PROGRAMMING IN HASKELL

Chapter 15 – Lazy Evaluation

Introduction

We have not looked at how Haskell expressions are evaluated.

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

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

Evaluating Expressions

Expressions are evaluated or reduced by successively applying definitions until no further simplification is possible.

Recall the Lambda-calculus.

Example 1a

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

One sequence to reduce:

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

Example 1b

square n = n * n

Another sequence to reduce:

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

Orders of function application

Fact: in Haskell, two different (but terminating) ways of evaluating the same expression will always give the same final result.

But.

One may be longer than the other.

Evaluation strategies

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

A reducible expression is called a redex.

Note that not every expression can be reduced (recall our Lambda-calculus problem).

How to choose which order or reduction?

Redex examples

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

mult (1+2,2+3)
has three redexes:
1+2, 2+3, and mult (1+2,2+3)

Alternative first reduction steps are:
mult (3,2+3)
mult (1+2,5)
(1+2) * (2+3)

Evaluation - innermost

The innermost evaluation strategy: evaluate an innermost redex, e.g. a redex that contains no other redex:

mult (1+2,2+3)

1+2 or 2+3 are both innermost. Convention is to evaluate the leftmost first.

Evaluation – pass by value

Another way to think of innermost evaluation is that arguments must be fully evaluated before functions are applied to them – pass by value.

mult (1+2,2+3)

from 1+2 get 3, and from 2+3 get 5; then pass 3 and 5 as arguments to mult.

innermost evaluation is referred to as call-by-value.

Evaluation - outermost

The outermost evaluation strategy: evaluate an outermost redex, e.g. a redex that is contained in no other redex:

mult (1+2,2+3)

mult is outermost.

Functions are applied before their arguments are evaluated.

Evaluation – pass by name

The arguments of the function are fed in before they are evaluated – pass by name.

mult (1+2,2+3)

(1+2) * (2+3)

outermost evaluation is referred to as call-by-name.

Comparing strategies

We earlier saw some differences between the strategies.

Will compare them with respect to the properties of termination and number of reduction steps each requires.

These are relevant for the treatment of infinity in a programming language.

Termination

Consider:

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

inf abbreviates infinity, since it is defined as the recursive successor of itself.

1 + (1 + (1 + inf))....

Termination

Apply fst to a pair with inf, where fst outputs the first member of a pair: fst (x,y) = x

fst (0, inf) with the call-by-value evaluation gives an infinite sequence. No output:

fst (0, 1 + (1 + (1 + inf))....

The call-by-name evaluation gives a result:

fst (0, inf) = 0

Termination

The call-by-name evaluation may produce a result when call-by-value evaluation fails to terminate.

In general: 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 result.

call-by-name evaluation is preferable to call-by-value to ensure that evaluation terminates as often as possible.

Number of reductions

Consider again:

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

call-by-value evaluation

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

Note that the expression (3+4) is evaluated once.

call-by-name evaluation

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

(3+4) is evaluated twice.

Numbers of evaluations

arguments evaluated once using call-by-value.

arguments can be evaluated multiple times using call-by-name.

Seems call-by-name is less efficient.

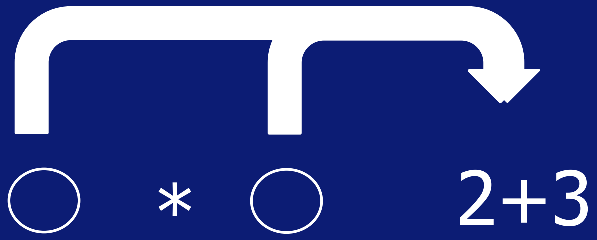
However, this efficiency problem can be resolved using pointers.

Pointers

Keep one ‘original’ copy of the argument; where the argument is used in a function, point to the original, evaluate, and copy the result.

square (3+4)

=



Lazy evaluation

call-by-name evaluation together with sharing is known as lazy evaluation.

Haskell is a lazy programming language.

call-by-name ensures that evaluation terminates as often as possible.

sharing ensures that lazy evaluation never requires more steps than call-by-value.

Infinite structures

call-by-name evaluation (and lazy evaluation) allows us to program with infinite structures. We saw this earlier.

More interesting are infinite lists.

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

This list recurses and adds 1s to the list infinitely. No strategy helps.

Infinite structures

The function head selects the first element of a list.
call-by-value does not terminate:

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

call-by-name (lazy evaluation) does terminate:

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

Infinite structures

call-by-name (lazy evaluation) does terminate:

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

Applying head to (1 : ones) selects 1 and ignores ones.

Lazy evaluation – only evaluate expressions as much as required by the context in which they are used.

Infinite structures

Example 1:

```
-- All natural numbers
    nn = [1..]
```

Example 2:

```
-- The squares of all natural numbers

    nnSquares = map square nn
    where square x = x * x
```

Working infinite structures

An infinite list (as earlier) is only potentially infinite. We only work with as much as we might need.

- take 10 [1..]

  [1,2,3,4,5,6,7,8,9,10]
- take 5 nnSquares

  [1, 4, 9, 16, 25]
- filter (<20) nnSquares

  [1, 4, 9, 16]
- filter (<20) (filter (>10) nnSquares)

  [16]

Exercise Fibonacci numbers

Using a list comprehension, define an expression

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

that generates the infinite 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.

Use zip and tail.