计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2022年09～12月

Adapted from Graham's Lecture slides

第15章：计算模型—Lazy Evaluation

主要知识点：

计算，计算策略

无限数据，模块化程序设计

应用：素数序列计算

计算：function application

```haskell
inc :: Int -> Int
inc n = n + 1
```

```
     inc (2 * 3)
=    { applying * }
     inc 6
=    { applying inc }
     6 + 1
=    { applying + }
     7
```

```
     inc (2 * 3)
=    { applying inc }
     (2 * 3) + 1
=    { applying * }
     6 + 1
=    { applying + }
     7
```

✤ Any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate.

计算策略

- Reducible expression （redex)
  - 一个 function application
  - 称这个表达式 is reducible，因为可以将这个 function application 替换为对应的定义
- 注意：一个 redex 中可能包含更细粒度的一个或多个 redex

```
inc (2 * 3)
```

- Reduce的策略
  1. 最内策略 (innermost)
  2. 最外策略 (outermost)

最内策略

```haskell
mult :: (Int, Int) -> Int
mult (x, y) = x * y
```

最外策略

最内策略

```
    mult (1+2, 2+3)
=     { applying the first + }
    mult (3, 2+3)
=     { applying + }
    mult (3, 5)
=     { applying mult }
    3 * 5
=     { applying * }
    15
```

最外策略

```
    mult (1+2, 2+3)
=     { applying mult }
    (1+2) * (2+3)
=     { applying the first + }
    3 * (2+3)
=     { applying + }
    3 * 5
=     { applying * }
    15
```

✳ 注意：很多 built-in functions （如 ∗, +）要求它们的参数必须首先被求值

```haskell
mult :: Int -> Int -> Int
mult x = \y -> x * y
```

最内策略

```
    mult (1+2) (2+3)
= { applying the first + }
    mult 3 (2+3)
= { applying mult }
    (\y -> 3 * y) (2+3)
= { applying + }
    (\y -> 3 * y) 5
= { applying the lambda }
    3 * 5
= { applying * }
    15
```

最外策略

```
    mult (1+2) (2+3)
= { applying the mult}
    (\y -> (1+2) * y) (2+3)
= { applying the lambda }
    (1+2) * (2+3)
= { applying the first + }
    3 * (2+3)
= { applying + }
    3 * 5
= { applying * }
    15
```

✳ Note: the only operation that can be performed on a function is that of applying it to an argument.

```
  (\x -> 1 + 2) 0
= { applying the lambda }
    1 + 2
= { applying + }
    3
```

The function `\x -> 1 + 2` is deemed to be black box, even though its body contains the redex 1 + 2.

- Using innermost and outermost evaluation, but not within lambda expressions, is normally referred to as call-by-value and call-by-name evaluation, respectively.

Termination (终止性)

```
inf :: Int
inf = 1 + inf
```

```
     inf
= { applying inf }
     1 + inf
= { applying inf }
     1 + (1 + inf)
= { applying inf }
     1 + (1 + (1 + inf))
= { applying inf }
     …
```

Termination (终止性)

```
inf :: Int
inf = 1 + inf
```

最内策略

```
    fst (0, inf)
= { applying inf }
    fst (0, 1 + inf)
= { applying inf }
    fst (0, 1 + (1 + inf))
= { applying inf }
    fst (0, 1 + (1 + (1 + inf)))
= { applying inf }
      ⋮
```

最外策略

```
    fst (0, inf)
= { applying fst }
    0
```

If there exists any evaluation sequence that terminates for a given expression, then call-by-name evaluation will also terminate for this expression, and produce the same final result.

Number of reductions (需要进行多少次reduction，才能完成求值)

```
square :: Int -> Int
square n = n * n
```

最内策略

```
    square (1+2)
=  { applying + }
    square 3
= { applying square }
    3 * 3
=  { applying * }
    9
```

最外策略

```
    square (1+2)
=  { applying square }
    (1+2) * (1+2)
=  { applying the first + }
    3 * (1+2)
=  { applying + }
    3 * 3
=  { applying * }
    9
```

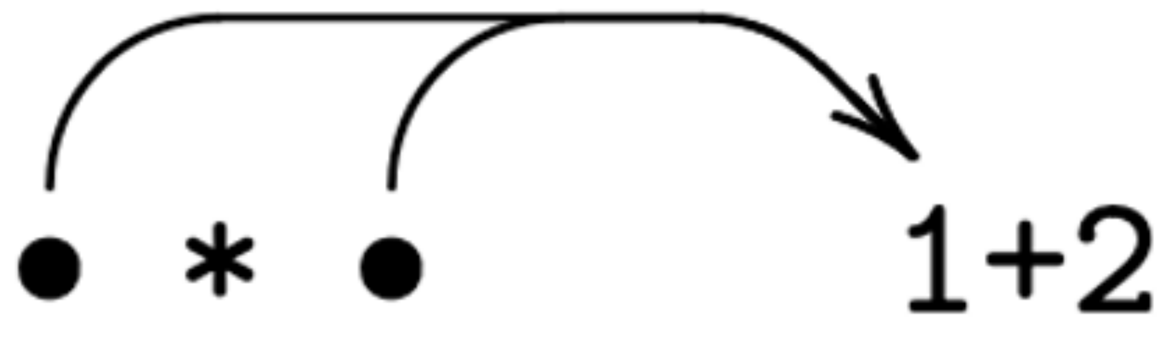
Arguments are evaluated precisely once using call-by-value evaluation, but may be evaluated many times using call-by-name.

Lazy Evaluation / 惰性求值

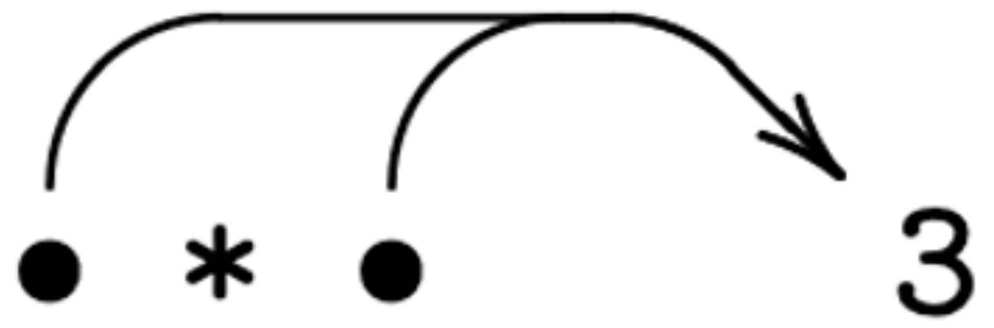
Call-by-name evaluation
in conjunction with Sharing

`square (1+2)`

= { applying `square` }



= { applying + }



= { applying * }

Infinite structures / 无限结构

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

```
    ones
= { applying ones }
    1 : ones
= { applying ones }
    1 : (1 : ones)
= { applying ones }
    1 : (1 : (1 : ones))
= { applying ones }
    ..
```

```
    head ones
= { applying ones }
    head (1 : ones)
= { applying head }
    1
```

Modular Programming 之 将数据和控制分开

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

```haskell
replicate :: Int -> a -> [a]
replicate n = take n  . repeat

repeat :: a -> [a]
repeat x = x : repeat x
```

应用示例：素数序列计算

2	3	4	5	6	7	8	9	10	11	12	13	14	15	···
	3		5	_	7		9		11	_	13		15	···
			5		7			_	11		13		_	···
					7				11		13	_		···
									11		13			···
											13			···

```haskell
primes :: [Int]
primes = sieve [2..]

sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [ x | x <- xs, x `mod` p /= 0 ]
```

Strict application of functions

- Haskell uses lazy evaluation by default, but also provides a special strict version of function application, written as $!
- An expression of the form f $! x is only a redex once evaluation of the argument x, using lazy evaluation as normal, has reached the point where it is known that the result is not an undefined value, at which point the expression can be reduced to the normal application f x

Strict application of functions

- An expression of the form f $! x is only a redex once evaluation of the argument x, using lazy evaluation as normal, has reached the point where it is known that the result is not an undefined value, at which point the expression can be reduced to the normal application f x

```
square $! (1+2)
=      { applying + }
square $! 3
=      { applying $! }
square 3
=      { applying square }
3 * 3
=      { applying * }
9
```

Strict application of functions

- If `f` is a curried function with two arguments, an application of the form `f x y` can be modified to have three different behaviours:

`(f $! x) y` forces top-level evaluation of `x`

`(f x) $! y` forces top-level evaluation of `y`

`(f $! x) $! y` forces top-level evaluation of `x` and `y`

- In Haskell, strict application is mainly used to improve the space performance of programs.

```
sumwith :: Int -> [Int] -> Int
sumwith v []     = v
sumwith v (x:xs) = sumwith (v+x) xs
```

```
    sumwith 0 [1,2,3]
=       { applying sumwith }
    sumwith (0+1) [2,3]
=       { applying sumwith }
    sumwith ((0+1)+2) [3]
=       { applying sumwith }
    sumwith (((0+1)+2)+3) []
=       { applying sumwith }
    ((0+1)+2)+3
=       { applying the first + }
    (1+2)+3
=       { applying the first + }
    3+3
=       { applying + }
    6
```

```
sumwith v []     = v
sumwith v (x:xs) = (sumwith $! (v+x)) xs
```

```
    sumwith 0 [1,2,3]
=       { applying sumwith }
    (sumwith $! (0+1)) [2,3]
=       { applying + }
    (sumwith $! 1) [2,3]
=       { applying $! }
    sumwith 1 [2,3]
=       { applying sumwith }
    (sumwith $! (1+2)) [3]
=       { applying + }
    (sumwith $! 3) [3]
=       { applying $! }
    sumwith 3 [3]
=       { applying sumwith }
    (sumwith $! (3+3)) []
=       { applying + }
    (sumwith $! 6) []
=       { applying $! }
    sumwith 6 []
=       { applying sumwith }
    6
```

Generalising from the above example, the library `Data.Foldable` provides a strict version of the higher-order library function `foldl` that forces evaluation of its accumulator prior to processing the tail of the list:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f v []       = v
foldl' f v (x:xs) = ((foldl' f) $! (f v x)) xs
```

```
sumwith = foldl' (+).
```

- However, strict application is not a silver bullet that automatically improves the space behaviour of Haskell programs.
- Even for relatively simple examples, the use of strict application is a specialist topic that requires careful consideration of the behaviour of lazy evaluation.

作业

15-1 Using a list comprehension, define an expression fibs :: [Integer] that generates the infinite sequence of Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

using the following simple procedure:

- the first two numbers are 0 and 1;
- the next is the sum of the previous two;
- return to the second step.

✳ Hint: make use of the library functions zip and tail.

✳ Note: numbers in the Fibonacci sequence quickly become large, hence the use of the type Integer of arbitrary-precision integers above.

15-2 Newton's method for computing the square root of a (non-negative) floating-point number n can be expressed as follows:

- start with an initial approximation to the result;
- given the current approximation a, the next approximation is defined by the function next a = (a + n/a) / 2;
- repeat the second step until the two most recent approximations are within some desired distance of one another, at which point the most recent value is returned as the result.

Define a function sqroot :: Double -> Double that implements this procedure.

Hint: first produce an infinite list of approximations using the library function iterate. For simplicity, take the number 1.0 as the initial approximation, and 0.00001 as the distance value.

第15章：计算模型—Lazy Evaluation

就到这里吧