

CSc 372

Comparative Programming Languages

13 : Haskell — Lazy Evaluation

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Lazy evaluation

- Haskell evaluates expressions using a technique called **lazy evaluation**:
 - ① No expression is evaluated until its value is needed.
 - ② No shared expression is evaluated more than once; if the expression is ever evaluated then the result is shared between all those places in which it is used.
- Lazy functions are also called **non-strict** and evaluate their arguments **lazily** or **by need**.
- C functions and Java methods are **strict** and evaluate their arguments **eagerly**.

Don't Evaluate Until Necessary

- The first of these ideas is illustrated by the following function:

```
ignoreArgument x = "I didn't evaluate x"
```

- Since the result of the function `ignoreArgument` doesn't depend on the value of its argument `x`, that argument will not be evaluated:

```
$ hugs +s  
> ignoreArgument (1/0)  
I didn't evaluate x  
(246 reductions, 351 cells)
```

Don't Evaluate Until Necessary...

- The function `seq` forces `strict evaluation` when that is necessary:

```
> seq ignoreArgument (1/0)
Inf
(32 reductions, 78 cells)
```

Evaluate Shared Expressions Once

- The second basic idea behind lazy evaluation is that no shared expression should be evaluated more than once.
- For example, the following two expressions can be used to calculate $3 * 3 * 3 * 3$:

```
$ hugs +s
```

```
> square*square where square = 3*3
```

```
81
```

```
(30 reductions, 67 cells)
```

```
> (3*3)*(3*3)
```

```
81
```

```
(34 reductions, 45 cells)
```

Evaluate Shared Expressions Once...

- Notice that the first expression requires fewer reduction than the second.
- A **reduction** is the basic step of evaluating a Haskell expression, by applying a function to its argument.

Saving Reductions

- Consider these sequences of reductions:

```
square * square where square = 3 * 3
-- calculate the value of square by
-- reducing 3*3==>9 and replace each
-- occurrence of square with this result
==> 9 * 9
==> 81
```

```
(3 * 3) * (3 * 3) -- evaluate first (3*3)
==> 9 * (3 * 3) -- evaluate second (3*3)
==> 9 * 9
==> 81
```

- Lazy evaluation means that only the minimum amount of calculation is used to determine the result of an expression.

Taking the Minimum

- Consider the task of finding the smallest element of a list of integers.

```
> minimum [100,99..1]
```

```
1
```

```
(2355 reductions, 3211 cells)
```

- `[100,99..1]` denotes the list of integers from 1 to 100 arranged in decreasing order.
- Instead, we could first sort and then take the head of the result:

```
> :load List
```

```
> sort [100,99..1]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, ..., 99, 100]
```

```
(3430 reductions, 8234 cells)
```


Taking the Minimum...

- However, thanks to lazy evaluation, calculating just the first element of the sorted list actually requires less work in this particular case than the first solution using `minimum`:

```
> head (sort [100,99..1])
```

```
1
```

```
(1877 reductions, 3993 cells)
```

```
> minimum [100,99..1]
```

```
1
```

```
(2355 reductions, 3211 cells)
```

Infinite data structures

- Lazy evaluation makes it possible for functions in Haskell to manipulate 'infinite' data structures.
- The advantage of lazy evaluation is that it allows us to construct infinite objects piece by piece as necessary
- The function `ones` below generates an infinite list of 1s:

```
ones = 1 : ones
```

```
> take 10 ones
```

```
[1,1,1,1,1,1,1,1,1,1]
```

```
(277 reductions, 389 cells)
```

Infinite data structures...

- Consider the following function which can be used to produce infinite lists of integer values:

```
countFrom n = n : countFrom (n+1)
```

```
> countFrom 1
```

```
[1, 2, 3, 4, 5, 6, 7, 8, ^CInterrupted!]
```

Infinite data structures...

- For practical applications, we are usually only interested in using a finite portion of an infinite data structure.
- We can find the sum of the integers 1 to 10:

```
> sum (take 10 (countFrom 1))  
55  
(278 reductions, 440 cells)
```
- `take n xs` evaluates to a list containing the first `n` elements of the list `xs`.

Infinite data structures...

- Infinite data structures enable us to describe an object without being tied to one particular application of that object.
- The following definitions for infinite list of powers of two [1, 2, 4, 8, ...]:

```
powersOfTwo = 1 : map double powersOfTwo
              where double n = 2*n
```

```
> take 10 powersOfTwo
```

```
[1,2,4,8,16,32,64,128,256,512]
```

Infinite data structures...

- `xs !! n` evaluates to the n :th element of the list `xs`.
- We can define a function to find the n th power of 2 for any given integer n :

```
powersOfTwo = 1 : map (*2) powersOfTwo
```

```
twoToThe n = powersOfTwo !! n
```

```
> twoToThe 5
```

```
32
```

Fibonacci

- Here's a definition of a function that generates an infinite list of all the fibonacci numbers:

```
fib = 1:1: [a+b | (a,b) <- zip fib (tail fib)]
```

```
> take 10 fib
```

```
[1,1,2,3,5,8,13,21,34,55]
```

Acknowledgements

- These slides were derived mostly from the Gofer manual.
Functional programming environment, Version 2.20
© Copyright Mark P. Jones 1991.
- We're using hugs here rather than ghci since ghci doesn't have an easy way to show number of reductions.