Functional Programming

WS 2019/20

Torsten Grust
University of Tübingen

Lazy Evaluation

To execute a program, *Haskell reduces expressions to values.*
Haskell uses **normal order reduction** to select the next expression
to reduce:

- The *outermost* reducible expression (the so-called *head redex*)
  is reduced first.
  $\Rightarrow$ Function applications are reduced *before* their arguments.
- If no further redex is found, the expression is in **normal form**
  and reduction terminates.

(Reducing a function application *f e*: replace application by body
of *f* in which the formal parameter is replaced by argument *e*.)

Lazy Evaluation

Example for normal order (≡ outermost redex first) reduction:

```
fst :: (a,b) -> a
fst (x,y) = x

sqr :: Num a => a -> a
sqr x = x * x

-- ↠: "reduces to"
fst (sqr (1 + 3), sqr 2)    ↠    sqr (1 + 3)         [fst]
                            ↠    (1 + 3) * (1 + 3)   [sqr]
                            ↠    4 * 4               [+/+]
                            ↠    16                  [*]
```
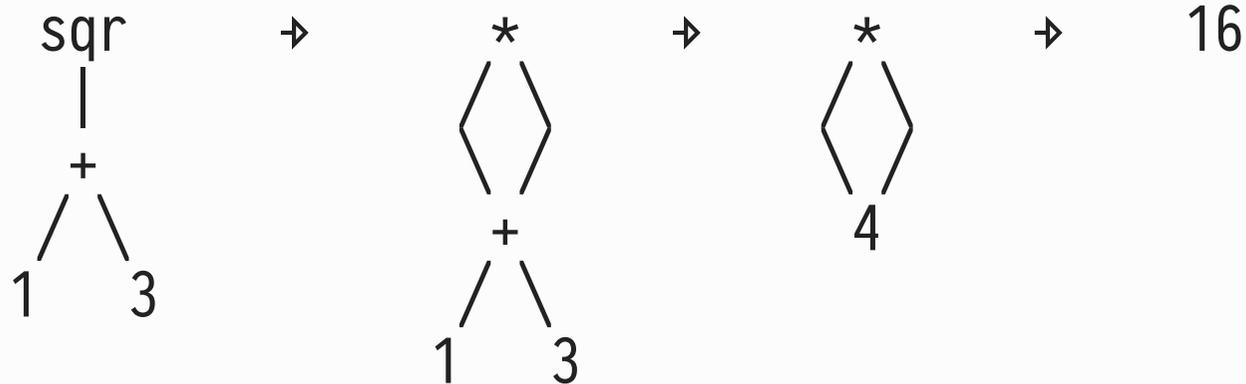
Graph Reduction

Haskell avoids the duplication of work through **graph reduction.** Expressions are shared (referenced more than once) instead of duplicated.

Example (reduction of sqr (1 + 3)):

```
  sqr        →        *        →        *        →     16
   |                 ⟨ ⟩              ⟨ ⟩
   +                  |                 |
  / \                 +                 4
 1   3               / \
                    1   3
```

Graph reduction and sharing …

- … makes normal order reduction never perform more reduction steps (⇸) than applicative order reduction,

- … can implement let…in efficiently,

- … depends on the language semantics to be **free of side effects:**
  - sharing affects the number of evaluations of an expression,
  - (the number of) side effects are observable for an outsider.

Weak Head Normal Form (WHNF)

To save further evaluation (= reduction) effort, Haskell stops expression reduction once **weak head normal form** has been reached:

- Expression e is in **weak head normal form (WHNF)** if it is of the following forms:

  1. v (where v is an atomic value of type Integer, Bool, Char, ...),
  2. $c\ e_1\ e_2\ …\ e_n$ (where c an n-ary constructor function, like (:)),
  3. $f\ e_1\ e_2\ …\ e_m$ (where f is an n-ary function with $m < n$).

- **NB:** The arguments e_i need *not* be in WHNF for e to be in WHNF.

- Haskell reduces values to WHNF only ($\equiv$ stop criterion for reduction) unless we explicitly request reduction to normal form (e.g. when printing results).

- **Example:** Expressions in WHNF:

```
42                  -- 1. atomic value
(sqr 2, sqr 4)      -- 2. tuple constructor (,)
f x : map f xs      -- 2. list constructor (:)
Just (40 + 2)       -- 2. Maybe constructor (Just)
(* (40 + 2))        -- 3. binary (*) applied to one argument only
(\x -> 40 + 2)      -- 3. lambda applied to no argument at all
```

- Haskell expressions may have the value **bottom** (⊥).
  - Examples: error "…", undefined, bomb (see above).

- **Lazy evaluation** admits functions that return a non-bottom value even if they receive ⊥ as argument (these are the so-called **non-strict** functions):

  - A n-ary function f is **strict** in its i-th argument, if
    $f\ e_1\ \dots\ e_{i-1}\ \bot\ e_{i+1}\ \dots\ e_n = \bot$.

- **Examples:**
  - const :: a -> b -> a: strict in first, non-strict in second argument
  - && :: Bool -> Bool -> Bool: dito

Lazy Evaluation and Bottom (⊥)

⚠️ If a function **pattern matches** on an argument, Haskell semantics define it to be **strict** in that argument.

- **Example:**

```haskell
data T = T Int

f :: T -> Int
f (T x) = 42            -- x not needed to produce result

f undefined      ⇸ ⊥
f (T undefined) ⇸ 42  -- argument evaluated but only
                       -- until pattern match can be decided
```

- Note: Haskell supports *lazy pattern matching* via syntax *~‹pat›*.

- To find the minimum in a non-empty list xs :: Ord a => [a]:
  1. sort xs in ascending order (here: use *insertion sort*, $O(n^2)$), then
  2. return the first element:

```
min :: Ord a => [a] -> a
min xs = (head . isort) xs
```

- 💡 Lazy evaluation never needs xs sorted in its entirety. Hmm…

- ⚠️ The following depends on our use of *insertion sort* (isort) as the sorting algorithm.

Proposed implementations of min and isort:

```
min :: Ord a => [a] -> a
min = head . isort                                    -- [min]

isort :: Ord a => [a] -> [a]
isort []     = []                                     -- [isort.1]
isort (x:xs) = ins x (isort xs)                       -- [isort.2]
  where
    ins x []                 = [x]                     -- [ins.1]
    ins x (y:ys) | x < y     = x:y:ys                  -- [ins.2]
                 | otherwise = y:ins x ys              -- [ins.3]
```

- Label the branches of function definitions via $[f.n]$ to refer to them during reduction.

Reduce min [8,6,1,7,5], use stop criterion WHNF:

```
min [8,6,1,7,5]
  ↬  (head . isort) [8,6,1,7,5]                     [min]
  ↬  head (isort [8,6,1,7,5])                       [(.)]
  ↬  head (ins 8 (ins 6 (ins 1 (ins 7 (ins 5 []))))) [isort.2⁺]
  ↬  head (ins 8 (ins 6 (ins 1 (ins 7 [5]))))       [ins.1]
  ↬  head (ins 8 (ins 6 (ins 1 (5 : ins 7 []))))    [ins.3] ⊠
  ↬  head (ins 8 (ins 6 (1 : (5 : ins 7 []))))      [ins.2]
  ↬  head (ins 8 (1 : ins 6 (5 : ins 7 [])))        [ins.3]
  ↬  head (1 : ins 8 (ins 6 (5 : ins 7 [])))        [ins.3]
  ↬  1                                               [head]
```

⊠ (5 : ins 7 []) is in WHNF ⟹ do not reduce any further.

Observing Reduction in GHCi

- Command :sprint <e> in ghci reduces <e> to WHNF.

- **Example:** observe behavior of function delete of pre-packaged
  module Data.List:

```
Prelude> :doc delete
delete :: (Eq a) => a -> [a] -> [a]
base Data.List
delete x removes the first occurrence of x
from its list argument. For example,

  delete 'a' "banana" == "bnana"

It is a special case of deleteBy, which allows the programmer
to supply their own equality test.
```

Infinite Lists (and other Data Structures)

A welcome consequence of lazy evaluation: programs can handle **infinite lists** as long as any run will inspect only a finite prefix of such a list.

- Enables a **modular style of programming** in which

  1. *generator functions* produce an infinite number of solutions/approximations/...
  2. *test functions* select one (or a finite number of) solutions from this infinite stream.

- Modularity: can formulate generator and test functions **independently.**

- **Idea:** Iteratively approximate the square root of x:

  1. a_0 $= x / 2$
  2. $a_{i+1} = (a_i + x / a_i) / 2, i \geqslant 1$

- To compute this series of a_i, employ

  - **generator** iterate :: (a -> a) -> a -> [a]:
    iterate f x = [x, f x, f (f x), …
  - **test** within :: (Ord a, Num a) => a -> [a] -> a:
    within ε xs consumes xs until two adjacent elements differ
    less than ε (for the first time).

- **Idea:** To compute $\int (f\ x)\ dx$ between x_1 and x_2, keep subdividing the interval $[x_1,x_2]$ until it is reasonable to assume that f is linear in the interval. Build on additive property of integration:

$$\int_{x_1}^{x_2} (f\ x)\ dx = \int_{x_1}^{m} (f\ x)\ dx + \int_{m}^{x_2} (f\ x)\ dx$$

$$= \int_{x_1}^{m_1} (f\ x)\ dx + \int_{m_1}^{m} (f\ x)\ dx + \int_{m}^{m_2} (f\ x)\ dx + \int_{m_2}^{x_2} (f\ x)\ dx$$