FP101x - Functional Programming

Programming in Haskell – Lazy Evaluation

Erik Meijer

TU Delft
Delft University of Technology

Challenge the future

Introduction

Up to now, we have not looked in detail at how Haskell expressions are evaluated.

In fact they are evaluated using a simple technique that, among other things:

1. Avoids doing unnecessary evaluation
2. Allows programs to be more modular
3. Allows us to program with infinite lists

The evaluation technique is called lazy evaluation, and Haskell is a lazy functional language.

Evaluating Expressions

Basically, expressions are evaluated or <u>reduced</u> by successively <u>applying definitions</u> until no further simplification is possible.

For example, given the definition:

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

The expression <u>square(3 + 4)</u> can be evaluated using the following sequence of reductions:

Evaluating Expressions

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

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

However, this is not the only possible reduction sequence. Another is the following:

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

Now we have applied square before doing the addition, but the final result is the same.

FACT

In Haskell, two <u>different</u> (but terminating) ways of evaluating <u>the same</u> expression will always give <u>the same</u> final result.

Reduction Strategies

At each stage during evaluation of an expression, there may be many possible subexpressions that can be reduced by applying a definition.

There are two common strategies for deciding which redex (reducible subexpression) to choose:

1. Innermost reduction: An innermost redex is always reduced
2. Outermost reduction: An outermost redex is always reduced

How do the two strategies compare … ?

Termination

Given the definition: `loop = tail loop`

Let's evaluate the expression `fst (1, loop)` using these two reduction strategies.

Innermost reduction

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

This strategy does not terminate.

Outermost reduction

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

This strategy gives a result in one step.

Facts

- Outermost reduction may give a result when innermost reduction fails to terminate.
- For a given expression if there exists any reduction sequence that terminates, then outermost reduction also terminates, with the same result.

Number of reductions

Innermost

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

Outermost

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

The outermost version is inefficient: the subexpression 3 + 4 is duplicated when square is reduced, and so must be reduced twice.

Fact Outermost reduction may require more steps than innermost reduction.

The problem can be solved by using pointers to indicate sharing of expressions during evaluation:

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

This gives a new reduction strategy:

Lazy evaluation = Outermost reduction + sharing

Facts

- Never requires more steps than innermost reduction
- Haskell uses lazy evaluation

Infinite lists

In addition to the termination advantages, using lazy evaluation allows us to program with infinite lists of values!

Consider the recursive definition:

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

Unfolding the recursion a few times gives:

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

That is, ones is the infinite list of 1’s.

Now consider evaluating the expression

```
head ones
```

using <u>innermost reduction</u>

and <u>lazy evaluation</u>.

Innermost reduction

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

In this case, evaluation does not terminate.

Lazy evaluation

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

In this case, evaluation gives the result 1.

That is, using lazy evaluation only the first value in the infinite list ones is actually produced, since this is all that is required to evaluate the expression head ones as a whole.

Lazy evaluation

In general we have the slogan:

Using lazy evaluation, expressions are only evaluated as <u>much as required</u> to produce the final result.

We see now that

```
ones = 1 : ones
```

really only defines a <u>potentially infinite</u> list that is only evaluated as much as required by the context in which it is used.

Modular programming

We can generate **finite** lists by taking elements from infinite lists.

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

Lazy evaluation allows us to make programs more modular, by separating control from data:

$$\underbrace{\texttt{take 5}}_{\texttt{control}}\quad\underbrace{\texttt{[1..]}}_{\texttt{data}}$$

Using lazy evaluation, the data is only evaluated as much as required by the control part.

Example: generating primes

A simple procedure for generating the infinite list of prime numbers is as follows:

1. Write down the list 2, 3, 4, … ;
2. Mark the first value p in the list as prime;
3. Delete all multiples of p from the list;
4. Return to step 2.

Example: generating primes

The first few steps can be pictured by:

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

This procedure is known as the "seive of Eratosthenes", after the Greek mathematician who first described it.

It can be translated directly into Haskell:

```
primes :: [Int]
primes = seive [2..]

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

and executed as follows:

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

By freeing the generation of primes from the constraint of finiteness, we obtain a modular definition on which different boundary conditions can be imposed in different situations:

Selecting the first 10 primes:

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

Selecting the primes less than 15:

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

Lazy evaluation is powerful programming tool!

Fun exercises

Define a program

```
fibs :: [Integer]
```

that generates the infinite Fibonacci sequence

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

using the following simple procedure:

1. The first two numbers are 0 and 1;
2. The next is the sum of the previous two;
3. Return to step 2.

Fun exercises

Define a program

```
fib :: Int -> Integer
```

that calculates the **n**th Fibonacci number.

Happy Hacking!