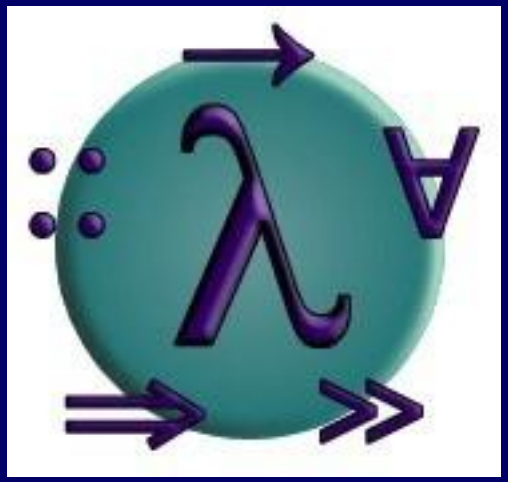
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 12 – Lazy Evaluation
遅延評価

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

用語

- 評価(evaluation, evaluate)と簡約(reduction, reduce)
  - 同じと考えて良い
  - 評価: 式の値を求めること
  - 簡約: 数学では、式をより簡単な式に置き換えること
    $a * x + a * y \Rightarrow a\,(x + y)$
    Haskell では、関数を適用し本体で置き換えること
    - 展開(unfold)も関数の本体で置き換える意味で用いられる
- 簡約可能式(リデックス、redex)
- 評価戦略
  - リデックスの集合から実際に簡約する式を選択する指針
  - 最内簡約、最外簡約、遅延評価
    - 遅延簡約とは(なぜか)言わない

導入

- ここまで、Haskell の式がどのように評価されるのかの詳細に触れなかった
- 実際のところ、式は以下の 3 つの性質を満たす単純な方法で評価される:
  1. 不要な評価をしない
  2. 評価順序を気にしないでプログラムを書ける
  3. 無限リストを扱える
- この評価方法は遅延評価(lazy evaluation)と呼ばれ、Haskell は遅延関数型プログラミング言語である

式の評価

- 基本的に、式は簡約できなくなるまで、関数定義を順に適用することにより評価される

- 例えば、定義

```
square n = n * n
```

を考えると、式 square (3 + 4) は以下のように評価される:

```
    square (3 + 4)        {+ を適用}
  = square 7              {square を適用}
  = 7 * 7                 {* を適用}
  = 49
```

- しかし、これは唯一の評価方法ではなく、他の方法も考えられる:

```
    square (3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 49
```

{square を適用}
{最初の + を適用}
{+ を適用}
{* を適用}

- + より先に square を適用しても、結果は等しい

- FACT: Haskell において、同じ式に対する 2 つの異なる評価方法が停止するならば、その結果は等しい

望ましい性質だが、手続き型言語では成り立たない

評価戦略(Reduction Strategies)

- 式を評価するとき、複数の部分式に関数を適用できることがある
- このような式をリデックス(簡約可能式)と呼ぶ
- 評価するリデックスを選択するための、よく知られた2 つの戦略:
  1. 最内簡約
     最も内側の(複数の場合、最左の)リデックスを簡約する
  2. 最外簡約
     最も外側の(最左)リデックスを簡約する
- 2 つの戦略はどう違うのか?

正確には最左最内簡約

教科書では"簡約可能式"を使っているが、スライドでは"リデックス"を使う

停止性(Termination)

- 以下の定義を考えてみる
  loop = tail loop

- 式 fst (1, loop) を 2 つの評価戦略で評価してみる:
  1. 最内簡約

```
      fst (1, loop)
    = fst (1, tail loop)
    = fst (1, tail (tail loop))
    = …
```

  評価は停止しない

2. 最外簡約

```
fst (1, loop)
= 1
```

1 ステップで結果が出る

- FACTS
  - 最内簡約が停止しないときでも、最外簡約は停止することがある
  - ある式に対して停止する評価系列があるならば、最外簡約も停止して同じ結果となる

最外簡約は強力

簡約の回数

- 再度、以下の評価系列を考えてみる:

最内簡約(3ステップ):

```
  square (3 + 4)
= square 7
= 7 * 7
= 49
```

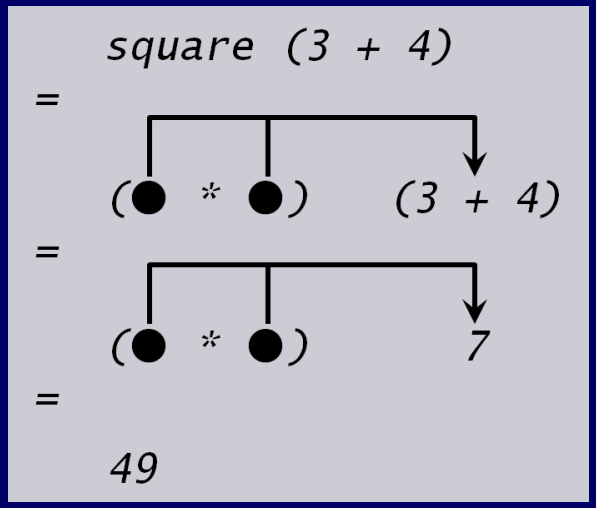
最外簡約(4ステップ):

```
  square (3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 49
```

- 最外簡約は効率が悪い: square の評価で複製された部分式 3 + 4 が 2 回評価される
- FACT: 最外簡約のステップ数は、最内簡約よりも多くなることがある

ここで効率は時間効率(ステップ数)であり、とりあえず空間効率は考えない

- この問題は、評価中に式をポインターで指して共有することで解決される:



最外簡約の強力な停止性と
式の共有による効率を兼ね備えている

- 改良された新しい評価戦略:
  遅延評価 = 最外簡約 + 式の共有 (Sharing)
- FACTS
  - 遅延評価のステップ数は最[内外]簡約よりも多くならない
  - Haskell は遅延評価を採用している

無限リスト

- 最外簡約から受け継いだ停止性に関する有利さに加えて、遅延評価のおかげで無限リストを使ったプログラムが可能となる!
- 以下の再帰的な定義を考えてみる

```
ones :: [Int]
ones = 1 : ones
```

- 再帰の展開を数回行ってみる:

```
ones = 1 : ones
     = 1 : 1 : ones
     = 1 : 1 : 1 : ones
     = …
```

ones は 1 の無限リストである

- 最内簡約と遅延評価で head ones を評価してみる:

1. 最内簡約

```
head ones = head (1 : ones)
          = head (1 : 1 : ones)
          = head (1 : 1 : 1 : ones)
          = …
```

   評価は停止しない

2. 遅延評価

```
head ones = head (1 : ones)
          = 1
```

   結果は 1 となる

- 遅延評価により、無限リスト ones における最初の値だけが実際に求められているのは、これが式 head ones を評価するのに必要な全てであるから

- 一般化したスローガン:
  遅延評価により、式の結果を求めるのに必要な部分だけを評価しよう

- ここで、
  ```
  ones = 1 : ones
  ```
  は無限リストではなく、それが使用された文脈が要求する部分だけが評価される潜在的な無限リストだと分かる

モジュラープログラミング

モジュラー: 組み合わせが容易になるように規格化されているという意味

- 無限リストから、いくつかの要素を抜き出せば、有限なリストを生成できる。例えば:

```
? take 5 ones
[1,1,1,1,1]
? take 5 [1..]
[1,2,3,4,5]
```

制御: データを参照する側
データ: データを生成する側

- 制御とデータを分離することにより、遅延評価はプログラムをよりモジュラーにする:

```
take 5 [1..]
```

制御　データ

- 遅延評価により、制御から要求される分だけデータが評価される

例: 素数の生成

- 素数の無限リストを生成する簡単な手続き:
  1. 無限リスト 2,3,4,... を生成する
  2. リストの先頭の値 p を素数として印を付ける
  3. p の倍数をリストから除く
  4. ステップ 2 に戻る

- 最初の数ステップを以下に示す:

2	3	4	5	6	7	8	9	10	11	12	...
	3		5		7		9		11		...
			5		7				11		...
					7				11		...
									11		...

- この手続きは、考案者であるギリシャの数学者に因んで “エラトステネスのふるい” と呼ばれる
- 手続きはそのまま Haskell に変換できる:

```
primes         :: [Int]
primes          = sieve [2..]
sieve          :: [Int] -> [Int]
sieve (p:xs) = p:sieve [x|x ← xs, x `mod` p ≠ 0]
```

- 以下のように実行できる:

```
? primes
[2,3,5,7,11,13,17,19,23,29,31,
 37,41,43,47,53,59,61,67,…
```

- どこまで計算するかという制約から素数の生成を解き放つことにより、モジュラーな定義が可能となり、状況に応じた異なる制約を与えることもできる
- 最初の 10 個の素数を選択:

```
? take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

- 15 未満の素数を選択:

```
? takeWhile (< 15) primes
[2,3,5,7,11,13]
```

- 遅延評価は強力なプログラミングツール!

まとめ(12章)

- 簡約可能式(リデックス)
- 評価戦略
  - リデックスの集合から実際に簡約する式を選択する指針
  - 最内簡約:　　値渡し(C や Java が採用)
  - 最外簡約:　　名前渡し(字面渡し)
    - 停止する評価系列があるならば、停止して同じ結果となる
    - 最内簡約よりも効率が悪いことがある
  - 遅延評価:　　最外簡約 + 式の共有 (Haskell が採用)
    - 停止性の有利さはそのままに、式の共有により効率化
    - 遅延評価の効率は最[内外]簡約と同等以上
- 無限リストとモジュラー性
  - 遅延評価により、潜在的な無限リストが可能になる
  - 制御(消費側制約)とデータ(生成)を分離して見通しを向上

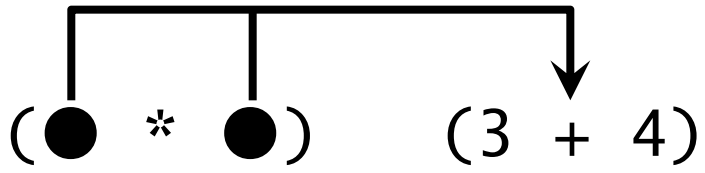
練習問題 – 12章

1. フィボナッチ数列の無限リスト
   [0,1,1,2,3,5,8,13,21,34,...
   を生成する関数
   fibs :: [Integer]
   を、以下の簡単な手続きに従って定義せよ:
   a. 数列の最初の 2 つは 0 と 1 である
   b. 次の数は直前の 2 つの数の和である
   c. ステップ b に戻る

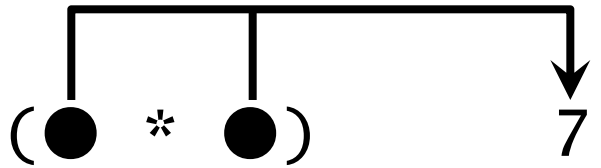
2. フィボナッチ数列の n 番目を求める関数
   fib :: Int → Integer
   を定義せよ

```
square (3 + 4)
```

=



=



=

```
49
```