ちょっと変なプログラミング言語 — 遅延評価を行なう関数型言語について*

香川 考司†
京都大学数理解析研究所

May 11, 1994

Abstract

一般に良く知られているプログラミング言語 (C や、もしかしたら Lisp など) の関数 (手続き) 呼び出しは、すべて先行評価という戦略に基づいている。つまり、関数の引数は関数に渡される前に完全に計算される。これに対して、関数の引数を必要になるまで計算しない、という戦略に基づく、変な言語が世の中にはある。この変な言語での変なプログラムを紹介する。

1 はじめに

たいていのプログラミング言語は、関数[1]呼び出しの機構を持っています。つまり、何度も繰り返される処理を関数として独立させて、プログラミングの手間を軽減することができます。これらの関数には引数[2]を与えることができます。引数によって関数の振舞いは少しずつ異なってきます。

しかし、引数は関数に渡される前に完全に計算されてしまうという点では、上にあげた中ではどの言語でも似たようなものです。ふつう、関数の引数としては必要なものしか渡さないから、これは人間にとっても自然であるように見えます。例えば、`double` という (つまらない) 関数を次のように定義したとしましょう。

```
double x = x + x
```

(一応、具体的な文法としては、これから説明する遅延評価型の関数型言語の代表的な言語である Haskell[HW+92] というプログラミング言語のものを用います。しかし、その文法は数学で使われる記法と良く似ているので、特にわかりにくいと思われる点以外は取り立てて説明することはしません。) `double (2+2)` という式を (手で) 計算して下さいと言われれば、たいていの人は次のように計算するは

*About a Little Curious Programming Languages — Lazy Functional Languages

†E-mail:kagawa@kurims.kyoto-u.ac.jp

[1] これは C や Lisp などでの呼び方で、Fortran や Pascal, BASIC では手続き (procedure) あるいはサブルーチン (subroutine) と呼ぶ (んだと思う)。

[2] “ひきすう” と読みます。

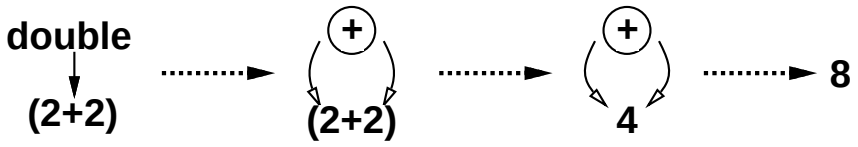
ずです。

$$\begin{aligned} \textit{double}\ (2+2) &= \textit{double}\ 4 \\ &= 4+4 \\ &= 8 \end{aligned}$$

しかし、このような呪縛 (思い込みと言っても良い) を離れることによって**新しい世界**が姿を現すのです。**遅延評価**[3]**(lazy evaluation)** というのは関数の計算を次のような方法で行なうことです。

$$\begin{aligned} \textit{double}\ (2+2) &= (2+2)+(2+2) \\ &= 4+4 \\ &= 8 \end{aligned}$$

つまり、*double* の引数である (2 + 2) は計算されないまま、まず *double* の定義にしたがって式が展開されます。そして、本当に必要になって (この場合は + の引数ですから必要です。) はじめて計算されるのです。人間が手で計算をする場合にこのような方法で計算しないのは、一つには通常、数学では引数を必ず必要とするような関数しか考えないこと、もう一つは上の例でわかるように、計算の手間が増えるからです。(つまり、2 + 2 を 2 回計算しています。) しかしこれは紙の上で計算するからであってコンピュータが実行する場合には必ずしも後者はあてはまりません。つまり、コンピュータの中では上の計算をグラフの形で表して、2 + 2 を一度しか計算しないようにアレンジすることができるのです。



遅延評価の良い点は何といっても概念的には無限のデータ構造を扱えること[4]でこれに尽きると言っても良いと思います。このためいろいろと面白いことができるのです。もちろん下に述べるようないろいろな問題点があるので[5]、現実的なプログラムを書くのにはあまり使われていないのですが、もう少し注目されても良い[6]言語だと思います。

遅延評価に対して、最初に示した関数の引数をまず計算する戦略を**先行評価 (eager evaluation)** といいます。先行評価の良い点は、何と言っても効率の良い処理系を作ることが簡単だと言うことです。また、代入文や入出力などの**副作用 (side effect)** がある場合にも、遅延評価というのは問題となると一般に信じられています。つまり、式はじっさいに必要となるまでは評価されませんから、どこで代入文などが行なわれるか予測しがたいと考えられているのです。そこで、遅延評価を行なう言語では一般に副作用というものを持っていません。つまり、“関数”というのはまさに数学的な関数であり、字面が同じ式は、何度計算しても同じ値

[3]**評価(evaluation)** というのは独特の術語ですが、計算と同じくらいの意味だと考えて下さい。
[4]もちろん、コンピュータのメモリには限りがありますので本当に無限ではありませんが。
[5]少しずつ、改善されてきていますが。
[6]少なくとも理学部のような現実離れしたところでは。

を返すのです。副作用を持つ言語では、そんなことは決してありません。まさにこの点が、多くの人に遅延評価が非現実的だと信じられている理由だと思います。皆さんも、入出力文や代入文なしでどのように実際のプログラムを書くのか、という疑問を当然抱くと思います。実は、このへんの話題は、現在のコンピュータサイエンスの一部で、ホットな話題 [Wad90, Wad92, PJW93] なのですが、ここでは残念ながら、とてもそのようなアドバンストな話題には触れられません。

遅延評価という言葉については、いま一応説明しました。では**関数型言語 (functional language)** という言葉は何をあらわすのでしょうか? 上で述べたように副作用のない言語では、関数はまさに数学的な関数です。しかし、このことをもって関数型言語と呼ぶのではありません。副作用を持つ言語の中にも関数型言語と呼ばれる言語はあるのです。(ML[Pau91] という言語や Scheme[A$^+$91, AS85] という言語などです。これらの言語はもちろん先行評価を採用しています。) しかし、C や Fortran などは関数呼び出しという機構はあっても関数型言語とは普通呼びません。通常は **高階関数 (higher-order function)** があるかどうか、あるいは関数を **first-class object** として扱えるかどうかで関数型言語かどうかということを区別します。これらの言葉の意味については、その概念が出てきたところで説明することにしましょう。なお、つけくわえると、関数を first-class として扱えるかどうかという点はプログラミング言語が**ゴミ集め (garbage collection)** の機構を持っているかという点に深く関係しています。ゴミ集めというのは必要のなくなったメモリを自動的に解放する機構です[7]。

さて、それでは Haskell でのプログラミングについて具体的に見ていきましょう。

2 リストについて

リスト (list) は、実用上重要なデータ構造であり、また関数型言語、特に遅延評価の記述力を示す例として最適です。リストというのは簡単に言えばデータの並び (sequence) です。集合と違ってリストでは要素の順番や数が重要になります。Haskell ではリストは ブラケット ([,]) で要素をくくることによって表すことができます。

```
[1, 2, 3, 4, 5]                 :: [Int]
["Imadegawa", "Oike", "Sanjo"] :: [String]
```

`::`の後ろは **型 (type)** です。つまり、`[1..5]` は `Int` (integer, 整数) のリストの型であるということを表しています。型のことについては、あまり立ち入るつもりはありませんが、理解の助けになると思われる時にはこのように明示することにします。

2.1 リストの構成

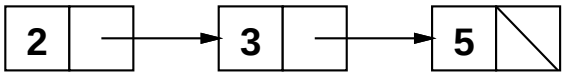
では、リストはコンピュータの中でどのように表現されているのでしょうか? リストは次のうちのどちらかです。

[7] 知ってる人にしかわからないことを覚悟で言うと、C にはゴミ集めの機構がないので、ヒープに確保したデータが必要なくなれば `free` などを用いてメモリを解放しなければならないのです。

1. 空のリスト、何も要素のないリストである。
   空のリストを **nil** と呼び、`[]` という記号で表します。

2. 少なくとも 1 つ要素を含んでいるリストである。
   この場合、リストを先頭の要素と残りの尾部のリストに分けて考えます。先頭の要素 (**head**) を `a`、残りのリスト (**tail**) を `as` とすると、このようなリストは、`a:as` と表現し、`a` と `as` の **cons**[8]と呼びます。

つまり、`[2, 3, 5]` は、`2:(3:(5:[]))` の略記法なのです。また、このような“`:`”(infix cons, 中置記法の cons) を使った書き方の場合、“`:`” を右に結合する演算子 (right assosiative operator) と考えて、単に `2:3:5:[]` と書きます。図を用いて次のように表現することもあります。



この記法を**箱矢印記法 (box-and-pointer notation)** と言います。斜線が入っているところは nil を表します[9]。また**配列 (array)** というデータ構造と混同されがちですが、配列は大きさが固定されたデーター構造でリストのように伸び縮みしません。コンピュータの中では配列は連続した領域で表すことができます。

2.2 リストの内包表記

Haskell には**リストの内包表記 (list comprehension)** という便利な記法があります。これは、おなじみの集合の内包表記 (set comprehension) に似ています。例で示すのが簡単でしょう。

```
? [ x+x | x <-[1..10], odd x]
[2, 6, 10, 14, 18]
(138 reductions, 195 cells)
? [ (x, y) | x <- [1, 2, 3], y <- [4, 5]]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
(107 reductions, 246 cells)
```

これは実際に Gofer という処理系を使った対話例で `?`から行末までがユーザの入力を表し、他はすべて Gofer の出力したものです。次のような集合の内包表記と比べてみましょう。

$$\{x + x | x \in \{1, \ldots, 10\}, x : odd\}$$

$$\{(x, y) | x \in \{1, 2, 3\}, y \in \{4, 5\}\}$$

とても良く似ていることが一目瞭然です。ただし、リストの場合には順番、重複なども無視できないことは覚えておいて下さい。上のようにリストの内包表記には、`x <-[1..10]` のような**生成式 (generator)** か、`odd x` のような論理値 (boolean, **真 (true)** か **偽 (false)**) をとる式かのどちらかをコンマで区切って書くことができます。

[8] Lisp 以来の歴史的な呼び名ですが、constructor の略です。

[9] これもまた知ってる人にしかわからないと思いますが、C などでリストを表す時は矢印の部分がポインタになります。

実はリストの内包表記は単なる**構文上の糖衣(syntactic sugar)**であり、構文解析時かその直後に下で定義する関数 (`map`, `filter`, `concat` など) を用いた式に翻訳されます。

2.3 関数定義

ここで、リストを扱うのに必要な関数をいくつか定義しておきましょう。同時にHaskell での関数定義の方法を詳しく見ていきましょう。関数は次のような等式(の集まり) により定義します。

```
even, odd :: Int -> Bool
even x   =   (x `rem` 2) == 0
odd  x   =   (x `rem` 2) /= 0
```

`even`, `odd` は整数から論理値への関数です。`x `rem` 2` は `x` を `2` で割った余りを表す式です。これが `0` と等しいか (==)、否か (/=) で偶奇を判定します。ここで "=" は関数を定義するための等号ですが、これを 2 つ並べたもの "==" は、数値などが等しいかどうかを判断して真偽値を返す演算子であることに注意して下さい。(C でも同じような記法を用いていると思います。) また数学では $even(x)$ のように関数の引数に括弧をつけて次のように書くのが普通だと思いますが

$$even(x) \stackrel{\text{def}}{=} x \; rem \; 2 = 0$$
$$odd(x) \stackrel{\text{def}}{=} x \; rem \; 2 \neq 0$$

Haskell をはじめとする多くの関数型言語はこれを省略します。

Haskell では次のような**パターンマッチング (pattern matching)** による関数の定義をすることができます。

```
head              :: [a] -> a
head (x:xs)        = x

tail              :: [a] -> [a]
tail (x:xs)        = xs

null              :: [a] -> Bool
null []            = True
null (x:xs)        = False
```

例えば、`null` という関数は、引数が cons か nil かの場合分けによって定義します。また `head` や `tail` の定義を見てわかるように、パターンの中に現れた変数を右辺で用いることができます。また例えば、`head []` や `tail []` は上の定義の中にないのでエラーになります[10]。

以下にこの資料で用いる関数の定義をまとめてあげておきます。これらの多くは多引数の関数です。多引数の関数は単に引数を並べて書きます[11]。

[10] これらの関数の型の中で `a` などは**型変数 (type variable)** を表します。型変数は `Int` や `Char` などに具体化できます。

[11] 実際には、多引数の関数はカリー化という手段を用いて一引数の関数として表 n されています。`map` を例にとると、この関数は、`a -> b` という型の関数を引数にとり、`[a] -> [b]` という型の**関数**

```
length            :: [a] -> Int
length []          = 0
length (x:xs)      = 1 + (length xs)

map               :: (a -> b) -> [a] -> [b]
map f []           = []
map f (x:xs)       = (f x) : (map f xs)

zip               :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs)  = (a, b) : (zip as bs)
zip _      _       = []

filter            :: (a -> Bool) -> [a] -> [a]
filter _ []        = []
filter p (x:xs)    = if p x then x : (filter p xs) else filter p xs

append            :: [a] -> [a] -> [a]
append [] ys       = ys
append (x:xs) ys   = x : (append xs ys)

concat            :: [[a]] -> [a]
concat []          = []
concat (xs:xss)    = append xs (concat xss)
```

パターンマッチングを用いると関数の定義を簡潔に行なうことができることがわかります。_というパターンはワイルドカード (wild card) として用いることができます。length はリストの長さを求める関数です。また、map は、第 1 引数の関数を第 2 引数のリストの各要素に適用する関数です。zip は 2 つのリストを綴じ合わせます。filter はリストの要素のうち第 1 引数の述語を True にするもののみを返します。append は 2 つのリストをくっつけます。また concat はリストのリストをフラットなリストにします。

```
? map double  [1, 2, 3]
[2, 4, 6]
(17 reductions, 45 cells)
? length [1, 2, 3]
3
(12 reductions, 23 cells)
? zip [1, 2, 3] [4, 5, 6]
[(1,4), (2,5), (3,6)]
(19 reductions, 84 cells)
? filter odd [1, 2, 3]
[1, 3]
(24 reductions, 46 cells)
```

を返す関数として表されます。つまり、(a -> b) -> [a] -> [b] は、(a -> b) -> ([a] -> [b]) を表します。

```
? append [1, 2, 3] [4, 5]
[1, 2, 3, 4, 5]
(15 reductions, 58 cells)
? concat [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[1, 4, 7, 2, 5, 8, 3, 6, 9]
(36 reductions, 125 cells)
```

次の 3 つの関数は特に無限リストを取り扱う際に必要になります。

```
take              :: Int -> [a] -> [a]
take 0     _       = []
take _     []      = []
take n     (x:xs)  = x : (take (n-1) xs)

takeWhile         :: (a -> Bool) -> [a] -> [a]
takeWhile p []     = []
takeWhile p (x:xs) = if p x then x : (takeWhile p xs)
                            else []

iterate           :: (a -> a) -> a -> [a]
iterate f x        = x : (iterate f (f x))
```

take はリストの最初の n 個を取り出す関数、takeWhile は、述語が満たされる間、リストの要素を取り出す関数です[12]。iterate は無限リストを作る時に基本となる関数です。

```
? [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
(77 reductions, 155 cells)
? take 5 [1..10]
[1, 2, 3, 4, 5]
(44 reductions, 96 cells)
? takeWhile (/=0) [1, 2, 3, 4, 0, 5, 6, 7, 8]
[1, 2, 3, 4]
(34 reductions, 80 cells)
? take 10 (iterate double 1)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
(60 reductions, 155 cells)
```

3 無限リスト

遅延評価を採用することにより、無限リストを有限のリストと全く同様に扱うことができます。例えば、ひじょうに単純な例ですが

```
ones = 1 : ones
```

[12] ここで、(/=0) は零でないという意味の述語 (真偽値を返す関数) です。例えば (+1) は 1 を足す関数、(1.0/) は逆数を求める関数です。このような記法は**セクション**(**section**) といいます。

で 1 ばかりからなる無限のリストを定義できます。また、

```
nats = from 1
   where from n = n : from (n+1)
```

はすべての自然数のリストを表します。一般に [m..] は、m から始まる整数の無限リストを表します。また、無限リストの無限リストなどなども可能です。

遅延評価がどのようにこのような無限のリストを可能にするのかもう少し詳しく見てみましょう。例えば take 3 nats という式がどのように計算されるか考えてみます。まず、最初に take の関数の定義を展開しますが、この時、第 2 引数である nats が cons か nil かを確かめる必要があります。つまり、take 3 nats の計算は nats を計算することからはじまります。

```
take 3 nats → take 3 (1:from (1+1)) → 1:(take 2 (from (1+1))) →
1:(take 2 (2:from (2+1))) → 1:2:(take 1 (from (2+1))) → ··· →
1:2:3:(take 0 (from (3+1))) → 1:2:3:[] (=[1, 2, 3])
```

何度も言うようにリストと集合を混同してはいけません。内包表記
[x^2 | x <- [1..], x^2 < 10] を表示させると、(x^2 は x^2 を表します。)

```
? [ x^2 | x <- [1..], x^2 < 10]
[1, 4, 9{Interrupted!}

(61217 reductions, 118082 cells, 1 garbage collection)
?
```

のようになってしまい、永久に計算を続けようとします。つまりコンピュータには 4 以降に 2 乗して 10 より小さい数がないということは自明ではありません。ただし、これは、内包表記を等価な式 (filter (<10) (map (^2) [1..])) に書き換え、さらに filter を takeWhile に書き換えることによって、避けることができます。

```
? filter (<10) (map (^2) [1..])
[1, 4, 9{Interrupted!}

(38030 reductions, 64905 cells, 1 garbage collection)
? takeWhile (<10) (map (^2) [1..])
[1, 4, 9]
(84 reductions, 162 cells)
?
```

4 素数の生成

無限リストを利用して、エラトステネス (Eratosthenes) のふるいを実装します。このおなじみのアルゴリズムを言葉で表現すると次のようになります。

① 2 以上の自然数を並べる。

② 先頭の数を取り除き、その倍数を同時にとり除く。

③ ②を繰り返す。

この時に先頭に現れた数を順番に並べたものが素数の列です。
　文章で書かれたものをそのまま翻訳すると、次のようになります。

```
primes          = map head (iterate sieve [2..])
sieve (p:xs)    = [x | x <- xs, x `mod` p /= 0]
```

無限のリストが含まれていようが全く気にすることはありません。このプログラムは無限リストの無限リストを用いていることに注意しましょう。そしてこれで立派な Haskell のプログラムなのです。このようにして、素数を無限リストとして表現することで、さまざまな “境界条件” に対応することができます。C などで実装しようとすると、1000 までの素数というのは配列を用いて簡単に求めることができますが、最初の 100 個の素数を求めるのは急に難しくなります。
　実際に実行してみると、

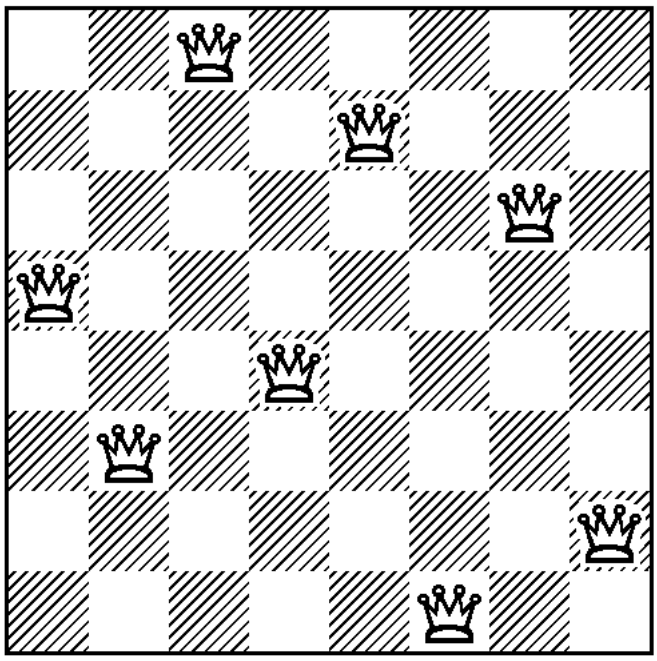
```
? take 20 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71]
(1592 reductions, 2572 cells)
? takeWhile (< 1000) primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563,
569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643,
647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739,
743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829,
839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
941, 947, 953, 967, 971, 977, 983, 991, 997]
(83278 reductions, 132044 cells, 1 garbage collection)
?
```

となります。

5　8 クイーンの問題

8 個のクイーンを、お互いにとることができないように、チェス盤の上に置くという問題です。可能な解はいくつか存在します。この可能な解の集まりをリストとして表現します。ここでは無限リストは使用しませんが、遅延評価は効率の点で決定的な役割を果たします。

クイーンの配置は、ここでは数のリストで表します。[4, 6, 1, 5, 2, 8, 3, 7] は次のような配置を表します。



safe p n は、$m-1$ (length p) 列までのクイーンの配置が p というリストで与えられた時、第 m (length p + 1) 列の第 n 行にクイーンを置くことができるかどうかを示す関数です。

```
safe p n  = all not [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
       where m = 1 + length p

check (i,j) (m,n) = j==n || (i+j==m+n) || (i-j==m-n)
```

ここで、

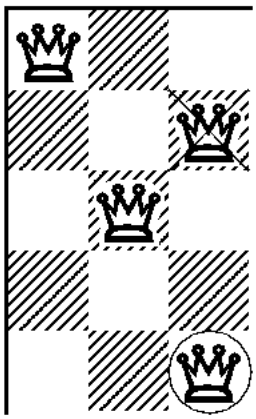
```
all               :: (a -> Bool) -> [a] -> Bool
all p []           = True
all p (x:xs)       = if p x then all p xs else False
```

です。例えば、

```
? safe [1, 3] 5
True
(53 reductions, 114 cells)
? safe [1, 3] 2
False
```

```
(45 reductions, 107 cells)
?
```

となります。



順に、最初の m 列のすべての安全な配置を調べていきます。そのために、そのようなすべての配置 (のリスト) を返す queens という関数を定義します。

```
queens 0  = [[]]
queens m  = [ append p [n] | p<-queens (m-1), n<-[1..8], safe p n ]
```

例えば

```
? queens 1
[[1], [2], [3], [4], [5], [6], [7], [8]]
(172 reductions, 387 cells)
? queens 2
[[1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [1, 8], [2, 4], [2, 5],
 [2, 6], [2, 7], [2, 8], [3, 1], [3, 5], [3, 6], [3, 7], [3, 8],
 [4, 1], [4, 2], [4, 6], [4, 7], [4, 8], [5, 1], [5, 2], [5, 3],
 [5, 7], [5, 8], [6, 1], [6, 2], [6, 3], [6, 4], [6, 8], [7, 1],
 [7, 2], [7, 3], [7, 4], [7, 5], [8, 1], [8, 2], [8, 3], [8, 4],
 [8, 5], [8, 6]]
(2683 reductions, 5767 cells)
```

となります。そうすると head (queens 8) で最初の解を求めることができます。

```
? head (queens 8)
[1, 5, 8, 6, 3, 7, 2, 4]
(65105 reductions, 127798 cells, 1 garbage collection)
```

遅延評価を用いているので、最初の解を求めるためには本当に必要な部分の簡約しか行ないません。つまり、上の [1, 5, 8, 6, 3, 7, 2, 4] を求めるのに、queens 7 の計算をすべて行なっているわけではなく、この最初の解を求めるのに必要なだけの部分の計算をしているのです。これは、queens 7 を完全に計算させると head (queens 8) よりも計算に時間がかかることからもわかります。

```
?  queens 7
[[1, 3, 5, 7, 2, 4, 6], [1, 3, 5, 8, 2, 4, 6], [1, 3, 8, 6, 4, 2, 5],
(略)
[8, 6, 1, 3, 5, 7, 4], [8, 6, 4, 1, 7, 5, 3], [8, 6, 4, 2, 7, 5, 3]]
(999888 reductions, 1975221 cells, 21 garbage collections)
```

ここでは、遅延評価は Prolog でいうところの**後戻り (back track)** と同じような効果を実現しています。1 つだけ解を求める計算とすべての解を求める計算を別々に記述する必要はありません。しかも、1 つだけ解を求めるためには、それに必要なだけの計算しか行ないません。

ちなみに queens 8 を計算させると、

```
? queens 8
[[1, 5, 8, 6, 3, 7, 2, 4], [1, 6, 8, 3, 7, 4, 2, 5],
(略)
, [8, 3, 1, 6, 2, 5, 7, 4], [8, 4, 1, 3, 6, 2, 7, 5]]
(1230216 reductions, 2416480 cells, 25 garbage collections)
```

解はすべてで 92 個あることがわかります。

6 その他の遅延評価リストの応用について

6.1 無限精度計算

コンピュータ内では通常は数値の表現は 32 ビットとかの有限精度です。しかし、リストを用いれば無限精度の計算パッケージを作成することも可能です。(これには、必ずしも遅延評価が必要なわけではありませんが、いろいろと有利だと考えられます。また、もちろん本質的に無限の計算を行なうのは無理でしょう。例えば $1 - 0.\dot{9} = 0$ を計算するのは無理でしょう。)

6.2 入出力ストリーム

ユーザとのインタラクションを含むプログラムを、コンピュータからユーザへの出力、ユーザからコンピュータへの入力を遅延リストと考えて、String -> String (ここで、String は、[Char] のことです。) と考えることができます。

7 処理系について

遅延評価型関数型言語の処理系については現在研究が進んでいますが、やはり、今のところは研究者の趣味の領域を出ていないと言っても過言ではないと思います。

残念ながらシリアスなアプリケーション (つまり、OS とか**ゲーム**とか) を Haskell などで書くのに成功したという事例はあまり聞きません。(もちろん、“あまり” という言葉には幅があるわけで、“全く” ないわけではありません。たとえば、いくつかの Haskell のコンパイラは Haskell 自身で書かれています。) しかし、そんなに効率が悪いかと言うと別にそうとも言い切れません。たとえばここでたびたびお世話になった Gofer という Haskell のサブセットの処理系があります。

```
Gofer Version 2.28b  Copyright (c) Mark P Jones 1991-1993
```

この処理系は、そんなに真面目に最適化の処理をしているわけではありませんが、少なくとも、この資料に出てくるような “あそびの” プログラムならワークステーション上ならすぐに (一瞬にとは言わないが) 実行してくれます。Gofer はパソコン (IBM-PC 互換機, Macintosh, Atari, Amiga など) でも動作するので、以上のプログラムを皆さんで実際に試すことができます。なお 98 でも再コンパイルして動作を確認しています[13]。Towns で動くのかどうか、筆者は試していませんがソースがついているのでコンパイルし直せば大丈夫だと思います。Gofer の最新バージョン (1994 年 5 月 1 日現在 2.28b) は次のところから ftp できます。

```
nebula.cs.yale.edu:pub/haskell/gofer
```

また、数理研にもできるだけ最新バージョンを置いておくようにします。

```
ftp.kurims.kyoto-u.ac.jp:pub/src/{gofer,Gofer}*
```

Ftpって何だかわからない人は KCSS に入りましょう。

また、遅延評価ではない関数型言語としては ML や scheme に対しても ftp-available な処理系があります。

8 参考文献について

遅延評価を行なう関数型言語でのプログラミングについては [BW88] が標準的です。(日本語訳もあります。) また [AS85, Pau91] もそれぞれ、Scheme, ML についての教科書ですが、推薦できます。[AS85] には日本語訳があります。また、遅延評価型関数型言語の実装技術については、[Pey87] が良いと思います。また論文としては、[Hud89, Hug89] などが最初に読むのには良いと思います。[Hud89] には日本語訳もあります。

謝辞

この資料の改良に寄与してくれた数理解析研究所の古瀬淳氏と KCSS の勝股審也氏に感謝します。

[13] KCSS の勝股君が確認したそうです。

References

[A+91] H. Abelson et al. Revised[4] report on the algorithmic language scheme. ftp-available, November 1991.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985. (邦訳: プログラムの構造と実行 [上・下]. 元吉文男 訳. マグロウヒル.).

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988. (邦訳: 関数プログラミング. 武市正人 訳. 近代科学社.).

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989. (邦訳: 関数プログラム言語の概念・発展・応用. 武市正人 訳. **bit** 別冊 コンピュータ・サイエンス 1989 年度版, pp. 37–87, 1992 年 9 月.).

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[HW+92] Paul Hudak, Philip Wadler, et al. Report on the prgramming language Haskell, a non-strict purely functional laguage (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1993.

[Wad90] Philip Wadler. Comprehending monads. In *ACM Symp. on Lisp and Functional Programming*, pages 61–78, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1992.