プログラミング入門
第 13 回「無限ストリーム」

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

3.5.2 無限ストリーム

先週はストリームの取り扱いについてみてきました。実際に計算の必要な箇所のみ計算するため、不要な計算があれば行われないこと、また、メモ化によって、同じ計算を二度と行わない、ということをみてきました。このような遅延評価を用いると、驚くべき事にストリームは無限に長いリストを表現することに使えます。

(stream-range a b)はaからbまでの整数列のストリームを作成しましたが、ここで、正の整数のストリームを次のように定義します。

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

このように終わりがないストリームは無限ストリームと呼ばれます。force によるアクセスが有限である限り通常のストリームと同様に扱えます。

さらに、このようなストリームから、他の無限ストリーム、例えば、7 で割り切れない整数のストリームのようなものを定義できます。

```
(define (divisible? x y) (= (remainder x y) 0))


(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

すると、7 で割り切れない整数を順次みつけることができるようになります。

```
> (stream-ref no-sevens 100)
117
```

補助関数の再帰呼び出しによる無限ストリーム

フィボナッチ数列の無限ストリームも作成できます。

```
(define (fibgen a b)
  (cons a (delay (fibgen b (+ a b)))))


(define fibs (fibgen 0 1))
```

これに対し、

```
> (stream-ref fibs 3)
2
> (stream-ref fibs 4)
3
> (stream-ref fibs 5)
```

```
5
> (stream-ref fibs 6)
8
> (stream-ref fibs 7)
13
```

とフィボナッチ数列が得られることがわかります。さらに、

```
> (map (lambda (x) (stream-ref fibs x)) (range 0 20))
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

とすると、fibs からフィボナッチ数列が得られていることがわかります。このように補助関数を用いて定義する無限ストリームのことを「補助関数の再帰呼び出しによる無限ストリーム」と呼ぶことにしましょう。

delay と force だけの簡単な仕組みで実現されるストリームですが、だんだんと今までのプログラミングと異なるパラダイムのプログラミングスタイルを実現できることがわかってきたと思います。次はさらに衝撃的なエラトステネスの篩 (sieve of Eratosthenes)による素数の無限ストリームの例です。まず、最初の素数である 2 から始まる整数ストリームを作ります。次に、先頭の 2 をとりだして、2 以降のストリームから 2 の倍数を全てフィルタします。次にそのフィルタされたストリームの最初の要素をみます。それは 3 になるのですが、3 を取り出して、3 以降のストリームから、3 の倍数を全てフィルタします。続いて、そのフィルタされたストリームの先頭は 5 になるので、ストリームの残りから 5 の倍数を全てフィルタする、ということを繰り返し行えば、素数のストリームが得られます。

```
(define (sieve stream)
  (cons
   (stream-car stream)
   (delay (sieve (stream-filter
                  (lambda (x) (not (divisible? x (stream-car stream))))
```

```
                    (stream-cdr stream))))))
(define primes (sieve (integers-starting-from 2)))
```

これに対して、素数のストリームをみてみると、

```
> (stream-ref primes 0)
2
> (stream-ref primes 1)
3
> (stream-ref primes 2)
5
> (stream-ref primes 3)
7
> (stream-ref primes 4)
11
> (stream-ref primes 5)
13
```

と素数のストリームが得られている感じがします。map でまとめて表示させると、

```
> (map (lambda (x) (stream-ref primes x)) (range 0 20))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73)
```

とうまく素数列が得られていることがわかります。

stream-map の拡張

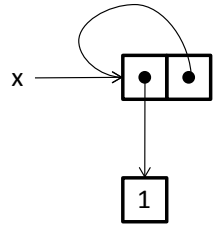
stream-map を拡張することを考えます。今までの stream-map は一本のストリームしか受け取ることができませんでしたが、次のように定義することで、二本以上のストリームを受け取り、各ストリームの要素に対し、proc を実行します。

```
(define (stream-map proc . argstreams)
  (if (null? (car argstreams))
      '()
      (cons
       (apply proc (map (lambda (x) (stream-car x)) argstreams))
       (delay (apply stream-map
                     (cons proc (map (lambda (x) (stream-cdr x))
argstreams)))))))
```

一本しかストリームを渡さない場合は今までの stream-map と同じ出力となります。上で用いられている apply は特殊な関数で、関数とリストの二つの引数をとります。apply はリストの要素を引数として関数を作用させます。例えば、(apply + (list 1 2 3 4)) は 10 を返します。例えば、s1 を 1,2,3,..と続く整数のストリーム、s2 を 2,4,6,…と続く偶数のストリーム、s3 を 3,6,9,…と続く 3 の倍数のストリームとすると、(stream-map + s1 s2 s3)は 6, 12, 18, …と 6 の倍数のストリームを返します。これは各ストリームの各要素を+で足しているからです。

自分自身を用いて定義する無限ストリーム

無限ストリームでなくとも、次のような構造を作れば無限のリストが得られます。



これは例えば

```
(define x (list 1))

(set-cdr! x x)
```

として、x の cdr を x 自身とすれば作成できます。もしくは、

```
(define x (cons 1 x))
```

とやっても同じ構造がつくれそうですが、x を定義する上でまだ x が定義されていないため、この定義はエラーとなってしまいます。そこで、遅延評価を用いると次のように無限の列を実現することができます。

```
(define ones (cons 1 (delay ones)))
```

これは 1 が無限に続くストリームになっています。先ほどまでは補助関数を用いて無限ストリームを表現していましたが、このようにすると、補助関数を用いてなくても無限ストリームを定義することができます。このように自分自身を用いて定義されたストリームのことを「自分自身を用いて定義する無限ストリーム」と呼ぶことにしましょう。

続いてストリームを足し算で合成する関数を作ることを考えます。

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

すると、最初に定義した integers も次のように定義することができます。

```
(define integers (cons 1 (delay (add-streams ones integers))))
```

第一要素が 1 で、残りが ones と integers の和であるストリームとしています。つまり、第二要素が、ones の先頭の要素(=1)と integers の先頭の要素(=1)の和となり、第二要素は 2 となります。続いて第三要素は、ones の第二要素(=1)と integers の第二要素(=2)の和となるので、第三要素は 3 となります。以下同様にして、4, 5, 6, …と定義できていることがわかります。このようにして、自分自身を利用して自分自身を定義する、という複雑な定義を与えることもストリームでは可能となります。

フィボナッチ数列の無限ストリームも、自分自身を用いて定義する無限ストリームによって定義することができます。

```
(define fibs

  (cons 0 (delay (cons 1 (delay
```

```
                (add-streams (stream-cdr fibs)

                             fibs))))))
```

この定義では先頭が 0 と 1 のストリームで残りは、自分自身と、cdr で一つずらした自身のストリームとの足し算になっている、ということです。

```
   1 1 2 3 5  8 13 21 ... = (stream-cdr fibs)
   0 1 1 2 3  5  8 13 ... = fibs
-------------------------------------------
0 1 1 2 3 5 8 13 21 34 ... = fibs
```

まとめると、今まで、(1) 補助関数の再帰呼び出しによる無限ストリームを実現する方法、(2) フィルターやマップを使って無限ストリームから無限ストリームを作る方法、(3) 自分自身を用いて定義する無限ストリームを実現する方法、またその際にストリームの合成も用いる、ということを学びました。

3.5.3 ストリームパラダイムの開発

反復をストリームプロセスとして形式化する方法について考えていきます。まず、平方根を求める関数について考えてみましょう。これについては、以前次の更新式で x に対する guess を改良することができるということを第 3 回の講義で学びました。

```
(define (sqrt-improve guess x)
   (/ (+ guess (/ x guess)) 2))
> (sqrt-improve 1.0 2)
1.5
> (sqrt-improve (sqrt-improve 1.0 2) 2)
1.4166666666666665
```

ここで、最初の予測値から始まる予測値の無限ストリームを考えれば反復をストリームで表現することができます。ここで、重要なのは実行の時間経過に伴う実行結果がストリームとして表現されている点です。

```
(define (sqrt-stream x)
  (define guesses
    (cons 1.0
          (delay (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses))))
  guesses)
```

これに対し、

```
> (map (lambda (x) (stream-ref (sqrt-stream 2.0) x)) (range 0 5))
(1.0 1.5 1.4166666666666665 1.4142156862745097 1.4142135623746899
1.414213562373095)
```

という結果が得られ、ストリームが平方根の反復改良を表していることがわかります。

3.5.4 ストリームと遅延評価

ストリームを使っていると、ある関数を実装する際にその引数にくるべきオブジェクトが、delay で作られた遅延オブジェクトなのか、それとも普通のオブジェクトなのか気にする必要があります。前者ならば一度 force で本当のオブジェクトを手に入れる必要があるし、後者だったら、force をかける必要がありません。このような違いを気にしなくてすむようにする一つの手は、全てのオブジェクトを遅延オブジェクトにしてしまう、という方法があります。そして、正規順序で評価し、遅延オブジェクトは可能な限りあとで評価する、という非常に美しいやり方も考えられます。しかし、この方法では、代入を同時に用いることが非常に難しくなります。評価の順序に依存する代入と、評価の順序を変えてしまう遅延評価は非常に相性が悪いと言えます。また、多くのプログラミングで代入を用いずに効率の良いプログラムを書くことが難しい、ということが知られていて、代入操作は必ず必要な操作となっています。つまり、正規順序＋全て遅延オブジェクトによる評価を実現するのは難しい、ということです。こういった遅延評価の枠組みを有効に活用しているのがスレッドプログラミングを含む並列プログラミングといえますが、順番を気にせずに正しい計算を行い、かつ美しいという並列プログラミング言語の設計を考えるということは大きな研究テーマの一つになっています。