INF2810: Funksjonell Programmering

Strømmer og utsatt evaluering

Stephan Oepen

Universitetet i Oslo

30. mars 2017

Forrige forelesning

- Mer om (prosedyre)navn, bindinger, og verditilordning
- Nok en ny abstrakt muterbar datatype basert på lister: tabeller
- Memoisering
- Repetisjons-quiz med Kahoot!
- Sekvensoperasjoner (repetisjon)

I dag

- Sekvensoperasjoner (repetisjon)
- Strømmer
- Realisering av data ved behov
- Uendelige datastrukturer
- Utsatt evaluering
- Tilbake til sekvensoperasjoner

Repetisjon: Sekvensoperasjoner

- Vi har tidligere jobbet mye med sekvensoperasjoner definert for lister.
- F.eks `map`, `reduce` (`accumulate`), `filter`, m.m.

```
(define (reduce proc init items)
  (if (null? items)
      init
      (proc (car items)
            (reduce proc init (cdr items)))))

? (reduce * 1 '(2 2 2)) → 8

(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

? (interval 0 5) → (0 1 2 3 4 5)

? (reduce + 0 (interval 0 5)) → 15
```

Inkrementell beregning vs sekvensoperasjoner

```
(define (mystery low high)
  (define (recurse count sum)
    (cond ((> count high) sum)
          ((prime? count)
           (recurse (+ count 1) (+ count sum)))
          (else (recurse (+ count 1) sum))))
  (recurse low 0))

? (mystery 1 5) → 10
```

- Hva er formålet med `mystery`, og hva returnerer eksempelkallet?
- (Vi later som vi allerede har `prime?`.)
- Med høyreordensprosedyrer over sekvenser kan det løses mer elegant:

```
(define (sum-primes low high)
  (reduce + 0 (filter prime? (interval low high))))
```

- Er det egentlig relevante forskjeller i tids- eller plasskompleksitet?

Begrensninger ved "sekvenstilnærmingen"

- Å uttrykke beregninger som manipulasjon av sekvenser kan være elegant, konsist og modulært.
- Men i visse tilfeller kan teknikken bli veldig (og unødvendig) lite effektiv:

```
? (car (cdr (filter prime? (interval 100 100000)))) → 103
```

- Det beregnes én lang sekvens, så én til, men bare 4 elementer "brukes".
- Med en mer tradisjonell programmeringsstil ville vi gjort beregningen inkrementelt, med de ulike operasjonene sammenvevd.
- (F.eks ved å iterativt teste en tellervariabel, og returnere så fort vi hadde kommet til det andre primtallet.)
- Kan vi beholde den elegante strukturen ved sekvensoperasjonene uten å miste effektiviteten ved inkrementelle beregninger?
- Løsningen: strømmer.

Strømmer

```
? (stream-car
    (stream-cdr
      (stream-filter prime? (stream-interval 100 100000))))
→ 103
```

- Ideen: strømmen konstrueres bare delvis:
- Kun elementene vi trenger å se på genereres.
- Hvis vi forsøker å aksessere en del av strømmen som ikke er konstruert ennå, genererer strømmen automatisk mer av seg selv.
- Lar oss samle data i sekvenser på samme måte som lister, *men*:
- elementene (untatt det første) blir først evaluert når de 'brukes'.
- Utsatt evaluering (*delayed evaluation*).

Strømmer: Grensesnittet

- En strøm skal ha en 'kontrakt' som ligner på vanlige lister:

```
(stream-car (cons-stream x y))   ≡   x
(stream-cdr (cons-stream x y))   ≡   y
```

- I tillegg har vi et objekt for den tomme strømmen `the-empty-stream`
- og predikatet `stream-null?`.
- (For øyeblikket skal vi bare late som om disse var innebygget i Scheme.)
- På overflaten kan vi bruke strømmer som om de var lister. . .
- men `cdr`-verdien av en strøm lages ikke ved `cons-stream`, men først ved `stream-cdr`.

cons og cons-stream

- `cons-stream` gir et "løfte" om at cdr-verdien kan beregnes ved behov.
- Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- Etterspørselen styrer beregningstidspunkt (*computing on demand*).

```
? (cons (+ 1 2)
        (+ 3 4))
→ (3 . 7)

? (cons-stream (+ 1 2)
               (+ 3 4))
→ (3 . #<promise>)

? (stream-cdr
    (cons-stream (+ 1 2)
                 (+ 3 4)))
→ 7
```

Sekvensoperasjoner for strømmer

- Liknende grensesnitt som lister, så listeoperasjonene kan lett tilpasses:

```
(define (filter pred seq)
  (cond ((null? seq) '())
        ((pred (car seq))
         (cons (car seq)
               (filter pred (cdr seq))))
        (else (filter pred (cdr seq)))))


(define (stream-filter pred seq)
  (cond ((stream-null? seq) the-empty-stream)
        ((pred (stream-car seq))
         (cons-stream
          (stream-car seq)
          (stream-filter pred (stream-cdr seq))))
        (else (stream-filter pred (stream-cdr seq)))))
```

Eksempel med interval

```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
    the-empty-stream
    (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)

? (stream-interval 1 10) →  (1 . #<promise>)

? (stream-cdr (stream-interval 1 10)) →  (2 . #<promise>)
```

Tilbake til primtalleksemplet

```
? (stream-car
    (stream-cdr
      (stream-filter prime? (stream-interval 100 100000))))
→ 103

? (car (cdr (filter prime? (interval 100 100000))))
→ 103
```

- Begge uttrykkene returnerer den andre primtallet større enn 100.
- Listeversjonen kaller predikatet ca 100.000 ganger og lager ca 110.000 cons-celler.
- Strømversjonen kaller predikatet 4 ganger og lager 6 cons celler.

Evalueringsstrategier (repetisjon fra 2. forelesning)

Applicative-order evaluation

- Evaluer argumentene: kall prosedyre på verdi.
- Standard i Scheme.
- Andre navn: *call-by-value*, *strict / eager evaluation*.

Normal-order evaluation

- Prosedyren kalles med argument*uttrykkene*: evalueres først ved behov.
- Andre navn: *call-by-name*, *call-by-need*, *non-strict / lazy evaluation*.

- Gir samme resultat så lenge vi holder oss til ren funksjonell kode (og endelige datastrukturer), men kan gi forskjeller i effektivitet.
- `delay` gir oss kontroll til å velge selv når uttrykk evalueres.
- Skal se at den kan brukes for å implementere `cons-stream`.

Utsatt evaluering

- `delay` er en *special form* som tar et uttrykk som argument og returnerer et "løfte" om at uttrykket kan evalueres seinere.
- Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt "satt på vent" med `delay`.

```
? (define foo (* 21 2))
? foo →  42
? (define bar (delay (* 21 2)))
? bar →  #<promise>
? (force bar) →  42
```

Å implementere strømmer i Scheme

- `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y)  ≡  (cons x (delay y))
```

- Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  ...(force (cdr stream)))

(define the-empty-stream '())

(define (stream-null? stream)
  ...(null? stream))
```

Hvordan så implementere `delay` og `force`?

- `delay` og `force` er innebygd i Scheme, men det kan være opplysende å reflektere rundt hvordan vi kunne definert dem selv.
- Evaluering av et uttrykk kan utsettes ved å gjøre det til prosedyrekropp:

```
(delay exp)  ≡  (lambda () exp)
```

- Dette kan så regnes som et løfte om å beregne `exp` senere.
- For å utføre selve beregningen kaller vi bare prosedyreobjektet:

```
(define (force promise)
  ...(promise))
```

Et eksempel i mer detalj

```
? (define s (stream-interval 1 10))
? s → (1 . #<promise>)
? (stream-cdr s) → (2 . #<promise>)
```

- Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)
⇒ (cons-stream 1 (stream-interval 2 10)
⇒ (cons 1 (delay (stream-interval 2 10)))
⇒ (cons 1 (lambda () (stream-interval 2 10)))
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))
⇒ (force (lambda () (stream-interval 2 10)))
⇒ ((lambda () (stream-interval 2 10)))
⇒ (stream-interval 2 10)
⇒ (cons-stream 2 (stream-interval 3 10))
 ...
```

En mer realistisk implementasjon av `delay`

```
(delay exp)  ≡  (lambda () exp)
```

- Definisjonen av `delay` over er tilstrekkelig for utsatt evaluering...
- men den kan gjøres mye mer effektiv!
- I motsetning til i/o-'strømmer' støtter strømmer *random access*.
- Strømelementer kan brukes flere ganger:
- Med definisjonen så langt må vi da innfri samme løfte flere ganger.
- Memoisering ('dynamisk programmering') kan bygges inn i `delay`.

Memoisering og utsatt evaluering

- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
      result)))
```

```
? (define (foo x)
    (display "I was called!")
    x)
? (define bar (delay (foo 42)))
? bar → #<promise>
? (force bar)
↝ I was called!
→ 42
? bar → #<promise>
? (force bar) →  42
```

Utsatt evaluering og destruktive operasjoner

- Utsatt evaluering går ikke så bra i hop med mutasjoner:

```
? (define a 4)
? (define b (delay (+ 1 a)))
? (set! a 0)
? (force b) → 1
```

- Verdien til den utsatte evalueringen avhenger av tidspunktet for når den faktisk kalles: kan lett skape mye forvirring (og bugs)!
- Enda mer forrvirrende hvis det er optimisert med memoisering i tillegg:

```
? (set! a 100)
? (force b) → 1
```

Noen hjelpeprosedyrer fra Oblig (3a)

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

? (stream-ref (stream-interval 0 1000000) 7)
→ 7
```

```
(define (show-stream stream n)
  (cond ((= n 0) (display "...\n"))
        ((stream-null? stream)  (newline))
        (else (display (stream-car stream))
              (display " ")
              (show-stream (stream-cdr stream) (- n 1)))))

? (show-stream (stream-interval 0 1000000) 7)
⇝ 0 1 2 3 4 5 6 ...
```

- Antar også at vi har `stream-map` fra obligen.

Uendelige strømmer

- Vi har sett at en strøm er en sekvens som konstrueres mens den brukes.
- Kan dermed definere en *uendelig* sekvens, uten å faktisk beregne den.
- De naturlige tallene via en rekursiv strømgenerator:

```
? (define (integers-starting-from n)
    (cons-stream n (integers-starting-from (+ n 1))))

? (define nats (integers-starting-from 1))
```

- Veldefinert pga utsatt evaluering.
  - Hva mangler i forhold til rekursjonen vi har sett før?
  - Hva ville skjedd om vi brukte `cons`?
- Kan kombineres med andre strømprosedyrer og strømmer, f.eks.

```
? (define odds (stream-filter odd? nats))

? (show-stream odds 5) ⇝ 1 3 5 7 9 ...
```

Neste gang

- Mer om (uendelige) strømmer og utsatt evaluering.
- Hvordan definere egne *special forms*?
- Makroer: lar oss transformere kode før den blir evaluert eller kompilert.
- Strømmer, tid og tilstand.
- Evaluator.