PROGRAMMING IN HASKELL

Chapter 15 – Lazy Evaluation

Introduction

We haven't looked into how Haskell expressions are evaluated. They are evaluated using a simple technique, that amongst other things:

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

This technique is called lazy evaluation and Haskell is called a lazy functional language.

Evaluating Expressions

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

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

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

However, this is not the only evaluation sequence.

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

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

Reduction Strategies

At each stage during evaluation of an expression there may be <u>many</u> possible subexpressions that can be reduces by applying a definition.

Two common strategies for deciding which Redex (reducible expression) to choose:

1. Innermost reduction: an innermost reduction is always reduced.

2. Outermost reduction: an outermost reduction is always reduced.

Termination

```
loop = tail loop
```

Evaluate the expression `fst (1, loop)` using both evaluation strategies.

1. Innermost reduction

```
fst (1, loop)
= fst (1, tail loop)
= fst (1, tail (tail loop))
= … does not terminate!
```

Termination

2. Outermost reduction

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

- Outermost reduction may give a result when the innermost reduction fails to terminate.

- For a given expression, if there exists any reduction sequence that terminates, then outmost also terminates with the same result.

Number of reductions

Innermost

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

Outmost

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

FACT: Outmost reduction may require more steps than innermost reduction.

Thunks

Outmost reduction is inefficient because (3+4) is duplicated when square is reduced and thus must be reduced twice. Therefore: use <u>sharing</u>.

```
square (3+4)

=

    (● * ●)    (3+4)

=

    (● * ●)      7

= 49
```

Lazy Evaluation

New evaluation strategy:

$\boxed{\text{Lazy Evaluation = Outmost reduction + Sharing}}$

FACTS

- Lazy Evaluation never requires more reduction steps than innermost reduction.

- Haskell uses Lazy Evaluation.

Infinite Lists

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

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

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

Infinite Lists

1. Innermost reduction

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

2. Lazy evaluation

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

Infinite Lists

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

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

Really defines a potentially infinite list that is only evaluated as much as required by the context it is used in.

Modular Programming

We can create finite lists by taking elements from infinite lists. For example:

```
? take 5 ones
[1,1,1,1,1]
```

```
? take 5 [1..]
[1,1,1,1,1]
```

Lazy evaluation allows to make programs more modular by separating control from data.

$$\underbrace{\texttt{take 5}}_{\text{Control}}\quad\underbrace{\texttt{[1..]}}_{\text{Data}}$$

Example: Generating Primes

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

1. Write down the list 2, 3, 4, 5, …
2. Mark the first prime *p* in the list as prime.
3. Delete all multiples of *p* from the list.
4. Return to step 2.

Example: Generating Primes

2 3 4 5 6 7 8 9 10 11 12 …

3 5 7 9 11 …

5 7 11 …

7 11 …

11 …

Named "the sieve of Eratosthenes" after the Greek mathematician who first described it.

Example: Generating Primes

The sieve of Eratosthenes can be translated directly into Haskell:

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

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

```
? primes
= [2,3,5,7,11,13,17,19,23,29,31,37,41,…
```

Example: Generating Primes

By separating the generation of the primes from the constraint of finiteness, we obtain a <u>modular</u> definition on which different <u>boundary</u> <u>conditions</u> can be imposed for different solutions.

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

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

Lazy evaluation is a powerful programming tool!

Exercises

1. 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:

a) The first two numbers are 0 and 1.
b) The next is the sum of the previous two.
c) Return to step a)

Exercises

2. Define a function

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

that calculates the nth Fibonnaci number.