IN2040: Funksjonell Programmering

Utsatt evaluering og mer om strømmer

Martin Steffen

Universitetet i Oslo

Uke #10 (25. Okt. 2022)

Tema

Forrige gang

- Ny datastruktur, ny teknikk:
- Strømmer
- Utsatt evaluering

I dag

- Mer om utsatt evaluering,
- og (uendelige) strømmer
- Makroer: syntaktiske transformasjoner
- Strømmer og tidslinjer

Repetisjon: Utsatt evaluering

- I stedet for å umiddelbart evaluere et uttrykk kan vi representere et løfte (*promise*) om at det kan evalueres senere dersom verdien trengs.
- Løftet kan representeres ved å pakke uttrykket inn i en anonym prosedyrekropp uten parametre, som så kan innfris ved å kalles.

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

- Slike innkapslinger (*closures*) kalles noen ganger *thunks*.
- Prosedyreobjektet har en peker til omgivelsen der det opprettes slik at vi kan finne verdiene til uttrykkets frie variabler ved evaluering.

```
? (define p
    (let ((foo 42))
      (delay (* foo 2))))
? (force p) → 84
```

≡

```
? (define p
    (let ((foo 42))
      (lambda () (* foo 2))))
? (p) → 84
```

Repetisjon: Utsatt evaluering

- Evalueringsstrategien i Scheme er *eager evaluation* / *call-by-value*.
- Kan bruke teknikken over for å implementere en form for *call-by-name*, der uttrykk ikke evalueres før verdien skal brukes.
- Risikerer å måtte evaluere samme uttrykk flere ganger.
- Kan optimeres med memoisering; *lazy evaluation* / *call-by-need*, der uttrykket kun evalueres én gang og verdien lagres for gjenbruk:

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

- (`delay` som er innebygget i Scheme er memoisert.)
- Både utsatt evaluering og memoisering (og dermed strømmer) går dårlig sammen med side-effekter: bør brukes med funksjonell kode.

Et lite sidespor om destruktive operasjoner

- I kapittel 3 fikk vi operatorer for verditilordning og mutasjoner inn i språket, såkalt destruktive operasjoner.

- Lite hensiktsmessig å tenke på all bruk av disse som like 'destruktive'.
- Enkelte ganger brukes verditilordning kun for effektivisering: gjenbruk av cons-celler, memoisering, osv.
- Dette betyr ikke nødvendigvis at vi gir opp en funksjonell stil.
- I eksemplet vårt med utsatt evaluering + memoisering (som benyttet `set!`) så vi at dette nettopp var mest egnet til funksjonell kode.

Å definere *special forms*

- Vi har så langt beskrevet `delay` med '$\equiv$'-notasjon.
- Men hva om vi selv ønsker å implementere dette?
- For å definere våre egne *special forms* trengs det noe mer enn `define`.
- Makroer: Syntaktisk omskriving av Scheme-uttrykk.
- Lar oss transformere kode før den blir evaluert eller kompilert.
- Kode som genererer kode!

Syntaktisk omskriving med makroer

- Et Scheme-uttrykk kan dekomponeres og omskrives før evaluering.
- Implementeres ved `define-syntax` og `syntax-rules`:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (memoize (lambda () exp)))))
```

- `syntax-rules` definerer mønstre og ekspansjoner.
- Når Scheme ser et uttrykk som begynner med `delay` vil det nå skrives om ('ekspanderes') til uttrykket på høyresiden før det evalueres.

Hygieniske makroer

```
(define-syntax ⟨identifier⟩
  (syntax-rules (⟨keyword⟩*)
    (⟨pattern⟩ ⟨expansion⟩)
    ...))
```

- ⟨identifier⟩ i operatorposisjon vil fra nå av ekspanderes som makro.
- ⟨pattern⟩ brukes for å 'bryte opp' uttrykket og binde parameterne.
- ⟨expansion⟩ er oppskriften for å 'sette sammen' bitene igjen.
- ⟨keyword⟩ kan deklarere syntaktiske nøkkelord (f.eks `else` i `cond`).
- Kalles for 'hygieniske makroer':
- Har garanti for at eksisterende bindinger ikke overskygges av variabler som introduseres i ekspansjonen (*accidental capture*).

Enkelt makroeksempel: `when`

- Makroer lar oss gjøre ting vi ikke kan gjøre med vanlige prosedyrer.
- Kan utvide syntaksen til språket.

```
? (define-syntax when
    (syntax-rules ()
      ((when condition . body)
       (if condition (begin . body)))))

? (when (> -2 100)
    (display "I was evaluated!\n")
    42)

? (when (> 3 1)
    (display "I was evaluated!\n")
    42)
⇝ I was evaluated!
→ 42
```

- Umulig å skrive `when` som vanlig prosedyre. (Prøv!)

Enkelt makroeksempel: `when`

- En annen variant som viser bruk av syntaktiske *keywords*.

```
? (define-syntax when
    (syntax-rules (do else)
      ((when condition do body1 else body2)
       (if condition body1 body2))))

? (when (< 2 1)
   do (display "this")
   else (display "that"))
↝ that
```

Repetisjon: Strømmer via utsatt evaluering

- Strømmer: en sentral datastruktur i funksjonell programmering.
- Kan tenke på dem som 'utsatte lister' / *lazy lists*:
- En sekvensiell datastruktur der elementene realiseres ved behov.
- `cons-stream` utsetter evaluering av `cdr`-uttrykket.
  - En *special form* som bruker `delay` på `cdr`-argumentet.
- `stream-cdr` tvinger evaluering.

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

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

Strømmer implementert via utsatt evaluering

- Kan implementre `cons-stream` med en makro:

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream head tail)
     (cons head (delay tail)))))
```

- Resten av strømgrensesnittet kan defineres som vanlige prosedyrer:

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

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

(define (stream-null? stream)
  (null? stream))

(define the-empty-stream '())   ;; en konstant
```

Strømmer og sekvensmanipulasjon

```
? (reduce + 0 (filter prime? (interval 100 100000)))
```

- Mange beregninger kan formuleres som sekvensmanipulasjoner.
- En av hovedfordelene: modularitet.
- Komplekse operasjoner kan deles opp i flere mindre komponenter, med delvis prosesserte sekvenser som midlertidig resultat.
- Med lister må vi holde hele strukturen i minnet samtidig.
- Med strømmer kan vi jobbe på ett og ett element av gangen.
- Trenger ikke vente på at ett trinn er ferdig før neste begynner.
- Jf. pipeline av Un*x-kommandoer:

```
$ cat big.data | grep query | sort | more
```

Uendelige strømmer

- Veldefinert p.g.a. utsatt evaluering.
- De naturlige tallene via en rekursiv prosedyre som genererer en strøm:

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

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

- Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt (SICP: 'implisitt strømdefinisjon'):

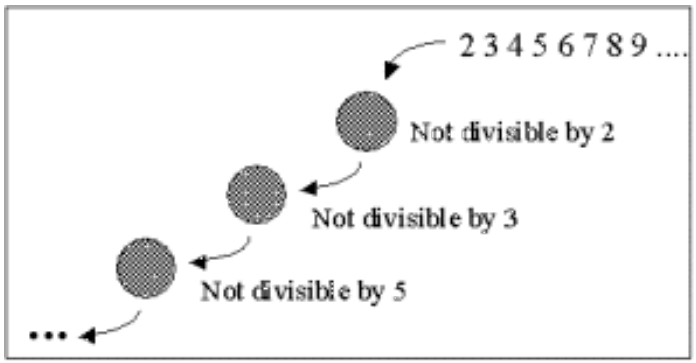
```
? (define ones (cons-stream 1 ones))

? (define (add-streams stream1 stream2)
    (stream-map + stream1 stream2))

? (define nats
    (cons-stream 1 (add-streams ones nats)))
```

Nok en uendelig strøm: Sielen til Eratosthenes

- Også primtallene danner en uendelig sekvens: $2, 3, 5, 7, 11, 13, 17, 19, \ldots$
- Altså en strøm der ingen element er delbart på 'tidligere' elementer.
- Kan modelleres som lag på lag av strømmer: hvert lag filtrerer videre:




```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve
    (stream-filter
     (lambda (x) (not (divisible? x (stream-car stream))))
     (stream-cdr stream)))))
(define primes (sieve (integers-starting-from 2)))
? (stream-ref primes 3) → 7
```

Uendelige strømmer, fallgruver

- Har sett at de fleste listeoperasjoner enkelt kan 'oversettes' til strømmer.
- Og alle vanlige strømoperasjoner fungerer også for uendelige strømmer.
- Vi må likevel ha tunga rett i munnen. Hva blir utfallet her:

```
? (stream-null? (stream-filter odd? (stream-filter even? nats)))
→
```

- `stream-null?` vil aldri bli kalt: det yttereste kallet på `stream-filter` returnerer aldri.

Uendelige strømmer, fallgruver (forts.)

- Kan du se noen problemer som kan oppstå med `stream-append`?

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

- Hvis `s1` er uendelig får vi aldri se elementer fra `s2`.
- Uendelige strømmer bør kombineres på andre måter, f.eks med;

```
(define (stream-interleave s1 s2)
  (if (stream-null? s1)
    s2
    (cons-stream (stream-car s1)
                 (stream-interleave s2 (stream-cdr s1)))))
```

Strømmer og lister

- Med lister så får vi levert dataene som på en pall: alt på en gang.
- Med strømmer får vi dataene våre som på et samlebånd: én av gangen.
- Avveining: plass vs. tid.
- Strømmer mer komplekse: brukes når vi har uendelige eller veldig lange sekvenser som det ikke er praktisk eller mulig å lagre i minnet.
- Egnet for diskret representasjon av tid...

Strømmer, tid og tilstand

```
(define (input-stream)
  (cons-stream (read) (input-stream)))

(define balances
  (cons-stream 100
               (add-streams balances (input-stream))))

? (show-stream balances 5) ⇝ ???
```

- Funksjonell kode (minus `set!` i `memoize`).
- Likevel vil det for en bruker virke som at systemet som helhet her har tilstand, som forandrer seg over tid:
- Ser en oppdatert saldo for hvert innskudd på kontoen.
- Jf. de tidligere kontooperasjonene våre med `(set! balance ...)`

Et anvendt eksempel: CMA

- Cumulative Moving Average (CMA): Et løpende oppdatert gjennomsnitt for en 'strøm' av tidsdata: $x_1, x_2, x_3, x_4, \ldots$.
- Typisk eksempel: en investor vil vite snittpris for en aksje over tid.
- CMA er gjennomsnittet for alle dataene opp til nåværende tidspunkt:

$$\mathrm{CMA}_i = \frac{x_1 + x_2 + \ldots + x_i}{i}$$

- Kan oppdateres fortløpende etterhvert som mer data kommer inn:

$$\mathrm{CMA}_{i+1} = \frac{x_{i+1} + i\mathrm{CMA}_i}{i+1} \qquad \text{hvor } \mathrm{CMA}_0 = 0$$

```
(define (moving-average stream)
  (define (generate s prev i)
    (let ((new (/ (+ (stream-car s) (* i prev))
                  (+ i 1.0))))
      (cons-stream new (generate (stream-cdr s) new (+ i 1)))))
  (generate stream 0 0))
```

Et anvendt eksempel: CMA (forts.)

```
(define (moving-average stream)
  (define (generate s prev i)
    (let ((new (/ (+ (stream-car s) (* i prev))
                  (+ i 1.0))))
      (cons-stream new (generate (stream-cdr s) new (+ i 1)))))
  (generate stream 0 0))

? (define avg-nats (moving-average nats))

? (show-stream avg-nats 10)
⇝ 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 ...

? (show-stream nats 10)
⇝ 1 2 3 4 5 6 7 8 9 10 ...
```

- Strøm av løpende oppdaterte gjennomsnitt for saldo over tid:

```
? (define avg-balance (moving-average balances))
```

Oppsummering: Objekter, strømmer, tid og tilstand

- To nye programmeringsstrategier i kap. 3:

Objekter med lokal tilstand

- Organiserer programmet som distinkte objekter med egenskaper som kan forandre seg over tid.
- Modellen forsøkt inndelt på samme måte som vi oppfatter verden.
- Tilstand og endring i verden modellert med tilstandsvariabler og verditilordning.
- Simulert tid gitt ved evaluering.

Strømmer

- Organiserer programmet som strømmer av informasjon som flyter mellom ulike komponenter.
- Deler opp og representerer tid eksplisitt: som distinkte punkter i en sekvens av hendelser.
- Simulert tid sammenfaller ikke nødvendigvis med evaluering.

Neste gang

- ‘Metalingvistisk abstraksjon’
- ‘Metasirkulær evaluator’
- Enklere enn det høres ut!
- Vi ser på en evaluator for Scheme, skrevet i Scheme.