プログラミング入門
第 12 回「ストリーム」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

標準部品化力、オブジェクトおよび状態(第 3 章)

3.5 ストリーム

代入は時間に応じて変化する状態を実現するための強力な手段でした。しかし、一方、その代償として、モデルが複雑になり、関数の性質が失われたり、データ構造の共有の違いが問題になるなど、様々な問題が起きることもわかりました。

代入以外に時間的変化を扱う手段として、ストリームという概念があります。この節ではストリームについて勉強していきます。基本的には、ストリームはリストとほぼ同じような概念ですが、リストに遅延評価 (delayed evaluation)がついたものと考えれば差し支え有りません。ストリームが代入の代わりになるわけではありませんが、ストリームも時間の変化を伴う状態を扱う一つの強力な手段となります。また、それ以外にも、整数列をストリームとして生成し、ループ処理を実現するなどの方法にも使えます。(例えば、python ではループをそのようにして実現しています)

3.5.1 ストリームは遅延リスト

リストは map や filter, reduce などの抽象化された関数で容易に様々な処理が行えることを学びました。しかし、一方で、メモリ空間的には、その都度大きなリストを作成することになり、最初のリストが大きなサイズであったり、途中で大きなサイズのリス

トになると、非常にメモリ空間的に無駄があることになります。例えば、次の二つのプログラムを比較してみましょう。最初に再帰的プログラムを使った場合です。

```
(define (sum-evens a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((even? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

次に reduce, filter, range を使ったプログラムです。

```
(define (sum-evens2 a b)
  (reduce + 0 (filter even? (range a b))))
```

後者のほうが簡単に実現できますが（簡単のはずですが…）、合計を記憶する変数があれば良いだけにもかかわらず、a から b までの完全な整数のリストを生成・保持しなくてはいけないため、計算時間的にも、空間的にも効率が良くありません。特に b の値が非常に大きい場合、リストを生成するだけでメモリを食い尽くしてしまうかもしれません。

ストリームはリスト処理のわかりやすさ優美さを残したまま良いメモリ効率を実現するすぐれたデータ構造です。ストリームは遅延評価(delayed evaluation)と呼ばれる機構で実現されます。まず遅延評価について説明します。

遅延評価は delay と force という二つの特殊形式で実現できます。(delay ⟨*exp*⟩)の評価式は式⟨*exp*⟩を評価せず、この評価を将来行うという「約束」をした遅延オブジェクト(delayed object)を返します。force は遅延オブジェクトを受け取り、その「約束」された式を評価します。例えば、

```
> (define x (delay (+ a 10)))
> (define a 20)
> (force x)
```

30

ということができます。この(delay (+ a 10))はその場で(+ a 10)を実行することはせず、(force x)とした時点で始めてこの(+ a 10)が実行されます。また、この遅延評価はメモ化(memoization)と呼ばれる機構も実現しています。上記の例に続き次の式を評価してみましょう。

> (set! a 100)

> (force x)

30

a に 100 を代入しているので、(force x)で遅延オブジェクトを評価すると、(+ 100 10)で 110 が返ってきそうですが、実は、force を実行した次点で(+ a 10)が実行され、その結果 30 が遅延オブジェクトに格納されるようになります。一度 force で評価を行った遅延オブジェクトにはそのときの結果が格納され、その後また同じオブジェクトに force を実行した場合はそのときの結果が返ってくるようになっています。メモ化機能のついた遅延評価は、オンデマンドな計算(必要になるまで計算をしないこと)と、計算結果のキャッシング(計算結果の一時的な記憶)を同時に可能にします。

　続いてこの delay と force をつかってストリームと呼ばれる遅延評価リストを実現します。例えば、リスト(1 2 3)に対するストリームは次のように作られます。

(define x (cons 1 (delay (cons 2 (delay (cons 3 (delay '())))))))

これに対し、

> x

(1 . #<promise>)

と DrRacket では返ってきます。この#<promise>とかかれているところが遅延オブジェクトを表していて、この(cons 2 …)の部分はまだ評価されていない、ということになります。上のストリームの例えば、3 を取り出したいときは、

> (car (force (cdr (force (cdr x)))))

3

とすれば良いことになります。このようなデータ構造に対する、range や map, filter を実装すれば、効率も表現も良いリスト、すなわちストリーム版の range, map, filter ができあがることになります。まず、range からみてみましょう。

```
(define (stream-range low high)
  (if (> low high)
      '()
      (cons low (delay (stream-range (+ low 1) high)))))
```

このように cons の cdr 部の評価は遅延されることになります。続いて、car や cdr の代わりに便利なストリームに対する car と cdr も用意しましょう。

```
(define (stream-car x) (car x))
(define (stream-cdr x) (force (cdr x)))
```

これらを用いると、リストの list-ref に対応するストリーム版の stream-ref を実現できます。

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

map のストリーム版も次のように作成できます。

```
(define (stream-map proc s)
  (if (null? s)
      '()
      (cons (proc (stream-car s))
            (delay (stream-map proc (stream-cdr s))))))
```

これに対し、

```
> (define x (stream-map (lambda (x) (* 2 x))
                        (stream-range 1 10)))
```

と実行すると、2, 4, 6, 8, 10, 12, 14, 16, 18, 20 のストリームが得られます。(最初の状態ではもちろん先頭の 2 だけが作られていて、4 以降は後に評価される、ということになります)

このような x に対し、次の評価を行うと、

```
> (stream-ref x 5)
12
```

となります。Stream のための for である stream-for は次のようになります。

```
(define (stream-for s proc)
  (if (null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for (stream-cdr s) proc))))
```

例えば、次のように実行したとします。

```
> (define y (stream-map (lambda (x) (* 2 x))
                        (stream-range 1 10)))
> (stream-for y (lambda (x) (display x) (display " ")))
2 4 6 8 10 12 14 16 18 20 done
```

すると上記のようにストリームの全要素をみることができました。

このように(cons $\langle x \rangle$ $\langle y \rangle$)の代わりに、(cons $\langle x \rangle$ (delay $\langle y \rangle$))という式でペアを作るようにすればストリームが実現できる、ということになります。つまり、Scheme で (stream-cons $\langle x \rangle$ $\langle y \rangle$)を(cons $\langle x \rangle$ (delay $\langle y \rangle$))に自動変換する糖衣構文 (syntax sugar, つまり、マクロのこと)が定義できればストリーム版の cons が実現できる、ということになります。

delay と force の実装

マクロを用いれば糖衣構文による delay と force を実装することができます。(delay ⟨*exp*⟩)は(lambda () ⟨*exp*⟩)の糖衣構文です。これに対し、force は単に遅延オブジェクトを評価さえできればよいので、

```
(define (force delayed-object)
    (delayed-object))
```

とすればよいということになります。

しかし、これだけでは実は不十分で、上で説明したメモ化の機能を入れ込むことも必要となります。そこで、次のような手続きを与えます。

```
(define (memo-proc proc)
  (let ((already-run? #f)
        (result #f))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? #t)
                 result)
          result))))
```

これを用いて、(delay ⟨*exp*⟩)を(mem-proc (lambda () ⟨*exp*⟩))の糖衣構文とすればメモ化が実現されるようになります。force は元のままで ok です。