Functional Programming
FLOLAC 2018

Shin-Cheng Mu

July, 2018

Academia Sinica and National Taiwan University

Values, and Wholemeal Programming

Course Materials and Tools

- Course homepage: `http://flolac.iis.sinica.edu.tw/flolac18/`
- We will be using the Glasgow Haskell Compiler (GHC).
    - A Haskell compiler written in Haskell, with an interpreter that both interprets and runs compiled code.
    - Installation: the Haskell Platform: `http://hackage.haskell.org/platform/`
- Early parts of the course material are adapted from Bird, which I highly recommend.

Function Definition

- A function definition consists of a type declaration, and the definition of its body:

$$\begin{aligned} &\mathit{square} && :: \mathit{Int} \to \mathit{Int} \\ &\mathit{square}\ x && = x \times x \\ \\ &\mathit{smaller} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ &\mathit{smaller}\ x\ y && = \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \end{aligned}$$

- The GHCi interpreter evaluates expressions in the loaded context:

$$\begin{aligned} &?\ \mathit{square}\ 3768 \\ &14197824 \\ &?\ \mathit{square}\ (\mathit{smaller}\ 5\ (3 + 4)) \\ &25 \end{aligned}$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3+4)$:

$$\mathit{square}\ (3+4)$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3+4)$:

$$\begin{array}{cl} & \mathit{square}\ (3+4) \\ = & \quad \{\ \text{definition of} + \ \} \\ & \mathit{square}\ 7 \end{array}$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3+4)$:

$$
\begin{array}{ll}
 & \mathit{square}\ (3+4) \\
= & \quad \{\ \text{definition of } +\ \} \\
 & \mathit{square}\ 7 \\
= & \quad \{\ \text{definition of } \mathit{square}\ \} \\
 & 7 \times 7
\end{array}
$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3 + 4)$:

$$
\begin{array}{cl}
 & \mathit{square}\ (3+4) \\
= & \quad \{\ \text{definition of } +\ \} \\
 & \mathit{square}\ 7 \\
= & \quad \{\ \text{definition of } \mathit{square}\ \} \\
 & 7 \times 7 \\
= & \quad \{\ \text{definition of } \times\ \} \\
 & 49
\end{array}
$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\mathit{square}\ (3 + 4)$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{aligned} & \mathit{square}\ (3+4) \\ = \quad & \{\ \text{definition of } \mathit{square}\ \} \\ & (3+4) \times (3+4) \end{aligned}$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{aligned}
& \mathit{square}\ (3+4) \\
= \quad & \{\ \text{definition of } \mathit{square}\ \} \\
& (3+4) \times (3+4) \\
= \quad & \{\ \text{definition of } +\ \} \\
& 7 \times (3+4)
\end{aligned}$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{array}{cl}
 & \mathit{square}\ (3+4) \\
= & \{\ \text{definition of } \mathit{square}\ \} \\
 & (3+4) \times (3+4) \\
= & \{\ \text{definition of } +\ \} \\
 & 7 \times (3+4) \\
= & \{\ \text{definition of } +\ \} \\
 & 7 \times 7
\end{array}$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{aligned}
& \mathit{square}\ (3+4) \\
= \quad & \{\ \text{definition of } \mathit{square}\ \} \\
& (3+4) \times (3+4) \\
= \quad & \{\ \text{definition of } +\ \} \\
& 7 \times (3+4) \\
= \quad & \{\ \text{definition of } +\ \} \\
& 7 \times 7 \\
= \quad & \{\ \text{definition of } \times\ \} \\
& 49
\end{aligned}$$

- In this sequence the rule for *square* is applied first. The final result stays the same.
- Do different evaluations orders always yield the same thing?

A Non-terminating Reduction

- Consider the following program:

$$\begin{aligned} &\mathit{three} && :: \mathit{Int} \to \mathit{Int} \\ &\mathit{three}\ x && = 3 \end{aligned}$$

$$\begin{aligned} &\mathit{infinity} && :: \mathit{Int} \\ &\mathit{infinity} && = \mathit{infinity} + 1 \end{aligned}$$

- Try evaluating *three infinity*. If we simplify *infinity* first:

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = \quad & \{\ \text{definition of } \mathit{infinity}\ \} \\ & \mathit{three}\ (\mathit{infinity} + 1) \\ = \quad & \mathit{three}\ ((\mathit{infinity} + 1) + 1) \ldots \end{aligned}$$

- If we start with simplifying *three*:

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = \quad & \{\ \text{definition of } \mathit{three}\ \} \\ & 3 \end{aligned}$$

Evaluation Order

- There can be many other evaluation orders. As we have seen, some terminates while some do not.
- *normal form*: an expression that cannot be reduced anymore.
    - 49 is in normal form, while 7×7 is not.
    - Some expressions do not have a normal form. E.g. *infinity*.
- A corollary of the Church–Rosser theorem: an expression has at most one normal form.
    - If two evaluation sequences both terminate, they reach the same normal form.

Evaluation Order

- Applicative order evaluation: starting with the innermost reducible expression (a redex).
- Normal order evaluation: starting with the outermost redex.
- If an expression has a normal form, normal order evaluation delivers it. Hence the name.
- For now you can imagine that Haskell uses normal order evaluation. A way to implement normal order evaluation is called *lazy evaluation*.

Forced Computation

How to evaluate $\mathit{positive}\ (3 + 4)$?

$$\begin{aligned} \mathit{positive}\ 0 &= 1 \\ \mathit{positive}\ n &= n \times n \quad . \end{aligned}$$

There is only one way:

$$\begin{aligned} & \mathit{positive}\ (3 + 4) \\ = \;& \mathit{positive}\ 7 \\ = \;& 7 \times 7 = 49 \quad . \end{aligned}$$

Huh? Shouldn't the outermost identifier be expanded and reduced in normal order evaluation?

Forced Computation

In this case, the outermost redex is not *postive*, but $3 + 4$.

An inner sub-expression is the redex when we need to examine its value to determine how to carry on.

This could happen at the site of

- pattern matching,
- guarded expressions,
- **case** expressions,
- some built-in functions such as $(<)$, $(\leq)$, etc...

Booleans

The datatype *Bool* can be introduced with a *datatype declaration*:

$$\textbf{data}\ \mathit{Bool}\ =\ \mathit{False} \mid \mathit{True}$$

(But you need not do so. The type *Bool* is already defined in the Haskell Prelude.)

Datatype Declaration

- In Haskell, a **data** declaration defines a new type.

$$
\begin{aligned}
\textbf{data}\ \mathit{Type} &= \mathit{Con}_1\ \mathit{Type}_{11}\ \mathit{Type}_{12} \ldots \\
&\mid \mathit{Con}_2\ \mathit{Type}_{21}\ \mathit{Type}_{22} \ldots \\
&\mid \quad \vdots
\end{aligned}
$$

- The declaration above introduces a new type, *Type*, with several cases.
- Each case starts with a constructor, and several (zero or more) arguments (also types).
- Informally it means "a value of type *Type* is either a Con_1 with arguments Type_{11}, Type_{12}..., or a Con_2 with arguments Type_{21}, Type_{22}..."
- Types and constructors begin in capital letters.

Functions on Booleans

Negation:

$$\begin{aligned} &\mathit{not} && :: \mathit{Bool} \rightarrow \mathit{Bool} \\ &\mathit{not\ False} && = \mathit{True} \\ &\mathit{not\ True} && = \mathit{False} \end{aligned}$$

- Notice the definition by *pattern matching*. The definition has two cases, because *Bool* is defined by two cases. The shape of the function follows the shape of its argument.

Functions on Booleans

Conjunction and disjunction:

$$\begin{aligned}
(\wedge),(\vee) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{False} \wedge x &= \mathit{False} \\
\mathit{True} \wedge x &= x \\
\mathit{False} \vee x &= x \\
\mathit{True} \vee x &= \mathit{True}
\end{aligned}$$

I use the symbols $\wedge$ and $\vee$ due to mathematical convension. In your Haskell code, $\wedge$ should be written `&&`, and $\vee$ should be `||`.

Functions on Booleans

Equality check:

$$\begin{aligned} (\equiv), (\neq) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\ x \equiv y &= (x \wedge y) \vee (\mathit{not}\ x \wedge \mathit{not}\ y) \\ x \neq y &= \mathit{not}\ (x \equiv y) \end{aligned}$$

- $=$ is a definition, while $\equiv$ is a function.
- $\equiv$ and $\neq$ are written respectively written $==$ and $/=$ in ASCII.

Characters

- You can think of *Char* as a big **data** definition:

  $$\textbf{data}\ Char\ =\ \texttt{'}a\texttt{'} \mid \texttt{'}b\texttt{'} \mid \ldots$$

  with functions:

  $$\begin{aligned} &ord\ :: Char \to Int \\ &chr\ :: Int \to Char \end{aligned}$$

- Characters are compared by their order:

  $$\begin{aligned} &isDigit\ \ :: Char \to Bool \\ &isDigit\,x = \texttt{'}0\texttt{'} \leq x \land x \leq \texttt{'}9\texttt{'} \end{aligned}$$

Equality Check

- Of course, you can test equality of characters too:

  $$(==) :: Char \to Char \to Bool$$

- $(==)$ is an *overloaded* name — one name shared by many different definitions of equalities, for different types:
  - $(==) :: Int \to Int \to Bool$
  - $(==) :: (Int, Char) \to (Int, Char) \to Bool$
  - $(==) :: [Int] \to [Int] \to Bool$ ...
- Haskell deals with overloading by a general mechanism called *type classes*. It is considered a major feature of Haskell.
- While the type class is an interesting topic, we might not cover much of it since it is orthogonal to the central message of this course.

Tuples

- The polymorphic type (a, b) is essentially the same as the following declaration:

$$\textbf{data}\ \mathit{Pair}\ a\ b = \mathit{MkPair}\ a\ b$$

- Or, had Haskell allow us to use symbols:

$$\textbf{data}\ (a, b) = (a, b)$$

- Two projections:

$$\begin{aligned} &\mathit{fst} && :: (a, b) \to a \\ &\mathit{fst}\ (a, b) && = a \\ &\mathit{snd} && :: (a, b) \to b \\ &\mathit{snd}\ (a, b) && = b \end{aligned}$$

Lists in Haskell

- Traditionally an important datatype in functional languages.
- In Haskell, all elements in a list must be of the same type.
    - $[1, 2, 3, 4] :: [Int]$
    - $[True, False, True] :: [Bool]$
    - $[[1, 2], [], [6, 7]] :: [[Int]]$
    - $[] :: List\ a$, the empty list (whose element type is not determined).
- $String$ is an abbreviation for $[Char]$; "abcd" is an abbreviation of $['a', 'b', 'c', 'd']$.

List as a Datatype

- $[] :: List\ a$ is the empty list whose element type is not determined.
- If a list is non-empty, the leftmost element is called its *head* and the rest its *tail*.
- The constructor $(:) :: a \to List\ a \to List\ a$ builds a list. E.g. in $x : xs$, x is the head and xs the tail of the new list.
- You can think of a list as being defined by

$$\textbf{data}\ List\ a\ =\ []\ |\ a : List\ a$$

- $[1, 2, 3]$ is an abbreviation of $1 : (2 : (3 : []))$.

Head and Tail

- $head :: List\ a \to a$. e.g. $head\ [1,2,3] = 1$.
- $tail :: List\ a \to List\ a$. e.g. $tail\ [1,2,3] = [2,3]$.
- $init :: List\ a \to List\ a$. e.g. $init\ [1,2,3] = [1,2]$.
- $last :: List\ a \to a$. e.g. $last\ [1,2,3] = 3$.
- They are all partial functions on non-empty lists. e.g. $head\ [] = \bot$.
- $null :: List\ a \to Bool$ checks whether a list is empty.

List Generation

- [0..25] generates the list [0, 1, 2..25].
- [0, 2..25] yields [0, 2, 4..24].
- [2..0] yields [].
- The same works for all *ordered* types. For example *Char*:
    - ['a'..'z'] yields ['a', 'b', 'c'..'z'].
- [1..] yields the *infinite* list [1, 2, 3..].

List Comprehension

- Some functional languages provide a convenient notation for list generation. It can be defined in terms of simpler functions.
- e.g. $[x \times x \mid x \leftarrow [1..5], \mathit{odd}\ x] = [1, 9, 25]$.
- Syntax: $[e \mid Q_1, Q_2..]$. Each Q_i is either
    - a generator $x \leftarrow xs$, where x is a (local) variable or pattern of type a while xs is an expression yielding a list of type $\mathit{List}\ a$, or
    - a guard, a boolean valued expression (e.g. $\mathit{odd}\ x$).
    - e is an expression that can involve new local variables introduced by the generators.

List Comprehension

Examples:

- $[(a, b) \mid a \leftarrow [1..3], b \leftarrow [1..2]] =$ $[(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]$
- $[(a, b) \mid b \leftarrow [1..2], a \leftarrow [1..3]] =$ $[(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2)]$
- $[(i, j) \mid i \leftarrow [1..4], j \leftarrow [i + 1..4]] =$ $[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]$
- $[(i, j) \mid \leftarrow [1..4], \mathit{even}\ i, j \leftarrow [i + 1..4], \mathit{odd}\ j] = [(2, 3)]$

Two Modes of Programming

- Functional programmers switch between two modes of programming.
    - Inductive/recursive mode: go into the structure of the input data and recursively process it.
    - Combinatorial mode: compose programs using existing functions (combinators), process the input in stages.
- We will try the latter style today. However, that means we have to familiarise ourselves to a large collection of library functions.
- In the next lecture we will talk about how these library functions can be defined, in the former style.

Length and Indexing

- $(!!) :: \mathit{List}\ a \rightarrow \mathit{Int} \rightarrow a$. List indexing starts from zero. e.g. $[1, 2, 3]!!0 = 1$.
- $\mathit{length} :: \mathit{List}\ a \rightarrow \mathit{Int}$. e.g. $\mathit{length}\ [0..9] = 10$.

Append and Concatenation

- Append: $(+\!\!+) :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$. In ASCII one types $(++)$.
    - $[1,2] +\!\!+ [3,4,5] = [1,2,3,4,5]$
    - $[] +\!\!+ [3,4,5] = [3,4,5] = [3,4,5] +\!\!+ []$
- Compare with $(:) :: a \to \mathit{List}\ a \to \mathit{List}\ a$. It is a type error to write $[] : [3,4,5]$. $(+\!\!+)$ is defined in terms of $(:)$.
- $\mathit{concat} :: \mathit{List}\ (\mathit{List}\ a) \to \mathit{List}\ a$.
    - e.g. $\mathit{concat}\ [[1,2],[],[3,4],[5]] = [1,2,3,4,5]$.
    - concat is defined in terms of $(+\!\!+)$.

Take and Drop

- $take\ n$ takes the first n elements of the list.
    - For example, $take\ 0\ xs = []$
    - $take\ 3\ \text{"abcde"} = \text{"abc"}$
    - $take\ 3\ \text{"ab"} = \text{"ab"}$
- Working with infinite list: $take\ 5\ [1..] = [1, 2, 3, 4, 5]$. Thanks to normal order (lazy) evaluation.
- Dually, $drop\ n$ drops the first n elements of the list.
    - For example, $drop\ 0\ xs = xs$
    - $drop\ 3\ \text{"abcde"} = \text{"cd"}$
    - $drop\ 3\ \text{"ab"} = \text{""}$
- $take\ n\ xs +\!\!+\ drop\ n\ xs = xs$, as long as $n \neq \perp$.

Map and λ

- $map :: (a \to b) \to List\ a \to List\ b$. e.g. $map\ (1+)\ [1, 2, 3, 4, 5] = [2, 3, 4, 5, 6]$.
- $map\ square\ [1, 2, 3, 4] = [1, 4, 9, 16]$.
- Every once in a while you may need a small function which you do not want to give a name to. At such moments you can use the λ notation:
    - $map\ (\lambda x \to x \times x)\ [1, 2, 3, 4] = [1, 4, 9, 16]$
    - In ASCII λ is written $\backslash$.
- λ is an important primitive notion. We will talk more about it later.

Filter

- $filter :: (a \to Bool) \to List\ a \to List\ a$.
    - e.g. $filter\ even\ [2,7,4,3] = [2,4]$
    - $filter\ (\lambda n \to n\ `mod`\ 3 == 0)\ [3,2,6,7] = [3,6]$
- Application: count the number of occurrences of $'a'$ in a list:
    - $length \cdot filter\ ('a' ==)$
    - Or $length \cdot filter\ (\lambda x \to 'a' == x)$
- **Note** a list comprehension can always be translated into a combination of primitive list generators and *map*, *filter*, and *concat*.

Zip

- $zip :: List\ a \rightarrow List\ b \rightarrow List\ (a, b)$
- e.g. zip "abcde" $[1, 2, 3] = [('a', 1), ('b', 2), ('c', 3)]$
- The length of the resulting list is the length of the shorter input list.

Positions

- Exercise: define $positions :: Char \to String \to List\ Int$, such that $positions\ x\ xs$ returns the positions of occurrences of x in xs. E.g. $positions$ `'o'` `"roodo"` $= [1, 2, 4]$.
- $positions\ x\ xs = map\ snd\ (filter\ ((x ==) \cdot fst)\ (zip\ xs\ [0..]))$
- Or,
  $positions\ x\ xs = map\ snd\ (filter\ (\lambda(y, i) \to x == y)\ (zip\ xs\ [0..]))$
- What if you want only the position of the *first* occurrence of x?

$$\begin{aligned} pos &:: Char \to String \to Int \\ pos\ x\ xs &= head\ (positions\ x\ xs) \end{aligned}$$

  - Due to lazy evaluation (normal order evaluation), positions of the other occurrences are *not* evaluated!

Morals of the Story

- Lazy evaluation helps to improve modularity.
    - List combinators can be conveniently re-used. Only the relevant parts are computed.
- The combinator style encourages “wholemeal programming”.
    - Think of aggregate data as a whole, and process them as a whole!

λ EXPRESSIONS

- $\lambda x \rightarrow e$ denotes a function whose argument is x and whose body is e.
- $(\lambda x \rightarrow e_1)\ e_2$ denotes the function $(\lambda x \rightarrow e_1)$ applied to e_2. It can be reduced to e_1 with its *free* occurrences of x replaced by e_2.
- E.g.

$$\begin{array}{rl} & (\lambda x \rightarrow x \times x)\ (3+4) \\ = & (3+4) \times (3+4) \\ = & 49 \quad . \end{array}$$

- λ expression is a primitive and essential notion. Many other constructs can be seen as syntax sugar of λ's.
- For example, our previous definition of *square* can be seen as an abbreviation of

$$\begin{aligned}&\mathit{square} :: \mathit{Int} \to \mathit{Int}\\ &\mathit{square} = \lambda x \to x \times x \ .\end{aligned}$$

  - Indeed, *square* is merely a value that happens to be a function, which is in turn given by a λ expression.
- λ's are like all values — they can appear inside an expression, be passed as parameters, returned as results, etc.
- In fact, it is possible to build a complete programming language consisting of only λ expressions and applications. Look up "λ calculus".

- $\lambda x \rightarrow \lambda y \rightarrow e$ is abbreviated to $\lambda x\, y \rightarrow e$.
- The following definitions are all equivalent:

$$\begin{aligned}
\mathit{smaller}\ x\ y &= \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller}\ x &= \lambda y \rightarrow \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller} &= \lambda x \rightarrow \lambda y \rightarrow \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y \\
\mathit{smaller} &= \lambda x\, y \rightarrow \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y\ .
\end{aligned}$$

Replacing Constructors

- The function *foldr* is among the most important functions on lists.

$$foldr :: (a \to b \to b) \to b \to List\ a \to b$$

- One way to look at $foldr\ (\oplus)\ e$ is that it replaces $[\,]$ with e and $(:)$ with $(\oplus)$:

$$\begin{aligned} & foldr\ (\oplus)\ e\ [1,2,3,4] \\ = \ & foldr\ (\oplus)\ e\ (1 : (2 : (3 : (4 : [\,])))) \\ = \ & 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))). \end{aligned}$$

- $sum = foldr\ (+)\ 0$.
- One can see that $id = foldr\ (:)\ [\,]$.

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:

- Function *prod* returns the product of a list:

- Function *and* returns the conjunction of a list:

- Lets emphasise again that *id* on lists is a fold:

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:
    - *maximum* $=$ *foldr max* -∞.
- Function *prod* returns the product of a list:
- Function *and* returns the conjunction of a list:
- Lets emphasise again that *id* on lists is a fold:

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:
  - $maximum = foldr\ max$ -∞.
- Function *prod* returns the product of a list:
  - $prod = foldr\ (\times)\ 1$.
- Function *and* returns the conjunction of a list:
- Lets emphasise again that *id* on lists is a fold:

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:
  - $maximum = foldr\ max\ \text{-}\infty$.
- Function *prod* returns the product of a list:
  - $prod = foldr\ (\times)\ 1$.
- Function *and* returns the conjunction of a list:
  - $and = foldr\ (\wedge)\ True$.
- Lets emphasise again that *id* on lists is a fold:

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:
  - $maximum = foldr\ max\ \text{-}\infty$.
- Function *prod* returns the product of a list:
  - $prod = foldr\ (\times)\ 1$.
- Function *and* returns the conjunction of a list:
  - $and = foldr\ (\wedge)\ True$.
- Lets emphasise again that *id* on lists is a fold:
  - $id = foldr\ (:)\ [\,]$.

Some Slightly Complex Folds

- $length = foldr\ (\lambda x\ n \rightarrow 1 + n)\ 0$.
- $map\ f = foldr\ (\lambda x\ xs \rightarrow f\ x : xs)\ [\,]$.
- $xs +\!\!+\ ys = foldr\ (:)\ ys\ xs$. Compare this with *id*!
- $filter\ p = foldr\ (fil\ p)\ [\,]$
  where $fil\ p\ x\ xs = \mathbf{if}\ p\ x\ \mathbf{then}\ (x : xs)\ \mathbf{else}\ xs$.

The Ubiquitous Fold

- In fact, *any* function that takes a list as its input can be written in terms of *foldr* — although it might not be always practical.
- With fold it comes one of the most important theorem in program calculation — the fold-fusion theorem. We will talk about it later.

Fold-Left

There is another, sometimes useful fold on lists: *foldl*.

$$foldl :: (b \to a \to b) \to b \to List\ a \to b \quad .$$

One may see from its type that it brackets the elements of the given list from the different direction:

$$\begin{aligned} & foldl\ (\oplus)\ e\ [1,2,3,4] \\ = \quad & (((e \oplus 1) \oplus 2) \oplus 3) \oplus 4. \end{aligned}$$

It has advantages for some applications. We will talk about it in the last lecture.

Definition and Proof by Induction

Total Functional Programming

- The next few lectures concerns inductive definitions and proofs of datatypes and programs.
- While Haskell provides allows one to define nonterminating functions, infinite data structures, for now we will only consider its total, finite fragment.
- That is, we temporarily
    - consider only finite data structures,
    - demand that functions terminate for all value in its input type, and
    - provide guidelines to construct such functions.
- Infinite datatypes and non-termination will be discussed later in this course.

The So-Called "Mathematical Induction"

- Let P be a predicate on natural numbers.
- We've all learnt this principle of proof by induction: to prove that P holds for all natural numbers, it is sufficient to show that
  - $P\,0$ holds;
  - $P\,(1 + n)$ holds provided that $P\,n$ does.

Proof by Induction on Natural Numbers

- We can see the above inductive principle as a result of seeing natural numbers as defined by the datatype [1]

$$\mathbf{data}\ \mathit{Nat} \ = \ 0 \mid \mathbf{1}_{+}\ \mathit{Nat} \ .$$

- That is, any natural number is either 0, or $\mathbf{1}_{+}\ n$ where n is a natural number.
- In this lecture, $\mathbf{1}_{+}$ is written in bold font to emphasise that it is a data constructor (as opposed to the function $(+)$, to be defined later, applied to a number 1).

[1] Not a real Haskell definition.

A Proof Generator

Given $P\,0$ and $P\,n \Rightarrow P\,(1_+\,n)$, how does one prove, for example, $P\,3$?

$$
\begin{array}{ll}
 & P\,(1_+\,(1_+\,(1_+\,0))) \\
\Leftarrow & \{\ P\,(1_+\,n) \Leftarrow P\,n\ \} \\
 & P\,(1_+\,(1_+\,0)) \\
\Leftarrow & \{\ P\,(1_+\,n) \Leftarrow P\,n\ \} \\
 & P\,(1_+\,0) \\
\Leftarrow & \{\ P\,(1_+\,n) \Leftarrow P\,n\ \} \\
 & P\,0\,.
\end{array}
$$

Having done math. induction can be seen as having designed *a program that generates a proof* — given any $n :: Nat$ we can generate a proof of $P\,n$ in the manner above.

Inductively Defined Functions

- Since the type *Nat* is defined by two cases, it is natural to define functions on *Nat* following the structure:

$$\begin{aligned} &exp &&:: Nat \to Nat \to Nat \\ &exp\ b\ 0 &&= 1 \\ &exp\ b\ (\mathbf{1}_{+}\ n) &&= b \times exp\ b\ n \quad . \end{aligned}$$

- Even addition can be defined inductively

$$\begin{aligned} &(+) &&:: Nat \to Nat \to Nat \\ &0 + n &&= n \\ &(\mathbf{1}_{+}\ m) + n &&= \mathbf{1}_{+}\ (m + n) \quad . \end{aligned}$$

- Exercise: define $(\times)$?

A Value Generator

Given the definition of exp, how does one compute $exp\ b\ 3$?

$$
\begin{array}{ll}
 & exp\ b\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ 0))) \\
= & \quad \{\ \text{definition of } exp\ \} \\
 & b \times exp\ b\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ 0)) \\
= & \quad \{\ \text{definition of } exp\ \} \\
 & b \times b \times exp\ b\ (\mathbf{1}_{+}\ 0) \\
= & \quad \{\ \text{definition of } exp\ \} \\
 & b \times b \times b \times exp\ b\ 0 \\
= & \quad \{\ \text{definition of } exp\ \} \\
 & b \times b \times b \times 1 \quad .
\end{array}
$$

It is a program that generates a value, for any $n :: Nat$.
Compare with the proof of P above.

Moral: Proving is Programming

An inductive proof is a program that generates a proof for any given natural number.

An inductive program follows the same structure of an inductive proof.

Proving and programming are very similar activities.

Without the $n + k$ Pattern

- Unfortunately, newer versions of Haskell abandoned the "$n + k$ pattern" used in the previous slides:

$$\begin{aligned} &exp &&:: Int \to Int \to Int \\ &exp\ b\ 0 &&= 1 \\ &exp\ b\ n &&= b \times exp\ b\ (n - 1)\ . \end{aligned}$$

- *Nat* is defined to be *Int* in `MiniPrelude.hs`. Without `MiniPrelude.hs` you should use *Int*.
- For the purpose of this course, the pattern $1 + n$ reveals the correspondence between *Nat* and lists, and matches our proof style. Thus we will use it in the lecture.
- Remember to remove them in your code.

Proof by Induction

- To prove properties about Nat, we follow the structure as well.
- E.g. to prove that $exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n$.
- One possibility is to preform induction on m. That is, prove $P\,m$ for all $m :: Nat$, where $P\,m \equiv (\forall n :: exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n)$.

Proof by Induction

Recall $P\, m \equiv (\forall n :: exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n)$.

Case $m := 0$. For all n, we reason:

$$exp\ b\ (0 + n)$$

Proof by Induction

Recall $P\,m \equiv (\forall n :: exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n)$.

Case $m := 0$. For all n, we reason:

$$\begin{aligned} & exp\ b\ (0 + n) \\ = & \quad \{\ \text{defn. of } (+)\ \} \\ & exp\ b\ n \end{aligned}$$

Proof by Induction

Recall $P\,m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := 0$. For all n, we reason:

$$
\begin{array}{ll}
 & \mathit{exp}\ b\ (0 + n) \\
= & \quad \{\ \text{defn. of}\ (+)\ \} \\
 & \mathit{exp}\ b\ n \\
= & \quad \{\ \text{defn. of}\ (\times)\ \} \\
 & 1 \times \mathit{exp}\ b\ n
\end{array}
$$

Proof by Induction

Recall $P\, m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := 0$. For all n, we reason:

$$\begin{array}{ll}
& \mathit{exp}\ b\ (0 + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
& \mathit{exp}\ b\ n \\
= & \quad \{\ \text{defn. of } (\times)\ \} \\
& 1 \times \mathit{exp}\ b\ n \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
& \mathit{exp}\ b\ 0 \times \mathit{exp}\ b\ n \quad .
\end{array}$$

We have thus proved $P\, 0$.

Proof by Induction

Recall $P\, m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := \mathbf{1}_+\ m$. For all n, we reason:

$$\mathit{exp}\ b\ ((\mathbf{1}_+\ m) + n)$$

Proof by Induction

Recall $P\,m \equiv (\forall n :: exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n)$.

Case $m := \mathbf{1}_{+}\ m$. For all n, we reason:

$$\begin{aligned}
& exp\ b\ ((\mathbf{1}_{+}\ m) + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
& exp\ b\ (\mathbf{1}_{+}\ (m + n))
\end{aligned}$$

Proof by Induction

Recall $P\ m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := \mathbf{1}_{+}\ m$. For all n, we reason:

$$
\begin{array}{ll}
 & \mathit{exp}\ b\ ((\mathbf{1}_{+}\ m) + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
 & \mathit{exp}\ b\ (\mathbf{1}_{+}\ (m + n)) \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
 & b \times \mathit{exp}\ b\ (m + n)
\end{array}
$$

Proof by Induction

Recall $P\ m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := \mathbf{1}_{+}\ m$. For all n, we reason:

$$\begin{array}{ll}
 & \mathit{exp}\ b\ ((\mathbf{1}_{+}\ m) + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
 & \mathit{exp}\ b\ (\mathbf{1}_{+}\ (m + n)) \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
 & b \times \mathit{exp}\ b\ (m + n) \\
= & \quad \{\ \text{induction}\ \} \\
 & b \times (\mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)
\end{array}$$

Proof by Induction

Recall $P\,m \equiv (\forall n :: \mathit{exp}\ b\ (m + n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := \mathbf{1}_{+}\ m$. For all n, we reason:

$$
\begin{array}{ll}
 & \mathit{exp}\ b\ ((\mathbf{1}_{+}\ m) + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
 & \mathit{exp}\ b\ (\mathbf{1}_{+}\ (m + n)) \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
 & b \times \mathit{exp}\ b\ (m + n) \\
= & \quad \{\ \text{induction}\ \} \\
 & b \times (\mathit{exp}\ b\ m \times \mathit{exp}\ b\ n) \\
= & \quad \{\ (\times)\ \text{associative}\ \} \\
 & (b \times \mathit{exp}\ b\ m) \times \mathit{exp}\ b\ n
\end{array}
$$

Proof by Induction

Recall $P\,m \equiv (\forall n :: \mathit{exp}\ b\ (m+n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n)$.

Case $m := \mathbf{1}_+\ m$. For all n, we reason:

$$
\begin{aligned}
& \mathit{exp}\ b\ ((\mathbf{1}_+\ m) + n) \\
= & \quad \{\ \text{defn. of } (+)\ \} \\
& \mathit{exp}\ b\ (\mathbf{1}_+\ (m+n)) \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
& b \times \mathit{exp}\ b\ (m+n) \\
= & \quad \{\ \text{induction}\ \} \\
& b \times (\mathit{exp}\ b\ m \times \mathit{exp}\ b\ n) \\
= & \quad \{\ (\times) \text{ associative}\ \} \\
& (b \times \mathit{exp}\ b\ m) \times \mathit{exp}\ b\ n \\
= & \quad \{\ \text{defn. of } \mathit{exp}\ \} \\
& \mathit{exp}\ b\ (\mathbf{1}_+\ m) \times \mathit{exp}\ b\ n \quad .
\end{aligned}
$$

We have thus proved $P\ (\mathbf{1}_+\ m)$, given $P\ m$.

Structure Proofs by Programs

- The inductive proof could be carried out smoothly, because both $(+)$ and exp are defined inductively on its lefthand argument (of type Nat).
- The structure of the proof follows the structure of the program, which in turns follows the structure of the datatype the program is defined on.

Lists and Natural Numbers

- We have yet to prove that $(\times)$ is associative.
- The proof is quite similar to the proof for associativity of $(+\!\!+)$, which we will talk about later.
- In fact, *Nat* and lists are closely related in structure.
- Most of us are used to think of numbers as atomic and lists as structured data. Neither is necessarily true.
- For the rest of the course we will demonstrate induction using lists, while taking the properties for *Nat* as given.

An Inductively Defined Set?

- For a set to be “inductively defined”, we usually mean that it is the *smallest* fixed-point of some function.
- What does that maen?

Fixed-Point and Prefixed-Point

- A *fixed-point* of a function f is a value x such that $f\,x = x$.
- **Theorem**. f has fixed-point(s) if f is a *monotonic function* defined on a complete lattice.
  - In general, given f there may be more than one fixed-point.
- A *prefixed-point* of f is a value x such that $f\,x \leq x$.
  - Apparently, all fixed-points are also prefixed-points.
- **Theorem**. the smallest prefixed-point is also the smallest fixed-point.

Example: *Nat*

- Recall the usual definition: *Nat* is defined by the following rules:
  1. 0 is in *Nat*;
  2. if n is in *Nat*, so is $\mathbf{1}_{+}\ n$;
  3. there is no other *Nat*.
- If we define a function F from sets to sets: $F\,X = \{0\} \cup \{\mathbf{1}_{+}\ n \mid n \in X\}$, 1. and 2. above means that $F\,Nat \subseteq Nat$. That is, *Nat* is a prefixed-point of F.
- 3. means that we want the *smallest* such prefixed-point.
- Thus *Nat* is also the least (smallest) fixed-point of F.

Least Prefixed-Point

Formally, let $F\,X = \{0\} \cup \{\mathbf{1}_+\, n \mid n \in X\}$, Nat is a set such that

$$F\,Nat \subseteq Nat \quad , \tag{1}$$

$$(\forall X : F\,X \subseteq X \;\Rightarrow\; Nat \subseteq X) \quad , \tag{2}$$

where (1) says that Nat is a prefixed-point of F, and (2) it is the least among all prefixed-points of F.

Mathematical Induction, Formally

- Given property P, we also denote by P the set of elements that satisfy P.
- That $P\,0$ and $P\,n \Rightarrow P\,(\mathbf{1}_{+} n)$ is equivalent to $\{0\} \subseteq P$ and $\{\mathbf{1}_{+}\,n \mid n \in P\} \subseteq P$,
- which is equivalent to $F\,P \subseteq P$. That is, P is a prefixed-point!
- By (2) we have $\mathit{Nat} \subseteq P$. That is, all Nat satisfy P!
- This is "why mathematical induction is correct."

COINDUCTION?

There is a dual technique called *coinduction* where, instead of least prefixed-points, we talk about *greatest postfixed points*. That is, largest x such that $x \leq f x$.

With such construction we can talk about infinite data structures.

Inductively Defined Lists

- Recall that a (finite) list can be seen as a datatype defined by: [2]

$$\mathbf{data}\ \mathit{List}\ a \;=\; [\,] \mid a : \mathit{List}\ a \quad .$$

- Every list is built from the base case $[\,]$, with elements added by $(:)$ one by one: $[1, 2, 3] = 1 : (2 : (3 : [\,]))$.

[2] Not a real Haskell definition.

All Lists Today are Finite

But what about infinite lists?

- For now let's consider finite lists only, as having infinite lists make the *semantics* much more complicated. [3]
- In fact, all functions we talk about today are total functions. No $\perp$ involved.

[3]What does that mean? We will talk about it later.

Set-Theoretically Speaking...

The type *List a* is the *smallest* set such that

1. $[\,]$ is in *List a*;
2. if xs is in *List a* and x is in a, $x : xs$ is in *List a* as well.

Inductively Defined Functions on Lists

- Many functions on lists can be defined according to how a list is defined:

$$\begin{aligned}
&\mathit{sum} && :: \mathit{List}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sum}\ [\,] && = 0\\
&\mathit{sum}\ (x : \mathit{xs}) && = x + \mathit{sum}\ \mathit{xs} \quad .
\end{aligned}$$

$$\begin{aligned}
&\mathit{map} && :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b\\
&\mathit{map}\ f\ [\,] && = [\,]\\
&\mathit{map}\ f\ (x : \mathit{xs}) && = FX : \mathit{map}\ f\ \mathit{xs} \quad .
\end{aligned}$$

List Append

- The function $(+\!\!+)$ appends two lists into one

$$\begin{aligned}
(+\!\!+) &:: \mathit{List}\ a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
[\,] +\!\!+ ys &= ys \\
(x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys) \quad .
\end{aligned}$$

- Compare the definition with that of $(+)$!

Proof by Structural Induction on Lists

- Recall that every finite list is built from the base case $[\,]$, with elements added by $(:)$ one by one.
- To prove that some property P holds for all finite lists, we show that
  1. $P\ [\,]$ holds;
  2. forall x and xs, $P\ (x : xs)$ holds provided that $P\ xs$ holds.

For a Particular List...

Given $P\,[\,]$ and $P\,xs \Rightarrow P\,(x : xs)$, for all x and xs, how does one prove, for example, $P\,[1,2,3]$?

$$
\begin{array}{ll}
 & P\,(1 : 2 : 3 : [\,]) \\
\Leftarrow & \quad \{\ P\,(x : xs) \Leftarrow P\,xs\ \} \\
 & P\,(2 : 3 : [\,]) \\
\Leftarrow & \quad \{\ P\,(x : xs) \Leftarrow P\,xs\ \} \\
 & P\,(3 : [\,]) \\
\Leftarrow & \quad \{\ P\,(x : xs) \Leftarrow P\,xs\ \} \\
 & P\,[\,]\ .
\end{array}
$$

Appending is Associative

To prove that $xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) = (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$.

Let $P\ xs = (\forall ys, zs :: xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) = (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs)$, we prove P by induction on xs.

Case $xs := [\,]$. For all ys and zs, we reason:

$$\begin{array}{ll}
 & [\,] \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \\
= & \quad \{\ \text{defn. of } (\mathbin{+\!\!+})\ \} \\
 & ys \mathbin{+\!\!+} zs \\
= & \quad \{\ \text{defn. of } (\mathbin{+\!\!+})\ \} \\
 & ([\,] \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \quad .
\end{array}$$

We have thus proved $P\ []$.

Appending is Associative

Case $xs := x : xs$. For all ys and zs, we reason:

$$\begin{array}{ll}
 & (x : xs) \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \\
= & \quad \{\ \text{defn. of } (\mathbin{+\!\!+})\ \} \\
 & x : (xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)) \\
= & \quad \{\ \text{induction}\ \} \\
 & x : ((xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs) \\
= & \quad \{\ \text{defn. of } (\mathbin{+\!\!+})\ \} \\
 & (x : (xs \mathbin{+\!\!+} ys)) \mathbin{+\!\!+} zs \\
= & \quad \{\ \text{defn. of } (\mathbin{+\!\!+})\ \} \\
 & ((x : xs) \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \quad .
\end{array}$$

We have thus proved $P\ (x : xs)$, given $P\ xs$.

Do We Have To Be So Formal?

- In our style of proof, every step is given a reason. Do we need to be so pedantic?
- Being formal *helps* you to do the proof:
    - In the proof of $exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n$, we expect that we will use induction to somewhere. Therefore the first part of the proof is to generate $exp\ b\ (m + n)$.
    - In the proof of associativity, we were working toward generating $xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$.
- By being formal we can work on the *form*, not the *meaning*. Like how we solved the knight/knave problem
- Being formal actually makes the proof easier!
- *Make the symbols do the work.*

Length

- The function *length* defined inductively:

$$\begin{aligned} &\mathit{length} && :: \mathit{List}\ a \to \mathit{Nat} \\ &\mathit{length}\ [\,] && = 0 \\ &\mathit{length}\ (x : xs) && = \mathbf{1}_{+}\ (\mathit{length}\ xs)\ \ . \end{aligned}$$

- Exercise: prove that *length* distributes into $(+\!\!+)$:

$$\mathit{length}\ (xs +\!\!+ ys) = \mathit{length}\ xs + \mathit{length}\ ys$$

Concatenation

- While $(+\!\!+)$ repeatedly applies $(:)$, the function *concat* repeatedly calls $(+\!\!+)$:

$$\begin{aligned}
&\mathit{concat} && :: \mathit{List}\ (\mathit{List}\ a) \rightarrow \mathit{List}\ a\\
&\mathit{concat}\ [\,] && = [\,]\\
&\mathit{concat}\ (\mathit{xs} : \mathit{xss}) && = \mathit{xs} +\!\!+ \mathit{concat}\ \mathit{xss}\quad .
\end{aligned}$$

- Compare with *sum*.
- Exercise: prove $\mathit{sum} \cdot \mathit{concat} = \mathit{sum} \cdot \mathit{map}\ \mathit{sum}$.

Definition by Induction/Recursion

- Rather than giving commands, in functional programming we specify values; instead of performing repeated actions, we define values on inductively defined structures.
- Thus induction (or in general, recursion) is the only "control structure" we have. (We do identify and abstract over plenty of patterns of recursion, though.)
- To inductively define a function f on lists, we specify a value for the base case ($f\,[\,]$) and, assuming that $f\,xs$ has been computed, consider how to construct $f\,(x : xs)$ out of $f\,xs$.

Filter

- *filter p xs* keeps only those elements in *xs* that satisfy *p*.

$$
\begin{aligned}
&\mathit{filter} && :: (a \rightarrow \mathit{Bool}) \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
&\mathit{filter}\ p\ [\,] && = [\,] \\
&\mathit{filter}\ p\ (x : xs) && \mid p\ x = x : \mathit{filter}\ p\ xs \\
& && \mid \mathbf{otherwise} = \mathit{filter}\ p\ xs \quad .
\end{aligned}
$$

Take and Drop

- Recall *take* and *drop*, which we used in the previous exercise.

$$\begin{aligned}
&\mathit{take} && :: \mathit{Nat} \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
&\mathit{take}\ 0\ \mathit{xs} && = [\,] \\
&\mathit{take}\ (\mathbf{1}_{+}\ n)\ [\,] && = [\,] \\
&\mathit{take}\ (\mathbf{1}_{+}\ n)\ (x : \mathit{xs}) && = x : \mathit{take}\ n\ \mathit{xs} \quad .
\end{aligned}$$

$$\begin{aligned}
&\mathit{drop} && :: \mathit{Nat} \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
&\mathit{drop}\ 0\ \mathit{xs} && = \mathit{xs} \\
&\mathit{drop}\ (\mathbf{1}_{+}\ n)\ [\,] && = [\,] \\
&\mathit{drop}\ (\mathbf{1}_{+}\ n)\ (x : \mathit{xs}) && = \mathit{drop}\ n\ \mathit{xs} \quad .
\end{aligned}$$

- Prove: $\mathit{take}\ n\ \mathit{xs} \mathbin{+\!\!+} \mathit{drop}\ n\ \mathit{xs} = \mathit{xs}$, for all n and xs.

TakeWhile and DropWhile

- $takeWhile\ p\ xs$ yields the longest prefix of xs such that p holds for each element.

$$\begin{array}{lll} takeWhile :: (a \to Bool) \to List\ a \to List\ a \\ takeWhile\ p\ [\,] & = [\,] \\ takeWhile\ p\ (x : xs) & \mid p\ x = x : takeWhile\ p\ xs \\ & \mid \mathbf{otherwise} = [\,] \quad . \end{array}$$

- $dropWhile\ p\ xs$ drops the prefix from xs.

$$\begin{array}{lll} dropWhile :: (a \to Bool) \to List\ a \to List\ a \\ dropWhile\ p\ [\,] & = [\,] \\ dropWhile\ p\ (x : xs) & \mid p\ x = dropWhile\ p\ xs \\ & \mid \mathbf{otherwise} = x : xs \quad . \end{array}$$

- Prove: $takeWhile\ p\ xs +\!\!+ dropWhile\ p\ xs = xs$.

List Reversal

- $reverse\ [1, 2, 3, 4] = [4, 3, 2, 1]$.

$$\begin{aligned}
&reverse && :: List\ a \to List\ a \\
&reverse\ [\,] && = [\,] \\
&reverse\ (x : xs) && = reverse\ xs +\!\!+ [x] \quad .
\end{aligned}$$

All Prefixes and Suffixes

- $inits\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$

$$\begin{aligned} &inits && :: List\ a \rightarrow List\ (List\ a) \\ &inits\ [\,] && = [[\,]] \\ &inits\ (x : xs) && = [\,] : map\ (x :)\ (inits\ xs) \quad . \end{aligned}$$

- $tails\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$

$$\begin{aligned} &tails && :: List\ a \rightarrow List\ (List\ a) \\ &tails\ [\,] && = [[\,]] \\ &tails\ (x : xs) && = (x : xs) : tails\ xs \quad . \end{aligned}$$

Totality

- Structure of our definitions so far:

$$\begin{aligned} &f\,[\,] &&= \ldots \\ &f\,(x : xs) &&= \ldots f\,xs \ldots \end{aligned}$$

  - Both the empty and the non-empty cases are covered, guaranteeing there is a matching clause for all inputs.
  - The recursive call is made on a “smaller” argument, guranteeing termination.
- Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

Variations with the Base Case

- Some functions discriminate between several base cases. E.g.

$$\begin{aligned}
&\mathit{fib} && :: \mathit{Nat} \rightarrow \mathit{Nat} \\
&\mathit{fib}\ 0 && = 0 \\
&\mathit{fib}\ 1 && = 1 \\
&\mathit{fib}\ (2+n) && = \mathit{fib}\ (\mathbf{1}_{+} n) + \mathit{fib}\ n \quad .
\end{aligned}$$

- Some functions make more sense when it is defined only on non-empty lists:

$$\begin{aligned} f\,[x] &= \ldots \\ f\,(x : xs) &= \ldots \end{aligned}$$

- What about totality?
  - They are in fact functions defined on a different datatype:

    $$\mathbf{data}\ \mathit{List}^{+}\ a \ = \ \mathit{Singleton}\ a \mid a : \mathit{List}^{+}\ a \quad .$$

  - We do not want to define *map*, *filter* again for $\mathit{List}^{+}\ a$. Thus we reuse *List a* and pretend that we were talking about $\mathit{List}^{+}\ a$.
  - It's the same with *Nat*. We embedded *Nat* into *Int*.
  - Ideally we'd like to have some form of *subtyping*. But that makes the type system more complex.

Lexicographic Induction

- It also occurs often that we perform *lexicographic induction* on multiple arguments: some arguments decrease in size, while others stay the same.
- E.g. the function *merge* merges two sorted lists into one sorted list:

$$
\begin{aligned}
&\mathit{merge} :: \mathit{List\ Int} \to \mathit{List\ Int} \to \mathit{List\ Int}\\
&\mathit{merge}\ [\,]\ [\,] && = [\,]\\
&\mathit{merge}\ [\,]\ (y : ys) && = y : ys\\
&\mathit{merge}\ (x : xs)\ [\,] && = x : xs\\
&\mathit{merge}\ (x : xs)\ (y : ys)\\
&\qquad \mid x \le y = x : \mathit{merge}\ xs\ (y : ys)\\
&\qquad \mid \mathbf{otherwise} = y : \mathit{merge}\ (x : xs)\ ys \quad .
\end{aligned}
$$

Zip

Another example:

$$\begin{aligned}
&\mathit{zip} :: \mathit{List}\ a \to \mathit{List}\ b \to \mathit{List}\ (a, b)\\
&\mathit{zip}\ [\,]\ [\,] && = [\,]\\
&\mathit{zip}\ [\,]\ (y : ys) && = [\,]\\
&\mathit{zip}\ (x : xs)\ [\,] && = [\,]\\
&\mathit{zip}\ (x : xs)\ (y : ys) && = (x, y) : \mathit{zip}\ xs\ ys \quad .
\end{aligned}$$

Non-Structural Induction

- In most of the programs we've seen so far, the recursive call are made on direct sub-components of the input (e.g. $f\ (x : xs) = ..f\ xs..$). This is called *structural induction*.
    - It is relatively easy for compilers to recognise structural induction and determine that a program terminates.
- In fact, we can be sure that a program terminates if the arguments get "smaller" under some (well-founded) ordering.

Mergesort

- In the implemenation of mergesort below, for example, the arguments always get smaller in size.

```
msort      :: List Int → List Int
msort [ ]  = [ ]
msort [x]  = [x]
msort xs   = merge (msort ys) (msort zs)  ,
   where n = length xs ‘div‘ 2
         ys = take n xs
         zs = drop n xs  .
```

  - What if we omit the case for [x]?
- If all cases are covered, and all recursive calls are applied to smaller arguments, the program defines a total function.

A Non-Terminating Definition

- Example of a function, where the argument to the recursive does not reduce in size:

$$\begin{aligned} f &:: Int \rightarrow Int \\ f\,0 &= 0 \\ f\,n &= f\,n \quad . \end{aligned}$$

- Certainly f is not a total function. Do such definitions “mean” something? We will talk about these later.

Internally Labelled Binary Trees

- This is a possible definition of internally labelled binary trees:

$$\textbf{data}\ \mathit{Tree}\ a\ =\ \mathrm{Null} \mid \mathrm{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\quad ,$$

- on which we may inductively define functions:

$$\begin{aligned}
&\mathit{sumT} &&::\ \mathit{Tree}\ \mathit{Nat} \to \mathit{Nat}\\
&\mathit{sumT}\ \mathrm{Null} &&=\ 0\\
&\mathit{sumT}\ (\mathrm{Node}\ x\ t\ u) &&=\ x + \mathit{sumT}\ t + \mathit{sumT}\ u\quad .
\end{aligned}$$

Exercise: given $(\downarrow) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$, which yields the smaller one of its arguments, define the following functions

1. $\mathit{minT} :: \mathit{Tree}\ \mathit{Nat} \to \mathit{Nat}$, which computes the minimal element in a tree.
2. $\mathit{mapT} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b$, which applies the functional argument to each element in a tree.
3. Can you define $(\downarrow)$ inductively on Nat? [4]

[4] In the standard Haskell library, $(\downarrow)$ is called *min*.

Induction Principle for *Tree*

- What is the induction principle for *Tree*?
- To prove that a predicate P on *Tree* holds for every tree, it is sufficient to show that

Induction Principle for *Tree*

- What is the induction principle for *Tree*?
- To prove that a predicate P on *Tree* holds for every tree, it is sufficient to show that
  1. P Null holds, and;
  2. for every x, t, and u, if $P\ t$ and $P\ u$ holds, P (Node $x\ t\ u$) holds.

Induction Principle for *Tree*

- What is the induction principle for *Tree*?
- To prove that a predicate P on *Tree* holds for every tree, it is sufficient to show that
  1. P Null holds, and;
  2. for every x, t, and u, if $P\ t$ and $P\ u$ holds, P (Node $x\ t\ u$) holds.
- Exercise: prove that for all n and t, $\mathit{minT}\ (\mathit{mapT}\ (n+)\ t) = n + \mathit{minT}\ t$. That is, $\mathit{minT} \cdot \mathit{mapT}\ (n+) = (n+) \cdot \mathit{minT}$.

Induction Principle for Other Types

- Recall that **data** $Bool = False \mid True$. Do we have an induction principle for $Bool$?
- To prove a predicate P on $Bool$ holds for all booleans, it is sufficient to show that

Induction Principle for Other Types

- Recall that **data** $Bool = False \mid True$. Do we have an induction principle for $Bool$?
- To prove a predicate P on $Bool$ holds for all booleans, it is sufficient to show that
    1. $P\ False$ holds, and
    2. $P\ True$ holds.
- Well, of course.

- What about $(A \times B)$? How to prove that a predicate P on $(A \times B)$ is always true?
- One may prove some property P_1 on A and some property P_2 on B, which together imply P.
- That does not say much. But the “induction principle” for products allows us to extract, from a proof of P, the proofs P_1 and P_2.

- *Every inductively defined datatype comes with its induction principle.*
- We will come back to this point later.

Program Calculation

A Quick Review

- Functions are the basic building blocks. They may be passed as arguments, may return functions, and can be composed together.
- While one issues commands in an imperative language, in functional programming we specify values, and computers try to reduce the values to their normal forms.
- Formal reasoning: reasoning with the form (syntax) rather than the semantics. Let the symbols do the work!
- ‘Wholemeal’ programming: think of aggregate data as a whole, and process them as a whole.

A Quick Review

- Once you describe the values as algebraic datatypes, most programs write themselves through structural recursion.
- Programs and their proofs are closely related. They share similar structure, by induction over input data.
- Properties of programs can be reasoned about in equations, just like high school algebra.

Data Representation

- So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
- Take lists for example. Recall the definition: **data** *List a* $=$ [ ] | *a* : *List a*.
- Our representation of lists is biased. The left most element can be fetched immediately.
    - Thus. (:), *head*, and *tail* are constant-time operations, while *init* and *last* takes linear-time.
- In most implementations, the list is represented as a linked-list.

List Concatenation Takes Linear Time

- Recall $(+\!\!+)$:

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

List Concatenation Takes Linear Time

- Recall $(+\!\!+)$:

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

List Concatenation Takes Linear Time

- Recall $(+\!\!+)$:

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

- Consider $[1, 2, 3] +\!\!+ [4, 5]$:

$$\begin{aligned} &(1 : 2 : 3 : [\,]) +\!\!+ (4 : 5 : [\,]) \\ =\ & 1 : ((2 : 3 : [\,]) +\!\!+ (4 : 5 : [\,])) \\ =\ & 1 : 2 : ((3 : [\,]) +\!\!+ (4 : 5 : [\,])) \\ =\ & 1 : 2 : 3 : ([\,] +\!\!+ (4 : 5 : [\,])) \\ =\ & 1 : 2 : 3 : 4 : 5 : [\,] \end{aligned}$$

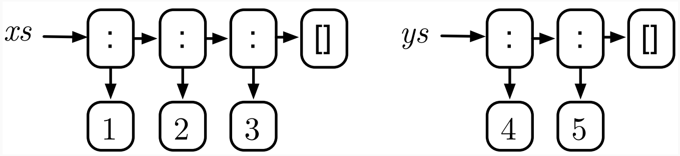
- $(+\!\!+)$ runs in time proportional to the length of its left argument.

Full Persistency

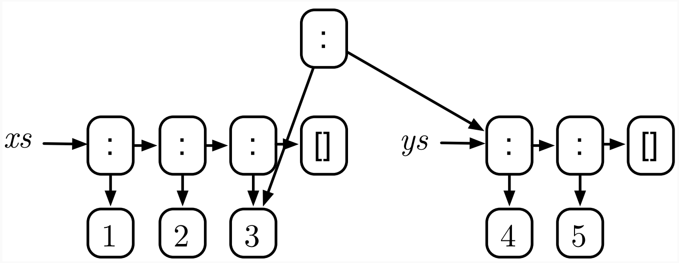
- Compound data structures, like simple values, are just values, and thus must be *fully persistent*.
- That is, in the following code:

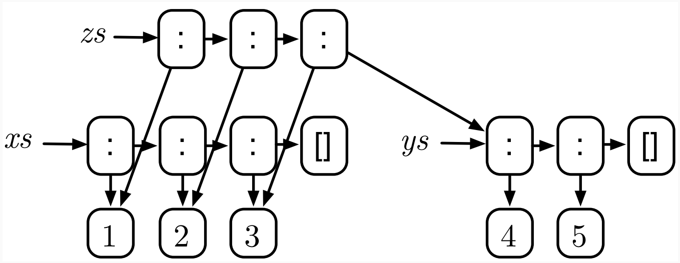
$$\begin{aligned}
\textbf{let}\ & xs = [1,2,3] \\
& ys = [4,5] \\
& zs = xs \mathbin{+\!\!+} ys \\
\textbf{in}\ & \ldots body \ldots
\end{aligned}$$

- The *body* may have access to all three values. Thus $\mathbin{+\!\!+}$ cannot perform a destructive update.



xs
:
:
:
[]
1
2
3
ys
:
:
[]
4
5


:
xs
:
:
:
[]
ys
:
:
[]
1
2
3
4
5



zs
xs
ys
:
[]
1
2
3
4
5

Linked v.s. Block Data Structures

- Trees are usually represented in a similar manner, through links.
- Fully persistency is easier to achieve for such linked data structures.
- Accessing arbitrary elements, however, usually takes linear time.
- In imperative languages, constant-time random access is usually achieved by allocating lists (usually called arrays in this case) in a consecutive block of memory.

Linked v.s. Block Data Structures

- Consider the following code, where *xs* is an array (implemented as a block), and *ys* is like *xs*, apart from its 10th element:

```
let xs = [1..100]
    ys = update xs 10 20
in ...body...
```

- To allow access to both *xs* and *ys* in *body*, the *update* operation has to duplicate the entire array.
- Thus people have invented some smart data structure to do so, in around $O(\log n)$ time.
- On the other hand, *update* may simply overwrite *xs* if we can somehow make sure that *nobody* other than *ys* uses *xs*.
- Both are advanced topics, however.

Another Linear-Time Operation

- Taking all but the last element of a list:

$$\begin{aligned} &init\ [x] &&= \\ &init\ (x : xs) &&= \end{aligned}$$

- Consider $init\ [1, 2, 3, 4]$:

Another Linear-Time Operation

- Taking all but the last element of a list:

$$\begin{aligned} \mathit{init}\ [x] &= [\,] \\ \mathit{init}\ (x : xs) &= x : \mathit{init}\ xs \end{aligned}$$

- Consider $\mathit{init}\ [1, 2, 3, 4]$:

Another Linear-Time Operation

- Taking all but the last element of a list:

$$\begin{aligned} \mathit{init}\ [x] &= [\,] \\ \mathit{init}\ (x : xs) &= x : \mathit{init}\ xs \end{aligned}$$

- Consider $\mathit{init}\ [1, 2, 3, 4]$:

$$\begin{aligned} & \mathit{init}\ (1 : 2 : 3 : 4 : [\,]) \\ = {} & 1 : \mathit{init}\ (2 : 3 : 4 : [\,]) \\ = {} & 1 : 2 : \mathit{init}\ (3 : 4 : [\,]) \\ = {} & 1 : 2 : 3 : \mathit{init}\ (4 : [\,]) \\ = {} & 1 : 2 : 3 : [\,] \end{aligned}$$

Sum, Map, etc

- Functions like *sum*, *maximum*, etc. needs to traverse through the list once to produce a result. So their running time is definitely $O(n)$.
- If f takes time $O(t)$, *map* f takes time $O(n \times t)$ to complete. Similarly with *filter* p.
    - In a lazy setting, *map* f produces its first result in $O(t)$ time. We won't need lazy features for now, however.

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  Specification: $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map}\ \mathit{square}$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\mathit{sumsq}\ [\,]$$

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  Specification: $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map}\ \mathit{square}$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{array}{ll}
 & \mathit{sumsq}\ [\,] \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
 & (\mathit{sum} \cdot \mathit{map}\ \mathit{square})\ [\,]
\end{array}
$$

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map}\ \mathit{square}$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{aligned}
& \mathit{sumsq}\ [\,] \\
= \quad & \{\ \text{definition of } \mathit{sumsq}\ \} \\
& (\mathit{sum} \cdot \mathit{map}\ \mathit{square})\ [\,] \\
= \quad & \{\ \text{function composition}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ [\,])
\end{aligned}
$$

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $sumsq = sum \cdot map\ square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{aligned}
& sumsq\ [\,] \\
= \quad & \{\ \text{definition of } sumsq\ \} \\
& (sum \cdot map\ square)\ [\,] \\
= \quad & \{\ \text{function composition}\ \} \\
& sum\ (map\ square\ [\,]) \\
= \quad & \{\ \text{definition of } map\ \} \\
& sum\ [\,]
\end{aligned}$$

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $sumsq = sum \cdot map\ square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{array}{ll}
 & sumsq\ [\,] \\
= & \{\ \text{definition of } sumsq\ \} \\
 & (sum \cdot map\ square)\ [\,] \\
= & \{\ \text{function composition}\ \} \\
 & sum\ (map\ square\ [\,]) \\
= & \{\ \text{definition of } map\ \} \\
 & sum\ [\,] \\
= & \{\ \text{definition of } sum\ \} \\
 & 0
\end{array}$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

  $$sumsq\ (x : xs)$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$\begin{aligned} & \mathit{sumsq}\ (x : \mathit{xs}) \\ = & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\ & \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : \mathit{xs})) \end{aligned}$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{aligned}
& \mathit{sumsq}\ (x : \mathit{xs}) \\
= &\quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : \mathit{xs})) \\
= &\quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\ (\mathit{square}\ x : \mathit{map}\ \mathit{square}\ \mathit{xs})
\end{aligned}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{aligned}
& \mathit{sumsq}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : \mathit{xs})) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\ (\mathit{square}\ x : \mathit{map}\ \mathit{square}\ \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sum}\ \} \\
& \mathit{square}\ x + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs})
\end{aligned}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{aligned}
& \mathit{sumsq}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : \mathit{xs})) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\ (\mathit{square}\ x : \mathit{map}\ \mathit{square}\ \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sum}\ \} \\
& \mathit{square}\ x + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{square}\ x + \mathit{sumsq}\ \mathit{xs}
\end{aligned}
$$

Alternative Definition for *sumsq*

- From $sumsq = sum \cdot map\ square$, we have proved that

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x : xs) &= square\ x + sumsq\ xs \end{aligned}$$

- Equivalently, we have shown that $sum \cdot map\ square$ is a solution of

$$\begin{aligned} f\ [\,] &= 0 \\ f\ (x : xs) &= square\ x + f\ xs \end{aligned}$$

- However, the solution of the equations above is unique.
- Thus we can take it as another definition of $sumsq$. Denotationally it is the same function; operationally, it is (slightly) quicker.
- Exercise: try calculating an inductive definition of $count$.

How Far Can We Get?

- Specification of maximum segment sum:

$$\begin{aligned}
&\mathit{mss} && :: \mathit{List\ Int} \to \mathit{Int} \\
&\mathit{mss} && = \mathit{maximum} \cdot \mathit{map\ sum} \cdot \mathit{segments} \\
&\mathit{segments} && :: \mathit{List\ a} \to \mathit{List\ (List\ a)} \\
&\mathit{segments} && = \mathit{concat} \cdot \mathit{map\ inits} \cdot \mathit{tails}
\end{aligned}$$

- From the specification we can calculate a linear time algorithm.

Remark: Why Functional Programming?

- Time to muse on the merits of functional programming. Why functional programming?
  - Algebraic datatype? List comprehension? Lazy evaluation? Garbage collection? These are just language features that can be migrated.
  - No side effects.[5] But why taking away a language feature?
- By being pure, we have a simpler semantics in which we are allowed to construct and reason about programs.
  - In an imperative language we do not even have $f\,4 + f\,4 = 2 \times f\,4$.
- Ease of reasoning. That's the main benefit we get.

[5]Unless introduced in a disciplined way.

Steep Lists

- A *steep list* is a list in which every element is larger than the sum of those to its right:

$$\begin{aligned} &\mathit{steep} && :: \mathit{List\ Int} \to \mathit{Bool} \\ &\mathit{steep}\ [\,] && = \mathit{True} \\ &\mathit{steep}\ (x : xs) && = \mathit{steep}\ xs\ \wedge\ x > \mathit{sum}\ xs. \end{aligned}$$

- The definition above, if executed directly, is an $O(n^2)$ program. Can we do better?
- Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

- Recall that $fst\ (a, b) = a$ and $snd\ (a, b) = b$.
- It is hard to quickly compute $steep$ alone. But if we define

$$steepsum\ xs\ = (steep\ xs, sum\ xs),$$

- and manage to synthesise a quick definition of $steepsum$, we can implement $steep$ by $steep = fst \cdot steepsum$.
- We again proceed by case analysis. Trivially,

$$steepsum\ []\ = (True, 0).$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$steepsum\ (x : xs)$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{array}{ll} & \mathit{steepsum}\ (x : xs) \\ = & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\ & (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \end{array}$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\ (x : \mathit{xs}) \\
= \quad & \{\ \text{definition of } \mathit{steepsum}\ \} \\
& (\mathit{steep}\ (x : \mathit{xs}), \mathit{sum}\ (x : \mathit{xs})) \\
= \quad & \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
& (\mathit{steep}\ \mathit{xs} \wedge x > \mathit{sum}\ \mathit{xs}, x + \mathit{sum}\ \mathit{xs})
\end{aligned}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{array}{ll}
 & \mathit{steepsum}\ (x : xs) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
 & (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \\
= & \quad \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
 & (\mathit{steep}\ xs \wedge x > \mathit{sum}\ xs, x + \mathit{sum}\ xs) \\
= & \quad \{\ \text{extracting sub-expressions involving } xs\ \} \\
 & \textbf{let}\ (b, y) = (\mathit{steep}\ xs, \mathit{sum}\ xs) \\
 & \textbf{in}\ (b \wedge x > y, x + y)
\end{array}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{array}{cl}
 & \mathit{steepsum}\ (x : xs) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
 & (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \\
= & \quad \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
 & (\mathit{steep}\ xs \wedge x > \mathit{sum}\ xs, x + \mathit{sum}\ xs) \\
= & \quad \{\ \text{extracting sub-expressions involving } xs\ \} \\
 & \mathbf{let}\ (b, y) = (\mathit{steep}\ xs, \mathit{sum}\ xs) \\
 & \mathbf{in}\ (b \wedge x > y, x + y) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
 & \mathbf{let}\ (b, y) = \mathit{steepsum}\ xs \\
 & \mathbf{in}\ (b \wedge x > y, x + y).
\end{array}
$$

Synthesised Program

We have thus come up with a $O(n)$ time program:

$$\begin{aligned}
steep &= fst \cdot steepsum \\
steepsum\ [\,] &= (True, 0) \\
steepsum\ (x : xs) &= \mathbf{let}\ (b, y) = steepsum\ xs \\
&\quad\ \mathbf{in}\ (b \wedge x > y, x + y),
\end{aligned}$$

Being Quicker by Doing More?

- A more generalised program can be implemented more efficiently?
    - A common phenomena! Sometimes the less general function cannot be implemented inductively at all!
    - It also often happens that a theorem needs to be generalised to be proved. We will see that later.
- An obvious question: how do we know what generalisation to pick?
- There is no easy answer — finding the right generalisation one of the most difficulty act in programming!
- Sometimes we simply generalise by examining the form of the formula.

Reversing a List

- The function *reverse* is defined by:

$$\begin{aligned} \mathit{reverse}\ [\,] &= [\,], \\ \mathit{reverse}\ (x : xs) &= \mathit{reverse}\ xs ++ [x]. \end{aligned}$$

- E.g.
  $\mathit{reverse}\ [1,2,3,4] = ((([\,] ++ [4]) ++ [3]) ++ [2]) ++ [1] = [4,3,2,1]$.
- But how about its time complexity? Since $(++)$ is $O(n)$, it takes $O(n^2)$ time to revert a list this way.
- Can we make it faster?

Introducing an Accumulating Parameter

- Let us consider a generalisation of *reverse*. Define:

$$\begin{aligned} &\mathit{revcat} &&:: [a] \to [a] \to [a] \\ &\mathit{revcat}\ \mathit{xs}\ \mathit{ys} &&= \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} \mathit{ys}. \end{aligned}$$

- If we can construct a fast implementation of *revcat*, we can implement *reverse* by:

$$\mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ [\,].$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$revcat\ [\,]\ ys$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$\begin{array}{cl} & \mathit{revcat}\ [\,]\ \mathit{ys} \\ = & \quad \{\ \text{definition of } \mathit{revcat}\ \} \\ & \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \end{array}$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$
\begin{aligned}
& \mathit{revcat}\ [\,]\ \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& [\,] \mathbin{+\!\!+} \mathit{ys}
\end{aligned}
$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$\begin{array}{cl}
 & \mathit{revcat}\ [\,]\ \mathit{ys} \\
= & \{\ \text{definition of } \mathit{revcat}\ \} \\
 & \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \\
= & \{\ \text{definition of } \mathit{reverse}\ \} \\
 & [\,] \mathbin{+\!\!+} \mathit{ys} \\
= & \{\ \text{definition of } (\mathbin{+\!\!+})\ \} \\
 & \mathit{ys}.
\end{array}$$

Reversing a List, Inductive Case

Case $x : xs$:

$$revcat\ (x : xs)\ ys$$

Reversing a List, Inductive Case

Case $x : xs$:

$$\begin{array}{ll} & \mathit{revcat}\ (x : xs)\ ys \\ = & \quad \{\ \text{definition of } \mathit{revcat}\ \} \\ & \mathit{reverse}\ (x : xs) +\!\!+\ ys \end{array}$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{array}{ll}
& \mathit{revcat}\ (x : xs)\ ys \\
= & \quad \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) +\!\!+\ ys \\
= & \quad \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs +\!\!+ [x]) +\!\!+\ ys
\end{array}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= & \quad \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) +\!\!+\, ys \\
= & \quad \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs +\!\!+ [x]) +\!\!+\, ys \\
= & \quad \{\ \text{since } (xs +\!\!+\, ys) +\!\!+\, zs = xs +\!\!+ (ys +\!\!+\, zs)\ \} \\
& \mathit{reverse}\ xs +\!\!+ ([x] +\!\!+\, ys)
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{since } (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)\ \} \\
& \mathit{reverse}\ xs \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} ys) \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{revcat}\ xs\ (x : ys).
\end{aligned}$$

Linear-Time List Reversal

- We have therefore constructed an implementation of *revcat* which runs in linear time!

$$\begin{aligned} \mathit{revcat}\ []\ \mathit{ys} &= \mathit{ys} \\ \mathit{revcat}\ (x : \mathit{xs})\ \mathit{ys} &= \mathit{revcat}\ \mathit{xs}\ (x : \mathit{ys}). \end{aligned}$$

- A generalisation of *reverse* is easier to implement than *reverse* itself? How come?
- If you try to understand *revcat* operationally, it is not difficult to see how it works.
    - The partially reverted list is *accumulated* in ys.
    - The initial value of ys is set by $\mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ []$.
    - Hmm... it is like a *loop*, isn't it?

Tracing Reverse

$$\begin{aligned}
& \mathit{reverse}\ [1,2,3,4] \\
= \quad & \mathit{revcat}\ [1,2,3,4]\ [\,] \\
= \quad & \mathit{revcat}\ [2,3,4]\ [1] \\
= \quad & \mathit{revcat}\ [3,4]\ [2,1] \\
= \quad & \mathit{revcat}\ [4]\ [3,2,1] \\
= \quad & \mathit{revcat}\ [\,]\ [4,3,2,1] \\
= \quad & [4,3,2,1]
\end{aligned}$$

$$\begin{aligned}
\mathit{reverse}\ xs &= \mathit{revcat}\ xs\ [\,] \\
\mathit{revcat}\ [\,]\ ys &= ys \\
\mathit{revcat}\ (x : xs)\ ys &= \mathit{revcat}\ xs\ (x : ys)
\end{aligned}$$

```
$xs, ys \leftarrow XS, [\,];$
while $xs \neq [\,]$ do
    $xs, ys \leftarrow (\mathit{tail}\ xs), (\mathit{head}\ xs : ys);$
return $ys$
```

Tail Recursion

- Tail recursion: a special case of recursion in which the last operation is the recursive call.

$$f\,x_1\ \dots\ x_n = \{\text{base case}\}$$
$$f\,x_1\ \dots\ x_n = f\,x_1'\ \dots\ x_n'$$

- To implement general recursion, we need to keep a stack of return addresses. For tail recursion, we do not need such a stack.
- Tail recursive definitions are like loops. Each x_i is updated to x_i' in the next iteration of the loop.
- The first call to f sets up the initial values of each x_i.

Accumulating Parameters

- To efficiently perform a computation (e.g. *reverse xs*), we introduce a generalisation with an extra parameter, e.g.:

$$\mathit{revcat}\ \mathit{xs}\ \mathit{ys} = \mathit{reverse}\ \mathit{xs} +\!\!+\ \mathit{ys}.$$

- Try to derive an efficient implementation of the generalised function. The extra parameter is usually used to “accumulate” some results, hence the name.
  - To make the accumulation work, we usually need some kind of associativity.
- A technique useful for, but not limited to, constructing tail-recursive definition of functions.

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x : xs) &= square\ x + sumsq\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $ssp\ xs\ n = \qquad\qquad\qquad .$
- Initialisation: $sumsq\ xs = \qquad\qquad .$
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} &\mathit{sumsq}\ [\,] &&= 0 \\ &\mathit{sumsq}\ (x : xs) &&= \mathit{square}\ x + \mathit{sumsq}\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $\mathit{ssp}\ xs\ n = \mathit{sumsq}\ xs + n$.
- Initialisation: $\mathit{sumsq}\ xs =$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the “sum of squares” problem:

$$\begin{aligned} \mathit{sumsq}\ [\,] &= 0 \\ \mathit{sumsq}\ (x : xs) &= \mathit{square}\ x + \mathit{sumsq}\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $\mathit{ssp}\ xs\ n = \mathit{sumsq}\ xs + n$.
- Initialisation: $\mathit{sumsq}\ xs = \mathit{ssp}\ xs\ 0$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} \mathit{sumsq}\ [\,] &= 0 \\ \mathit{sumsq}\ (x : xs) &= \mathit{square}\ x + \mathit{sumsq}\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $\mathit{ssp}\ xs\ n = \mathit{sumsq}\ xs + n$.
- Initialisation: $\mathit{sumsq}\ xs = \mathit{ssp}\ xs\ 0$.
- Construct ssp:

$$\begin{aligned} \mathit{ssp}\ [\,]\ n &= 0 + n \ = \ n \\ \mathit{ssp}\ (x : xs)\ n &= (\mathit{square}\ x + \mathit{sumsq}\ xs) + n \\ &= \mathit{sumsq}\ xs + (\mathit{square}\ x + n) \\ &= \mathit{ssp}\ xs\ (\mathit{square}\ x + n). \end{aligned}$$

Labelling a List

- Consider the task of labelling elements in a list with its index.

$$\begin{array}{l} \mathit{index} :: \mathit{List}\ a \to \mathit{List}\ (\mathit{Int}, a) \\ \mathit{index} = \mathit{zip}\ [0..] \end{array}$$

- To construct an inductive definition, the case for $[\,]$ is easy. For the $x : xs$ case:

$$\begin{array}{ll} & \mathit{index}\ (x : xs) \\ = & \mathit{zip}\ [0..]\ (x : xs) \\ = & (0, x) : \mathit{zip}\ [1..]\ xs \end{array}$$

- Alas, $\mathit{zip}\ [1..]$ cannot be fold back to index!
- What if we turn the varying part into…a variable?

Labelling a List, Second Attempt

- Introduce $\mathit{idxFrom} :: \mathit{List}\ a \to \mathit{Int} \to \mathit{List}\ (\mathit{Int}, a)$:

  $$\mathit{idxFrom}\ \mathit{xs}\ n = \mathit{zip}\ [n..]\ \mathit{xs}$$

- Initialisation: $\mathit{index}\ \mathit{xs} =$.

Labelling a List, Second Attempt

- Introduce $idxFrom :: List\ a \to Int \to List\ (Int, a)$:

  $$idxFrom\ xs\ n = zip\ [n..]\ xs$$

- Initialisation: $index\ xs = idxFrom\ xs\ 0$.

Labelling a List, Second Attempt

- Introduce $idxFrom :: List\ a \to Int \to List\ (Int, a)$:

$$idxFrom\ xs\ n = zip\ [n..]\ xs$$

- Initialisation: $index\ xs = idxFrom\ xs\ 0$.
- We reason:

$$\begin{aligned} & idxFrom\ (x : xs)\ n \\ = \quad & zip\ [n..]\ (x : xs) \\ = \quad & (n, x) : zip\ [n + 1..]\ xs \\ = \quad & (n, x) : idxFrom\ xs\ (n + 1) \end{aligned}$$

Summing Up a List in Reverse

- Prove: $sum \cdot reverse = sum$, using the definition $reverse\ xs = revcat\ xs\ [\,]$. That is, proving $sum\ (revcat\ xs\ [\,]) = sum\ xs$.
- Base case trivial. For the case $x : xs$:

$$\begin{aligned} & sum\ (reverse\ (x : xs)) \\ = \quad & sum\ (revcat\ (x : xs)\ [\,]) \\ = \quad & sum\ (revcat\ xs\ [x]) \end{aligned}$$

- Then we are stuck, since we cannot use the induction hypothesis $sum\ (revcat\ xs\ [\,]) = sum\ xs$.
- Again, generalise $[x]$ to a variable.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$\mathit{sum}\ (\mathit{revcat}\ \mathit{xs}\ \mathit{ys}) =$$

- By letting $\mathit{ys} = [\,]$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$sum\ (revcat\ (x : xs)\ ys)$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned} & sum\ (revcat\ (x : xs)\ ys) \\ = \quad & sum\ (revcat\ xs\ (x : ys)) \end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned}
& sum\ (revcat\ (x : xs)\ ys) \\
= \quad & sum\ (revcat\ xs\ (x : ys)) \\
= \quad & \{\ \text{induction hypothesis}\ \} \\
& sum\ xs + sum\ (x : ys)
\end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned}
& sum\ (revcat\ (x : xs)\ ys) \\
= \quad & sum\ (revcat\ xs\ (x : ys)) \\
= \quad & \ \{\ \text{induction hypothesis}\ \} \\
& sum\ xs + sum\ (x : ys) \\
= \quad & sum\ xs + x + sum\ ys \\
= \quad & sum\ (x : xs) + sum\ ys
\end{aligned}$$

Work Less by Proving More

- A stronger theorem is easier to prove! Why is that?
- By strengthening the theorem, we also have a stronger induction hypothesis, which makes an inductive proof possible.
    - Finding the right generalisation is an art — it's got to be strong enough to help the proof, yet not too strong to be provable.
- The same with programming. By generalising a function with additional arguments, it is passed more information it may use, thus making an inductive definition possible.
    - The speeding up of *revcat*, in retrospect, is an accidental "side effect" — *revcat*, being inductive, goes through the list only once, and is therefore quicker.

A Real Case

- A property I actually had to prove for a paper:

$$\begin{aligned} & \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{xs})) = \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{win}\ \mathit{xs})) \\ & \quad \Leftarrow\ \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{xs})) >_d \mathit{mds}\ \mathit{xs} \end{aligned}$$

- It took me a week to construct the right generalisation:

$$\begin{aligned} & \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{xs})) = \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{win}\ \mathit{xs})) \\ & \quad \Leftarrow\ \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{xs})) >_d \mathit{mds}\ \mathit{xs} \end{aligned}$$

- Once the right property is found, the actual proof was done in about 20 minutes.
- “Someone once described research as ‘finding out something to find out, then finding it out’; the first part is often harder than the second.”

Remark

- The $sum \cdot reverse$ example is superficial — the same property is much easier to prove using the $O(n^2)$-time definition of $reverse$.
- That's one of the reason we defer the discussion about efficiency — to prove properties of a function we sometimes prefer to roll back to a slower version.
- In our exercises there is an example where you need $revcat$ to prove a property about $reverse$.
    - Show that $reverse \cdot reverse = id$

A Common Pattern We've Seen Many Times...

$$\begin{aligned} \mathit{sum}\ [\,] &= 0 \\ \mathit{sum}\ (x : \mathit{xs}) &= x + \mathit{sum}\ \mathit{xs} \end{aligned}$$

$$\begin{aligned} \mathit{length}\ [\,] &= 0 \\ \mathit{length}\ (x : \mathit{xs}) &= 1 + \mathit{length}\ \mathit{xs} \end{aligned}$$

$$\begin{aligned} \mathit{map}\ f\ [\,] &= [\,] \\ \mathit{map}\ f\ (x : \mathit{xs}) &= f\ x : \mathit{map}\ f\ \mathit{xs} \end{aligned}$$

This pattern is extracted and called *foldr*:

$$\begin{aligned} \mathit{foldr}\ f\ e\ [\,] &= e, \\ \mathit{foldr}\ f\ e\ (x : \mathit{xs}) &= f\ x\ (\mathit{foldr}\ f\ e\ \mathit{xs}). \end{aligned}$$

Replacing Constructors

$$\begin{aligned} &foldr\ f\ e\ [\,] &&= e \\ &foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs) \end{aligned}$$

One way to look at $foldr\ (\oplus)\ e$ is that it replaces $[\,]$ with e and $(:)$ with $(\oplus)$:

$$\begin{aligned} & foldr\ (\oplus)\ e\ [1,2,3,4] \\ = \ & foldr\ (\oplus)\ e\ (1 : (2 : (3 : (4 : [\,])))) \\ = \ & 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))). \end{aligned}$$

- $sum = foldr\ (+)\ 0$.
- $length = foldr\ (\lambda x\ n.1 + n)\ 0$.
- $map\ f = foldr\ (\lambda x\ xs.f\ x : xs)\ [\,]$.
- One can see that $id = foldr\ (:)\ [\,]$.

Some Trivial Folds on Lists

Function *max* returns the maximum element in a list:

- $$\begin{aligned} \mathit{maximum}\,[\,] &= -\infty, \\ \mathit{maximum}\,(x : xs) &= x \uparrow \mathit{maximum}\,xs. \end{aligned}$$

Function *prod* returns the product of a list:

- $$\begin{aligned} \mathit{product}\,[\,] &= 1, \\ \mathit{product}\,(x : xs) &= x \times \mathit{product}\,xs. \end{aligned}$$

Function *and* returns the conjunction of a list:

- $$\begin{aligned} \mathit{and}\,[\,] &= \mathit{true}, \\ \mathit{and}\,(x : xs) &= x \wedge \mathit{and}\,xs. \end{aligned}$$

Some Trivial Folds on Lists

Function *max* returns the maximum element in a list:

- $$\begin{aligned} \mathit{maximum}\ [\,] &= \text{-}\infty, \\ \mathit{maximum}\ (x : xs) &= x \uparrow \mathit{maximum}\ xs. \end{aligned}$$
- $\mathit{maximum} = \mathit{foldr}\ (\uparrow)\ \text{-}\infty$.

Function *prod* returns the product of a list:

- $$\begin{aligned} \mathit{product}\ [\,] &= 1, \\ \mathit{product}\ (x : xs) &= x \times \mathit{product}\ xs. \end{aligned}$$

Function *and* returns the conjunction of a list:

- $$\begin{aligned} \mathit{and}\ [\,] &= \mathit{true}, \\ \mathit{and}\ (x : xs) &= x \wedge \mathit{and}\ xs. \end{aligned}$$

Some Trivial Folds on Lists

Function *max* returns the maximum element in a list:

- $$\begin{aligned} \mathit{maximum}\ [\,] &= \text{-}\infty, \\ \mathit{maximum}\ (x : xs) &= x \uparrow \mathit{maximum}\ xs. \end{aligned}$$
- $\mathit{maximum} = \mathit{foldr}\ (\uparrow)\ \text{-}\infty.$

Function *prod* returns the product of a list:

- $$\begin{aligned} \mathit{product}\ [\,] &= 1, \\ \mathit{product}\ (x : xs) &= x \times \mathit{product}\ xs. \end{aligned}$$
- $\mathit{product} = \mathit{foldr}\ (\times)\ 1.$

Function *and* returns the conjunction of a list:

- $$\begin{aligned} \mathit{and}\ [\,] &= \mathit{true}, \\ \mathit{and}\ (x : xs) &= x \wedge \mathit{and}\ xs. \end{aligned}$$

Some Trivial Folds on Lists

Function *max* returns the maximum element in a list:

- $$\begin{aligned} maximum\ [\,] &= \text{-}\infty, \\ maximum\ (x : xs) &= x \uparrow maximum\ xs. \end{aligned}$$
- $maximum = foldr\ (\uparrow)\ \text{-}\infty$.

Function *prod* returns the product of a list:

- $$\begin{aligned} product\ [\,] &= 1, \\ product\ (x : xs) &= x \times product\ xs. \end{aligned}$$
- $product = foldr\ (\times)\ 1$.

Function *and* returns the conjunction of a list:

- $$\begin{aligned} and\ [\,] &= true, \\ and\ (x : xs) &= x \wedge and\ xs. \end{aligned}$$
- $and = foldr\ (\wedge)\ true$.

Some Trivial Folds on Lists

Function *max* returns the maximum element in a list:

- $$\begin{aligned} \mathit{maximum}\ [\,] &= \text{-}\infty, \\ \mathit{maximum}\ (x : xs) &= x \uparrow \mathit{maximum}\ xs. \end{aligned}$$
- $\mathit{maximum} = \mathit{foldr}\ (\uparrow)\ \text{-}\infty$.

Function *prod* returns the product of a list:

- $$\begin{aligned} \mathit{product}\ [\,] &= 1, \\ \mathit{product}\ (x : xs) &= x \times \mathit{product}\ xs. \end{aligned}$$
- $\mathit{product} = \mathit{foldr}\ (\times)\ 1$.

Function *and* returns the conjunction of a list:

- $$\begin{aligned} \mathit{and}\ [\,] &= \mathit{true}, \\ \mathit{and}\ (x : xs) &= x \wedge \mathit{and}\ xs. \end{aligned}$$
- $\mathit{and} = \mathit{foldr}\ (\wedge)\ \mathit{true}$.

Some Functions We Have Seen...

.

$$\begin{aligned}
&(+\!\!+) && :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ ys && = \; ys \\
&(x : xs) +\!\!+ ys && = \; x : (xs +\!\!+ ys) \quad .
\end{aligned}$$

$concat =$.

$$\begin{aligned}
&concat && :: [[a]] \to [a] \\
&concat\,[\,] && = \; [\,] \\
&concat\,(xs : xss) && = \; xs +\!\!+ concat\; xss \quad .
\end{aligned}$$

Some Functions We Have Seen...

$(+\!\!+\, ys) = foldr\ (:)\ ys.$

$$\begin{aligned}
(+\!\!+) &:: [a] \to [a] \to [a] \\
[\,] +\!\!+ ys &= ys \\
(x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys) \quad .
\end{aligned}$$

$concat = \qquad\qquad .$

$$\begin{aligned}
concat &:: [[a]] \to [a] \\
concat\ [\,] &= [\,] \\
concat\ (xs : xss) &= xs +\!\!+ concat\ xss \quad .
\end{aligned}$$

Some Functions We Have Seen...

$(\mathbin{+\!\!+}\, ys) = \mathit{foldr}\ (:)\ ys.$

$$\begin{aligned}
&(\mathbin{+\!\!+}) &&:: [a] \to [a] \to [a] \\
&[\,] \mathbin{+\!\!+} ys &&= ys \\
&(x : xs) \mathbin{+\!\!+} ys &&= x : (xs \mathbin{+\!\!+} ys) \quad .
\end{aligned}$$

$\mathit{concat} = \mathit{foldr}\ (\mathbin{+\!\!+})\ [\,].$

$$\begin{aligned}
&\mathit{concat} &&:: [[a]] \to [a] \\
&\mathit{concat}\ [\,] &&= [\,] \\
&\mathit{concat}\ (xs : xss) &&= xs \mathbin{+\!\!+} \mathit{concat}\ xss \quad .
\end{aligned}$$

Replacing Constructors

- Understanding *foldr* from its type. Recall

$$\mathbf{data}\ [a] \ = \ [\,] \ \mid \ a : [a] \ \ .$$

- Types of the two constructors: $[\,] :: [a]$, and $(:) :: a \rightarrow [a] \rightarrow [a]$.
- *foldr* replaces the constructors:

$$\begin{array}{lcl} foldr & :: & (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \ \ . \end{array}$$

Why Folds?

- "What are the three most important factors in a programming language?"

Why Folds?

- "What are the three most important factors in a programming language?" Abstraction, abstraction, and abstraction!

Why Folds?

- “What are the three most important factors in a programming language?” Abstraction, abstraction, and abstraction!
- Control abstraction, procedure abstraction, data abstraction,...can programming patterns be abstracted too?

Why Folds?

- "What are the three most important factors in a programming language?" Abstraction, abstraction, and abstraction!
- Control abstraction, procedure abstraction, data abstraction,...can programming patterns be abstracted too?
- Program structure becomes an entity we can talk about, reason about, and reuse.
    - We can describe algorithms in terms of fold, unfold, and other recognised patterns.
    - We can prove properties about folds,
    - and apply the proved theorems to all programs that are folds, either for compiler optimisation, or for mathematical reasoning.

WHY FOLDS?

- "What are the three most important factors in a programming language?" Abstraction, abstraction, and abstraction!
- Control abstraction, procedure abstraction, data abstraction,...can programming patterns be abstracted too?
- Program structure becomes an entity we can talk about, reason about, and reuse.
    - We can describe algorithms in terms of fold, unfold, and other recognised patterns.
    - We can prove properties about folds,
    - and apply the proved theorems to all programs that are folds, either for compiler optimisation, or for mathematical reasoning.
- Among the theorems about folds, the most important is probably the *fold-fusion* theorem.

The Fold-Fusion Theorem

The theorem is about when the composition of a function and a fold can be expressed as a fold.

Theorem (*foldr*-Fusion)
Given $f :: a \to b \to b$, $e :: b$, $h :: b \to c$, and $g :: a \to c \to c$, we have:

$$h \cdot foldr\ f\ e \ = \ foldr\ g\ (h\ e) \quad ,$$

if $h\ (f\ x\ y) = g\ x\ (h\ y)$ for all x and y.

For program derivation, we are usually given h, f, and e, from which we have to construct g.

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ [a, b, c]) \\
= & \quad \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ a\ (f\ b\ (f\ c\ e)))
\end{aligned}$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{aligned}
& h\ (foldr\ f\ e\ [a, b, c]) \\
= \quad & \{\ \text{definition of } foldr\ \} \\
& h\ (f\ a\ (f\ b\ (f\ c\ e))) \\
= \quad & \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (h\ (f\ b\ (f\ c\ e)))
\end{aligned}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$\begin{aligned}
& h\ (foldr\ f\ e\ [a, b, c]) \\
= & \quad \{\ \text{definition of } foldr\ \} \\
& h\ (f\ a\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (h\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (g\ b\ (h\ (f\ c\ e)))
\end{aligned}$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{aligned}
& h\ (foldr\ f\ e\ [a, b, c]) \\
= & \quad \{\ \text{definition of } foldr\ \} \\
& h\ (f\ a\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (h\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (g\ b\ (h\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (g\ b\ (g\ c\ (h\ e)))
\end{aligned}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ [a,b,c]) \\
= & \quad \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ a\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (h\ (f\ b\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (g\ b\ (h\ (f\ c\ e))) \\
= & \quad \{\ \text{since } h\ (f\ x\ y) = g\ x\ (h\ y)\ \} \\
& g\ a\ (g\ b\ (g\ c\ (h\ e))) \\
= & \quad \{\ \text{definition of } \mathit{foldr}\ \} \\
& \mathit{foldr}\ g\ (h\ e)\ [a,b,c]\ \ .
\end{aligned}
$$

Sum of Squares, Again

- Consider $sum \cdot map\ square$ again. This time we use the fact that $map\ f = foldr\ (mf\ f)\ [\,]$, where $mf\ f\ x\ xs = f\ x : xs$.
- $sum \cdot map\ square$ is a fold, if we can find a ssq such that $sum\ (mf\ square\ x\ xs) = ssq\ x\ (sum\ xs)$. Let us try:

$$\begin{aligned}
& sum\ (mf\ square\ x\ xs) \\
= & \quad \{\ \text{definition of } mf\ \} \\
& sum\ (square\ x : xs) \\
= & \quad \{\ \text{definition of } sum\ \} \\
& square\ x + sum\ xs \\
= & \quad \{\ \text{let } ssq\ x\ y = square\ x + y\ \} \\
& ssq\ x\ (sum\ xs)\ .
\end{aligned}$$

Therefore, $sum \cdot map\ square = foldr\ ssq\ 0$.

Sum of Squares, without Folds

Recall that this is how we derived the inductive case of *sumsq* yesterday:

$$\begin{aligned}
& \mathit{sumsq}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : \mathit{xs})) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\ (\mathit{square}\ x : \mathit{map}\ \mathit{square}\ \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sum}\ \} \\
& \mathit{square}\ x + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{square}\ x + \mathit{sumsq}\ \mathit{xs} \quad .
\end{aligned}$$

Comparing the two derivations, by using fold-fusion we supply only the “important” part.

More on Folds and Fold-fusion

- Compare the proof with the one yesterday. They are essentially the same proof.
- Fold-fusion theorem abstracts away the common parts in this kind of inductive proofs, so that we need to supply only the "important" parts.
- Tupling can be seen as a kind of fold-fusion. The derivation of $\mathit{steepsum}$, for example, can be seen as fusing:

$$\mathit{steepsum} \cdot \mathit{id} = \mathit{steepsum} \cdot \mathit{foldr}\ (:)\ [\,].$$

  - Recall that $\mathit{steepsum}\ \mathit{xs} = (\mathit{steep}\ \mathit{xs}, \mathit{sum}\ \mathit{xs})$. Reformulating $\mathit{steepsum}$ into a fold allows us to compute it in one traversal.
- Not every function can be expressed as a fold. For example, $\mathit{tail} :: [a] \to [a]$ is not a fold!

Longest Prefix

- The function call $\mathit{takeWhile}\ p\ \mathit{xs}$ returns the longest prefix of xs that satisfies p:

$$\begin{aligned}
&\mathit{takeWhile}\ p\ [\,] &&= [\,]\\
&\mathit{takeWhile}\ p\ (x : \mathit{xs}) &&=\\
&\quad \mathbf{if}\ p\ x\ \mathbf{then}\ x : \mathit{takeWhile}\ p\ \mathit{xs}\\
&\quad \mathbf{else}\ [\,]\ \ .
\end{aligned}$$

- E.g. $\mathit{takeWhile}\ ({\leq}\ 3)\ [1, 2, 3, 4, 5] = [1, 2, 3]$.
- It can be defined by a fold:

$$\begin{aligned}
\mathit{takeWhile}\ p &= \mathit{foldr}\ (\mathit{tke}\ p)\ [\,],\\
\mathit{tke}\ p\ x\ \mathit{xs} &= \mathbf{if}\ p\ x\ \mathbf{then}\ x : \mathit{xs}\ \mathbf{else}\ [\,].
\end{aligned}$$

- Its dual, $\mathit{dropWhile}\ ({\leq}\ 3)\ [1, 2, 3, 4, 5] = [4, 5]$, is not a fold.

All Prefixes

- The function *inits* returns the list of all prefixes of the input list:

$$\begin{aligned} \mathit{inits}\ [\,] &= [[\,]], \\ \mathit{inits}\ (x : xs) &= [\,] : \mathit{map}\ (x :)\ (\mathit{inits}\ xs). \end{aligned}$$

- E.g. $\mathit{inits}\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$.
- It can be defined by a fold:

$$\begin{aligned} \mathit{inits} &= \mathit{foldr}\ \mathit{ini}\ [[\,]], \\ \mathit{ini}\ x\ xss &= [\,] : \mathit{map}\ (x :)\ xss. \end{aligned}$$

All Suffixes

- The function *tails* returns the list of all suffixes of the input list:

$$\begin{aligned} tails\ [\,] &= [[\,]],\\ tails\ (x : xs) &= \textbf{let}\ (ys : yss) = tails\ xs\\ &\quad\ \textbf{in}\ (x : ys) : ys : yss. \end{aligned}$$

- E.g. $tails\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$.
- It can be defined by a fold:

$$\begin{aligned} tails &= foldr\ til\ [[\,]],\\ til\ x\ (ys : yss) &= (x : ys) : ys : yss. \end{aligned}$$

Scan

- $scanr\ f\ e = map\ (foldr\ f\ e) \cdot tails$.
- E.g.

$$\begin{aligned} & scanr\ (+)\ 0\ [1,2,3] \\ = \ & map\ sum\ (tails\ [1,2,3]) \\ = \ & map\ sum\ [[1,2,3],[2,3],[3],[\,]] \\ = \ & [6,5,3,0]. \end{aligned}$$

- Of course, it is slow to actually perform $map\ (foldr\ f\ e)$ separately. By fold-fusion, we get a faster implementation:

$$\begin{aligned} scanr\ f\ e &= foldr\ (sc\ f)\ [e], \\ sc\ f\ x\ (y : ys) &= f\ x\ y : y : ys. \end{aligned}$$

- Folds are a specialised form of induction.
- Inductive datatypes: types on which you can perform induction.
- Every inductive datatype give rise to its fold.
- In fact, an inductive type can be defined by its fold.

Fold on Natural Numbers

- Recall the definition:

$$\textbf{data}\ \mathit{Nat} \ = \ 0 \ | \ \mathbf{1}_+\ \mathit{Nat} \ .$$

- Constructors: $0 :: \mathit{Nat}$, $(\mathbf{1}_+) :: \mathit{Nat} \rightarrow \mathit{Nat}$.
- What is the fold on Nat?

$$\mathit{foldN} \ :: \qquad\qquad\qquad \rightarrow \mathit{Nat} \rightarrow a$$

Fold on Natural Numbers

- Recall the definition:

$$\textbf{data}\ \mathit{Nat} \ = \ 0 \ \mid \ \mathbf{1}_{+}\ \mathit{Nat} \ \ .$$

- Constructors: $0 :: \mathit{Nat}$, $(\mathbf{1}_{+}) :: \mathit{Nat} \to \mathit{Nat}$.
- What is the fold on *Nat*?

$$\mathit{foldN} \ :: \ (a \to a) \to a \to \mathit{Nat} \to a$$

Fold on Natural Numbers

- Recall the definition:

$$\textbf{data}\ \mathit{Nat} \;=\; 0 \;\mid\; \mathbf{1}_{+}\ \mathit{Nat}\ \ .$$

- Constructors: $0 :: \mathit{Nat}$, $(\mathbf{1}_{+}) :: \mathit{Nat} \to \mathit{Nat}$.
- What is the fold on Nat?

$$\begin{aligned}
&\mathit{foldN} \;::\; (a \to a) \to a \to \mathit{Nat} \to a\\
&\mathit{foldN}\ f\ e\ 0 \qquad\quad = \; e\\
&\mathit{foldN}\ f\ e\ (\mathbf{1}_{+}\ n) \;=\; f\ (\mathit{foldN}\ f\ e\ n)\ \ .
\end{aligned}$$

Examples of *foldN*

$$
\begin{aligned}
& \quad . \\
0 + n &= n \\
(1_{+}\ m) + n &= 1_{+}\ (m + n) \quad .
\end{aligned}
$$

$$
\begin{aligned}
& \quad . \\
0 \times n &= 0 \\
(1_{+}\ m) \times n &= n + (m \times n) \quad .
\end{aligned}
$$

$$
\begin{aligned}
& \quad . \\
\mathit{even}\ 0 &= \mathit{True} \\
\mathit{even}\ (1_{+}\ n) &= \mathit{not}\ (\mathit{even}\ n) \quad .
\end{aligned}
$$

Examples of *foldN*

- $(+n) = foldN\ (1_+)\ n$.

$$\begin{aligned} 0 + n &= n \\ (1_+\ m) + n &= 1_+\ (m + n)\ . \end{aligned}$$

.

$$\begin{aligned} 0 \times n &= 0 \\ (1_+\ m) \times n &= n + (m \times n)\ . \end{aligned}$$

.

$$\begin{aligned} even\ 0 &= True \\ even\ (1_+\ n) &= not\ (even\ n)\ . \end{aligned}$$

Examples of *foldN*

- $(+n) = foldN\ (\mathbf{1}_+)\ n$.

$$\begin{aligned} 0 + n &= n \\ (\mathbf{1}_+\ m) + n &= \mathbf{1}_+\ (m + n) \quad . \end{aligned}$$

- $(\times n) = foldN\ (n+)\ 0$.

$$\begin{aligned} 0 \times n &= 0 \\ (\mathbf{1}_+\ m) \times n &= n + (m \times n) \quad . \end{aligned}$$

$$.$$

$$\begin{aligned} even\ 0 &= True \\ even\ (\mathbf{1}_+\ n) &= not\ (even\ n) \quad . \end{aligned}$$

Examples of *foldN*

- $(+n) = foldN\ (\mathbf{1}_+)\ n$.

$$\begin{aligned} 0 + n &= n \\ (\mathbf{1}_+\ m) + n &= \mathbf{1}_+\ (m + n) \quad . \end{aligned}$$

- $(\times n) = foldN\ (n+)\ 0$.

$$\begin{aligned} 0 \times n &= 0 \\ (\mathbf{1}_+\ m) \times n &= n + (m \times n) \quad . \end{aligned}$$

- $even = foldN\ not\ True$.

$$\begin{aligned} even\ 0 &= True \\ even\ (\mathbf{1}_+\ n) &= not\ (even\ n) \quad . \end{aligned}$$

Fold-Fusion for Natural Numbers

Theorem ($foldN$-Fusion)
Given $f :: a \to a$, $e :: a$, $h :: a \to b$, and $g :: b \to b$, we have:

$$h \cdot foldN\ f\ e \ = \ foldN\ g\ (h\ e) \quad ,$$

if $h\ (f\ x) = g\ (h\ x)$ for all x.

Exercise: fuse $even$ into $(+)$?

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned} \textbf{data}\ \mathit{ITree}\ a &= \mathsf{Null} \mid \mathsf{Node}\ \alpha\ (\mathit{ITree}\ a)\ (\mathit{ITree}\ a)\ , \\ \textbf{data}\ \mathit{ETree}\ a &= \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a)\ . \end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\mathit{foldIT} :: \qquad \mathit{ITree}\ a \to b$$

- The fold for *ETree*, is given by:

$$\mathit{foldET} :: \qquad \mathit{ETree}\ a \to b$$

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned} \textbf{data}\ \mathit{ITree}\ a &= \mathsf{Null} \mid \mathsf{Node}\ \alpha\ (\mathit{ITree}\ a)\ (\mathit{ITree}\ a) \quad , \\ \textbf{data}\ \mathit{ETree}\ a &= \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a) \quad . \end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\mathit{foldIT} :: (a \to b \to b \to b) \to b \to \mathit{ITree}\ a \to b$$

- The fold for *ETree*, is given by:

$$\mathit{foldET} :: \qquad\qquad\qquad \mathit{ETree}\ a \to b$$

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned}\textbf{data}\ \mathit{ITree}\ a &= \mathrm{Null} \mid \mathrm{Node}\ \alpha\ (\mathit{ITree}\ a)\ (\mathit{ITree}\ a)\ ,\\ \textbf{data}\ \mathit{ETree}\ a &= \mathrm{Tip}\ a \mid \mathrm{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a)\ .\end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\begin{aligned}&\mathit{foldIT} :: (a \to b \to b \to b) \to b \to \mathit{ITree}\ a \to b\\ &\mathit{foldIT}\ f\ e\ \mathrm{Null} = e\\ &\mathit{foldIT}\ f\ e\ (\mathrm{Node}\ a\ t\ u) = f\ a\ (\mathit{foldIT}\ f\ e\ t)\ (\mathit{foldIT}\ f\ e\ u)\ .\end{aligned}$$

- The fold for *ETree*, is given by:

$$\mathit{foldET} :: \qquad\qquad\qquad \mathit{ETree}\ a \to b$$

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned} &\textbf{data}\ \mathit{ITree}\ a \;=\; \mathrm{Null} \mid \mathrm{Node}\ \alpha\ (\mathit{ITree}\ a)\ (\mathit{ITree}\ a) \;, \\ &\textbf{data}\ \mathit{ETree}\ a \;=\; \mathrm{Tip}\ a \mid \mathrm{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a) \;. \end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\begin{aligned} &\mathit{foldIT} :: (a \to b \to b \to b) \to b \to \mathit{ITree}\ a \to b \\ &\mathit{foldIT}\ f\ e\ \mathrm{Null} \qquad\qquad = e \\ &\mathit{foldIT}\ f\ e\ (\mathrm{Node}\ a\ t\ u) = f\ a\ (\mathit{foldIT}\ f\ e\ t)\ (\mathit{foldIT}\ f\ e\ u) \;. \end{aligned}$$

- The fold for *ETree*, is given by:

$$\mathit{foldET} :: (b \to b \to b) \to (a \to b) \to \mathit{ETree}\ a \to b$$

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned} \textbf{data}\ \mathit{ITree}\ a &= \mathrm{Null} \mid \mathrm{Node}\ \alpha\ (\mathit{ITree}\ a)\ (\mathit{ITree}\ a) \ , \\ \textbf{data}\ \mathit{ETree}\ a &= \mathrm{Tip}\ a \mid \mathrm{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a) \ . \end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\begin{aligned} &\mathit{foldIT} :: (a \to b \to b \to b) \to b \to \mathit{ITree}\ a \to b \\ &\mathit{foldIT}\ f\ e\ \mathrm{Null} = e \\ &\mathit{foldIT}\ f\ e\ (\mathrm{Node}\ a\ t\ u) = f\ a\ (\mathit{foldIT}\ f\ e\ t)\ (\mathit{foldIT}\ f\ e\ u) \ . \end{aligned}$$

- The fold for *ETree*, is given by:

$$\begin{aligned} &\mathit{foldET} :: (b \to b \to b) \to (a \to b) \to \mathit{ETree}\ a \to b \\ &\mathit{foldET}\ f\ g\ (\mathrm{Tip}\ x) = g\ x \\ &\mathit{foldET}\ f\ g\ (\mathrm{Bin}\ t\ u) = f\ (\mathit{foldET}\ f\ g\ t)\ (\mathit{foldET}\ f\ g\ u) \ . \end{aligned}$$

Some Simple Functions on Trees

- To compute the size of an *ITree*:

$$sizeITree \ = \ foldIT\ (\lambda x\ m\ n \to \mathbf{1}_{+}\ (m+n))\ 0\ \ .$$

- To sum up labels in an *ETree*:

$$sumETree \ = \ foldET\ (+)\ id.$$

- To compute a list of all labels in an *ITree* and an *ETree*:

$$\begin{aligned} flattenIT\ &= foldIT\ (\lambda x\ xs\ ys \to xs +\!\!+ [x] +\!\!+ ys)\ [\,],\\ flattenET\ &= foldET\ (+\!\!+)\ (\lambda x \to [x]). \end{aligned}$$

- **Exercise**: what are the fusion theorems for *foldIT* and *foldET*?

Maximum Segment Sum

Finally we have introduced enough concepts to tackle the maximum segment sum problem!

Maximum Segment Sum: given a list of numbers, find the maximum possible sum of a consecutive segment.

Can be traced to 1984 in Dijkstra and Feijen's *Een methode van programmeren*,

Probably made famous by Bentley, and became a pet topic of the program derivation community after being given a formal treatment by Gries.

Perhaps the most popular example in program derivation. The calculation we present here is close to that of Gibbons.

Specifying Maximum Segment Sum

- A segment can be seen as a prefix of a suffix.
- The function *segs* computes the list of all the segments.

$$segs \;=\; concat \cdot map\ inits \cdot tails.$$

- Therefore, *mss* is specified by:

$$mss \;=\; max \cdot map\ sum \cdot segs.$$

The Derivation!

We reason:

$$max \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails$$

The Derivation!

We reason:

$$\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails}
\end{aligned}$$

The Derivation!

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\ \mathit{max}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails}
\end{aligned}
$$

The Derivation!

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\ \mathit{max}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f.g)\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails}\quad .
\end{aligned}
$$

Recall the definition $\mathit{scanr}\ f\ e = \mathit{map}\ (\mathit{foldr}\ f\ e) \cdot \mathit{tails}$. If we can transform $\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}$ into a fold, we can turn the algorithm into a *scanr*, which has a faster implementation.

Maximum Prefix Sum

Concentrate on $max \cdot map\ sum \cdot inits$:

$$
\begin{aligned}
& max \cdot map\ sum \cdot inits \\
= \quad & \{\ \text{definition of } init, ini\ x\ xss = [\,] : map\ (x :)\ xss\ \} \\
& max \cdot map\ sum \cdot foldr\ ini\ [[\,]]
\end{aligned}
$$

Maximum Prefix Sum

Concentrate on $\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}$:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits} \\
= \quad & \{\ \text{definition of } \mathit{init},\ \mathit{ini}\ x\ \mathit{xss} = [\,] : \mathit{map}\ (x\ :)\ \mathit{xss}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{foldr}\ \mathit{ini}\ [[\,]] \\
= \quad & \{\ \text{fold fusion, see below}\ \} \\
& \mathit{max} \cdot \mathit{foldr}\ \mathit{zplus}\ [0]\ \ .
\end{aligned}
$$

The fold fusion works because:

$$
\begin{aligned}
& \mathit{map}\ \mathit{sum}\ (\mathit{ini}\ x\ \mathit{xss}) \\
= \ & \mathit{map}\ \mathit{sum}\ ([\,] : \mathit{map}\ (x\ :)\ \mathit{xss}) \\
= \ & 0 : \mathit{map}\ (\mathit{sum} \cdot (x\ :))\ \mathit{xss} \\
= \ & 0 : \mathit{map}\ (x+)\ (\mathit{map}\ \mathit{sum}\ \mathit{xss})\ \ .
\end{aligned}
$$

Define $\mathit{zplus}\ x\ \mathit{yss} = 0 : \mathit{map}\ (x+)\ \mathit{yss}$.

Maximum Prefix Sum, 2nd Fold Fusion

Concentrate on $max \cdot map\ sum \cdot inits$:

$$
\begin{aligned}
& max \cdot map\ sum \cdot inits \\
= & \quad \{ \text{ definition of } init, ini\ x\ xss = [\,] : map\ (x\ :)\ xss \ \} \\
& max \cdot map\ sum \cdot foldr\ ini\ [[\,]] \\
= & \quad \{ \text{ fold fusion, } zplus\ x\ xss = 0 : map\ (x+)\ xss \ \} \\
& max \cdot foldr\ zplus\ [0] \\
= & \quad \{ \text{ fold fusion, let } zmax\ x\ y = 0 \uparrow (x + y) \ \} \\
& foldr\ zmax\ 0 \quad .
\end{aligned}
$$

The fold fusion works because $\uparrow$ distributes into $(+)$:

$$
\begin{aligned}
& max\ (0 : map\ (x+)\ xs) \\
= & 0 \uparrow max\ (map\ (x+)\ xs) \\
= & 0 \uparrow (x + max\ xs) \quad .
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\ \mathit{max}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f.g)\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails}
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\ \mathit{max}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f.g)\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= \quad & \{\ \text{reasoning in the previous slides}\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{foldr}\ \mathit{zmax}\ 0) \cdot \mathit{tails}
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= & \quad \{\ \text{since } \mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= & \quad \{\ \text{since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\ \mathit{max}\ \} \\
& \mathit{max} \cdot \mathit{map}\ \mathit{max} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails} \\
= & \quad \{\ \text{since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f.g)\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{max} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= & \quad \{\ \text{reasoning in the previous slides}\ \} \\
& \mathit{max} \cdot \mathit{map}\ (\mathit{foldr}\ \mathit{zmax}\ 0) \cdot \mathit{tails} \\
= & \quad \{\ \text{introducing } \mathit{scanr}\ \} \\
& \mathit{max} \cdot \mathit{scanr}\ \mathit{zmax}\ 0 \quad .
\end{aligned}
$$

Maximum Segment Sum in Linear Time!

- We have derived $mss = max \cdot scanr\ zmax\ 0$, where $zmax\ x\ y = 0 \uparrow (x + y)$.
- The algorithm runs in linear time, but takes linear space.
- A tupling transformation eliminates the need for linear space.

  $$mss \;=\; fst \cdot maxhd \cdot scanr\ zmax\ 0$$

  where $maxhd\ xs = (max\ xs, head\ xs)$. We omit this last step in the lecture.
- The final program is $mss = fst \cdot foldr\ step\ (0, 0)$, where $step\ x\ (m, y) = ((0 \uparrow (x + y)) \uparrow m, 0 \uparrow (x + y))$.

A Quick Note on Type Classes

The Type of *take*

- Recall the definition:

$$\begin{aligned} take\ 0\ xs &= [\,] \\ take\ (1+n)\ [\,] &= [\,] \\ take\ (1+n)\ (x : xs) &= x : take\ n\ xs \quad . \end{aligned}$$

- The first argument has to be of a numeric type (e.g. Int), since we pattern matched it against 0 and $1+$.
- The second argument must be a list, since we patten matched it against $[\,]$ and $(:)$.
- But the element of the list is not examined at all. It is merely copied to the output.

The Type of *take*

- The type of *take* can be
  - $Int \to List\,Int \to List\,Int$;
  - $Int \to List\,Char \to List\,Char$, etc.
- There is a *most general* type: $Int \to List\,a \to List\,a$.
  - The small letter means that a is a type variable. One can imagine that there is an implicit $\forall$ that quantifies all type varaibles: $\forall a.Int \to List\,a \to List\,a$.

The Identity Function

- For a more obvious example, consider the (simple but important) identity function

$$id\ x = x \quad .$$

- The argument is not touched at all.
- It may have type $Int \to Int$, $Char \to Char$, or even $(Int \to Int) \to (Int \to Int)$.
- The most general type is $a \to a$.

Filter

- Recall *filter*:

$$\begin{aligned}
&\mathit{filter} && :: \ (a \to \mathit{Bool}) \to \mathit{List}\, a \to \mathit{List}\, a \\
&\mathit{filter}\ p\ [\,] && = \ [\,] \\
&\mathit{filter}\ p\ (x : xs) && \mid p\ x \ = \ x : \mathit{filter}\ p\ xs \\
& && \mid \mathbf{otherwise} \ = \ \mathit{filter}\ p\ xs \quad .
\end{aligned}$$

- Still, in $\mathit{filter}\ p\ (x : xs)$ we merely passes x to p, without looking into x.
- Therefore *filter* works for any type a for which there exists functions of type $a \to \mathit{Bool}$ — which is true for all type a.

Counting Lowercase Characters

- For a counterexample, consider the following function:

$$\begin{aligned}
&\mathit{lowers} && :: \mathit{List\ Char} \to \mathit{Int} \\
&\mathit{lowers}\ [\,] && = 0 \\
&\mathit{lowers}\ (x : xs) && = \mathbf{if}\ \mathit{isLower}\ x \\
& && \quad\ \mathbf{then}\ 1 + \mathit{lowers}\ xs \\
& && \quad\ \mathbf{else}\ \mathit{lowers}\ xs \quad .
\end{aligned}$$

- The function counts the number of lowercase characters in a string.
- It is equivalent to *length* · *filter isLower*.
- *x* is passed to *isLower*, which forces *x* to be a *Char*.

Parametric Polymorphism

- *Polymorphism*: allowing a piece of code to have many types, such that it can be used in many occasions.
    - Indeed, take can be applied to all types of lists. We do not need to define a separate version for $\mathit{List}\ \mathit{Int}$, $\mathit{List}\ (\mathit{Int} \to \mathit{Int})$.
- *Parametric* polymorphic, as we have seen just now, is common in many functional programming languages.
- When $\mathit{take}\ n :: \mathit{List}\, a \to \mathit{List}\, a$ is applied to an argument, say $[1, 2, 3]$, the type variable a is instantiated to the type of the argument (Int in this case).
    - The type variable a behaves like a parameter, thus the name.
    - Observe: the same piece of code (e.g. take, filter) works for all instantiations of a .
- Object-oriented languages often adopt another kind of polymorphism for operator overloading, called *ad-hoc*

Membership Test

- Given the definition below, $elem\ x\ xs$ yields $True$ iff. x occurs in xs.

$$\begin{aligned} &elem\ x\ [\,] && = False \\ &elem\ x\ (y : xs) \mid x == y && = True \\ & \mid \mathbf{otherwise} && = elem\ x\ xs\ . \end{aligned}$$

- It could have type $Int \to List\,Int \to Bool$, $Char \to List\,Char \to Bool$, etc.
- We do not want to define $elem$ once for each type, thus we wish that it has a polymorphic type, say $a \to List\,a \to Bool$.
- However, not all values can be tested for equality! The operator $(==)$ is defined for some types, but not all types. For example, we cannot in general decide whether two functions are equal.
- Thus $elem$ cannot have type, for example, $(Int \to Int) \to List\,(Int \to Int) \to Bool$.

The *Eq* Class

- There is such a definition in the Standard Prelude:

  class *Eq a* **where**
  $$(==) :: a \to a \to Bool \quad .$$

- which says that a type a is in the *type class* *Eq* if there is an operator $(==)$, of type $a \to a \to Bool$, defined.
- *Int* is in *Eq* since we can define $(==)$ for numbers. So is *Char*, although $(==)$ for *Char* implements a different algorithm from that of *Int*.

Type of *elem*

- The most general type of *elem* is $Eq\ a \Rightarrow a \rightarrow List\, a \rightarrow Bool$,
    - which means that *elem* takes a value of type a and a list of type $List\, a$ and returns a *Bool*, provided that a is in *Eq*.
- The additional constraint arises from the fact that *elem* calls $(==)$.

Instance Declaration

- To use *elem* on concrete types, we have to teach Haskell how to check equality for each type. The following are defined somewhere in the Haskell Prelude:

```
    instance Eq Int where
      m == n =   {- how to check equality for Int -}

    instance Eq Char where
      m == n =   {- how to check equality for Char -}
```

- It is not possible to give a definition for, for example *Eq* $(a \to a)$. Thus *elem* cannot be applied to such types.

Instance Declaration

- When we define a new type, we might want to teach Haskell how to check equality:

$$\textbf{data}\ \mathit{Color} = \mathit{Red} \mid \mathit{Green} \mid \mathit{Blue} \ldots$$

-

$$\begin{aligned}&\textbf{instance}\ \mathit{Eq}\ \mathit{Color}\ \textbf{where}\\ &\quad \mathit{Red} == \mathit{Red} \quad = \mathit{True}\\ &\quad \mathit{Red} == \mathit{Green} = \mathit{False}\\ &\ldots\end{aligned}$$

Summary So Far...

- Class declaration:

  $$\textbf{class}\ Eq\ a\ \textbf{where}$$
  $$\quad (==) :: a \to a \to Bool\quad .$$

  - The *method* $(==)$ then has type $Eq\ a \Rightarrow a \to a \to Bool$.

- Instance declaration:

  $$\textbf{instance}\ Eq\ MyType\ \textbf{where}$$
  $$\quad x == y = \ldots$$

  - $(==)$ above should have type $MyType \to MyType \to Bool$, but the type is not written.

- A function that calls a function with class constraint $\mathit{Eq}\ a$ (e.g. $(==)$) also has the constraint in its type:

$$\begin{aligned} &\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to \mathit{List}\, a \to \mathit{Bool} \\ &\mathit{elem} = \ldots == \ldots \end{aligned}$$

- $\mathit{elem}\ 2\ [1, 2, 3]$ is allowed because there is an instance declaration for $\mathit{Eq}\ \mathit{Int}$, while $\mathit{elem}\ \mathit{id}\ [\mathit{id}, (1+), (2+)]$ is not (unless you define and instance $\mathit{Eq}\ (\mathit{Int} \to \mathit{Int})$).

Ad-hoc Polymorphism

- Note that $(==)$ for Int is a different program from that for $Char$.
- Type classes is thus a way to describe *operator loading* — using one name to refer to different piece of code.
- Such mechanisms are often called *ad-hoc* polymorphism.
- Compare with parametric polymorphism, where the same code, say, the same definition of $take$, works for all types.

Other Important Type Classes

- *Show*: things that can be printed (converted to string).
- *Read*: things that can be parsed from strings.
- *Num*: things that behave like numbers (with addition, multiplication, etc).
- *Integral*: things that behave like integers.
- *Monad*, *Functor*...hope we will be able to talk about them later!
- Use `:i` in `GHCi` to find out what methods and instances each class has!

Derived Instances

- The Haskell compiler may automatically construct some routine instance declarations, to save you some typing. E.g.

```
data Colors = Red | Green | Blue
  deriving (Eq, Show, Read)  .
```

Instance Inheritance

- How do we check whether two lists are equal? We can do so if we know how to check whether their elements are equal.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{List}\ a)\ \textbf{where} \\
\quad \begin{array}{ll}
[\,] == [\,] & = \mathit{True} \\
[\,] == (x : xs) & = \mathit{False} \\
(x : xs) == [\,] & = \mathit{False} \\
(x : xs) == (y : ys) & = x == y \ \wedge\ xs == ys\ .
\end{array}
\end{array}$$

- Note that in $x == y$, the $(==)$ refers to the method for type a, while the $(==)$ in $xs == ys$ is a recursive call.

Instance Inheritance

- Another example:

$$\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)\ \textbf{where}$$
$$(x_1, y_2) \mathrel{==} (x_2, y_2) \ = \ (x_1 \mathrel{==} x_2) \land (y_1 \mathrel{==} y_2) \ .$$

- All the three $(==)$ in the expression above refer to different methods!

The Type Class *Ord*

- Another type class *Ord* includes things are can be "ordered":

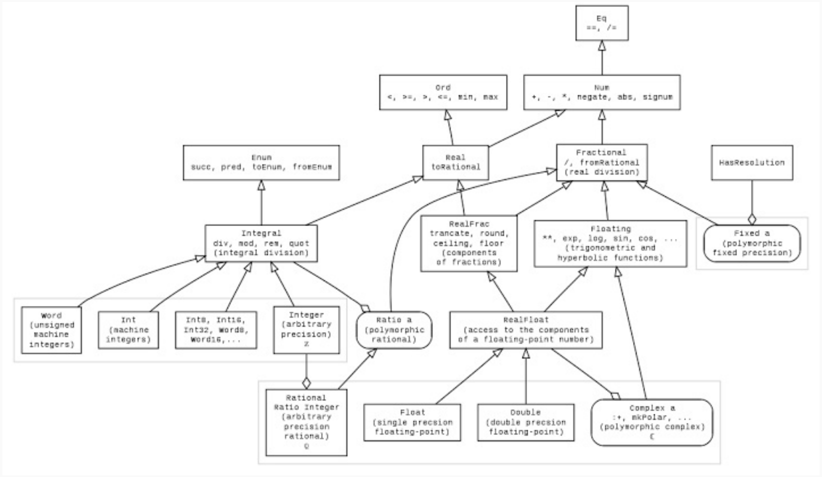
  $$\begin{aligned}&\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where}\\ &\quad (<) :: a \to a \to Bool\\ &\quad (\geq) :: a \to a \to Bool\\ &\quad (>) :: a \to a \to Bool\\ &\quad (\leq) :: a \to a \to Bool \quad \ldots\end{aligned}$$

- The declaration $Eq\ a \Rightarrow Ord\ a$ intends to mean that for a type a to be in class *Ord* it has to be in class *Eq*.
  - The methods $(<)$, $(\geq)$, etc, is allowed to use $(==)$.
  - Logically, it makes more sense to write $Eq\ a \Leftarrow Ord\ a$. But it's a historical mistake that has been made.
- The function *sort* that sorts a list might have type $Ord\ a \Rightarrow List\, a \to List\, a$.

Class Hierarchy

- Inheritance between *type classes* are not to be confused with inheritance between *types*.
- Through inheritance, type classes form a hierarchy.
- Types in the standard Haskell Prelude form a complex hierarchy.
- Other libraries may extend the existing hierarchy or build their own hierarchy.

Standard Haskell Numerical Classes



Eq ==, /=
Ord <, >=, >, <=, min, max
Num +, -, *, negate, abs, signum
Enum succ, pred, toEnum, fromEnum
Real toRational
Fractional /, fromRational (real division)
HasResolution
Integral div, mod, rem, quot (integral division)
RealFrac trancate, round, ceiling, floor (components of fractions)
Floating **, exp, log, sin, cos, ... (trigonometric and hyperbolic functions)
Fixed a (polymorphic fixed precision)
Word (unsigned machine integers)
Int (machine integers)
Int8, Int16, Int32, Word8, Word16,...
Integer (arbitrary precision) Z
Ratio a (polymorphic rational)
RealFloat (access to the components of a floating-point number)
Rational Ratio Integer (arbitrary precision rational) Q
Float (single precsion floating-point)
Double (double precsion floating-point)
Complex a :+, mkPolar, ... (polymorphic complex) C

Notes

- The name “type class” is merely a mechanism for operator loading and shall not be confused with classes in object oriented languages.
- Type classes are an important feature of Haskell. Use of type classes has extended far beyond the inventors had imagined.

Monads and Effects

Side Effects

It is a misconception that functional languages do not allow side effects. In fact, many of them allow a variety of effects.

It is just that side effects must be introduced in a disciplined manner.

Disciplined? Such that we can use side effects, and still be able to reason about programs.

Side effects: anything a function does other than returning a value.

- reading/writing to a variable,
- raising an exception,
- input/output,
- partialty (possible failure),
- non-determinism,
- non-termination… and many more.

Impurity Made Explicit in Type

Impure programs (programs in which a side effect may incur) and pure programs are separated by type.

An expression of type $m\ a$ denotes a computation that, if run, may yield a result of type a. During its execution, some side effects may incur.

Such expressions are built using operators that are expected to satisfy certain, agreed laws.

Programmers, Laws, Implementation

Programmers use operators to build programs. These operators are supposed to satisfy a set of agreed laws.

The laws specify behaviours of these operators, with which the programmers can reason about their programs.

Library implementors implement these operators, and ensure that they do satisfy these laws.

The laws form an interface between the programmers and the library implementators.

Monads

One of the ways to structure effectful programs is through *monads*.

Two operators:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ \quad \mathit{return}\ ::\ a \rightarrow m\ a\\ \quad (\gg\!=)\ ::\ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\ .\end{array}$$

$\mathit{return}\ x$ denotes a program that simply returns x, with no side effects incurred.

$m \gg\!= f$ is a program that, when run, executes m, sends the result to f, which generates a program, and runs the resulting program.

A Bland Program

$$\begin{aligned}&\mathit{return}\ (3+4) && \gg\!= \lambda x \rightarrow \\ &\mathit{return}\ (x \times x) && \gg\!= \lambda y \rightarrow \\ &\mathit{return}\ (y+1) && .\end{aligned}$$

This is a simple program that is expected to return 50. It may have type $\mathit{Monad}\ m \Rightarrow m\ \mathit{Int}$.

Hmmm... not very impressive. Isn't that just function application written the other way round?

That is because we have not introduce any effectful operators yet.

Monad Laws

Operators *return* and $(\gg=)$ should satisfy the three *monad laws*:

$$\begin{aligned}
\textbf{left unit}: &\quad return\ x \gg= f = f\ x \\
\textbf{right unit}: &\quad f \gg= return = f \\
\textbf{associativity}: &\quad (m \gg= f) \gg= g = m \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}$$

Programs That Can Fail

Define

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFail}\ m\ \textbf{where}$$
$$\quad \mathit{fail} :: m\ a \quad .$$

where *fail* denotes failure.

Programs That Can Fail

Define

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFail}\ m\ \textbf{where}$$
$$\quad \mathit{fail} :: m\ a\quad .$$

where *fail* denotes failure.

The only law we demand is

$$\mathit{fail} \gg\!= f \;=\; \mathit{fail}\quad .$$

On paper we sometimes write *fail* as $\emptyset$.

Exception Catching

Extending from *MonadFail*:

$$\begin{array}{l}\textbf{class } \mathit{MonadFail}\ m \Rightarrow \mathit{MonadExcept}\ m \textbf{ where} \\ \quad \mathit{catch} :: m\ a \rightarrow m\ a \rightarrow m\ a\ \ .\end{array}$$

Exception Catching

Extending from *MonadFail*:

$$\textbf{class}\ \mathit{MonadFail}\ m \Rightarrow \mathit{MonadExcept}\ m\ \textbf{where}$$
$$\quad \mathit{catch} :: m\ a \rightarrow m\ a \rightarrow m\ a \quad .$$

Laws: *catch* and $\emptyset$ form a monoid:

$$\begin{aligned} \mathit{catch}\ \emptyset\ h &= h \quad , \\ \mathit{catch}\ m\ \emptyset &= m \quad , \\ \mathit{catch}\ m\ (\mathit{catch}\ h\ h') &= \mathit{catch}\ (\mathit{catch}\ m\ h)\ h' \quad , \end{aligned}$$

and unexceptional computations needs no handler:

$$\mathit{catch}\ (\mathit{return}\ x)\ h = \mathit{return}\ x \quad .$$

Shortcut Product

Recall that *product* computes the product of all numbers in a given list:

$$\begin{aligned} &\mathit{product} && :: \mathit{List}\ \mathit{Int} \to \mathit{Int} \\ &\mathit{product}\ [\,] && = 1 \\ &\mathit{product}\ (x : \mathit{xs}) && = x \times \mathit{product}\ \mathit{xs} \quad , \end{aligned}$$

Shortcut Product

Recall that *product* computes the product of all numbers in a given list:

$$\begin{aligned} &\mathit{product} && :: \mathit{List}\ \mathit{Int} \to \mathit{Int} \\ &\mathit{product}\ [\,] && = 1 \\ &\mathit{product}\ (x : \mathit{xs}) && = x \times \mathit{product}\ \mathit{xs} \quad , \end{aligned}$$

But, if we know in advance that there is a 0 in the list, we can just return 0, right?

$$\begin{aligned} &\mathit{scutprod} \ :: \ \mathit{MonadExcept}\ m \Rightarrow \mathit{List}\ \mathit{Int} \to m\ \mathit{Int} \\ &\mathit{scutprod}\ \mathit{xs} \ = \\ &\quad \mathit{catch}\ (\mathbf{if}\ \mathit{elem}\ 0\ \mathit{xs}\ \mathbf{then}\ \mathit{fail} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{return}\ (\mathit{product}\ \mathit{xs})) \\ &\qquad\qquad (\mathit{return}\ 0) \quad . \end{aligned}$$

Show that $\mathit{scutprod}\ \mathit{xs} = \mathit{return}\ (\mathit{product}\ \mathit{xs})$.

Non-Deterministic Monad

Another possible extension of *MonadFail*:

$$\textbf{class}\ \mathit{MonadFail}\ m \Rightarrow \mathit{MonadNondet}\ m\ \textbf{where}$$
$$\quad (\mathbin{[\!]}) :: m\ a \to m\ a \to m\ a \quad .$$

Laws: $(\mathbin{[\!]})$ and $\emptyset$ form a monoid:

$$\begin{aligned} \emptyset \mathbin{[\!]} m &= m \quad , \\ m \mathbin{[\!]} \emptyset &= m \quad , \\ m \mathbin{[\!]} (n \mathbin{[\!]} k) &= (m \mathbin{[\!]} n) \mathbin{[\!]} k \quad , \end{aligned}$$

and a **left distributivity law**:

$$(m \mathbin{[\!]} n) \gg\!= f \;=\; (m \gg\!= f) \mathbin{[\!]} (n \gg\!= f) \quad .$$

Note: the class hierarchy of monads in this course is different from that of the standard Haskell library, but adapted from Gibbons and Hinze, which I find more suitable for teaching.

Insertion Into a List

The program *insert x ys* non-deterministically inserts *x* into one arbitrary position of *ys*.

$$
\begin{array}{lcl}
\mathit{insert} :: \mathit{MonadNondet}\ m \Rightarrow a \to \mathit{List}\ a \to m\ (\mathit{List}\ a) \\
\mathit{insert}\ x\ [\,] & = & \mathit{return}\ [x] \\
\mathit{insert}\ x\ (y : ys) & = & \mathit{return}\ (x : y : ys) \mathbin{[\!]} \\
 & & \mathit{insert}\ x\ ys \mathbin{>\!\!>\!\!=} (\mathit{return} \cdot (y\,:)) \quad .
\end{array}
$$

Prove:

$$
\begin{array}{l}
\mathit{insert}\ x\ ys \mathbin{>\!\!>\!\!=} (\mathit{return} \cdot \mathit{map}\ f) = \\
\quad \mathit{insert}\ (f\ x)\ (\mathit{map}\ f\ ys) \quad .
\end{array}
$$

Maybe as Exception

A natural choice for implementing *MonadFail* is Haskell's standard *Maybe* type:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \quad ,$$

such that *Nothing* denotes failure and *Just x* denotes a computation yielding result *x*.

```
instance Monad Maybe where
  return   =  Just
  Just x     >>= k = k x
  Nothing  >>= k = Nothing  .
```

```
instance MonadFail Maybe where
   fail = Nothing  ,

instance MonadExcept Maybe where
   catch Nothing  h = h
   catch (Just x)   h = Just x  .
```

The List Monad

Meanwhile, *List* also forms a monad.

$$\begin{aligned}&\mathbf{instance}\ \mathit{Monad}\ \mathit{List}\ \mathbf{where}\\ &\quad \mathit{return}\ x \quad = [x]\\ &\quad \mathit{xs} \gg\!= k = \mathit{concat}\ (\mathit{map}\ k\ \mathit{xs})\quad .\end{aligned}$$

Empty List as Failure

One may denote failed computation by the empty list, and a succeeded computation by a singleton list.

```
instance MonadFail List where
   fail = [] ,

instance MonadExcept List where
   catch [] h = h
   catch xs h = xs .
```

Lists Representing Non-determinism

In fact, lists are often used to represent non-determinism.

$$\begin{array}{l}\textbf{instance } \textit{MonadNondet List} \textbf{ where}\\ \quad ([\!]) = (+\!\!+) \ .\end{array}$$

Therefore, a non-deterministic computation, when implemented as lists, actually gives you the list of all its results.

Is this a faithful representation? That depends on what you expect from non-determinism. For now, it does satisfy the laws we expect. See Kiselyov for discussion on suitable implementations of non-deterministic monads.

Maybe as Non-determinism?

"If I do not need *all* results, but only one, can I implement *MonadNondet* using *Maybe*?"

Try a possible implemenation:

$$\begin{array}{l}
\textbf{instance}\ \mathit{MonadNondet}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{Nothing}\ [\!]\ m = m \\
\quad \mathit{Just}\ x\ \ \ \ [\!]\ m = \mathit{Just}\ x\ \ ,
\end{array}$$

which is... exactly like *catch*. That is usually a sign that something is wrong.

Verify: does this implementation satisfy the left-distributivity law?

List Monad for Backtracking

Lesson: even if we want only the first result, we still need to *backtrack* and compute the next result when the current first result fails.

This is in fact what happens in the *List* monad. With lazy evaluation we do not actually compute a *list* of results, but keeping enough information to backtrack. This is how backtracking is often represented in Haskell.

A Single, Unnamed Variable

For the State effect we introduce two operators that respectively reads and writes to an unnamed variable:[6]

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ \mathit{st}\ m\ \mathbf{where} \\
\quad \mathit{get}\ :: m\ \mathit{st} \\
\quad \mathit{put}\ :: \mathit{st} \to m\ () \quad .
\end{array}
$$

[6] Our naming convention here: st is the type of the state, while s, s_0, etc. are values whose type could be st.

Laws for State Effect

We usually assume the following rules:

$$\begin{aligned}
\textbf{get-put}: &\quad get \gg\!= put = return\ ()\ , \\
\textbf{put-get}: &\quad put\ s \gg get = put\ s \gg return\ s\ , \\
\textbf{put-put}: &\quad put\ s \gg put\ s' = put\ s'\ , \\
\textbf{get-get}: &\quad get \gg\!= \lambda s \to get \gg\!= \lambda s' \to f\ s\ s' = \\
&\qquad get \gg\!= \lambda s \to f\ s\ s\ ,
\end{aligned}$$

where $m \gg n = m \gg\!= \lambda s \to n$ is a shorthand we often use when n does not need the result from m.

Summing Up a List using a State

Well, this is perhaps among your first few imperative programs.

$$\begin{aligned} \mathit{loop}\ [\,] &= \mathit{get} \\ \mathit{loop}\ (x : xs) &= \mathit{get} \gg\!= \lambda s \rightarrow \\ &\quad\ \mathit{put}\ (s + x) \gg \mathit{loop}\ xs \quad , \end{aligned}$$

with which you can sum up a list by $\mathit{put} 0 \gg \mathit{loop}\ xs$.

Does it compute $\mathit{sum}\ xs$?

Loop and Fold-Left

Well, not quite... if $(+)$ were not associative. What *loop* actually computes is a *foldl*. Recall

$$\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to \mathit{List}\ a \to b \\
&\mathit{foldl}\ (\oplus)\ s\ [\,] &&= s \\
&\mathit{foldl}\ (\oplus)\ s\ (x : xs) &&= \mathit{foldl}\ (\oplus)\ (s \oplus x)\ xs \quad .
\end{aligned}$$

Loop and Fold-Left

Well, not quite... if $(+)$ were not associative. What *loop* actually computes is a *foldl*. Recall

$$\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to \mathit{List}\ a \to b \\
&\mathit{foldl}\ (\oplus)\ s\ [\,] &&= s \\
&\mathit{foldl}\ (\oplus)\ s\ (x : xs) &&= \mathit{foldl}\ (\oplus)\ (s \oplus x)\ xs \quad .
\end{aligned}$$

such that $\mathit{foldl}\ (\oplus)\ s\ [x_0, x_1, x_2]$ = $((s \oplus x_0) \oplus x_1) \oplus x_2$. It is like *foldr*, but associates the operands to the left.

Loop and Fold-Left

Well, not quite... if $(+)$ were not associative. What *loop* actually computes is a *foldl*. Recall

$$\begin{array}{lcl} \mathit{foldl} :: (b \to a \to b) \to b \to \mathit{List}\ a \to b & & \\ \mathit{foldl}\ (\oplus)\ s\ [\,] & = & s \\ \mathit{foldl}\ (\oplus)\ s\ (x : xs) & = & \mathit{foldl}\ (\oplus)\ (s \oplus x)\ xs \quad . \end{array}$$

If we define:

$$\begin{array}{lcl} \mathit{loop}\ (\oplus)\ [\,] & = & \mathit{get} \\ \mathit{loop}\ (\oplus)\ (x : xs) & = & \mathit{get} \gg\!\!= \lambda s \to \\ & & \quad \mathit{put}\ (s \oplus x) \gg \mathit{loop}\ (\oplus)\ xs \quad , \end{array}$$

We have $\mathit{put}\ s \gg \mathit{loop}\ (\oplus)\ xs = \mathit{put}\ (\mathit{foldl}\ (\oplus)\ s\ xs) \gg \mathit{get}$.

The do-Notation

Note: Haskell support a syntax called the **do**-notation which, informally, allows $m \gg\!\!= x \rightarrow n$ to be written as $n \leftarrow m$. The function *loop* can be written as:

$$\begin{aligned}
\mathit{loop}\ (\oplus)\ [\,] &= \mathit{get} \\
\mathit{loop}\ (\oplus)\ (x : xs) &= \mathbf{do}\ \ s \leftarrow \mathit{get} \\
&\qquad\quad\ \mathit{put}\ (s \oplus x) \\
&\qquad\quad\ \mathit{loop}\ (\oplus)\ xs\ \ ,
\end{aligned}$$

which looks more like an imperative program.

In this lecture I prefer to make $(\gg\!\!=)$ explicit, to facilitate reasoning. Nevertheless the **do**-notation is popular among programmers.

Implementing *MonadState*

A program that has access to a state can be seen as a function that maps an initial state to a pair of the returned value and the final state:[7]

$$\textbf{newtype}\ \mathit{StFun}\ \mathit{st}\ a \ = \ \mathit{MkS}\ (\mathit{st} \to (a, \mathit{st})) \ \ .$$

Given a computation having type *StFun st a* and an initial state, we may execute it by:

$$\begin{aligned} &\mathit{run} \ :: \ \mathit{StFun}\ \mathit{st}\ a \to \mathit{st} \to a \\ &\mathit{run}\ (\mathit{MkS}\ k)\ s_0 \ = \ k\ s_0 \ \ . \end{aligned}$$

[7]The same monad is called *State* in Haskell's library. We use a different name to avoid confusion.

State Operations

For all *st*, *StFun st* is a monad:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ (\mathit{StFun}\ st)\ \textbf{where} \\
\quad \mathit{return}\ x = \mathit{MkS}\ (\lambda s \to (x, s)) \\
\quad \mathit{MkS}\ f \mathbin{>\!\!>\!\!=} k = \mathit{MkS}\ (\lambda s \to \begin{array}[t]{l} \textbf{let}\ \begin{array}[t]{rl} (x, s') & = f\ s \\ \mathit{MkS}\ h & = k\ x \end{array} \\ \textbf{in}\ h\ s') \end{array}
\end{array}$$

,

and it is a state monad:

$$\begin{array}{l}
\textbf{instance}\ \mathit{MonadState}\ st\ (\mathit{StFun}\ st)\ \textbf{where} \\
\quad \begin{array}{lcl} \mathit{get} & = & \mathit{MkS}\ (\lambda s \to (s, s)) \\ \mathit{put}\ s' & = & \mathit{MkS}\ (\lambda s \to ((), s')) \end{array}
\end{array}$$

.

Destructive Update

This monad *StFun s* might not be what you expect: it simulates state passing and updating, but does not actually *update* any memory cell.

Currently in the Haskell library, an actual destructive update can be performed in two monads: *ST* and *IO*.

The *ST* Monad

The monad *ST* is more complex than the state effect we have discussed. Instead of *get* and *put*, there are methods for creating new references, new arrays, accessing and updating variables and arrays, etc.

In the type *ST s a*, the type variable *s* no longer stands for the type of the state. It is never instantiated. Instead it is used to ensure that the state cannot be leaked outside the monadic computation. See Launchbury and Peyton Jones for details.

It is possible to wrap *ST* with an interface such that it can work like *MonadState*. It is rarely done, though.

The *IO* Monad

The *IO* monad is where the programmer has access to all unsafe features: destructive update, reading, printing, file access, multi-threading using multable variables...

There is no way to privately “run” a computation of type *IO*. It can only be run by the system, in the topmost level.

ST can be converted to *IO*.

State and Non-Determinism

Can a monad be in both *MonadNondet* and *MonadState s*?

It must provide operators from both effects: *fail*, $(\|)$, *get*, *put*, satisfy all the laws, and perhaps some additional laws stating how the operators of different effects interact.

Local State

One possibility:

$$\textbf{newtype}\ StL\ st\ a\ =\ MkSL\ (st \to List\ (a, st))\ .$$

Each non-deterministic branch has its own final state.

How to define its *Monad*, *MonadNondet*, and *MonadState s* instance methods?

Global State

One might imagine a monad where all non-deterministic branches share one global state:

$$\textbf{newtype}\ \mathit{StG}\ \mathit{st}\ a \;=\; \mathit{MkSG}\ (\mathit{st} \rightarrow (\mathit{List}\ a, \mathit{st}))\ \ .$$

But no, this is not even a monad — the associativity of $(\gg\!=)$ fails to hold.

It is tricky designing monads involving multiple effects.

Yet, a real program may use several effects — exceptions, multiple states, input/output…

Modular Construction of Monads

Question: given a collection of desired effects, can we construct, in a modular manner, a monad supporting these effects?

Monad transformer is a popular approach. *Effect handing* is a recent new alternative.

Question: how to ensure that the monad so constructed obey the laws we specify?

This is still a research topic.