Lists and Abstraction*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin for slide material

List type

- Types we have seen
  - Base types: Int, Char, Bool, . . .
  - Type constructors: $(T_1, \ldots, T_k)$
- List types, a new type constructor

  If T is a type, then $[T]$ is a type
- Elements of $[T]$
  - Empty list $[\,] :: [T]$
  - If $x :: T$ and $xs :: [T]$, then $(x : xs) :: [T]$
- Short hand: $1 : (2 : (3 : [\,]))$ written as $[1, 2, 3]$

Examples and notation

```
? ['a','a','b'] :: [Char]
"aab" :: [Char]

? ['a','a','b'] == "aab"
True
```

Haskell supports various abbreviations

```
? [3..6]
[3, 4, 5, 6] :: [Integer]

? [6..3]
[] :: [Integer]
```

$[n, p..m]$ means count from n to m in steps of $p - n$

```
? [7,6..3]
[7, 6, 5, 4, 3] :: [Integer]

? [0.0, 0.3 .. 1.0]
[0.0,0.3,0.6,0.8999999999999999] :: [Double]
```

Functions on lists — sumList

- Function `sumList :: [Int] -> Int` must specify:
  - how to compute with the **empty list** $[\,]$
  - how to compute with the **non-empty list** $(x : xs)$
- Computation

  Empty list $[\,] \mapsto 0$
  Non-empty list $(x : xs) \mapsto x +$ sum of list xs

```
sumList []     = 0
sumList (x:xs) = x + sumList xs

? sumList [1..100]                -- 100 * 101 / 2
5050 :: Int
```

Standard functions on lists

- length

```
length []     = 0
length (x:xs) = 1 + length xs
```

- append (not only for strings!)

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

N.B.: $(++)$ and $(:)$ have different types!

```
? [2] ++ [3,4,5] == [2,3,4,5]
True

? 2 : [3,4,5] == [2,3,4,5]
True

? [2] : [3,4,5]
... Error ...
```

Patterns (lists and in general)

- Pattern matching has two purposes
  - checks if an argument has the proper form
  - binds values to variables
- Example: $(x : xs)$ matches with $[2, 3, 4]$ $(= 2 : 3 : 4 : [] = 2 : [3, 4])$

$$\begin{aligned} x &= 2 \\ xs &= [3, 4] \end{aligned}$$

- Another example (pattern matching during `let`-binding)

```
? let ([x,y,z],t) = ([1,2,3],(20,30)) in x + y
3 :: Int

? let ([x,y,z],t) = ([1,2,3,4],(20,30)) in x + y
Pattern match fails...
```

Patterns — details

- Patterns are inductively defined (additional constructors later)

  Constants: -2, $'1'$, $True$, $[\,]$
  Variables: x, foo
  Wild card: _
  Tuples: $(p_1, p_2, \ldots, p_k)$, where p_i are patterns
  Lists: $(p_1 : p_2)$, where p_i are patterns

- Moreover, patterns required to be **linear**.
  This means that each variable can occur at most once

- Examples: $[(x, foo), \_]$ $((x, y), \_)$ $1 : (2 : (x, y))$

- Counterexamples: $(x +\!\!+ y, z)$ and $[x, y, z, x]$

Pattern matching

- Define pattern p **matches** term a by recursion on p.

  Constant $p = c$**:** succeeds if $c = a$
  Variable $p = x$**:** succeeds and with binding $x = a$
  Wild card $p = \_$**:** succeeds but no binding
  Tuple $p = (p_1, \ldots, p_k)$**:** succeeds if $a = (a_1, \ldots, a_k)$ and p_i **matches** a_i, for $i \in \{1, \ldots, k\}$
  List $p = (p_1 : p_2)$**:** succeeds if a is a nonempty list $a_1 : a_2$ and p_1 **matches** a_1 and p_2 **matches** a_2

- Successful or not?
  - $([x], y)$ matches $([1], 2 + 3)$
  - $[x]$ matches $[1, 2]$? $[x, y]$ matches $[1]$?
  - $x : y$ matches $[1, 2]$? $x : (y : z)$ matches $[1, 2]$?
  - $[x, x]$ matches $[1, 1]$

Examples

- Zipper function

$$\begin{aligned} zip\ [2,3,4]\ [4,5,78] &= [(2,4),(3,5),(4,78)] \\ zip\ [2,3]\ [1,2,3] &= [(2,1),(3,2)] \end{aligned}$$

  N.B.: extra elements in a longer list are discarded.

- Implementing zip

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

- Which functions are implemented here?

```
f (False, False) = False
f (_,     _    ) = True
```

```
g (True, _   ) = True
g (_,    True) = True
g (_,    _   ) = False
```

Intermezzo — advice on recursion

> "Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it; it may seem impossible when you first try to do it yourself, but becomes simple and natural with practice." — G. Hutton, *Programming in Haskell*

Function *drop* removes the first n elements of a list.

Step 1: Define the type:

```
drop :: Int -> [Int] -> [Int]
```

Step 2: Enumerate the cases:

```
drop 0 []     = ...
drop 0 (x:xs) = ...
drop n []     = ...
drop n (x:xs) = ...
```

Step 3: Define the simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x:xs
drop n []     = []
```

Step 4: Define the other cases:

```
drop n (x:xs) = drop (n-1) xs
```

Step 5: Generalize and simplify. **Suggestions?**

Example: insertion sort

Insertion sort: $[7, 3, 9, 2]$

- First sort rest: $[2, 3, 9]$
- Insert head: $[2, 3, 7, 9]$

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = ins x (isort xs)

ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (x:xs)
  | a <= x    = a : (x : xs)
  | otherwise = x : ins a xs
```

```
isort [3,9,2]
    = ins 3 (isort [9,2])
    = ins 3 (ins 9 (isort [2]))
    = ins 3 (ins 9 (ins 2 (isort [])))
    = ins 3 (ins 9 (ins 2 []))
    = ins 3 (ins 9 [2])
    = ins 3 (2 : (ins 9 []))
    = 2 : (ins 3 (ins 9 []))
    = 2 : (ins 3 [9])
    = 2 : 3 : [9] = [2,3,9]

-- next evaluation step
-- evaluation of argument needed
```

Insertion sort — complexity

- Processing list $[x_1, \ldots, x_n]$ of size n results in

$$ins\ x_1\ (ins\ x_2\ (\ldots (ins\ x_n\ [\,])\ldots))$$

- Complexity of computing $ins\ a\ [x_1, x_2, \ldots, x_n]$

  Best case: $a \leq x_1$: 1 step
  Worst case: $a > x_n$: n steps
  Average: $n/2$ steps (assuming all input sequences equally likely)

- Complexity of insertion sort:

  Best case: n steps, i.e., $O(n)$
  Worst case: $1 + 2 + \cdots + n = n(n+1)/2$, i.e., $O(n^2)$
  Average: $1/2 + 2/2 + \cdots + (n-1)/2 + n/2$, also $O(n^2)$

Quick sort

- Quick sort: $[7, 3, 8, 2, 9]$
  - Split into head and tail: 7 and $[3, 8, 2, 9]$
  - Partition tail into parts ≤ 7 and > 7: $[3, 2]$ and $[8, 9]$
  - Recursively sort: $[2, 3]$ and $[8, 9]$
  - Concatenate with head in the middle: $[2, 3] +\!\!+ [7] +\!\!+ [8, 9] = [2, 3, 7, 8, 9]$

```
qsort [] = []
qsort (x:xs) =
  qsort (lesseq x xs) ++ [x] ++ qsort (greater x xs)
  where
    lesseq _ [] = []
    lesseq x (y:ys)
         | (y <= x)  = y : lesseq x ys
         | otherwise =     lesseq x ys
    greater _ [] = []
    greater x (y:ys)
         | (y > x)   = y : greater x ys
         | otherwise =     greater x ys
```

- Complexity: $O(n \log n)$ on average. Worst case?

List comprehension

- Notation for sequential processing of list elements
  - Analogous to set comprehension in set theory $\{2 \cdot x \mid x \in X\}$
  - Haskell notation: `[2*x | x <- xs]`

```
? [2*x | x <- [1,2,3,4,5]]
[2, 4, 6, 8, 10]

? [n `mod` 2 == 0 | n <- [2,4,7]]
[True,True,False]
```

- Can be augmented with guards: `[2*x | x <- xs, pred1(x), ...]`

```
? [2*x | x <- [0,1,2,3,4,5,6], x `mod` 2 == 0, x > 3]
[8, 12]
```

- What is computed here?

```
q []     = []
q (p:xs) = q [x | x<-xs, x <= p] ++ [p] ++ q [x | x<-xs, x > p]
```

Program development with list comprehensions
A larger example

- Objective: a (mini-)library database
- A design method for programs "in the small"
  1. Specify the requirements
  2. Fix the types (input/output representation)
  3. Implement each function
- For systems "in the large", design is substantially more difficult

  Topic for software engineering courses

Step 1: requirements analysis

Which functionality is required?

1. Given a person p, which books has p borrowed?
2. Given a book b, who has borrowed b?

   Assumption: many-to-many relation between (copies of) books and persons
3. Is a book lent out?

Step 2: types

- We define two types to represent books and people

```
type Person = String  -- Note that types always start with
type Book   = String  -- a capital letter
```

- Database: list of (Person, Book)-pairs

```
type Database = [(Person,Book)]
```

- Example

```
myDB = [("Alice", "Postman Pat"), ("Anna", "All Alone"),
                ("Alice", "Spot"), ("Rory", "Postman Pat")]
```

Types (cont.)

- Each kind of functionality implemented by a separate function

  1. Which books has person p borrowed?

     ```
     books :: Database -> Person -> [Book]
     ```

  2. Who has borrowed book b?

     ```
     borrowers :: Database -> Book -> [Person]
     ```

  3. Is a book b on loan?

     ```
     borrowed :: Database -> Book -> Bool
     ```

- Further functionality could also be specified

  E.g., checking out or returning a book

Step 3: implementation

- ```
  books:: Database -> Person -> [Book]

  books db p = [bk | (per,bk) <- db, per == p]
  ```

- ```
  borrowers :: Database -> Book -> [Person]

  borrowers db b = [per | (per,bk) <- db, bk == b]
  ```

- ```
  borrowed :: Database -> Book -> Bool

  borrowed db b = notEmpty (borrowers db b)
    where
      notEmpty [] = False
      notEmpty _  = True
  ```
  ```

Complete program and examples

```
type Person   = String
type Book     = String
type Database = [(Person,Book)]

myDB  = [("Alice", "Postman Pat"), ("Anna", "All Alone"),
                ("Alice", "Spot"), ("Rory", "Postman Pat")]

books     db p = [bk  | (per,bk) <- db, per == p]
borrowers db b = [per | (per,bk) <- db, bk == b]

borrowed db b = notEmpty (borrowers db b)
  where
    notEmpty [] = False
    notEmpty _  = True

? books myDB "Alice"
["Postman Pat", "Spot"]
? borrowers myDB "All Alone"
["Anna"]
? borrowed myDB "Postman Pat"
True
```

Correctness — Induction

Review: induction over natural numbers

- Rule for induction over $\mathcal{N}$.

  Proof by induction: to prove $P(n)$ for all natural numbers n.
  Base case: prove $P(0)$
  Step case: prove $P(n) \to P(n+1)$ for an arbitrary $n \in \mathcal{N}$, i.e.,
  Induction hypothesis: $P(n)$
  To prove: $P(n+1)$

- Once proven, we know, for example, $P(17)$ since
  - $P(0)$ holds, and
  - $P(0) \to P(1)$ . . . and. . . $P(16) \to P(17)$
  - since all $n \in \mathcal{N}$ are reachable in finitely many steps, we know that $P(n)$, for every $n \in \mathcal{N}$.

- Reflects that N is the least set such that $0 \in \mathcal{N}$ and if $n \in \mathcal{N}$, then so is $n + 1 \in \mathcal{N}$.

Induction over lists

- How are elements in $[T]$ constructed?

  $$[\,] :: [T] \quad \text{and} \quad (x : xs) :: [T], \text{ if } x :: T \text{ and } xs :: [T]$$

- Corresponds to following rule

  Proof by induction: to prove $P(xs)$ for all xs in $[T]$
  Base case: prove $P([\,])$
  Step case: prove $P(x : xs)$ under the assumption $P(xs)$, for an arbitrary $xs :: [T]$ and $x :: T$, i.e.,
  Induction hypothesis: $xs :: [T]$, $x :: T$, and $P(xs)$
  To prove: $P(x : xs)$

- Like with numbers: induction can be seen as a "machine" that establishes a property for all lists.

  E.g., $P([3, 20])$ follows since $P([\,]) \to P(20 : [\,]) \to P(3 : 20 : [\,])$.

Example 1: sum and double

```
sumList []     = 0                 double []     = []
sumList (x:xs) = x + sumList xs    double (x:xs) = (2*x) : double xs
```

Proof by induction: $P(xs) \equiv \texttt{sumList}\ (\texttt{double}\ xs) = 2 \cdot \texttt{sumList}\ xs$

Base case: $P([\,])$

$$\begin{aligned} \texttt{sumList}\ (\texttt{double}\ [\,]) &= \texttt{sumList}\ [\,] = 0 \\ 2 \cdot \texttt{sumList}\ [\,] &= 2 \cdot 0 = 0 \end{aligned}$$

Step case: $P(xs) \to P(x : xs)$. Let $x :: Int$ and $xs :: [Int]$ be arbitrary.

Induction hypothesis: $\texttt{sumList}\ (\texttt{double}\ xs) = 2 \cdot \texttt{sumList}\ xs$

To prove: $\texttt{sumList}\ (\texttt{double}\ (x : xs)) = 2 \cdot \texttt{sumList}\ (x : xs)$

$$\begin{aligned} \texttt{sumList}\ (\texttt{double}\ (x : xs)) &= \texttt{sumList}\ ((2 \cdot x) : \texttt{double}\ xs) \\ &= 2 \cdot x + \texttt{sumList}\ (\texttt{double}\ xs) \\ &= 2 \cdot x + 2 \cdot \texttt{sumList}\ xs \\ &= 2 \cdot (x + \texttt{sumList}\ xs) \\ &= 2 \cdot (\texttt{sumList}\ (x : xs)) \end{aligned}$$

Example 2: associativity of append

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Proof by induction:

$Q(xs) \equiv (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

Base case: $([\,] +\!\!+ ys) +\!\!+ zs = ys +\!\!+ zs = [\,] +\!\!+ (ys +\!\!+ zs)$

Step case: Let x and xs be arbitrary.

Induction hypothesis: $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

To prove: $((x : xs) +\!\!+ ys) +\!\!+ zs = (x : xs) +\!\!+ (ys +\!\!+ zs)$

$$\begin{aligned}((x : xs) +\!\!+ ys) +\!\!+ zs &= (x : (xs +\!\!+ ys)) +\!\!+ zs \\ &= x : ((xs +\!\!+ ys) +\!\!+ zs) \\ &= x : (xs +\!\!+ (ys +\!\!+ zs)) \\ &= (x : xs) +\!\!+ (ys +\!\!+ zs)\end{aligned}$$

Induction: difficulties

- How do we select an induction variable?

$$\begin{aligned} P(xs) &\equiv \texttt{sumList}\ (\texttt{double}\ xs) = 2 \cdot \texttt{sumList}\ xs \\ Q(???) &\equiv (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs) \end{aligned}$$

  - Induction on ys or zs fails, e.g., $Q(zs)$
  - In step case we must show $Q(z : zs)$

$$(xs +\!\!+ ys) +\!\!+ (z : zs) = xs +\!\!+ (ys +\!\!+ (z : zs))$$

  Not possible either to further simplify or to use $Q(zs)$.

- Requires planning in order to use the induction hypothesis

- Auxiliary lemmas are sometimes also needed.

Example: `rev` and `qrev`

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
qrev xs = aux (xs, [])
aux ([],     ys) = ys
aux ((x:xs), ys) = aux (xs, x:ys)
```

- Append is $O(n)$ in first argument

$$\begin{aligned}[a_1, a_2, \ldots, a_n] +\!\!+\, x &= a_1 : ([a_2, \ldots, a_n] +\!\!+\, x) \\ &= \ldots\ n-1 \text{ times } \ldots \\ &= a_1 : (a_2 : \cdots : (a_n : x) \ldots)\end{aligned}$$

- `rev` is $O(n^2)$ (average append: $n/2$ steps)

$$\begin{aligned}\texttt{rev}\ [a_1, a_2, \ldots, a_n] &= \texttt{rev}\ [a_2, \ldots, a_n] +\!\!+\, [a_1] \\ &= \ldots\ n-1 \text{ times } \ldots \\ &= (\ldots([\,] +\!\!+\, [a_n]) +\!\!+\, \ldots +\!\!+\, [a_2]) +\!\!+\, [a_1]\end{aligned}$$

Example (cont.)

- `qrev` is $O(n)$

$$\begin{aligned}
\texttt{qrev}\ [a_1, \ldots, a_n] &= \texttt{aux}\ ([a_1, \ldots, a_n], [\,]) \\
&= \texttt{aux}\ ([a_2, \ldots, a_n], [a_1]) \\
&= \ldots\ n-2 \text{ times } \ldots \\
&= \texttt{aux}\ ([\,], [a_n, \ldots, a_2, a_1]) \\
&= [a_n, \ldots, a_2, a_1]
\end{aligned}$$

Relationship between rev and $qrev$

- Is $\texttt{rev}\ xs = \texttt{qrev}\ xs$?

  Base case: $P([\,])$

  $$\texttt{rev}\ [\,] = [\,] = \texttt{aux}\ ([\,], [\,]) = \texttt{qrev}\ [\,]$$

  Step case: $P(xs) \to P(x : xs)$. Let x and xs be arbitrary.

  Induction hypothesis: $\texttt{rev}\ xs = \texttt{qrev}\ xs$

  To prove: $\texttt{rev}\ (x : xs) = \texttt{qrev}\ (x : xs)$

  $$\begin{aligned} \texttt{rev}\ (x : xs) &= \texttt{rev}\ xs +\!\!+ [x] \\ \texttt{qrev}\ (x : xs) &= \texttt{aux}\ (x : xs, [\,]) = \texttt{aux}\ (xs, [x]) \end{aligned}$$

- No proof!
  - We require $\texttt{rev}\ xs +\!\!+ [x] = \texttt{aux}\ (xs, [x])$
  - Induction hypothesis only says $\texttt{rev}\ xs = \texttt{qrev}\ xs = \texttt{aux}\ (xs, [\,])$

Next induction

- To prove: $\texttt{rev}\ xs +\!\!+ [x] = \texttt{aux}\ (xs, [x])$. Let's try induction again

  Base case: $P([\,])$

  $$\texttt{rev}\ [\,] +\!\!+ [x] = [\,] +\!\!+ [x] = [x] = \texttt{aux}\ ([\,], [x])$$

  Step case: $P(xs) \to P(a : xs)$. Let a and xs be arbitrary.

  Induction hypothesis: $\texttt{rev}\ xs +\!\!+ [x] = \texttt{aux}\ (xs, [x])$
  To prove: $\texttt{rev}\ (a : xs) +\!\!+ [x] = \texttt{aux}\ (a : xs, [x])$

  $$\begin{aligned} \texttt{rev}\ (a : xs) +\!\!+ [x] &= (\texttt{rev}\ xs +\!\!+ [a]) +\!\!+ [x] \\ &= \texttt{rev}\ xs +\!\!+ ([a] +\!\!+ [x]) \\ &= \texttt{rev}\ xs +\!\!+ [a, x] \\ \texttt{aux}\ (a : xs, [x]) &= \texttt{aux}\ (xs, [a, x]) \end{aligned}$$

- Problem: Induction hypothesis only proven for accumulator $[x]$.
  Rather than another induction, let's try a **generalization**!

Generalization

- To prove $\texttt{rev}\ xs +\!\!+ [x] = \texttt{aux}\ (xs, [x])$, we prove

$$\texttt{rev}\ xs +\!\!+ ys = \texttt{aux}\ (xs, ys)$$

  I.e., prove $\forall xs.\, P(xs)$, where
  $P(xs) \equiv \forall ys.\, \texttt{rev}\ xs +\!\!+ ys = \texttt{aux}\ (xs, ys)$

- Recall proof rules for $\forall x.Q(x)$
  - $\forall$-Introduction
    To prove: $\forall x.Q(x)$
    Prove: $Q(x)$, where x is arbitrary
  - $\forall$-Elimination
    From $\forall x.Q(x)$ we can conclude $Q(t)$, for every t

Prove: $\forall xs.\, \forall ys.\, \texttt{rev}\ xs +\!\!+ ys = \texttt{aux}\ (xs, ys)$

Proof by induction: $\forall xs.\, P(xs)$ by induction

$$P(xs) \equiv \forall ys.\, \texttt{rev}\ xs +\!\!+ ys = \texttt{aux}\ (xs, ys)$$

Base case: $P([\,])$

$$\forall ys.\, \texttt{rev}\ [\,] +\!\!+ ys = \texttt{aux}\ ([\,], ys)$$

Using $\forall$-Introduction, it is sufficient to prove:

$$\texttt{rev}\ [\,] +\!\!+ ys = \texttt{aux}\ ([\,], ys)$$

Holds as $\texttt{rev}\ [\,] +\!\!+ ys = [\,] +\!\!+ ys = ys = \texttt{aux}\ ([\,], ys)$

Proof (cont.)

Step case: $P(xs) \to P(x : xs)$. Let x and xs be arbitrary.

Induction hypothesis: $\forall ys.\, \texttt{rev}\ xs +\!\!+ ys = \texttt{aux}\ (xs, ys)$
To prove: $\forall ys.\, \texttt{rev}\ (x : xs) +\!\!+ ys = \texttt{aux}\ (x : xs, ys)$

Using $\forall$-introduction, we reduce the goal (for ys arbitrary) to

$$\texttt{rev}\ (x : xs) +\!\!+ ys = \texttt{aux}\ (x : xs, ys)$$

From the definition and associativity of $++$, we know that

$$\begin{aligned} \texttt{rev}\ (x : xs) +\!\!+ ys &= (\texttt{rev}\ xs +\!\!+ [x]) +\!\!+ ys \\ &= \texttt{rev}\ xs +\!\!+ ([x] +\!\!+ ys) = \texttt{rev}\ xs +\!\!+ (x : ys) \\ \texttt{aux}\ (x : xs, ys) &= \texttt{aux}\ (xs, x : ys) \end{aligned}$$

But an instance of the induction hypothesis ($\forall$-Elimination) is

$$\texttt{rev}\ xs +\!\!+ (x : ys) = \texttt{aux}\ (xs, x : ys) \qquad \textbf{QED}$$

Abstraction

Until now . . .

- We have only seen simple structuring techniques
  - sufficient though to construct all programs
- Difference: language **expressiveness** versus **usability/eloquence**
  - Assembler versus modern programming languages
  - Quick sort with/without list comprehension

```
q []     = []
q (p:xs) = q [x | x<-xs, x <= p] ++ [p] ++ q [x | x<-xs, x > p]
```

- We will now examine different ways of
  - structuring programs
  - simplifying programs
  - improving their reusability

Polymorphic types and reusability

- Consider following example:

```
length []     = 0
length (x:xs) = 1 + length xs

? length [17,3,149]
3 :: Int

? length ["eat","the","potato","Jane"]
4 :: Int
```

- What is type? `[Int] -> Int`, `[String] -> Int`, ...
  - The type is **polymorphic**: `[t] -> Int`, **for all types** t.
- Often called **parametric polymorphism**
  - Generics in Java/Eiffel: `static <T> int length(List<T> xs) ...`
  - Differs from **subtyping polymorphism**, where methods can be applied to objects only of sub-classes.

Types and reusability (cont.)

- Polymorphic types contain type variables

```
length :: [t] -> Int
```

- Function typeable for all instances.

  Definition: A type w for f is a **most general** (also called **principal**) **type** iff for all types s for f, s is an instance of w.

- Haskell has algorithms for **type checking** and **type reconstruction**
  - Haskell computes the principal type w, given a function f
  - If user provides a type t, then t must be an instance of w.
    I.e., one can only restrict the type
  - We will look at this in more detail later.

- **Type variables in Haskell start with a lower-case letter.**

Polymorphic types — examples

- Functions over lists

```
:type (++)
(++) :: [a] -> [a] -> [a]

:type zip
zip :: [a] -> [b] -> [(a, b)]

:type []
[] :: [a]
```

- Identity

```
id :: a -> a
id x = x
```

- Functions over tuples

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

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

Higher-order functions

First order: Arguments are base types or constructor types

```
Int -> [Int]
```

Second order: Arguments can be themselves functions

```
(Int -> Int) -> [Int]
```

Third order: Arguments may be functions, whose arguments are functions

```
((Int -> Int) -> Int) -> [Int]
```

Higher-order functions: Functions of arbitrary order

Example: map

```
double :: [Int] -> [Int]
double []     = []
double (x:xs) = (2*x) : double xs

triple :: [Int] -> [Int]
triple []     = []
triple (x:xs) = (3*x) : triple xs
```

Same control structure, only different function application

$$\texttt{double}\,[x_1, \ldots, x_n] = [2{\cdot}x_1, \ldots, 2{\cdot}x_n]$$
$$\texttt{triple}\,[x_1, \ldots, x_n] = [3{\cdot}x_1, \ldots, 3{\cdot}x_n]$$

Control structure can be abstracted

```
map :: (a -> b) -> [a] -> [b]
map f []     = []                -- higher order
map f (x:xs) = f x : map f xs    -- (function f is an argument)

times2 x = 2 * x
times3 x = 3 * x

double xs = map times2 xs
triple xs = map times3 xs
```

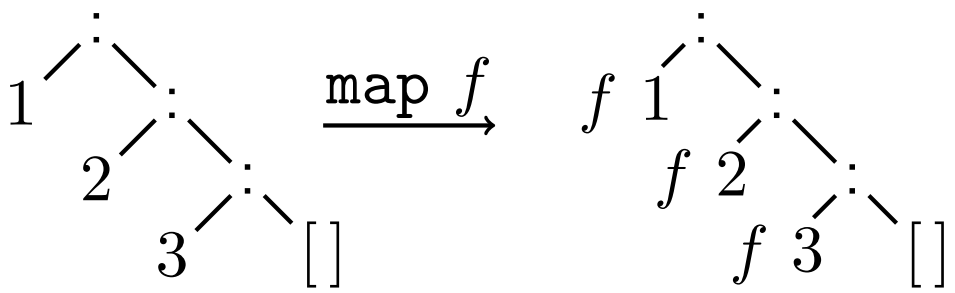
Example of execution

```
map times2 [2,3]
  = times2 2 : map times2 [3]
  = 4 : map times2 [3]
  = 4 :(times2 3 : map times2 [])
  = 4 : (6 : map times2 [])
  = 4 : (6 : [])
  = [4,6]
```

Visualizing map

```
map f []     = []
map f (x:xs) = f x : map f xs
```

$$\texttt{map}\ f\ [1,2,3] = [f\ 1, f\ 2, f\ 3]$$



Note similarity to list comprehension: `map f xs = [f x | x <- xs]`

Why functions as arguments?

```
-- without map
double []     = []
double (x:xs) = (2*x) : double xs

-- with map
double xs = map times2 xs
```

Advantages:

1. Definition is easier to understand
2. Parts are easier to modify
3. Parts are easier to reuse
4. Correctness is simpler to understand and show

Example: folding

- Consider `sumList [1,2,3] = 1+2+3`

```
sumList []     = 0
sumList (x:xs) = x + sumList xs
```

Is this an instance of `map`?

- Generalization

$$g\,[x_1, x_2, \ldots, x_k] = f(x_1, f(x_2, \ldots, f(x_k, 0)\ldots))$$

E.g. `sumList [1,2,3] = f(1, f(2, f(3, 0)))`, for `f = (+)`

- Program

```
foldr :: (a -> b -> b) -> b -> [a] -> b

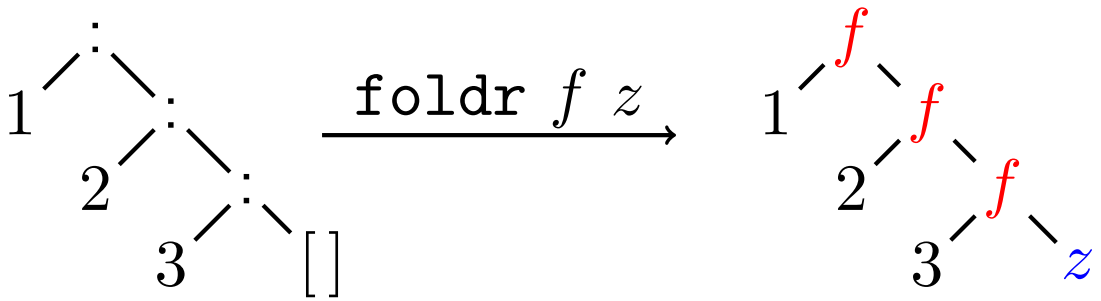
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

sumList xs = foldr (+) 0 xs
```

Visualizing foldr

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

$$\texttt{foldr}\ f\ z\ [1,2,3] = f(1,\ f(2,\ f(3,\ z))$$



`foldr` f z replaces (:) with f and [] with z.

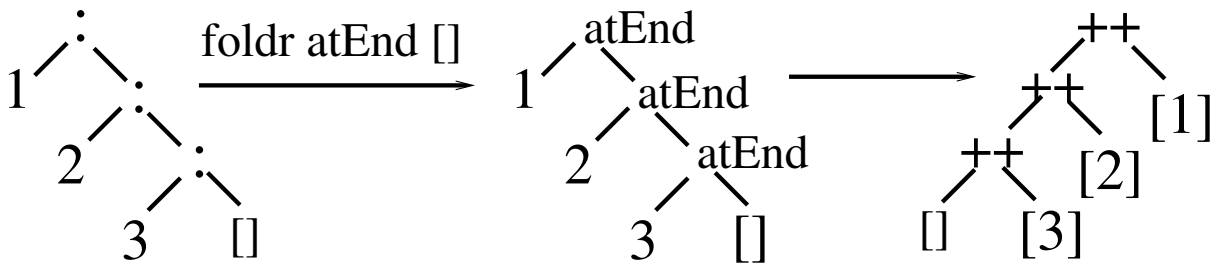
Example: reverse

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

$$
\begin{aligned}
\texttt{rev}\ [1,2,3] &= \texttt{rev}\ [2,3] \mathbin{+\!\!+} [1] \\
&= (\texttt{rev}\ [3] \mathbin{+\!\!+} [2]) \mathbin{+\!\!+} [1] \\
&= ((\texttt{rev}\ [\,] \mathbin{+\!\!+} [3]) \mathbin{+\!\!+} [2]) \mathbin{+\!\!+} [1] \\
&= (([\,] \mathbin{+\!\!+} [3]) \mathbin{+\!\!+} [2]) \mathbin{+\!\!+} [1] \\
&= \ldots \\
&= [3,2,1]
\end{aligned}
$$

Suggests program:

```
atEnd x xs = xs ++ [x]
rv xs = foldr atEnd [] xs
```



Example execution

```
atEnd x xs = xs ++ [x]
rv xs = foldr atEnd [] xs
```

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Executes as follows:

$$
\begin{aligned}
\texttt{rv}\ [1,2,3] &= \texttt{foldr atEnd}\ [\,]\ [1,2,3] \\
&= \texttt{atEnd}\ 1\ (\texttt{foldr atEnd}\ [\,]\ [2,3]) \\
&= (\texttt{foldr atEnd}\ [\,]\ [2,3]) +\!\!+ [1] \\
&= (\texttt{atEnd}\ 2\ (\texttt{foldr atEnd}\ [\,]\ [3])) +\!\!+ [1] \\
&= ((\texttt{foldr atEnd}\ [\,]\ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&= (\texttt{atEnd}\ 3\ (\texttt{foldr atEnd}\ [\,]\ [\,]) +\!\!+ [2]) +\!\!+ [1] \\
&= (((\texttt{foldr atEnd}\ [\,]\ [\,]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&= ([\,] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&= \ldots \\
&= [3,2,1]
\end{aligned}
$$

Correctness: prove $\forall xs.\, \texttt{rev}\ xs = \texttt{rv}\ xs$

More examples of foldr

We can now easily define some standard Haskell functions:

```
concat xs = foldr (++) [] xs
? concat [[1,2,3],[4],[5,6]]
[1,2,3,4,5,6] :: [Int]

and bs = foldr (&&) True bs
? and [True,True,False]
False :: Bool

or bs = foldr (||) False bs
? or [True,True,False]
True :: Bool

pcons (x,y) (xs,ys) = (x:xs, y:ys)
unzip xs = foldr pcons ([],[]) xs
? unzip [(1,2),(3,4),(5,6)]
([1,3,5],[2,4,6]) :: ([Int],[Int])
```

Exercise: Express `map` in terms of `foldr`.

λ-expressions

- Consider the following functions

```
times2 x = 2 * x                    atEnd x xs = xs ++ [x]
double xs = map times2 xs           rv xs = foldr atEnd [] xs
```

- Haskell provides notation to write functions like `times2` and `atEnd` in-line

```
? map (\x -> 2 * x)  [2,3,4]
[4,6,8]

? foldr (\x xs -> xs ++ [x]) [] [1,2,3,4]
[4,3,2,1]
```

N.B.: `\x xs -> xs ++ [x]` is a shorthand for `\x -> \xs -> xs ++ [x]`

- Church's λ-notation (character "\" used instead of "λ")

	usual	λ-calculus
Declaration	$f(x) = x + 3$	$\lambda x.\, x + 3$
Application	$f(5)$	$(\lambda x.\, x + 3)(5)$
Reduction (substitution)	$(x + 3)[x \leftarrow 5]$	$(x + 3)[x \leftarrow 5]$
Result (from evaluation)	8	8

Example: function composition and iteration

- Function composition $f \circ g$ as a function

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

- Function iteration

```
iter :: Int -> (a -> a) -> a -> a
iter 0 f x = x
iter n f x = f (iter (n-1) f x)
```

- Examples

```
times2 x = 2 * x
times3 x = 3 * x
```

```
? (times2 . times3) 5
30 :: Int

? iter 4 times2 1
16 :: Int

? iter 4 times3 1
81 :: Int
```

Functions as values

- Functions can be returned as values
- Example

```
twice :: (t -> t) -> (t -> t)
twice f = f . f

? twice times3 1
9 :: Int
? (twice . twice) times3 1          -- 3^4 = 81
81 :: Int
? twice (times3 . times3) 1
81 :: Int
```

Functions as values (cont.)

- Iter (higher-order argument)

```
iter 0 f x = x
iter n f x = f (iter (n-1) f x)
```

- Iter (higher-order argument and result)

```
id x = x

iter :: Int -> (a -> a) -> a -> a
iter 0 f = id
iter n f = f . (iter (n-1) f)
```

- Functions cannot be displayed, but they can be applied

```
? iter 2 times2
... Error ...

? let f = iter 2 times2 in f 5
20 :: Int
```

Evaluation example

```
iter 0 f = id
iter n f = f . (iter (n-1) f)
```

```
id x       = x
(f . g) x = f (g x)
times2 x   = 2 * x
```

```
let f = iter 2 times2 in f 5
```

Can be calculated as follows

```
let f = iter 2 times2 in f 5
 = (iter 2 times2) 5
 = (times2 . (iter (2-1) times2)) 5
 = times2 2 ((iter (2-1) times2) 5)
 = 2 * ((iter (2-1) times2) 5)
 = 2 * ((times2 . iter (1-1) times2) 5)
 = 2 * (times2 ((iter (1-1) times2) 5))
 = 2 * (2 * ((iter (1-1) times2) 5))
 = 2 * (2 * (id 5))
 = 2 * (2 * 5) = ... = 20
```

Example: Logging

- We want to log all recursive calls to `gcd` in a list.

```
gcdLog :: Integer -> Integer ->          Integer
gcdLog x y
  | x == y     =  x
  | x > y      = gcdLog (x-y) y
  | otherwise = gcdLog x (y-x)
```

```
> gcdLog 6 4
 2
```

Example: Logging

- We want to log all recursive calls to `gcd` in a list.

```
type Log = [(Integer, Integer)]

gcdLog :: Integer -> Integer -> Log -> (Integer, Log)
gcdLog x y log
  | x == y    = (x, log')
  | x > y     = gcdLog (x-y) y log'
  | otherwise = gcdLog x (y-x) log'
  where
    log' = log ++ [(x, y)]
```

```
> gcdLog 6 4 []
(2,[(6,4),(2,4),(2,2)])
```

Example: Logging

- We want to log all recursive calls to `gcd` in a list.

```
type Log = [(Integer, Integer)]

gcdLog :: Integer -> Integer -> Log -> (Integer, Log)
gcdLog x y log
  | x == y    = (x, log')
  | x > y     = gcdLog (x-y) y log'
  | otherwise = gcdLog x (y-x) log'
  where
    log' = log ++ [(x, y)]
```

```
> gcdLog 6 4 []
(2,[(6,4),(2,4),(2,2)])
```

- Appending to the log is linear in the log's size.
  Repeated appending causes quadratic run-time.

  Can we do better?

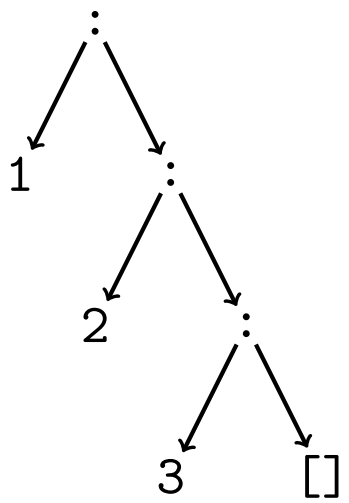
Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

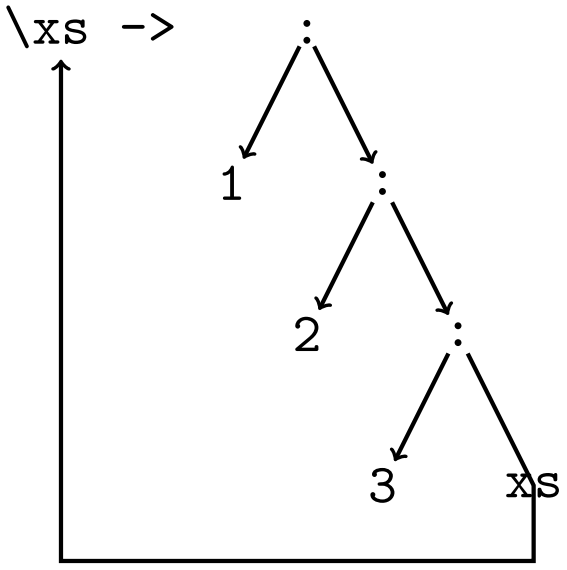
Example: `[1,2,3]` as a difference list

```
dl123 = \xs -> [1,2,3] ++ xs
```

ordinary list `[1,2,3]`



difference list `dl123`



The parameter is like a hole at the end of the list.

Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

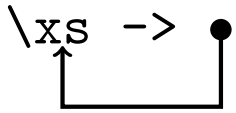
empty :: DList a                    -- empty list
empty = \xs -> xs

sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```



Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

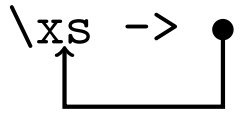
empty :: DList a                    -- empty list
empty = \xs -> xs

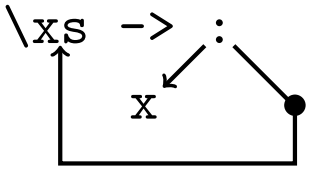
sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```





Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

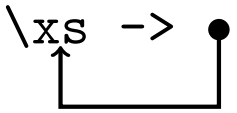
empty :: DList a                    -- empty list
empty = \xs -> xs

sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```



Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

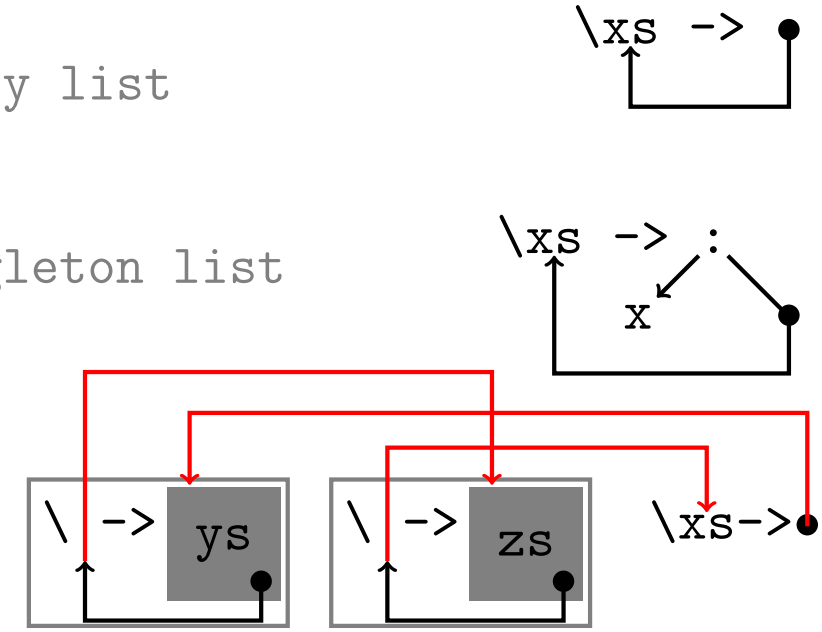
empty :: DList a                    -- empty list
empty = \xs -> xs

sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```



Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

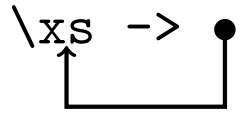
empty :: DList a                    -- empty list
empty = \xs -> xs

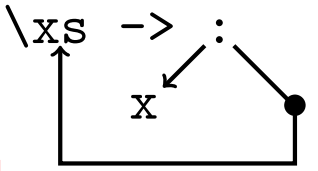
sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

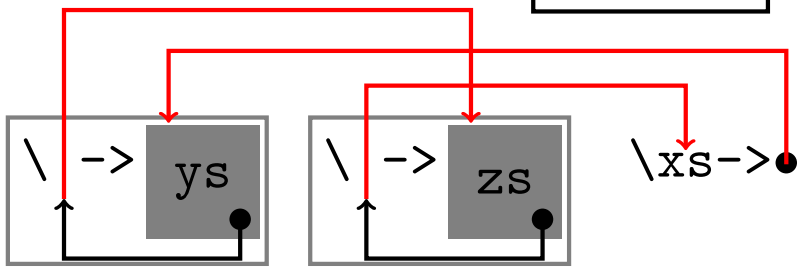
-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```







Difference lists

A **difference list** is a function `[a] -> [a]` that prepends itself to a list.

```
type DList a = [a] -> [a]

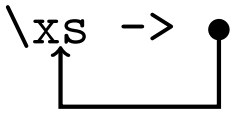
empty :: DList a                    -- empty list
empty = \xs -> xs

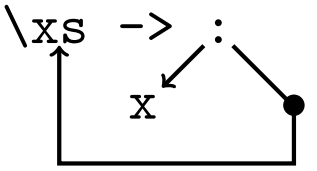
sngl :: a -> DList a                -- singleton list
sngl x = \xs -> x : xs

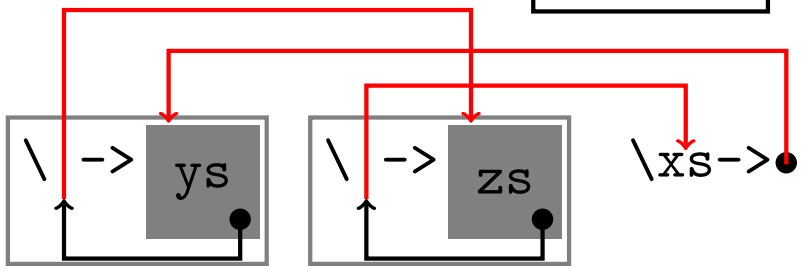
-- concatenation (higher-order!)
app :: DList a -> DList a -> DList a
ys `app` zs = \xs -> ys (zs xs)

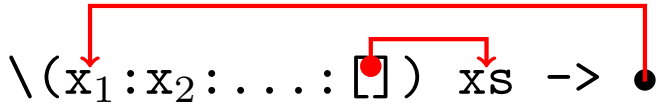
fromList :: [a] -> DList a          -- conversion from lists
fromList ys = \xs -> ys ++ xs

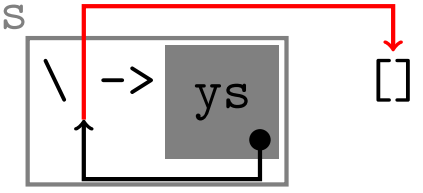
toList :: DList a -> [a]            -- conversion to lists
toList ys = ys []
```











Logging with difference lists I

```
type  Log =      [(Integer, Integer)]

gcdDLog :: Integer -> Integer ->  Log -> (Integer,  Log)
gcdDLog x y log
  | x == y    = (x, log')
  | x > y     = gcdLog (x-y) y log'
  | otherwise = gcdLog x (y-x) log'
  where
    log' = log       ++    [(x, y)]
```

Logging with difference lists I

```
type DLog = DList (Integer, Integer)

gcdDLog :: Integer -> Integer -> DLog -> (Integer, DLog)
gcdDLog x y log
  | x == y    = (x, log')
  | x > y     = gcdLog (x-y) y log'
  | otherwise = gcdLog x (y-x) log'
  where
    log' = log `app` sngl (x, y)
```

Logging with difference lists I

```
type DLog = DList (Integer, Integer)

gcdDLog :: Integer -> Integer -> DLog -> (Integer, DLog)
gcdDLog x y log
  | x == y    = (x, log')
  | x > y     = gcdLog (x-y) y log'
  | otherwise = gcdLog x (y-x) log'
  where
    log' = log `app` sngl (x, y)

gcdDLog 6 4 empty
 = gcdDLog 2 4 (empty `app` sngl (6,4))
 = gcdDLog 2 2 ((empty `app` sngl (6,4)) `app` sngl (2,4))
 = (2, ((empty `app` sngl (6,4)) `app` sngl (2,4)) `app` sngl (2,2))
```

Logging with difference lists II

```
empty        = \xs -> xs
sngl x       = \xs -> x : xs
ys `app` zs  = \xs -> ys (zs xs)
toList xs    = xs []
```

Let's evaluate the log:

```
toList (((empty `app` sngl (6,4)) `app` sngl (2,4)) `app` sngl (2,2))
 = (((empty `app` sngl (6,4)) `app` sngl (2,4)) `app` sngl (2,2)) []
 = ((empty `app` sngl (6,4)) `app` sngl (2,4)) (sngl (2,2) [])
 = (empty `app` sngl (6,4)) (sngl (2,4) (sngl (2,2) []))
 = empty (sngl (6,4) (sngl (2,4) (sngl (2,2) [])))
 = sngl (6,4) (sngl (2,4) (sngl (2,2) []))
 = (6,4) : sngl (2,4) (sngl (2,2) [])
 = (6,4) : (2,4) : sngl (2,2) []
 = (6,4) : (2,4) : (2,2) : []
```

- Appending to the end of a difference list takes $\mathcal{O}(1)$.
- Getting the head takes $\mathcal{O}(n)$ where $n =$ number of appends.
- Getting the remaining elements is $\mathcal{O}(1)$ each.

Partial application

- Functions of multiple arguments . . .

```
multiply :: Int -> Int -> Int
multiply a b = a * b
```

- . . . can be partially applied

```
? :type multiply 7
Int -> Int

? :type map
(a -> b) -> [a] -> [b]

? map (multiply 7) [1,2,3,4]
[7, 14, 21, 28] :: [Int]
```

- Application and types
  If $f :: t_1 \to t_2 \to \cdots \to t_n \to t$ and $e_1 :: t_1, \ldots, e_k :: t_k$
  then $f\ e_1 \ldots e_k :: t_{k+1} \to \cdots \to t_n \to t$

How many arguments do functions have?

- Each function takes exactly one argument
  - `multiply :: Int -> Int -> Int` means `multiply :: Int -> (Int -> Int)`
  - Application `multiply 2 3` means `(multiply 2) 3`
  - Partial application is consistent with the view (= illusion) that functions take multiple arguments
- Operator sections: if $\oplus$ is an infix binary operator

$$(a \oplus) \equiv \lambda x.\, a \oplus x$$

$$(\oplus\, a) \equiv \lambda x.\, x \oplus a$$

- Example

```
? map ((2*).(3*)) [1,2,3]
[6,12,18]
```

Multiple arguments versus tupling

```
f :: (Int,Int) -> Int
f (x,y) = x * y + 17
```

```
g :: Int -> Int -> Int
g x y = x * y + 17
```

- Tuple arguments: no partial application
- But equivalent in the following sense:

```
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a,b) -> c

curry f    = f' where f' x1 x2  = f (x1,x2)
uncurry f' = f  where f (x1,x2) = f' x1 x2
```

- "Curry" like "Haskell Brooks Curry"

Examples

```
f :: (Int,Int) -> Int
f (x,y) = x * y + 17
```

```
g :: Int -> Int -> Int
g x y = x * y + 17
```

```
? f (3,4)
29 :: Int

? curry f 3 4
29 :: Int

? g (3,4)
... Error ...

? g 3 4
29 :: Int

? uncurry g (3,4)
29 :: Int
```

Uncluttering notation

- `f $ x = f x` right-associative operator for function application. Avoids parentheses:

```
putStrLn (show (gcd 1000 (fib 6 + fib 10)))

putStrLn $ show $ gcd 1000 $ fib 6 + fib 10
```

- Partial application and sections avoid notational clutter.
- Example: operations on difference lists

```
empty = \xs -> xs
sngl x = \xs -> x : xs
ys `app` zs = \xs -> ys (zs xs)
fromList ys = \xs -> ys ++ xs
toList ys = ys []
```

```
empty = id
sngl = (:)
app = (.)
fromList = (++)
toList = ($[])
```

The operations on difference lists are predefined.