5
Continuations.

Motivation og definition.
Eksempler.
Kontekst-begrebet.
Escape funktioner.
Continuation-passing style.
Hale-rekursion igen.

Noter

Som sædvanlig findes der en fil med de Scheme definitioner (og lidt flere til), der er diskuteret i dette kapitel. Det er filen continuations.scm i normark/public/ps4.

Motivation.

- Vi har behov for en mekanisme, der tillader os at *håndtere undtagelser* (exceptions) i funktions-orienterede programmer.
- Der er behov for et semantisk begreb, som tillader os at *beskrive meningen af hop* (gotos) og tilsvarende primitiver, der eksplicit ændrer på kontrolforløbet i imperative programmer.
- Det vil være interessant at have en primitiv mekanisme, som tillader os at *definere nye kontrolstrukturer* (for såvel imperative som funktionsorienterede sprog).

Et muligt svar på ovenstående er **continuation begrebet**.

Noter

Continuations i funktions-orienterede sprog.

En *continuation* af et udtryk E i et omkringliggende udtryk C betegner det fremtidige forløb af beregningen, som venter på værdien af E i C.

En continuation af E fuldfører den beregning, som er igang, givet værdien af E.

Continuations i Scheme:

(call-with-current-continuation
(lambda (cont) ...))

- Continuation "indfanges", og bindes til cont.
- (lambda (cont) ...) kaldes.
- Hvis cont ikke kaldes, er værdien af ovenstående lig med værdien, som returneres af lambdaudtrykket.
- Hvis cont kaldes, skal der overføres én parameter til denne. Værdien af denne parameter bliver værdien af hele call-with-current-continuation kaldet.

Eksempel:

```
(+ (call-with-current-continuation
     (lambda (e)
       (* 4 (e 3))))
   5)                          => 8
```

Continuation, som kaldes e, er beregningen af summen.
Kald af e giver direkte en værdi til den første addend i summen.

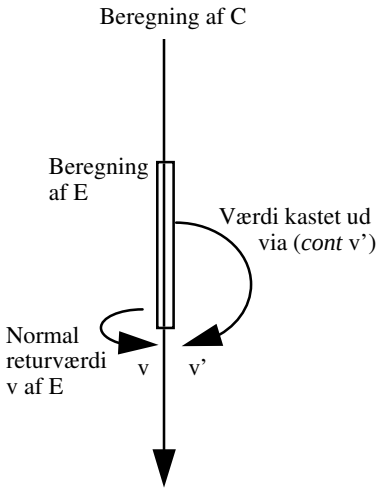
PS4 - Continuations

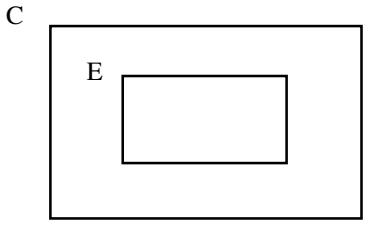
Noter

Primitivet call-with-current-continuation har et langt navn. Derfor benytter man ofte synonymet call/cc for denne. Hvis dette synonym ikke er defineret i dit Scheme system, er det let at gøre:

(define call/cc call-with-current-continuation)

Man kunne måske forfalde til at tro, at der er noget *orakulært* omkring continuations. Det er der ikke. Måden man via en continuation "fuldfører" den "fremtidige" beregning på, sker ved simpelthen at køre programmet.





Vi antager at vi i C omkring E indfanger continuation *cont* af E, som vi kan udnytte i kroppen af E:

(call-with-current-continuation(cont) E)

Continuations i Scheme: "Første klasses ulve i fåreklæder".

- Continuations i Scheme er objekter af *første klasse*.
  - Overføres som parametre.
  - Returneres som resultat af funktioner.
  - Gemmes i datastrukturer.
- Continuations i Scheme manifesterer sig som funktioner med én parameter.
  - Parameteren er et udtryk, hvis værdi substitueres ind i stedet for (call-with-current-continuation ...) kaldet, når "continuation-funktionen" kaldes.
- På implementationsniveau er en continuation
  - Et kontrolpunkt.
  - En reference til en stak af activation records, som afspejler den dynamiske kæde af funktionskald.

Noter

Vi har tidligere gjort et stort nummer ud af funktioner som første klasses objekter i Scheme, og funktions-orienterede sprog generelt. Da continuations i Scheme er klædt ud som en funktion, er det en ganske naturlig ting, at også continuations bliver af første klasse.

Lidt længere fremme i dette kapitel vil vi se eksempler på, at continuations kan gemmes i forskellige simple datastrukturer. Coroutine eksemplet kan specielt anbefales.

Fra en implementationsbetragtning betyder indførelse af continuations (lige som iøvrigt førsteklasses funktioner) at vi ikke kan realisere et Scheme program omkring én stak. En række af dynamiske funktionskald kan afbrydes ved aktivering af en eller anden tilgængelig continuation. Senere kan den igangværende beregning måske genoptages, hvis vi har fastholdt den relevante continuation på afbrydelsestidspunktet.

Eksempel på undtagelse: Længden af en uegentlig liste.

Exception mekanisme

```
(define (list-length l)
 (call-with-current-continuation
  (lambda (do-exit)
   (letrec ((list-length1
             (lambda (l)
               (cond ((null? l) 0)
                     ((pair? l) (+ 1 (list-length1 (cdr l))))
                     (else (do-exit 'improper-list))))))
     (list-length1 l)))))
```

```
> (list-length (list 5 7 9))
3
> (list-length (cons 5 (cons 7 (cons 9 '()))))
3
> (list-length (cons 5 (cons 7 9)))
improper-list
```

do-exit betegner en continuation, som giver den umiddelbare værdi af funktionen list-length.

Noter

Opgave: Ovenstående definition af funktionen list-length indeholder en lokal funktion list-length1, som laver det egentlige arbejde. For at overbevise sig om, at denne er nødvendig, kan man forsøge sig med følgende definition af list-length, som dog er forkert:

```
(define (list-length l)
    (call-with-current-continuation
     (lambda (do-exit)
       (cond ((null? l) 0)
             ((pair? l) (+ 1 (list-length (cdr l))))
             (else (do-exit 'improper-list))))))
```

Hvad er problemet?

Eksempel: Søgning i et et træ.

Exit mekanisme:

```
(define (find-in-tree tree pred)
 ;; traverse the tree, and return a node which satisfies the predicate pred.
 (call-with-current-continuation
  (lambda (found)
   (letrec
    ((find-in-tree1
            (lambda (tree pred)
                (if (pred tree)
                    (found tree)
                    (let ((subtrees (subtree-list tree)))
                        (for-each
                           (lambda (subtree) (find-in-tree1 subtree pred))
                           subtrees)))
                nil)))
     (find-in-tree1 tree pred)))))
```


Noter

Dette eksempel ligner det forrige i struktur, men formålet er et andet. Eksemplet består i, at vi gennemløber et træ ved først at konsultere roden, dernæst rekursivt alle deltræer. Ved binære træer svarer dette til et pre-order gennemløb. På hvert træ i gennemløbet kaldes prædikatet pred. Når denne første gang returnerer *sand*, ønsker vi det pågældende træ returneret. Generelt, vil vi være langt inde i det rekursive gennemløb, og vi ønsker blot at komme ud med værdien. Vi har sat en continuation op, som er i stand til at gribe den ønskede værdi. Vi kaster værdien ud til griberen ved at påkalde os den continuation, som vi har indfanget yderst i funktionen find-in-tree.

Scheme primitivet for-each er egentligt et imperativt primitiv, idet det kaldes for sin side-effekts skyld. Iøvrigt svarer det til uncurried map. Vi har ikke i det ovenstående brug for værdien af iterationen over subtræer. Hvis vi "falder helt igennem" (rekursionen slutter, og den yderste if returnerer) er resultatet af funktionskaldet nil (som altså her signalerer, at vi ikke fandt hvad vi søgte). I modsat fald har vi undervejs i rekursionen fundet hvad vi søgte efter, og dermed kalder vi continuation found, som også vil afsluttet kaldet af find-in-tree.

Se filen continuations.scm for definitioner og anvendelser omkring find-in-tree.

Eksempel: Lagring af continuations.

Lagring af en continuation, og senere genoptagelse af beregningen fra det indfangne punkt.

```
(define (fak n)
  (if (= n 0)
      (call-with-current-continuation
       (lambda(cont) (begin (set! retry cont) 1)))
      (* n (fak (- n 1)))))
```

```
> (define retry nil)
retry
> (fak 5)
120
> (retry 1)
120
> (retry 2)
240
```


Noter

Som man ser falder dette eksempel uden for funktionsorienteret programmering, idet vi benytter Scheme assignmentet set!.

Coroutiner med continuations.

```
(define other nil)

(define (producer)
  (let ((tal (list 5 8 7 5 9)))
    (while tal
      (call-with-current-continuation
       (lambda (c)
         ((exchange c) (car tal))))
      (set! tal (cdr tal)))))

(define (consumer)
  (call-with-current-continuation
   (lambda (exit)
     (set! other exit)
     (while #t
       (let ((val (call-with-current-continuation
                   (lambda (c)
                     ((exchange c) nil)))))
         (write (* val val)) (newline))))))
```

```
(define (exchange new)
  (let ((old-other other))
    (set! other new)
    old-other))
```

```
> (consumer)
#f
> (producer)
25
64
49
25
81
#f
```

PS4 - Continuations

Noter

I et coroutineforhold kan man stoppe udførelsen af en procedure, for senere at genoptage (resume) udførelsen på det sted, hvor udførelsen stoppede (og altså ikke forfra, som ved almindelige procedurekald). I det ovenstående er der en coroutine, der er producent af tal, som forbruges af den anden coroutine på den måde at tallene udskrives på kvadreret form. Ligesom det foregående eksempel tillader vi os selv brug af imperative primitiver til realisering af coroutineforholdet.

Hvordan virker ovenstående?

1. consumer kaldes, hvorved other assignes til exit. Når exit aktiveres vil consumer returnere umiddelbart.

2. Første exchange bevirker, at exit kaldes, hvorved consumeren terminerer. Dette svarer til et Detach, som det kendes fra et konventionelt (Simula-agtigt) coroutine-begreb. Rollen af exit er nu udspillet. Other betegner nu den indre continuation i consumer.

3. Producer kaldes. Den gennemløber listen, og for hvert element genoptages consumer, som udskriver det overførte tal, og vender tilbage til producer. Dette svarer til et Resume som det kendes fra et konventionelt (Simula-agtigt) coroutine-begreb. Producer slutter, og efterlader other til at være consumerens indre continuation.

Intentionen med other er altså, at denne variable på ethvert tidspunkt efter at produceren er startet op skal pege på den anden part i samarbejdet. Når exchange kaldes med en continuation c, lægges c over i other, medens den tidligere værdi af other returneres fra exchange. Dette befordrer coroutine begrebets “resume”.

Kontekster.

- En kontekst er en funktion med én parameter.
- *En kontekst af et vel-identificeret sted (deludtryk) X i en beregning af udtrykket E* er en funktion, som indfanger resten af beregningen af E fra stedet X, og givet X som parameter.
- Hvordan laves en kontekst af X i E:
  - Udfør E indtil X mødes.
  - Lav en funktion G = (lambda(y) F), således at (G X) giver samme værdi som E, og således at (lambda(y) F) repræsenterer resten af beregningen fra stedet X i E.

Eksempler:

Konteksten af (+ 5 6) i (+ 3 (* 4 (+ 5 6))): (lambda(v) (+ 3 (* 4 v)))

Konteksten af (* 3 4) i (if (= 0 5) (+ 3 (* 4 (+ 5 6))) (* (+ (* 3 4) 5) 2)) er (lambda(v) (* (+ v 5) 2))

Noter

Efter nu at have indført og (forhåbentligt) forstået continuationbegrebet på det intuitive plan vil vi indføre to andre begreber, hvormed continuationbegrebet kan præciseres.

Det første af disse begreber er *kontekster*, som diskuteres på denne side.

Det andet begreb er *escape-funktioner*, som diskuteres på efterfølgende sider.

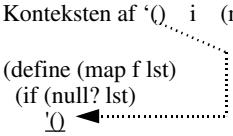
Fremstillingen af continuations ved brug af kontekster og escape-procedurer stammer fra bogen "Scheme and the Art of Programming" af Springer og Friedman fra MIT Press og McGraw-Hill.

I det første af eksemplerne ovenfor er (+ 5 6) det inderste udtryk, som beregnes før produktet og før den ydre sum, fordi Scheme har applicative-order evaluering. (lambda (v) (+ 3 (* 4 v))) udtrykker resten af beregningen, og ((lambda(v) (+ 3 (* 4 v))) (+ 5 6)) har samme værdi som det oprindelige udtryk (+ 3 (* 4 (+ 5 6))).

Det andet eksempel illustrerer, at vi udfører if og vælger else-delen inden vi former konteksten.

Flere eksempler på kontekster.

Konteksten af '() i (map (right-section + 1) (list 1 3 5)).



```
(define (map f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (map f (cdr lst)))))
```

er

```
(lambda (v)
  (cons 2 (cons 4 (cons 6 v))))
```


Noter

Eksemplet på denne side viser, at der kan være stor syntaktisk forskel på de kildeudtryk, vi arbejde med, og den kontekst, vi kan ende op med. I det ovenstående søger vi konteksten af et sted, som mødes inderst inde i en rekursiv proces. Da udtrykket (map (right-section + 1) (list 1 3 5)) tillader rekursionen at udvikle sig, kan vi let danne den faste del af konteksten, altså (cons 2 (cons 4 (cons 6 ...))). Bemærk, at vi benytter ikke-curried map.

Man kan spørge sig selv, om konteksten er *entydigt bestemt* ud fra definitionen. Definition sagde bl.a.:

> Lav en funktion G = (lambda(x) F), således at (G X) giver samme værdi som E.

Der er naturligvis mange sådanne funktioner. Så svaret på spørgsmålet er "nej". Men det er på den anden side sjældent særligt tvetydigt, hvordan man mest naturligt danner *funktionen,* som repræsenterer konteksten af et bestemt sted i et udtryk. Så i praksis er konteksten bestemt rimeligt entydigt.

Escape funktioner.

- En *escape funktion* er er en funktion, som ignorerer sin kontekst i ethvert kald.
- Værdien af en beregning, hvori der kaldes en escape funktion, er lig med værdien som returneres af den først kaldte escape funktion.

Antagelse: escaper er et funktionale, som givet en funktion giver en tilsvarende escape-funktion.

Eksempler:

```
> (+ ((escaper *) 5 2) 3)
10
```

```
>
(* (+ ((escaper
          (lambda (x)
            (- (* x 3) 7))) 4 )
      5)
   6)
5
```

Hvis e er en escape-funktion gælder at
(compose f e) = e
for alle funktioner f.

```
> ((compose
     length
     ((right-section (escaper cons) (list 1 2 3)) 4))
   4)
(4 1 2 3)
```


Noter

På næste slide vil vi vise, hvordan man kan definere escaper funktionalet ved brug af call-with-current-continuation.

Escaper funktionalet.

```
(define *top-level* "any continuation")

(define (receiver c)
  (set! *top-level* c)
  (*top-level* (lambda () (display "escaper defined"))))

((call-with-current-continuation receiver))

(define (escaper proc)
  (lambda args
    (*top-level*
      (lambda ()
        (apply proc args)))))
```


Noter

Vi definerer escaper ved brug af call-with-current-continuation. Dette er naturligvis lidt snyd, idet escape funktioner bruges til at fortolke betydningen af continuations. Ovenstående er teknisk lidt indviklet, så lad mig af bedste evne forklare, hvad der foregår.

I kaldet ((call-with-current-contination receiver)) infanges en continuation i receiver, hvor der blot forestår et procedurekald: det yderste parantespar i ((call-with-current-continuation receiver)). Fortsættelsen, der indfanges, tager som altid én parameter, som her skal være en parameterløs funktion! I kaldet af receiver fuldføres ((call-with-current-contination receiver)) ved at overføre (lambda () (display "escaper defined")) til den indfangede og gemte continuation.

I escaper returneres en funktion, som når den kaldes, vil overføre en funktion uden parametre til *top-level*. Når denne funktion kaldes afleveres værdien af (apply proc args) direkte som værdi til det yderste Scheme niveau.

Prøv den selv!

Continuations via kontekster og escape funktioner.

1. call-with-current-continuation er en funktion af ét argument, som kaldes modtager.
2. Modtageren er en funktion af ét argument, som er continuation.
3. continuation er en funktion af ét argument.
4. **(call-with-current-continuation modtager) ~ (modtager continuation)**
5. **continuation er (escaper c), hvor c er er konteksten af (call-with-current-continuation modtager).**

Eksempel:

(+ 3 (* 4 (call-with-current-continuation m)))

Konteksten af (call-with-current-continuation m) i
(+ 3 (* 4 (call-with-current-continuation m))) er
(lambda(v) (+ 3 (* 4 v)))

(+ 3 (* 4 (call-with-current-continuation m))) ~
(+ 3 (* 4 (m (escaper (lambda(v) (+ 3 (* 4 v)))))))

Noter

På denne slide giver vi en alternativ, og mere præcis definition på, hvad en continuation er.

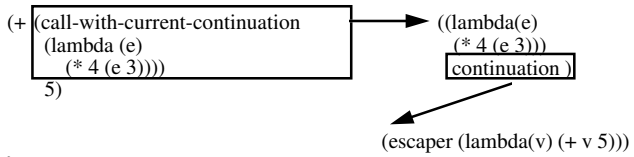
Escaper funktionalet og kontekster er hver i sær velforståede begreber. Punkt 4 siger, hvorledes vi transformerer kaldet (call-with-current-continuation ...). Punkt 5 siger, hvilken funktion der kan gøre det ud for fortsættelsen (altså continuation). Dette er netop escaper funktionalet anvendt på konteksten af (call-with-current-continuation ...) formen.

En systematisk transformation ved brug af ovenstående regler giver ikke særligt pæne resultater. Fordelen er dog, at vi helt og holdent kan forstå (og ræsonnere) om programmer, som bruger call-with-current-continuation, forudsat at vi forstår de simplere begreber: kontekster og escape funktioner.

Vi giver et par eksempler på næste side.

Eksempler på brug af kontekster og escape funktioner.

1.



Altså:

```
(+ (call-with-current-continuation
     (lambda (e)
       (* 4 (e 3))))
   5)
```

~

```
(+ ((lambda(e)
      (* 4 (e 3)))
    (escaper (lambda(v) (+ v 5))))
   5)
```

2.

```
(define (try a b)
 (call-with-current-continuation
   (lambda (c)
     (* 1000 (if (= a b)
                 (c a)
                 (+ a b))))))
```

~

```
(define (try a b)
  ((lambda (c)
     (* 1000 (if (= a b)
                 (c a)
                 (+ a b))))
   (escaper (lambda(v) v))))
```

PS4 - Continuations

Noter

I eksempel 2 er (call-with-current-continuation ...) yderst i funktionen try. Derfor er konteksten lig med identitetsfunktionen, som vi ovenfor har noteret direkte i lambda notation.

Continuation-passing style.

- Den implicitte continuation af en funktion gøres eksplicit.
- Alle funktioner tager én ekstra parameter: en continuation, som er en funktion af én parameter.
- Returnering af en værdi fra en funktion foregår ved at give værdien til continuation-parameteren.
- I et funktionskald overføres ud over almindelige parametre også en continuation, som repræsenterer *fortsættelsen* (resten) af beregningen.
- Med applicative-order beregning starter man indefra og ud: Udfør det inderste venstre funktionskald og overfør en continuation, som repræsenterer resten af beregningen.

Direct style

```
(define (p a b)
  (* (+ a b) (- a b)))
```

Continuation-passing style

```
(define (p a b k0)
  (plus a b (lambda(v1)
              ; givet summen af a og b i v1, beregn tilsvarende
               ; differens og multiplicer disse
                (sub a b (lambda(v2)
                           ; givet summen i v1, og differencen i v2, gennemfør
                            ; resten af beregningen, og returner resultatet til k0
                            (mult v1 v2 k0))))))
```

PS4 - Continuations

Noter

"Continuation-passing style" er en programmeringsstil, hvor fortsættelsen af ethvert funktionskald optræder eksplicit som en parameter (pr. konvention den sidste parameter). Derved er primitivet call-with-current-continuation overflødiggjort, idet vi ved hjælp af den ekstra parameter kan få hånd på fortsættelsen af ethvert kald. Prisen for dette er, som vi vil se tydeligt, at vi må reorganisere vore programmer ganske drastisk.

Når vi kalder allerede eksisterende funktioner, som ikke tager en continuation som sidste parameter, bliver vi naturligvis nødt til at gå på kompromis. Værdierne fra sådanne kald må vi selv kanalisere over i de continuations, hvorover vi har kontrol.

I dette eksempel benytter vi primitiverne mult, plus og sub i stedet for *, + og -. Mult, plus og sub tager hver en continuation ud over de to "almindelige parametre". Mult kan således defineres således:

```
(define (mult a b k2)
  (k2 (* a b)))
```

Plus og sub defineres tilsvarende.

Eksempel på continuation-passing-style (1).

Direct style

```
(define (w a b)
  (f (g a) (h b) (g b) (h a)))

(define (f a b c d)
  (+ a b c d))

(define (g a)
  (* a a))

(define (h a)
  (* a a a))
```

```
> (w 3 4)
116
```

Continuation-passing style

```
(define (w a b k0)
  (g a (lambda(v1)
         (h b (lambda (v2)
                (g b (lambda (v3)
                       (h a (lambda (v4)
                              (f v1 v2 v3 v4 k0)))))))))))

(define (f a b c d k1)
  (k1 (+ a b c d)))

(define (g a k2)
  (k2 (* a a)))

(define (h a k3)
  (k3 (* a a a)))
```

```
> (w 3 4 id)
116
```

PS4 - Continuations

Noter

Eksemplet viser, hvordan simple, ikke-rekursive funktionskald og definitioner kan skrives på continuation passing style. Funktionen id er identitetsfunktionen:

```
(define (id x) x)
```

Der vises en række kunstige funktioner, dels i den direkte almindelige stil, dels i continuation-passing stil. Først skal (g a) beregnes. Til denne beregning overføres endvidere en continuation, som er den yderste lambda-form. I denne continuation foretages resten af beregningen; dette udgøres af beregningen af formerne (h b), (g b) og (h a) som via parametre i nye lambda former bindes til hhv. v2, v3 og v4. Endelig kaldes f på de fire parametre. Den oprindelig overførte continuation k0 overføres til f, som til gengæld vil sørge for, at denne continuation bliver kaldt med f's værdi.

Eksempel på continuation-passing-style (2).

Direct style

```
(define (determinant a b c)
  (sub (square b)
       (mult (mult 4 a) c)))

(define (square a)
  (mult a a))

(define (mult a b)
  (* a b))

(define (sub a b)
  (- a b))
```

Continuation-passing style

```
(define (determinant a b c k0)
  (mult 4 a (lambda (v1)
              (mult v1 c (lambda(v2)
                           (square b (lambda(v3)
                                       (sub v3 v2 k0)))))))))
(define (square a k1)
  (mult a a k1))

(define (mult a b k2)
  (k2 (* a b)))

(define (sub a b k3)
  (k3 (- a b)))
```


Noter

I det ovenstående beregner vi 4*a*c før b*b. Det spiller naturligvis ikke nogen rolle i funktionsorinteret stil.

Følgende viser ovenstående, blot med en anderledes beregningsrækkefølge.

```
(define (determinant a b c k0)
  (square b (lambda (v1)
              (mult 4 a (lambda (v2)
                          (mult v2 c (lambda (v3)
                                       (sub v1 v3 k0)))))))))
```

Continuation-passing style: rekursive funktioner.

Direct style

```
(define (fak n)
  (if (= n 0)
      1
      (* n (fak (- n 1)))))
```

Continuation-passing style

```
(define (fak n k)
  (if (= n 1)
      (k 1)
      (fak (- n 1) (lambda(v)
                     ; v er (fak (- n 1)). Resten af beregningen er:
                     (k (* n v))))))
```

Ved overgangen fra den direkte stil til continuation-passing style bliver funktionen hale-rekursiv.

Til gengæld får vi skabt en række continuations, i form af funktioner, der modsvarer rekursionsudviklingen i den direkte stil.

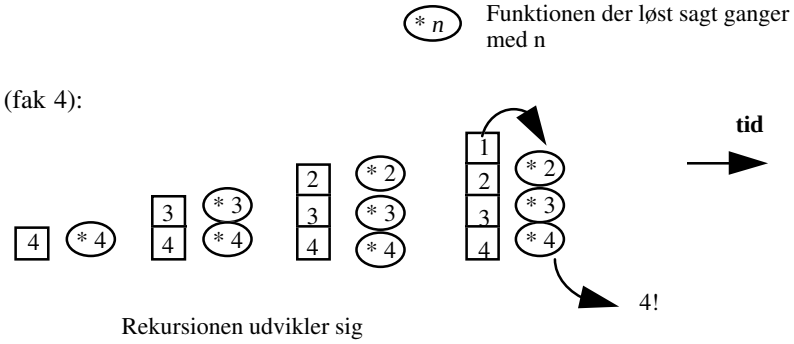
PS4 - Continuations

Noter

I rekursive funktioner med én eller flere parametre styrer parametrene udviklingen af rekursionen, ganske som i direkte stil. De egenlige beregninger foregår i de continuations, der bliver skabt som lambda-udtryk. Generelt vil en continuation c, skabt af kaldet (fak n ...) bliver overført til det rekursive kald, her (fak (- n 1) ...); i denne inkarnation vil c blive kaldt af en ny continuation, som skabes, osv.

Hvis vi kalder lambda-udtrykket, som bliver bragt på banen af (fak n ...) for Ln, vil kaldet (fak (- n 1) Ln) kalde Ln. Tilsvarende vil (fak (- n 2) ...) kalde Ln-1, osv. indtil (fak 1 L2) blot giver et 1-tal til L2. Vi ser at vi accumulerer en kæde af kald af contituation-funktioner, som ikke kan trække sig sammen inden 1-tallet gives til L2.

Nedenfor ser vi hvordan vi under rekursionsudviklingen skaber funktioner, der ganger med hhv n, n-1, ... , 2. Funktionen der ganger med m søger endvidere for, at funktionen der ganger med m+1 også bliver kaldt:



Rekursionen udvikler sig

Continuation-passing style: hale-rekursive funktioner.

Direct style

```
(define (fak n r)
  (if (= 0 n)
      r
      (fak (- n 1) (* r n))))
```

Continuation-passing style

```
(define (fak n r k)
  (if (= 0 n)
      (k r)
      (fak (- n 1) (* r n) (lambda (v)
                             ; Returner blot v til k:
                             (k v)))))
```

```
(define (fak n r k)
  (if (= 0 n)
      (k r)
      (fak (- n 1) (* r n) k)))
```

- En funktion er halerekursiv hvis funktionens værdi er værdien af et rekursivt kald.
- I continuation-passing-style betyder dette, at den oprindelige continuation til en rekursiv funktion kan overføres uændret til alle rekursive kald.
- Hale rekursion kræver ikke skabelse af en række nye continuations undervejs i rekursionens udvikling.

Noter

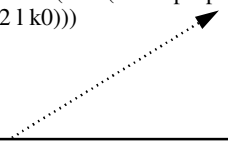
Hovedformålet med denne side er at karakterisere halerekursive funktioner, ved at studere deres omskrivning til continuation-passing style.

I den øverste af fak-funktionerne, returneres værdien, som er akkumuleret i r fra basistilfældet, gennem *n* indlejrede instantieringer af identitetsfunktionen. Med andre ord, ser vi præcist det samme mønster af lambda-udtryk som på forrige side. Dette er utilfredsstillende for hale-rekursive kald.

I den nederste af fak-funktionerne, returneres værdien, som er akkumuleret i r fra basistilfældet, direkte til den oprindelige continuation til kaldet af fak. Bemærk at med én anvendelse af én η reducering kommer vi fra den øverste hale-rekursive version til den nederste.

Continuation-passing style: undtagelser.

```
(define (list-length1 l k0)
    (letrec ((list-length2
              (lambda (l k1)
                (cond ((null? l) (k1 0))
                      ((pair? l) (list-length2 (cdr l) (lambda (v)
                                                         (k1 (+ 1 v)))))
                      (else (k0 'improper-list))))))
      (list-length2 l k0)))
```



- Vi overfører symbolet improper-list til fortsættelsen k0 i stedet for k1.
- Vi slipper for at indfange en continuation via call-with-current-continuation, idet den ønskede fortsættelse allerede er tilgængelig via parameteren k0.

Noter

Vi vender her tilbage til eksemplet, hvor vi indfanger continuations med henblik på håndtering af undtagelser. Vi har tidligere set, at call-with-current-continuation kan anvendes til netop dette formål. Her ser vi, hvordan vi kan klare os, når vi programmerer i continuation-passing style.