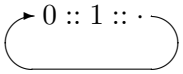
6　無限なデータ構造

ここまで紹介されたデータ構造のほとんどは有限なものであった。そもそも、コンピュータのメモリが有限だということを考えれば、実際のコンピュータの中には無限なデータ構造はありえない。

しかしながら、理論的に扱う構造は無限だったりする。例えば、自然数の集合、素数の集合、あるいは無限な木構造。

無限なデータ構造には、本質的に違う二種類がある。正則なデータとそうでないデータ。正則なデータは理論的には無限でも、有限な形で表現できる。例えば、0 と 1 が交互に現れる無限リストは ocaml で直接に定義できる。

```
# let rec zeroone = 0 :: 1 :: zeroone;;
val zeroone : int list = [0; 1; 0; 1; ...]
```



正則でないデータはある規則によって作られても、そのデータをメモリ上に完全に表現することができない。例えば、π の十進法による表現。全ての桁を計算するアルゴリズムが存在しても、その計算が終わるのに無限な時間と無限な資源が必要となる。

```
3 :: 1 :: 4 :: 1 :: 5 :: 9 :: 2 :: 6 :: ...
```

6.1　正則なデータ

上記の zeroone は生息なデータを定義しているが，このように let rec だけで定義しようとすると，定義できる値に厳しい制限が課される．新しいデータ構造を作れば，自由に作れるようになる．

```
type mutlist = Nil | Cons of mutcell
and mutcell = {hd: int; mutable tl: mutlist}

# let one = {hd = 1; tl = Nil};;
val one : mutcell = {hd = 1; tl = Nil}
# let zeroone = Cons {hd = 0; tl = Cons one};;
val zeroone : mutlist = Cons {hd = 0; tl = Cons {hd = 1; tl = Nil}}
# one.tl <- zeroone;;
```

そうすると、zeroone の値が再帰的になり、意味的には無限なものになる。使用するとき、zeroone = Cons{hd=0; tl=Cons{hd=1, tl=Cons{hd=0; tl=Cons ...}}} と思えばいい。実際にはメモリ上の番地を比べる以外、本当に無限なものと区別する方法はない。

グラフの表現

第 6 階のグラフの表現も正則なデータ構造を定義している．これによって，全体的なノードのリストか配列を参照しなくて済む．

```
type 'a node = {data: 'a; mutable neighbours: ('a node * int) list}
```

- 各頂点を頂点特有の情報 (ID 番号など) とそこから行ける頂点のリストとして表現する。
- ある頂点から同じ頂点に一周して戻ることがあるので、無限な木構造になったりするが、頂点と辺の数が有限なので正則なものである。

6.2 正則でないデータ

前節で見たように、データが正則なら、有限なものとほとんど変わらない扱いができる。実は、多くの場合では、理論的な無限性を忘れてそのデータを扱うことも多い。しかし、本質的に無限な (正則でない) データに関して、計算というもの自体が問題であり、扱いがもう少し複雑になる。

6.2.1 遅延評価

正則でない無限なデータを表現しようと思えば、上に述べたように、それを完全に計算してはいけない。無限な計算になってしまうからである。では、計算せずにデータを表現するのにどうすればよいか。単純なアイディアだが、待てばいい。そのデータの実際の値が必要になったとき初めて計算すればいい。それを遅延評価という。

正則でない最も簡単な例から始めよう。自然数のリストである。

```
type delaylist = Nil | Cons of int * (unit -> delaylist)

# let rec natfrom n = Cons (n, fun () -> natfrom (n+1));;
val natfrom : int -> delaylist = <fun>
# let nats = natfrom 0;;
val nats : delaylist = Cons (0, <fun>)
```

このリストの中身を見るのにアクセス関数を定義する。

```
let empty l = (l = Nil);;
val empty : delaylist -> bool = <fun>
let hd l =
  match l with Nil -> failwith "hd" | Cons(head,f) -> head ;;
val hd : delaylist -> int = <fun>
let tl l =
  match l with Nil -> failwith "tl" | Cons(head,f) -> f () ;;
val tl : delaylist -> delaylist = <fun>

# hd nats;;
- : int = 0
# tl nats;;
- : delaylist = Cons (1, <fun>)
# tl (tl nats);;
- : delaylist = Cons (2, <fun>)
```

こんなリストがあれば、π の全桁のリストも定義できる。ある桁を見に行かない限り、それは計算されないので、データの意味が無限でも、有限な形で表現できる。

しかし、このやり方には大きな欠点がある。それは、データを見に行く度に計算をしなければならないことだ。要するに、このリストは関数をデータに見せ掛けただけなのである。記憶機能を加えないと、このデータはあまりにも非効率的だ。

6.2.2 怠惰評価

怠惰評価というのは、遅延評価と同様に、必要になったとき始めて計算を行う。しかし、計算された結果は捨てずに、データ構造の中にあった関数と置き換える。このやり方では計算が最も少ないので、怠惰という。普通のリストみたいに必要としない値を計算する必要はないし、遅延評価みたいに何度も同じ計算を行わない。そのためにデータ構造の定義をまた拡張しないといけない。

```
type lazylist = Nil | Cons of lazycell
and lazycell = {hd: int; mutable tl: lazyval}
and lazyval = Known of lazylist | Unknown of (unit -> lazylist)

# let cons head f = Cons {hd = head; tl = Unknown f};;
val cons : int -> (unit -> lazylist) -> lazylist = <fun>
# let rec natfrom n = cons n (fun () -> natfrom (n+1));;
val natfrom : int -> lazylist = <fun>
# let nats = natfrom 0;;
val nats : lazylist = Cons {hd=0; tl=Unknown <fun>}
```

データ構造が複雑になったため cons 関数を定義したこと以外、ここまでは遅延評価とほとんど変わらない。違うのはアクセス関数である。値を読みに行って、もしもその値がまだ計算されていなければ、それを計算し、その結果をデータ構造の中で保存する。

```
let hd l =
  match l with Nil -> failwith "hd" | Cons cell -> cell.hd
val hd : lazylist -> int = <fun>
let tl l =
  match l with
    Nil -> failwith "tl"
  | Cons {tl = Known l} -> l
  | Cons ({tl = Unknown f} as cell) ->
      let l = f () in
      cell.tl <- Known l;
      l
val tl : lazylist -> lazylist = <fun>

# hd nats;;
- : int = 0
# tl nats;;
- : lazylist = Cons {hd=1, tl=Unknown <fun>}
# nats;;
- : lazylist = Cons {hd=0; tl=Known (Cons {hd=1, tl=Unknown <fun>})}
```

最後の nats の値で分かるように、リストの後部を見ること、計算済みの部分が変わり、表示される値が変わる。しかし、知られなかった結果が知られるようになるだけで、理論的には値は変わっていない。だから、その変更にも関わらず、nats を 0 からの自然数のリストと思ってもよい。

6.2.3 エラトステネスのふるい・怠惰版

数字に関するアルゴリズムは古代から数多く伝わってきている。その中で、この素数のリストを作る方式は有名である。

原理は簡単。ある N までの全ての素数を計算するには、それを小さいものから数えあげて、k が素数かどうかを判断するのに今まで見付けた素数のふるいに掛ける。そのどれかで割れれば、ふるいに引掛ったということで捨て、通れば素数のリストに加える。

普通のアルゴリズムでは、N をあらかじめ決め、それまでの素数を計算する。しかし、アルゴリズムは N と関係はない。今までの結果を利用すれば、それだけ次の素数が早く計算できる。最初から理論的に全ての素数のリストを作り、必要に応じてそれを計算すればいい。

```
let rec filter p l =
  match l with
```

```
    Nil -> Nil
  | Cons {hd = m} ->
      if p m then cons m (fun () -> filter p (tl l))
      else filter p (tl l)
val filter : (int -> bool) -> lazylist -> lazylist = <fun>
let filter_multiples l =
  let n = hd l in filter (fun m -> m mod n <> 0) (tl l);;
val filter_multiples : lazylist -> lazylist = <fun>
let rec primes l = cons (hd l) (fun () -> primes (filter_multiples l))
val primes : lazylist -> lazylist = <fun>
let all_primes = primes (natfrom 2)
val all_primes : lazylist = Cons {hd=2; tl=Unknown <fun>}

# tl(tl(tl(tl(tl all_primes))));;
- : lazylist = Cons {hd=13, tl=Unknown <fun>}
```

filter は l の中から n の倍数を除いた怠惰リストを返す。

filter_multiples はある怠惰リストを受けて、その先頭の要素で割れない要素の怠惰リストを返す。primes はそのふるいを再帰的に掛けて、l の中の素数の怠惰リストを作っていく。

結局、all_primes は primes を 2 からの自然数の怠惰リストに掛けて、全ての素数の怠惰リストを作る。

比較のために、ここに通常のリストを使ったプログラムを併記する。

```
let rec primes l =
  match l with
    [] -> []
  | n::l -> n :: primes (List.filter (fun m -> m mod n <> 0) l)
val primes : int list -> int list = <fun>
let rec nats lo hi =
  if lo <= hi then lo :: nats (lo+1) hi else []
val nats : int -> int -> int list = <fun>
# primes (nats 2 100);;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97]
```

練習問題 6.1

1. lazylist の最初の n 個を int list として返す関数を定義せよ.

```
val take : int -> lazylist -> int list
```

2. lazylist で既に計算されている値 *(*Known な部分*)* のリストを返す関数を定義せよ.

```
val ready : lazylist -> int list
```

3. lazylist の型定義を多相型'a lazylist に変更せよ.

4. lazylist を実現するクラスを定義せよ.

```
type 'a lazyval = Known of 'a | Unknown of (unit -> 'a)
class ['a] cons : 'a -> (unit -> 's) ->
  object ('s)
     val mutable tl : 's lazyval
     method hd : 'a
     method tl : 's
  end
```

take と ready は継承なしで定義できるか？filter は？

前回の解

連結成分でオブジェクトを使う

```
let rec connected_rec ~(graph:(_,_)#graph) ~reached i =
  if not reached.(i) then begin
    reached.(i) <- true;
    List.iter (fun (j,_) -> connected_rec ~graph ~reached j) (graph#edges i)
  end
val connected_rec : graph:('a, 'b) #graph -> reached:bool array -> int -> unit
let connected ~(graph:(_,_)#graph) ~start =
  let reached = Array.create (graph#nvertices) false in
  connected_rec ~graph ~reached (graph#index start);
  List.fold_right2
    (fun v r l -> if r then v :: l else l)
    graph#vertices
    (Array.to_list reached)
    []
val connected : graph:('a, 'b) #graph -> start:'a -> 'a list
```

add_vertices の仮想クラス

```
class virtual ['a,'b] graph_add = object
  val virtual vertices : 'a list
  val virtual edges : (int*'b) list array
  method add_vertices l =
    {< vertices = vertices @ l;
       edges = Array.append edges (Array.create (List.length l) []) >}
end
class virtual ['a, 'b] graph_add :
  object ('c)
    val virtual edges : (int * 'b) list array
    val virtual vertices : 'a list
    method add_vertices : 'a list -> 'c
  end
```

連結成分の直径

```
let radius ~graph ~start =
  let status = shortest_paths ~graph ~start in
  Array.fold_left
    (fun r s -> match s with Reached (d,_) -> max r d | _ -> r)
    0 status
val radius : graph:('a, int) graph -> start:'a -> int
let diameter ~graph ~member =
  let nodes = connected ~graph ~start:member in
  List.fold_left
    (fun dia start -> max dia (radius ~graph ~start))
    0 nodes ;;
val diameter : graph:('a, int) graph -> member:'a -> int
diameter graph "Imaike";;
- : int = 9
```

頂点の数が n，辺の総数が e なら，radius の計算量は $O(e) + O(n) = O(e)$ ($e \geq n$ と仮定して)，diameter は $O(ne)$．最大では辺の数が n^2 なので，そのときの diameter の計算量は $O(n^3)$.