text{kun } n = 0 \\ n \cdot (n-1)! & \text{muuten} \end{cases}$$

taipuu Scheme-kielelle lähes sellaisenaan:

```
(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1)))))
```

Rekursiivinen `if`-haara sanoo: "Pitää kertoa nyk
`n` luvulla joka saadaan kutsusta `(fac (- n 1))`.
Minäpä suoritan nyt tuon kutsun ja *palaan sen kanssa myöhemmin takaisin* tekemään kesken jä kertolaskun loppuun".

Scheme-toteutus joutuu siis pitämään kirjaa keskeneräisistä töistään; tätä kirjanpitoa voi aja jäljellä olevana lausekkeena. Suurilla parametrin arvoilla tämä kirjanpito ei enää mahdu muistiin; silloin joudutaan ohjelmaa muuttamaan kirjanpi pienentämiseksi.

Esimerkiksi laskettaessa kutsun `(fac 3)` arvoa aktiivinen sievennyskohta etenee yhä syvemmälle lausekkeessa:

```
(fac 3)
= ((lambda (n) (if (zero? n) 1 (* n (fac (- n 1))))) 3)
= (if (zero? 3) 1 (* 3 (fac (- 3 1))))
= (if #f        1 (* 3 (fac (- 3 1))))
= (* 3 (fac (- 3 1)))
= (* 3 (fac 2))
⋮
= (* 3 (* 2 (fac 1)))
⋮
= (* 3 (* 2 (* 1 (fac 0))))
⋮
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6
```

Sen eteen jää kirjanpito: polku

$$\underbrace{n \cdot ((m-1) \cdot ((m-2) \cdot \cdots \cdot ((m-p+1)}_{(\dagger)} \cdot (\texttt{fac}\ m-p)) \cdot$$

keskeneräisiä kertolaskuja, joka on juostava takaisin.

Rekursiivinen algoritmi voi esittää iteratiivista laskentaprosessia:

```
(define (mystery x)
  (if (= x 1)
      0
      (mystery (if (even? x)
                   (/ x 2)
                   (+ (* 3 x) 1)))))
```

esittää iteratiivisen algoritmin

while $x \neq 1$ **do**
  if x on parillinen **then**
    $x \leftarrow x/2$
  else
    $x \leftarrow 3 \cdot x + 1$
  end if
end while

(joka muuten näyttää pysähtyvän kaikilla alkuar x mutta tätä ei ole vielä kyetty todistamaan [H92,s.204]).

Siinä kirjanpitoa *ei tarvita* koska rekursiivisesta kutsusta saatu arvo on itsessään valmis vastaus ilman lisäkäsittelyä.

Tämä on iteratiivisen algoritmin idea: vastaavar prosessin tila voidaan aina lukea sen näkyvistä muuttujanarvoista (ja ohjelmakohdasta).

II.7.1 Loppukutsut

Iteratiiviset rekursiokutsut ovat siis niitä jotka eivät jätä muistijälkiä. Scheme-standardi vaatii (useimmista muista ohjelmointikielistä poiketen) että toteutus tekee sellaiset kutsut tilaa säästäen.

Ohjelmoijan on siis syytä tunnistaa sellaiset kutsut Scheme-ohjelmiensa (ajan- ja) tilantarpeen pienentämiseksi [KCR98,§3.5].

Ensiksi tunnistetaan annetun lambda-lausekkeen *loppuyhteydessä (tail context) olevat alilausekkeet*, intuitiivisesti ne jotka "tehdään lopuksi juuri ennen lambdasta poistumista".

Tarkastellaan kalvojen II.6.4 lambda-lausekkeen yleistä muotoa:

(**lambda** (x_1 ... x_k) d_1 ... d_m e_1 ... e_m)

missä kukin

- d_i on kalvojen II.6.2 paikallinen **define**-määritelmä.

  Kalvojen II.6.2 ja II.6.4 nojalla **lambda** siis av uuden paikallisen määrittelyalueen.

- e_j on suoritettava lauseke.

  Näistä viimeinen eli e_m antaa funktion arvor

  Edelliset eli $e_1, \ldots, e_{m-1}$ suoritetaan suorite *sivuvaikutustensa* (syöte/tulostus, uudelleensijoitus,. . .) vuoksi. Sivuvaikutuksi palataan kalvoilla II.9; vielä nyt Scheme-ohjelmissamme $m = 1$.

Koko tämän **lambda**-lausekkeen viimeinen vaihe laskea kutsulle arvo alilausekkeella e_m. Se ja vai on siis loppuyhteydessä.

Jos kalvojen II.5.4 ehtolauseke (**if** $e_?$ $e_{\#t}$ $e_{\#f}$) o loppuyhteydessä, niin myös sen haarat $e_{\#t}$ ja $e_{\#}$ ovat, koska jompi kumpi niistä tehdään ehtolau lopuksi. Ehto $e_?$ *ei* ole loppuyhteydessä, koska s jälkeen tehdään vielä jompi kumpi haaroista.

Jos lambda-lauseketta näin tutkittaessa "ulkoa sisäänpäin" päädytään loppuyhteydessä olevaan alilausekkeeseen

`((lambda (`$var_1 \ldots var_k$`)` $laus_1 \ldots laus_m$`)` $val_1 \ldots val_k$`)`

niin

1. selvitetään ne (lausekkeen $laus_m$ ali)lausekkeet jotka ovat loppuyhteydessä tähän sisempään lambdaan ja

2. otetaan ne myös ulomman lambdan loppuyhteyteen

koska tämän alilausekkeen viimeinen teko on laskea (oman ali)lausekkeensa $laus_m$ viimeinen teko.

(Vertaa kalvojen II.6.5 β-reduktio.)

Siis kalvojen II.6.4 `let`-rakenteessa

`(let ((`var_1 val_1`)...(`var_k val_k`))` $laus_1 \ldots laus_m$`)`

loppuyhteydessä on vain $laus_m$.

Näillä säännöillä voi (johdettuja lausekkeita purkaen) selvittää loppuyhteydessä olevat lausekkeet.

Esimerkiksi kalvojen II.7 funktiossa `mystery` loppuyhteydessä ovat vain vakio `0` ja rekursioku

Loppuyhteydessä olevaa funktionkutsua nimitet *loppukutsuksi* (tail call) ja sen Scheme-järjestel suorittaa "in situ", "paikallaan".

(Jos kutsu ei ole loppuyhteydessä, se varaa tiet tarvitsemansa kirjanpitomuistin.)

Rekursiivinen loppukutsu(jen jono) on Scheme- esittää iteraatio: esimerkiksi funktio `mystery` ja kalvojen II.6 funktio `newton` toimivat itse asiassa **while**-silmukoina sellaisinaan ilman että ohjelmo tarvitsee käyttää erillistä rakennetta.

Etuna on, että ohjelmoija voi kirjoittaa induktiiv määritelmän sellaisenaan, ja Scheme-toteutus p suorittamaan sen muistia säästäen. Vasta jos si onnistuta, joutuu ohjelmoija muokkaamaan koodiaan.

Tälläistä "epäaitoa" loppurekursiota kutsutaan *häntä- tai takarekursioksi* (tail recursion) ja Scheme-kielen standardin sanotaan vaativan se *poistoa* (elimination) jota pidetään tavallisesti (ohjelmoijien tai) ohjelmointikieli*toteutusten optimointi(palvelu)na*.

Kun imperatiivinen ohjelmoija haluaa kirjoittaa silmukan, hän kirjoittaa `while`-avainsanan tms. eikä hänen tarvitse nimetä silmukkaansa.

Kun Scheme-ohjelmoija haluaa kirjoittaa silmukan, hän siis esittää sen loppurekursiivisena funktiona, joka täytyy nimetä.

Kalvojen II.6.4 nimetön lambda-funktio ei nimittäin voi olla rekursiivinen (ilman temppuilua) – millä nimellä se kutsuisi itseään?

Scheme-stadardissa on kalvojen II.6.4 tapaan johdettuna lausekkeena *nimetty* `let` muotoa

`(let` *nimi* `(`$(var_1\ val_1)\ldots(var_k\ val_k)$`)` $laus_1 \ldots laus_m$`)`

jossa lisäksi rungon lausekkeisiin $laus_i$ – muttei muualle – näkyy k-parametrinen funktio *nimi*.

Tämän paikallisen funktion kutsu (*nimi* $val'_1 \ldots val'_k$) tekee rungon uudelleen arvoilla $var_i = val'_i$.

Loppukutsuna se siis vastaa silmukan seuraavaa kierrosta. Sitä voi myös käyttää tavallisena rekursiokutsuna muissa kuin loppuyhteyksissä.

Esimerkiksi kalvojen II.6.4 iteratiivinen aliohjelm
`newton-aux` voidaan korvata nimetyllä `let`-lausek
seuraavasti:

```
(define h 1/1000)

(define (newton f x)
  (define (d x)
    (/ (- (f (+ x h)) (f x)) h))
  (let apu ((x x))
    (if (< (abs (f x)) h)
        x
        (apu (- x (/ (f x) (d x)))))))

(define (juuri p)
  (lambda (x)
    (newton (lambda (y)
              (- (expt y p) x))
            x)))

(define neliojuuri (juuri 2))
(define kuutiojuuri (juuri 3))
```

Erityisesti alustuksessa `(x x)` silmukan sisäinen (vasemmanpuoleinen) `x` saa alkuarvokseen parametrina saadun (oikeanpuoleisen) arvon `x`, j
peittyy näkyvistä.

II.7.2 Rekursiosta iteraatioksi

Yleistä rekursiota käyttävä ohjelma(nosa) voidaan usein muuntaa iteratiiviseksi käyttämällä *kerääjiä* (accumulators).

Kerääjä on lisäparametri johon kootaan iteraation aikana niin paljon lisätietoa että sen perusteella voidaan lopuksi laskea alkuperäinen vastaus.

Esimerkiksi kalvojen II.7 kertomafunktio `fac` ei ollut iteratiivinen, koska kutsun (`fac` $m-p$) vastausta piti myöhemmin täydentää kertomalla se tulolla (†) eli $m \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot (m-p+1)$ lopullisen vastauksen $m!$ saamiseksi.

Kootaan siis tämä täydennysinformaatio kerääjäparametriin `a` ja pidetään näin huolta siitä että rekursiokutsu siirtyy loppuyhteyteen.

Kehitetään vastaava Scheme-funktio `fac2` vaihe vaiheelta *tehtävän spesifikaation ohjaamana ohjelmointityönä*.

- Ensiksi (`define (fac2 n) (fac-acc n` α)) mis `fac-acc` on iteratiivinen apufunktio jonka kerääjäparametrille `a` antaa alkuarvon vielä tuntematon Scheme-lauseke α.

- **Idea:** kehitetäänpä sellainen `fac-acc` että (`fac-acc m a`) $= m! \cdot a$.

  Idean jälkeen loppu on melko mekaanista.

  Esimerkiksi koska halutaan $n!$ = (`fac2 n`) = (`fac-acc n` α) $= n! \cdot \alpha$ on valittava $\alpha =$ `1`.

- Aloitetaan kehitys (`define (fac-acc m a)` β) missä vaaditaan siis $\beta = m! \cdot a$.

- Edetään *induktiolla parametrin* `m` *suhteen*.

  Koska $m \in \mathbb{N}$ jaetaan konstruktio β = (`if (zero? m)` γ δ) missä γ on perustap $m = 0$ ja δ induktiivinen tapaus $m > 0$.

  Induktiossa on muistettava palata vain tapaukseen $m-1$ jotta pysähtyminen perustapaukseen olisi taattu.

- Nyt mekaanisesti $\gamma = m \cdot a = 0! \cdot a =$ `a`.

- Samoin
  $\delta = m! \cdot a =$ `(fac-acc (- m 1) η)` $\underset{\text{ind.ol.}}{=} (m-1)! \cdot \eta$
  kunhan valitaan $\eta = m \cdot a =$ `(* m a)`.

Yhdistetään kaikki lausekkeet $\alpha, \ldots, \eta$ paikoilleen:

```
(define (fac2 n)
  (fac-acc n 1))

(define (fac-acc m a)
  (if (zero? m)
      a
      (fac-acc (- m 1)
               (* m a))))
```

- Iteratiivinen kuten pitikin:

```
(fac2 3 1)
= (fac2 (- 3 1) (* 1 3))
= (fac2 2 3)
⋮
= (fac2 (- 2 1) (* 3 2))
= (fac2 1 6)
⋮
= (fac2 (- 1 1) (* 6 1))
= (fac2 0 6)
⋮
= 6
```

- Korrekti jo *konstruktionsa nojalla*.

Vaikka tämä vaatikin hieman matematiikkaa, ol
silti helpompaa kuin

- keksiä ensin algoritmi ja todistaa se(n idea)
  myöhemmin oikeaksi

- kokeilla ja testata ja korjata ja testata ja ko
  ja...

Kalvojen II.3 funktionaalisessa ohjelmointitavass
säilytettiin Leibnizin periaate, ja sen nojalla voit
tehdä *yhtälöpäättelyä* "tämä lauseke = tuo laus

Tilaperustaisen ohjelmoinnin
"tämä käskysarja = tuo käskysarja" on vaikeam
koska

- on vaikeampi nähdä milloin kaksi eri sarjaa
  jotavat samaan lopputulokseen

- lopputulosten samuus voi riippua alkutilante

II.7.3 Korkeampaa iteroituvuutta

Koska kielessämme on kalvojen II.6 korkeamman kertaluvun funktiokäsite, voidaan kalvojen II.7.2 kerääjätekniikkaa yleistää keräämällä *funktio joka kertoo mitä nykyisen kutsun tuloksella pitikään tehdä*.

Otetaan esimerkiksi (yleistetyt) murtojatkeet (continued fractions)

$$\cfrac{1}{f(n)+\cfrac{1}{f(n-1)+\cfrac{1}{f(n-2)+\cdots\cfrac{1}{f(1)}}}}$$

jotka taipuvat rekursiiviseen Scheme-muotoon

```
(define (cf f n)
  (define (cf-aux m)
    (if (zero? m)
        0
        (/ (+ (f m)
              (cf-aux (- m 1))))))
  (cf-aux n))
```

Muokataan siitä iteratiivinen versio lisäämällä apufunktioon `cf-aux` kerääjä*funktio* `a` invariantti "kun parametrina annetaan *nykyisen* rekursioku antama tulos, saadaan vastauksena *koko* rekurs tulos": `(cf-acc m a)`=`(a (cf-aux m))`=`(cf f n)`.

Siis `a` kerää nyt tietoa siitä miten laskentaa pitä *jatkaa* nykyisen rekursiokutsun jälkeen.

Aluksi `(define (cf2 f n) (cf-acc n` α`))` missä

`(cf f n)` = `(cf-aux n)` = `(cf2 f n)` = `(cf-acc n`

pätee kun kerääjäfunktion `a` alkuarvoksi valitaan identtinen kuvaus α = `(lambda (v) v)`: ensimmä rekursiokutsun tulosta `v` ei tarvitse (eikä pidä) jatkokäsitellä, vaan se annetaan sellaisenaan vastauksena.

Jatkamalla kalvojen II.7.2 tapaan saadaan

`(define (cf-acc m a) (if (zero? m)` γ δ`))`

missä γ edustaa ei-rekursiivista haaraa `0`. Tällais haarassa tämä rekursiokutsu on antamassa tulostaan, ja `a` tietää miten se pitää jälkikäsitell joten γ = `(a 0)`.

Lauseke δ on puolestaan rekursiivinen haara. Koska apufunktiosta `cf-acc` halutaan iteratiivinen, on

$$\delta = \texttt{(cf-acc (- m 1)}\ \eta\texttt{)}\ \text{missä} \qquad (\dagger)$$
$$\eta = \texttt{(lambda (v) (a (/ (+ (f m) v))))}$$

on kerääjäfunktion `a` päivitetty muoto seuraavaa iteraatiokierrosta varten:

- Parametrina `v` saadaan (aikanaan) tämän rekursiokutsun ($\dagger$) vastaus $v = \texttt{(cf-aux (- m 1))}$.
- Tätä vastausta v jatkokäsitellään ensin paikallisesti lausekkeella
$$w = \frac{1}{f(m)+v} = \texttt{(cf-aux m)}.$$
- Paikallisen jatkokäsittelyn tulos w annetaan lopuksi parametrina saadulle kerääjäfunktiolle `a` ei-paikallista loppukäsittelyä varten.

Toistetaan sama järkeily formaalisti yhtälöillä:

`(cf-acc m a)` missä $\texttt{m} > 0$

$= \texttt{(a (cf-aux m))}$

kerääjän `a` oikeellisuusehto etuperin

$= \texttt{(a (/ 1 (+ (f m) (cf-aux (- m 1)))))}$

funktion `cf-aux` määritelmä kun $\texttt{m} > 0$

$= \texttt{((lambda (v) (a (/ 1 (+ (f m) v)))) (cf-aux (- m 1)))}$

kalvojen II.6.5 β-reduktio takaperin

$= \texttt{(cf-acc (- m 1)}\ \underbrace{\texttt{(lambda (v) (a (/ 1 (+ (f m) v))))}}_{\eta}$

kerääjän `a` oikeellisuusehto takaperin

Lopputulos on yhteen koottuna

```
(define (cf2 f n)
  (define (cf-acc m a)
    (if (zero? m)
        (a 0)
        (cf-acc (- m 1)
                (lambda (v)
                  (a (/ (+ (f m)
                           v)))))))
  (cf-acc n (lambda (v) v)))
```

jossa esimerkiksi kutsu `(cf2 g 2)` päättyy lausekkeen

```
((lambda (v1)
   ((lambda (v2)
      ((lambda (v)
         v)
       (/ (+ (g 2) v2))))
    (/ (+ (g 1) v1))))
 0)
```

arvon laskentaan. Tämä taas on kalvojen II.6.5 β-reduktioiden jälkeen sama lauseke

```
(/ (+ (g 2) (/ (+ (g 1) 0))))
```

jonka vastaava alkuperäinen kutsu `(cf g 2)` tuottaa rekursiokirjanpitonaan, kuten pitikin.

Abstrahoidaan ratkaisua edelleen:

- Nyt `cf-acc` palauttaa arvonaan kerääjäfunkt
- Tuloksen `a` kokoamiseen käytetään matematiikasta tuttua funktioiden f ja g yhdistämisoperaattoria $f \circ g$: $x \mapsto f(g(x))$.
- Tuloksen `a` alkuarvona on operaattorin $\circ$ neutraalialkio id: $x \mapsto x$.
- Tulokseen `a` lisätään murtojatkeessa toistuv lauseke.

```
(define (o f g)
  (lambda (x)
    (f (g x))))
(define id
  (lambda (v) v))
(define (cf2b f n)
  (define (cf-acc m a)
    (if (zero? m)
        a
        (cf-acc (- m 1)
                (o a
                   (lambda (v)
                     (/ (+ (f m)
                           v)))))))
  ((cf-acc n id) 0))
```

Oleellisesti sama funktio kuin kalvojen II.7.3 `fac`

II.7.4 Jatkeet

Kalvoilla II.7.3 tehtiin rekursiivisesta prosessista iteratiivinen keräämällä iteraation kuluessa funktiota joka kertoi *kuinka laskentaa on jatkettava* iteraation jälkeen.

Tälläisiä funktioita kutsutaan *jatkeiksi* (continuation) ja ohjelmointityyliä jossa ohjelmoija hallinnoi jatkeita itse – järjestelmän sijasta – *jatkeidenvälitystyyliksi (Continuation Passing Style, CPS)* [D96,§3.3-3.4].

+ Jatkeita käsittelemällä voi toteuttaa *minkä tahansa sarjallisen kontrollirakenteen*.

– Epästrukturoitu (eli vaikea) kontrollivuo joka jää ohjelmoijan vastuulle.

+ Korkeamman kertaluvun kieli ei siis tarvitse erillistä rekursiomekanismia.

  CPS onkin suosittu funktionaalisten kielten *toteutus*tason periaatteena.

– Rekursion tilavaativuus ei katoa vaan siirtyy järjestelmältä parametrinvälitykseen.

Tyylin tunteminen pelastaa kuitenkin usein pula
Esimerkkinä olkoon *virheidenkäsittely rekursioss*

Vaaditaan kalvojen II.7.3 murtojatkefunktiolta e
nollalla jakoon saa enää "kaatua" vaan silloin *k*
rekursion on palautettava `#f`.

Jos päivitämme alkuperäistä rekursiivista funkti
`cf` niin joudumme tarkastamaan apufunktiossa
`cf-aux`...

- rekursiokutsun jälkeen raportoiko se nollalla
  ja jos raportoi, niin lähetettävä raportti edel
  ja

- onnistuneenkin rekursiokutsun jälkeen onko
  jakajaksi nyt tulossa 0, eli pitääkö nyt lähet
  ensimmäinen virheraportti.

Tähän ongelmaan tarjoavat monet ohjelmointik
erilaisia *poikkeustenkäsittelymekanismeja* (excep
handling): `throw-catch` / Common Lisp [S98,§9.
`try-catch` / Java sekä C++ [S97,§14.1], `try-wi`
ML-kielet [CM98,§4.1],...

Scheme-kielessä ei ole tähän valmista mekanism

Mutta jos ongelmallinen rekursio kirjoitetaankin jatkeidenvälitystyylillä, tulee mahdolliseksi poistua rekursiosta "kesken kaiken": *Parametrina saatua "tee lopuksi tämä" -jatketta ei ole pakko kutsua!*

Sen sijaan voidaankin (vaikkapa) käsitellä virhe:

```
(define (cf3 f n)
  (define (cf-exit m a)
    (if (zero? m)
        (a 0)
        (cf-exit (- m 1)
                 (lambda (v)
                   (let ((denom (+ (f m) v)))
                     (if (zero? denom)
                         #f
                         (a (/ denom))))))))
  (cf-exit n (lambda (v) v)))
```

Tässä siis luodaan jatke, joka sisältää virheenkäsittelyn: jos jakajaksi `denom` on tulossa 0, niin *oikaistaan* ohi lopun "tulevaisuuden" `a` ja palataan suoraan kutsujaan virhearvolla `#f`.

Näin voidaan toimia silloin, kun virhe ilmenee rekursiosta paluumatkalla.

Jos virhe ilmenee jo rekursioon edetessä, niin käsittely on vielä helpompaa: koska rekursio on muutettu iteraatioksi, niin riittää yksinkertaisest poistua silmukasta — jälleen oikaisten ohi jatke

Oletetaan esimerkissämme lisäksi, että myös parametrina saatu funktio `f` voi antaa tuloksena numeron sijasta virhearvon `#f`:

```
(define (cf4 f n)
  (define (cf-exit m a)
    (if (zero? m)
        (a 0)
        (let ((fm (f m)))
          (if fm
              (cf-exit (- m 1)
                       (lambda (v)
                         (let ((denom (+ fm v)
                           (if (zero? denom)
                               #f
                               (a (/ denom))))
              #f))))
  (cf-exit n (lambda (v) v)))
```

Itse asiassa muiden kielten keskeytysmekanismie ajatella luovan *kaksi* jatketta sisältämälleen `try`-ohjelmanosalle: sen tavallisen (näkymättöm perusrekursiojatkeen ja `catch`-virhejatkeen.

II.7.5 Järjestelmän jatkeet

Kalvoilla II.7.4 nähtiin miten Scheme-kielessä voi ilmaista pakeneminen kesken rekursion ohjelmoijan ylläpitämillä jatkeilla.

Scheme-kielessä myös *järjestelmän* ylläpitämä tämänhetkinen suorituksessa oleva jatke on ohjelmoijan ulottuvilla: kirjastofunktiokutsulla

`(call-with-current-continuation (lambda (`c`) `e`))`

lausekkeessa e muuttuja c saa arvokseen yksiparametrisen *pakofunktion*. [KCR98,§6.4;D96,§5.6]. Usein käytetty synonyymi on `call/cc`.

Jos tehdään pakofunktion kutsu (c a) niin suoritus siirtyy *välittömästi ja palaamatta* jatkamaan vastaavasta `call/cc`-kutsusta ja antaa sille arvon a.

Toisin sanoen, `call/cc` paketoi normaalisti näkymättömän järjestelmäjatkeen funktioksi, jota kutsumalla nykyinen järjestelmäjatke vaihtuu paketoituun.

Esimerkiksi kalvojen II.7.4 ongelman olisi voinut ratkaista järjestelmäjatkeilla suoraan poistamatt ensin rekursiota:

```
(define (cf4 f n)
  (call/cc (lambda (exit)
             (define (cf-cc m)
               (if (zero? m)
                   0
                   (let ((denom
                          (+ (f m)
                             (cf-cc (- m 1)))))
                     (if (zero? denom)
                         (exit #f)
                         (/ denom)))))
             (cf-cc n))))
```

Kääntäen, `call/cc`-kutsuja käyttävä ohjelma voi aina kirjoittaa jatkeenvälitystyylillä ilman näitä kutsuja, mutta silloin saatetaan joutua kirjoittam tyylin mukaiset versiot kirjastofunktioistakin.

Koska paketti c on tavallinen Scheme-funktio, v suoritus "pomppia" lausekkeesta ulos ja takaisi sisään. Kirjastofunktiokutsulla

`(dynamic-wind` sisään lauseke ulos`)`

voi kertoa että (esim. alustukset) "sisään" on tehtävä joka kerran kun lausekkeeseen pompata ja "ulos" vastaavasti.

Kirjallisuutta

AS96 H.Abelson&G.J.Sussman: *Structure and Interpretation of Computer Programs. Second Edition.* MIT Press, 1996.

AHV95 S.Abiteboul,R.Hull,V.Vianu: *Foundations of Databases.* Addison Wesley, 1995.

AHU83 A.V.Aho,J.E.Hopcroft&J.D.Ullman: *Data Structures and Algorithms.* Addison Wesley, 1983.

B01 I.Bratko: *Prolog Programming for Artificial Intelligence. Third Edition.* Addison Wesley, 2001.

CM87 W.F.Clocksin&C.S.Mellish: *Programming in Prolog. Third, Revised and Extended Edition.* Springer-Verlag, 1987.

CM98 G.Cousineau&M.Mauny: *The Functional Approach to Programming.* Cambridge University Press, 1998.

DD98 C.J.Date&H.Darwen: *Foundation for Object/Relational Databases: The Third Manifesto.* Addison Wesley, 1998.

DEC96 P.Deransart,A.Ed-Dbali&L.Cervoni: *Prolo The Standard. Reference Manual.* Springer-Verlag, 1996.

D96 R.K.Dybvig: *The Scheme Programming Language: ANSI Scheme. Second Edition.* Prentice Hall, 1996.

H92 D.Harel: *Algorithmics. Second Edition.* Ac Wesley, 1992.

H97 J.R.Hindley: *Basic Simple Type Theory.* Cambridge University Press, 1997.

KCR98 R.Kelsey, W.Clinger&J.Rees (toim.): Revi Report on the Algorithmic Language Sche *Higher-Order and Symbolic Computation* 11(1), 1998 **tai** *ACM SIGPLAN Notices* 3 1998.

L87 J.W.Lloyd: *Foundations of Logic Programming. Second, Extended Edition.* Springer-Verlag, 1987.

P02 B.C.Pierce: *Type Theory and Programming Languages.* MIT Press, 2002.

S98 S.Slade: *Object-Oriented Common Lisp*. Prentice Hall, 1998.

SS86 L.Sterling&E.Shapiro: *The Art of Prolog*. MIT Press, 1986.

S97 B.Stroustrup. *The C++ Programming Language. Third Edition.* Addison-Wesley, 1997.

T99 S.Thompson: *Haskell: The Craft of Functional Programming. Second Edition.* Addison-Wesley, 1999.