Lazy Evaluation

Haskell uses \textit{Lazy (non-Strict) Evaluation}:
\begin{itemize}
  \item Expressions are only evaluated \textit{when} their value is needed
  \item In particular, argument expressions are not evaluated before a function is applied
  \item We find this approach allows us to write sensible programs not possible if strict-evaluation is used.
  \item However, it comes at a price . . .
\end{itemize}

Example: \texttt{isOdd}
\begin{itemize}
  \item We define a function checking for ‘oddness’ as follows:
    \[
    \text{isOdd } n \equiv n \mod 2 \equiv 1
    \]
  \item Consider the call \texttt{isOdd (1+2)}
  \item A strict (non-lazy) evaluation would be as follows:
    \[
    \text{isOdd (1+2)} \equiv \text{isOdd 3} \\
    \equiv 3 \mod 2 \equiv 1 \\
    \equiv 1 \equiv 1 \\
    \equiv \text{True}
    \]
  \item A non-strict (lazy) evaluation would be as follows:
    \[
    \text{isOdd (1+2)} \equiv (1+2) \mod 2 \equiv 1 \\
    \equiv 3 \mod 2 \equiv 1 \\
    \equiv 1 \equiv 1 \\
    \equiv \text{True}
    \]
  \end{itemize}

\texttt{1+2} is only evaluated \textit{when \texttt{mod}} needs its value to proceed.

len and down
\begin{itemize}
  \item We have a length function \texttt{len}:
    \[
    \text{len } xs \equiv \text{if } \text{null } xs \text{ then } 0 \text{ else } 1 + \text{len (tail } xs)\]
  \item We have a function \texttt{down} that generates a list, counting down from its numeric argument:
    \[
    \text{down } n \equiv \text{if } n \leq 0 \text{ then } [] \text{ else } n : (\text{down } (n-1))
    \]
    \text{For example, } \text{down 3 } = [3,2,1]
  \item We shall consider pattern matching versions shortly
\end{itemize}
Strict evaluation of \textit{len (down 1)}

\[
\begin{align*}
\text{len (down 1)} &= \text{len (if 1 \leq 0 the [ ] else 1 : (down (1-1)))} \\
&= \text{len (1 : (down (1-1)))} \\
&= \text{len (1 : (down 0))} \\
&= \text{len (1 : (if 0 \leq 0 the [ ] else 0 : (down (0-1))))) } \\
&= \text{len (1 : [ ])} \\
&= \text{if null (1 : []) then 0 else 1 + len (tail (1 : []))} \\
&= 1 + \text{len (tail (1 : []))} \\
&= 1 + \text{len []} \\
&= 1 + \text{len (if null [] then 0 else 1 + len (tail []))} \\
&= 1 + 0 \\
&= 1
\end{align*}
\]

We have 11 steps

Lazy evaluation of \textit{len (down 1)} (part 1)

\[
\begin{align*}
\text{len (down 1)} &= \text{if null \textit{xs}_1 then 0 else 1 + len (tail \textit{xs}_1)} \\
&\quad \text{where \textit{xs}_1 = \text{down 1}} \\
&= \text{if null \textit{xs}_1 then 0 else 1 + len (tail \textit{xs}_1)} \\
&\quad \text{where \textit{xs}_1 = \text{if 1 \leq 0 then [ ] else 1 : (down (1-1))}} \\
&= \text{if null \textit{xs}_1 then 0 else 1 + len (tail \textit{xs}_1)} \\
&\quad \text{where \textit{xs}_1 = 1 : (down (1-1))} \\
&= 1 + \text{len (tail \textit{xs}_1)} \quad \text{where \textit{xs}_1 = 1 : (down (1-1))} \\
&= 1 + \text{len (tail \textit{xs}_1)} \quad \text{where \textit{xs}_1 = \text{if null \textit{xs}_1 then 0 else 1 + len (tail \textit{xs}_1)}} \\
&\quad \text{where \textit{xs}_1 = \text{tail \textit{xs}_1}} \\
&= 1 + \text{len (tail \textit{xs}_1)} \quad \text{where \textit{xs}_1 = \text{if 0 \leq 0 then [ ] else 1 : (down (1-1))}} \\
&\quad \text{where \textit{xs}_1 = \text{if null \textit{xs}_2 then 0 else 1 + len (tail \textit{xs}_2)}} \\
&\quad \text{where \textit{xs}_2 = \text{tail \textit{xs}_1}} \\
&= 1 + \text{len (tail \textit{xs}_1)} \quad \text{where \textit{xs}_2 = [ ]} \\
&= 1 + 0 \\
&= 1
\end{align*}
\]

(continued overleaf)

Lazy evaluation of \textit{len (down 1)} (part 2)

\[
\begin{align*}
1 + ( \text{if null \textit{xs}_2 then 0 else 1 + len (tail \textit{xs}_2)} \\
&\quad \text{where \textit{xs}_2 = \text{tail \textit{xs}_1}} \\
&\quad \text{where \textit{xs}_1 = 1 : (down (1-1))} \\
&= 1 + ( \text{if null \textit{xs}_2 then 0 else 1 + len (tail \textit{xs}_2)} \\
&\quad \text{where \textit{xs}_2 = (if (1-1) \leq 0} \\
&\quad \quad \text{then [ ] else (1-1) : (down ((1-1)-1)))}) \\
&= 1 + ( \text{if null \textit{xs}_2 then 0 else 1 + len (tail \textit{xs}_2)} \\
&\quad \text{where \textit{xs}_2 = (if 0 \leq 0} \\
&\quad \quad \text{then [ ] else (1-1) : (down ((1-1)-1)))}) \\
&= 1 + ( \text{if null \textit{xs}_2 then 0 else 1 + len (tail \textit{xs}_2)} \\
&\quad \text{where \textit{xs}_2 = [ ]} \\
&= 1 + 0 \\
&= 1
\end{align*}
\]

We have only 10 steps (?), but each is more expensive (?).

Lazy evaluation of \textit{len (down 1)} (part 2)

Why the \textit{xs}_1 = ... ?

\begin{itemize}
\item Consider the first step:
\[
\text{len (down 1)} = \text{if null \textit{xs}_1 then 0 else 1 + len (tail \textit{xs}_1)} \\
&\quad \text{where \textit{xs}_1 = \text{down 1}}
\]
\item We don’t evaluate \textit{down 1} — we bind it to formal parameter \textit{xs}.
\item Parameter \textit{xs} occurs twice, but we don’t copy:
\[
\ldots \text{down 1} \ldots \text{down 1} \ldots
\]

Instead we share the reference, indicated by the \textit{where} clause:
\[
\ldots \text{xs} \ldots \text{xs} \ldots \text{where \textit{xs} = down 1}
\]
\item Function \textit{len} is recursive, so we get different instances of \textit{xs} which we label as \textit{xs}_1, \textit{xs}_2, ...
\item The grouping of an expression (\textit{down 1}) with a binding (\textit{xs}_1 = \text{down 1}) is called either a “closure”, or a “thunk”.
\item Building thunks is a necessary overhead for implementing lazy evaluation.
\end{itemize}
Lazy Evaluation: the costs

- Lazy evaluation has an overhead: building thunks
- Memory consumption per reduction step is typically slightly higher
- In our examples so far:
  `isOdd (1+2)`
  `len (down 1)` we needed to evaluate almost everything
- So far we have observed no advantage to lazy evaluation . . .

Advantages of Laziness (I)

- Imagine we have a function definition as follows:
  ```
  myfun carg struct1 struct2 = if f carg
  then g struct1
  else h struct2
  ```
  where `f`, `g` and `h` are internal functions
- Consider the following call:
  ```
  myfun val s1Expr s2Expr
  ```
  where both `s1Expr` and `s2Expr` are very expensive to evaluate.
- With strict evaluation we would have to compute both before applying `myfun`
- With lazy evaluation we evaluate `f val`, and then only evaluate one of either `s1Expr` or `s2Expr`, and then, only if `g` or `h` requires its value.

Laziness and Pattern Matching

- How does laziness interact with pattern matching?
- Consider a pattern matching version of `len`
  ```
  len [] = 0
  len (x:xs) = 1 + len xs
  ```
- How is call `len aListExpression` evaluated?
- In order to pattern match we need to know if `aListExpression` is empty, or a cons-node.
- We evaluate `aListExpression`, but only to the point were we know this difference
  If it is not null, we do not evaluate the head element, or the tail list.
- e.g. if `aListExpression = map (+1) (1:2:[])`, then we only evaluate as far as `((+1) 1) : (map (+) (2:[]))`

Advantages of Laziness (II)

- Prelude function `take n xs` returns the first `n` elements of `xs`
  ```
  take n [] = []
  take 0 xs = []
  take n (x:xs) = x : (take (n-1) xs)
  ```
- Function `from n` generates an infinite ascending list starting with `n`
  ```
  from n = n : (from (n+1))
  ```
- Evaluating `from n` will fail to terminate for any `n`.
- Evaluation of `take 2 (from 0)` depends on the evaluation method.
Strict Evaluation of `take 2 (from 0)`

```
take 2 (from 0)
= take 2 (0 : from 1)
= take 2 (0 : 1 : from 2)
= take 2 (0 : 1 : 2: from 3)
= take 2 (0 : 1 : 2: 3 : from 4)
= take 2 (0 : 1 : 2: 3 : 4 : from 5)
```

(You get the idea …)

Lazy Evaluation of `take 2 (from 0)`

```
take 2 (from 0)
= take 2 (0 : from 1)
= 0 : (take 1 (from 1))
= 0 : (1 : take 0 (from 1))
= 0 : (1 : [])
```

We are done! We only built the bit of `from 0` that we actually needed.

---

Evaluation Strategy and Termination

We can summarise the relationship between evaluation strategy and termination as:

- There are programs that simply do not terminate, no matter how they are evaluated
  e.g. `from 0`
- There are programs that terminate if evaluated lazily, but fail to terminate if evaluated strictly
  e.g. `take 2 (from 0)`
- There are programs that terminate regardless of chosen evaluation strategy
  e.g. `len (down 1)`
- However, there are no programs that terminate if evaluated strictly, but fail to terminate if evaluated lazily.

Laziness Example

- Consider Fibonacci’s sequence: \( f_1, f_2, f_3, f_4, \ldots \) where
  - \( f_1 = 1 \)
  - \( f_2 = 2 \)
  - \( f_{n+2} = f_n + f_{n+1} \)
- An encoding in Haskell without recursion:
  ```haskell```
  ```
  fib = 1 : 2 : zipWith (+) fib (tail fib)
  ```
  ```haskell```
- Just as efficient as a recursive version
  ```haskell```
  ```
  fibr n = fibber 1 2
  fibber m n = m : fibber n (n+m)
  ```
Strictness Analysis

▶ If a program requires everything to be evaluated, we have seen that strict evaluation is faster (no thunk overhead).
▶ In general, if an application \( f \ a \) always evaluates \( a \), then it is more efficient to use strict-evaluation to reduce it.
▶ Compilers for lazy languages often perform strictness analysis to detect cases were such a optimisation is possible.
▶ However complete strictness analysis is undecidable.
▶ So Haskell allows programmers to annotate arguments as strict. \( f \ ! a \) applies \( f \) to \( a \), but after forcing the evaluation of \( a \).
▶ Here \( ! \) is an right-associative infix operator:

\[
(\!)(:: ( a \rightarrow b) \rightarrow a \rightarrow b)
\]

Strictness Example

▶ \texttt{foldr} (Lazy)

\[
\begin{align*}
\text{foldr } & (z \ [\ ] = z \\
\text{foldr } & f z (x:xs) = f x (\text{foldr } f z xs)
\end{align*}
\]

▶ \texttt{foldl} (Lazy)

\[
\begin{align*}
\text{foldl } & (z \ [\ ] = z \\
\text{foldl } & f z (x:xs) = \text{foldl } f (f z x) xs
\end{align*}
\]

▶ \texttt{foldl'} (Strict)

\[
\begin{align*}
\text{foldl} & (z \ [\ ] = z \\
\text{foldl} & f z (x:xs) = \text{foldl} f (f z x) xs
\end{align*}
\]

▶ Lets try them with arguments \( (+) 0 \ [1,2,3] \)

Aside: the usefulness of \( ! \)

▶ \( (\! \) (a.k.a. “apply”) is the lazy (normal) version of \( ! \)
  ▶ It is a right-associative infix operator

\[
\begin{align*}
\text{f } & \! a = f a \quad \text{-- definition} \\
\text{f } & \! g \! h \! x = f \! (g \! (h \! x))
\end{align*}
\]

What is it good for?

▶ Consider a complex nested function application:

\[
f \ a \ (g \ b \ (h \ (k \ c \ d \ x)))
\]

▶ We could use function composition:

\[
(f \ a \ . \ g \ b \ . \ h \ . \ k \ c \ d \ x)
\]

where \( (f \ . \ g) x = f (g x) \)

▶ The \( ! \) notation allows us to drop all brackets:

\[
f \ a \ ! g \ b \ ! h \ ! k \ c \ d \ x
\]