Laziness

**EDAF40/EDAN40: Functional Programming**
**On Laziness**

Jacek Malec  
Dept. of Computer Science, Lund University, Sweden  
April 15th, 2019

References:
- [https://www.haskell.org/haskellwiki/Haskell/Lazy_evaluation](https://www.haskell.org/haskellwiki/Haskell/Lazy_evaluation)
- [https://www.haskell.org/haskellwiki/Lazy_vs._non-strict](https://www.haskell.org/haskellwiki/Lazy_vs._non-strict)

---

Laziness

Topic of today: non-strictness vs. laziness

Strict vs. non-strict

- Property of the *semantics* of the language
- Related to *reductions* (evaluations) of expressions
  - top-down
  - bottom-up
- If something evaluates to *bottom* (an error or endless loop) then
  - strict languages will always find the bottom value
  - non-strict languages not need to!

Property of an implementation!
Evaluate an expression only when its value is needed.
Common implementation technique for non-strict languages.
Not generally amenable to parallelisation.

Alternative: *lenient* (or optimistic) evaluation; somewhere between lazy and strict — more promising for parallelisation.

Laziness

Normal forms

- NF (RNF) – normal form (reduced normal form)
- HNF – head normal form
- WHNF – weak head normal form

Describe the amount of evaluation performed:
- NF – evaluated
- WHNF – evaluated only up to the outermost constructor


Laziness again

Haskell is not completely lazy!

E.g. pattern matching (a very common situation in any non-trivial piece of code) drives evaluation

Consequences of laziness

- purity
- space leaks
- short-circuiting operators by default
- infinite data structures
- efficient pipelining
- dynamic programming “for free” (Assignment N2)
**Space leaks**

- Space leaks (or foldl vs. foldl')

  \[\text{foldl} \ (+) \ 0 \ (1:2:3:[])\]
  \[= \text{foldl} \ (+) \ (0 + 1) \ (2:3:[])\]
  \[= \text{foldl} \ (+) \ ((0 + 1) + 2) \ (3:[])\]
  \[= \text{foldl} \ (+) \ (((0 + 1) + 2) + 3) \ []\]
  \[= (((0 + 1) + 2) + 3)\]

*Thunks* stored until needed.

How can we force evaluation?

\[
\text{foldl}' \ f \ a \ [] \ = \ a \\
\text{foldl}' \ f \ a \ (x:xs) \ = \ \text{let} \ a' = f \ a \ x \\
\quad \text{in} \ a' \ \text{`seq' foldl}' \ f \ a' \ xs
\]

**Strict application**

Haskell offers two application operators:

- \$ \ -- \ ordinary
- \$! \ -- \ strict

with the following effect:

\[
\text{square} \ x = x \ast x
\]

\[
\text{square} \ (1 + 2) \ -- \ evaluates \ (+) \ twice
\]

\[
\text{square} \ $! \ (1 + 2) \ -- \ only \ once!
\]

**Short-circuiting**

We get three possibilities for two arguments of a curried function:

- \((f \ $! \ x) \ y\) \ -- \ forces \ evaluation \ of \ \(x\)
- \((f \ x) \ $! \ y\) \ -- \ forces \ evaluation \ of \ \(y\)
- \((f \ $! \ x) \ $! \ y\) \ -- \ forces \ evaluation \ of \ both \ \(x\) \ and \ \(y\)

Strict evaluation saves space (sometimes)

- In strict languages: a special-case mechanism wired into language standards
- In lazy languages: the default

\[
\text{(&&)} :: \text{Bool} \to \text{Bool} \to \text{Bool}
\]

\[
\text{True} \ \&\& \ x = x \\
\text{False} \ \&\& \ _ = \text{False}
\]