Evaluation Order and Laziness

COMP2209 - Programming III

Dr Julian Rathke
Evaluation and Reduction

• When we consider a program in a given programming language we naturally think of “what the program does” or “what value it produces”

• Programs are expressions of a language and we call the latter point evaluation - that is, what value an expression produces under computation.

• In order to make sense of evaluation we tend to think of computation as proceeding in steps - it is clear why when one considers evaluation on a Von Neumann machine with a fetch-execute cycle.

• We often refer to an individual step of evaluation as a reduction

• This reflects that an expression is simplified, or reduced during evaluation until it becomes a value
Reducible Expressions

• A key concept in understanding evaluation is that of a reducible expression.
  • We use the shorthand name redex for this
• In Haskell a redex is an expression comprising a function applied to an argument expression.
• For example, \texttt{not (x==0)} is a redex because we have the function not applied to the expression \texttt{x==0}
• Whereas \texttt{(sqrt . negate) 45} is not itself a redex as the \texttt{lhs} needs to be evaluated to a function first.
• It does however contain the redex \texttt{(sqrt \ . \)} where \texttt{.} is the infix composition operator
• This tells us that redexes may appear nested within other expressions, and indeed inside other redexes
Example Redexes

• There are two redexes in

\[(1 + 2) \times (3 + 4)\]

• There are three redexes in

\[\text{not (not True)}\]

• This is easier to see if we write

\[\text{mult (add 1 2) (add 3 4)}\]
Evaluation Strategy

• Given that there may be multiple redexes in any given expression, how do we know which one to reduce first?
• The rules that determine this are what is known as an evaluation strategy.
• For any expression we call a redex innermost if it contains no other redex.
• For any expression we call a redex outermost if it is not contained in any redex.
• For example, in mult (add 1 2) (add 3 4) then
  • the (add 1) and (add 3) expressions are innermost redexes
  • and the mult (add 1 2) expression is an outermost redex
• This leads to two different evaluation strategies - innermost evaluation, in which the innermost redexes are evaluated first and outermost evaluation in which the outermost redexes are evaluated first.
Evaluation Strategy

• We have distinguished innermost from outermost evaluation but that is not enough to determine a strategy:

• Consider the expression and its innermost redexes

\[(1 + 2) \times (3 + 4)\]

• Yes, there are two of them!

• Which do we evaluate first?

• We also need to specify leftmost or rightmost also when it comes to giving an evaluation strategy.

• For example, Haskell uses outermost leftmost evaluation.

• Rightmost is not commonly used at all so the left/right distinction is often omitted.
Reduction and Lambda

- Recall that we can use λ notation for functions.
- This looks like $\lambda x \rightarrow e$
- Where $x$ is a variable that is *bound* in the expression $e$
- To reduce the redex $(\lambda x \rightarrow e_1) \ e_2$ we must replace every occurrence of the variable $x$ inside $e_1$ with $e_2$.
- Doing this is called **beta-reduction**
- One has to be careful with naming of variables to do this properly but we’ll just assume that all bound variable names are distinct for now.
- For example,

\[
\text{mult} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{mult} = \lambda x \rightarrow \lambda y \rightarrow x \ast y
\]

- $(\text{mult} \ 3)$ beta-reduces to $\lambda y \rightarrow 3 \ast y$
Reduction under lambda

- The expression $\lambda x \to (\text{add } 5 \ x)$ has a redex:
  - It is the (add 5) subexpression.
- The evaluation strategy for most programming languages, including Haskell, do not select such redexes for reduction.
- Any redex that appears within the scope of a $\lambda$ expression is not reduced.
- This has the effect that functions in $\lambda$ form are actually terminated values.
  - So $\lambda x \to (1 + 2)$ is actually a fully evaluated value of Haskell.
  - It is a “suspended” or “thunked” computation.
  - Were this function to be applied to a value, it would be “unthunked” and the (1+2) computation would occur.
Call-by-Name, Call-by-Value

- We have specific names for the evaluation strategies that maintain the “no reduction under lambdas” rule.

- (leftmost) innermost reduction with no reduction under lambda is more commonly known as **call-by-value** reduction.

- (leftmost) outermost reduction with no reduction under lambda is more commonly known as **call-by-name** reduction.

- Most mainstream languages, such as C, C++, Java, Scheme, OCaml, Python, Javascript etc use call-by-value reduction.

- The best known language that uses call-by-name reduction is in fact, Haskell.

- Let’s have a look at how the different choice of strategy affects programs.
(Non-)Termination

- We use recursion an awful lot in functional programming.
- We can easily write recursive expressions that simply do not terminate when we try to evaluate them.
  - e.g. \( \text{inf} = 1 + \text{inf} \) is valid Haskell but does not ever evaluate to a value:

Consider: \( \text{fst} (0, \text{inf}) \)

- Let’s reduce this using call-by-value

\[
\begin{align*}
\text{fst} (0, \text{inf}) &= \text{fst} (0, 1 + \text{inf}) \\
&= \text{fst} (0, 1 + (1 + \text{inf})) \\
&= \text{fst} (0, 1 + (1 + (1 + \text{inf}))) \\
&\quad \quad \vdots
\end{align*}
\]

This is the innermost redex

- And, using call-by-name

\[
\begin{align*}
\text{fst} (0, \text{inf}) &= 0
\end{align*}
\]

This is the outermost redex
Efficiency

• Remember the replicate function?

\[
\text{replicate 10 } x = x \cdot \text{replicate 10 } x
\]

• Think about what happens if we call \( \text{replicate 10 } (\pi^2) \)?
• Using a call-by-value strategy this would produce a list of 10 float values of (approx) 9.896
• Using a call-by-name strategy instead would produce a list of 10 unevaluated expressions \( \pi^2 \)
  • This would mean we potentially calculate \( \pi^2 \) ten times instead of a single time using call-by-name!
• This would suggest call-by-name is not a good idea, however it works well for avoiding non-termination.
Lazy Evaluation

- Haskell actually avoids the problem of repeated evaluation in the example on the previous slide.
- It does this by using a technique called **graph reduction**
- Using this, instead of copying an expression multiple times as part of beta-reduction, a single copy of the expression is kept and pointers to this expression are passed to the function body instead.
- Any reductions performed on the single copy of the expression are therefore shared between all instances of it in the function body.
- The combination of call-by-name reduction and graph reduction is referred to as **lazy evaluation**
- This is precisely the evaluation strategy in Haskell
Why Lazy?

- Consider the infinite list defined by \( \text{ones} = 1 : \text{ones} \)
- Does evaluation of this expression terminate?
- Yes, sort of - it depends what you do with it.
- For example, the expression \((\text{head ones})\) quite happily returns the value 1.
  
  \[
  \text{head (x:_)} = x
  \]
- \text{head} doesn’t need to evaluate anything but the first value of the argument list in order to reduce. It can lazily just return as soon as it has done enough computation.
- But if we type “ones” in to GHCi we will see an infinite list of 1s until we terminate it manually.
  
  - This is because GHCi consumes the list by calling \text{show} on the list, which in turn maps \text{show} across every element of the list.
  - In order to \text{show} each value, Haskell \text{needs} to compute each value
Modularity through Laziness

• Laziness actually allows a rather neat modular style of programming

• Consider the difference between the following two programs:

```haskell
ones = 1 : ones

take 0 _ = []
take _ [] = []
take n (x:xs) =
  x : take (n-1) xs

take 300 ones
```

```haskell
replicate 0 _ = []
replicate n x =
  x : replicate (n-1) x

replicate 300 1
```

The left hand code separates the production of data (ones) and control (take). The right hand code blends them together.

But we could re-use the take function for different purposes - Modularity!
Example

• The function `takeWhile :: (a → Bool) → [a] → [a]` accepts a predicate and a list and returns the longest prefix of the list where each element satisfies the predicate.

• This is really useful when combined lazily with functions that produce very long or infinite lists.

• Let’s use laziness to produce a list of prime numbers up to a given limit

• First we need to define a list of primes
  • This is best lazily defined as an infinite list
  • We can use the well-known Sieve of Eratosthenes here.
Sieve of Eratosthenes

1. Write down the infinite sequence 2, 3, 4, 5, 6, 7, ....
2. Mark the first number, p in the sequence as prime
3. Delete all multiples of p from the sequence
4. Repeat from step 2

\[
\text{sieve} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{sieve} \ (p : \ xs) = \\
\quad p : \text{sieve} \ [ x \mid x \leftarrow \ xs, x \ `\mod` \ p \ /= \ 0 ]
\]
\[
\text{primes} = \text{sieve} \ [2..]
\]
Using the Sieve

• Having lazily defined an infinite list of prime numbers we can make use of it.

• Clearly, just calling “primes” in GHCi will cause all elements of the list to be displayed - this is non-terminating.

• But we can just take some initial prefix of them
  • e.g. take 100 primes - returns the first 100 primes
  • takeWhile (< 10000) primes - returns all primes that are less than 10000.

• We don’t know in advance how many of these there will be so take is not appropriate there.

• The generation of the primes is separated from the check of whether to return the prime (i.e. < 10000)
Strict Application

• Remember when we discussed tail recursion and how it didn’t obviously help in Haskell due to lazy evaluation?

• What do you mean “no”? Sure you do:

\[
\text{sum'} \ acc \ [\ ] = acc \\
\text{sum'} \ acc \ (x : xs) = \text{sum'} \ (acc+x) \ xs
\]

\[
\begin{align*}
\text{sum'} \ 0 \ [1,2,3] \\
&= \text{sum'} \ (0+1) \ [2,3] \\
&= \text{sum'} \ ((0+1)+2) \ [3] \\
&= \text{sum'} \ (((0+1)+2)+3) \ [] \\
&= ((0+1)+2)+3
\end{align*}
\]

For large lists, this calculation will use a lot of heap space - it stores the pending computation.

In call-by-value languages this is not the case - the argument to \text{sum'} is evaluated each time

Sometimes you might just want non-lazy, or \textbf{strict}, evaluation
**Strict Application**

- Haskell provides a special application operator ($!) that provides a strict evaluation order.

- It behaves much like the lazy application ($) but enforces that the argument is fully evaluated before being passed to the function.

- For example, to reduce the expression - square $! (1 + 2) we evaluate the expression (1+2) first and to become the lazy application - square $ 3

- We can control evaluation very well using this

  \[
  (f \, $! \, x) \, y \quad - \text{forces top-level evaluation of } x \\
  (f \, x) \, $! \, y \quad - \text{forces top-level evaluation of } y \\
  (f \, $! \, x) \, $! \, y \quad - \text{forces top-level evaluation of } x \text{ and } y
  \]

and

\[
\text{sum'} \, \text{acc} \, [\,] = \text{acc} \\
\text{sum'} \, \text{acc} \, (x : \, xs) = (\text{sum'} \, \, $! \, (\text{acc+}x)) \, \, xs
\]

is way more space efficient than the lazy version!
YOUR QUESTIONS

Next Lecture:
Writing Interpreters