

Comp 205: Comparative Programming Languages

Lazy Evaluation

- Errors
- Evaluation Strategies
- Infinite Lists....

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Errors

Errors are reported when the Haskell interpreter evaluates an illegal expression:

```
Main> 3/0
program error: {primDivDouble 3.0 0.0}
Main> head (tail [2])
program error: {head []}
Main>
```

Reporting Errors

Errors can be raised by the programmer.

Consider the following function to compute the greatest number in a non-empty list:

```
maxList [] = error "empty argument list"
maxList (n : []) = n
maxList (n : ns) = max n (maxList ns)
```

Haskell is Lazy

Haskell only evaluates a subexpression if it's necessary to produce a result.

This is called lazy (or non-strict) evaluation

```
Main> maxList []
program error: empty argument list
Main> fst (0, maxList [])
0
Main>
```

Patterns Force Evaluation

Haskell will evaluate a subexpression to test if it matches a pattern. Suppose we define:

```
myFst (x, 0) = x
myFst (x, y) = x
```

Then the second argument is always evaluated:

```
Main> myFst (0, maxList [])
program error: empty argument list
Main>
```

Lazy But Productive

Haskell will produce as much of a result as possible:

```
Main> [1, 2, div 3 0, 4]
[1,2,
program error: {primQrmInteger 3 0}
Main> map (1/) [1, 2, 0, 7]
[1.0,0.5,
program error: {primDivDouble 1.0 0.0}
```

Lazy Evaluation

Lazy evaluation: a subexpression is evaluated only if it is necessary to produce a result.

The Haskell interpreter implements **topmost-outermost** evaluation:

Rewriting is done as near the "top" of the parse tree as possible.

For example:

```
reverse (1 : (2 : []))
```

Topmost-Outermost

```
reverse (n : ns) = swap n (reverse ns)
swap h r1 = r1 ++ [h]
```

```
reverse (1 : (2 : []))
=> (swap 1) (reverse (2 : []))
=> (reverse (2 : [])) ++ [1]
=> ((swap 2) (reverse [])) ++ [1]
=> ((reverse []) ++ [2]) ++ [1]
```

Topmost-Outermost

```
((reverse []) ++ [2]) ++ [1]
=> ([] ++ [2]) ++ [1]
=> [2] ++ [1]
=> [2,1]
```

Infinite Lists

Haskell has a "dot-dot" notation for lists:

```
Main> [0..7]
[0,1,2,3,4,5,6,7]
```

The upper bound can be omitted:

```
Main> [1..]
[1,2,3,4,5,6,7, ...
...
2918,2919,291<<not enough heap space --
task abandoned>>
```

Using Infinite Lists

Haskell gives up displaying a list when it runs out of memory, but infinite lists like `[1..]` can be used in programs that only use a part of the list:

```
Main> head (tail (tail (tail [1..])))
4
```

This style of programming is often summarised by the phrase "generators and selectors"

- `[1..]` is a generator
- `head.tail.tail.tail` is a selector

Generators and Selectors

Because Haskell implements lazy evaluation, it only evaluates as much of the generator as is necessary:

```
Main> head (tail (tail (tail [1..])))
5
Main> reverse [1..]
ERROR - Garbage collection fails to
reclaim sufficient space
Main>
```

A Selector

The built-in function `filter` removes elements in a list that don't satisfy a property `p`:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x : xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```

```
Main> filter (<10) [1, 11, 2, 13, 3]
[1,2,3]
```

Another Selector

The built-in function `takeWhile` returns the longest initial segment that satisfies a property `p`:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
  | p x      = x : takeWhile p xs
  | otherwise = []
```

```
Main> takeWhile (<10) [1, 2, 13, 3]
[1,2]
```

Selectors

Note that evaluation of `takeWhile` stops as soon as the given property doesn't hold, whereas evaluation of `filter` only stops when the end of the list is reached:

```
Main> takeWhile (<10) [1..]
[1,2,3,4,5,6,7,8,9]
Main> filter (<10) [1..]
[1,2,3,4,5,6,7,8,9]
```

Summary

- Lazy evaluation: only evaluate what's needed
- implemented by topmost-outermost evaluation
- Haskell allows infinite lists
- lazy evaluation allows using (selecting) parts of infinite (generator) lists

Next: more infinite lists