Functional Programming

Shin-Cheng Mu

FLOLAC 2024

0 To Begin With...

Prerequisites

If you have done the homework requested before this summer school, you should have familiarised yourself with

- values and types, and basic list processing,
- basics of type classes,
- defining functions by pattern matching,
- guards, $\mathbf{case}$, local definitions by $\mathbf{where}$ and $\mathbf{let}$,
- recursive definition of functions,
- and higher order functions.

Recommanded Textbooks

- *Introduction to Functional Programming using Haskell* [Bir98]. My recommended book. Covers equational reasoning very well.
- *Programming in Haskell* [Hut16]. A thin but complete textbook.
- *Learn You a Haskell for Great Good!* [Lip11], a nice tutorial with cute drawings!
- *Real World Haskell* [OSG98].
- *Algorithm Design with Haskell* [BG20].

1 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 non-terminating 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 can be modelled with more advanced theory, which we cannot cover in this course.

1.1 Induction on Natural Numbers

Recalling "Mathematical Induction"

- Let P be a predicate on natural numbers.
  - What is a predicate? Such a predicate can be seen as a function of type Nat → Bool.
  - So far, we see Haskell functions as simple mathematical functions too.
  - However, Haskell functions will turn out to be more complex than mere mathematical functions later. To avoid confusion, we do not use the notation Nat → Bool for predicates.
- 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.

1.1.1 Proof by Induction

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.

[1] Not a real Haskell definition.

- 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).

A Proof Generator

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

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

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.

1.1.2 Inductively Definition of Functions

Inductively Defined Functions

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

$$\begin{array}{ll} exp & :: Nat \to Nat \to Nat \\ exp\;b\;0 & = 1 \\ exp\;b\;(\mathbf{1}_+\,n) & = b \times exp\;b\;n\ \ . \end{array}$$

- Even addition can be defined inductively

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

- Exercise: define $(\times)$?

A Value Generator

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

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

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:

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

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

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

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

We have thus proved $P\,0$.

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

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

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.

1.1.3 A Set-Theoretic Explanation of Induction

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 $f\ x \leqslant x$.
  - 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

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

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

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 $Nat \subseteq P$. 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 $x \leqslant f\,x$.

With such construction we can talk about infinite data structures.

1.2 Induction on Lists

Inductively Defined Lists

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

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

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

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 $\bot$ involved.

Set-Theoretically Speaking...

The type $\mathit{List}\ a$ is the *smallest* set such that

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

Inductively Defined Functions on Lists

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

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

$$\begin{array}{ll} \mathit{map} & :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b \\ \mathit{map}\ f\ [\,] & = [\,] \\ \mathit{map}\ f\ (x : xs) & = F\ X : \mathit{map}\ f\ xs\ . \end{array}$$

  - $\mathit{sum}\ [1..10] = 55$
  - $\mathit{map}\ (\mathbf{1}_+)\ [1,2,3,4] = [2,3,4,5]$

[2]Not a real Haskell definition.

[3]What does that mean? Other courses in FLOLAC might cover semantics in more detail.

1.2.1 Append, and Some of Its Properties

List Append

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

$$\begin{array}{ll} (+\!\!+) & :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a \\ [\,] +\!\!+ ys & = ys \\ (x : xs) +\!\!+ ys & = x : (xs +\!\!+ ys)\ . \end{array}$$

- 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 $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$.

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

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

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

We have thus proved $P\ []$.

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

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

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 $\mathit{exp}\ b\ (m+n) = \mathit{exp}\ b\ m \times \mathit{exp}\ b\ n$, we expect that we will use induction to somewhere. Therefore the first part of the proof is to generate $\mathit{exp}\ b\ (m+n)$.
  - In the proof of associativity, we were working toward generating $xs +\!\!+ (ys +\!\!+ zs)$.
- 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:

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

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

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

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

1.2.2 More Inductively Defined Functions

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.)
- **Note** Terminology: an inductive definition, as we have seen, define "bigger" things in terms of "smaller" things. Recursion, on the other hand, is a more general term, meaning "to define one entity in terms of itself."
- 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

- $\mathit{filter}\ p\ xs$ keeps only those elements in xs that satisfy p.

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

Take and Drop

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

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

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

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

TakeWhile and DropWhile

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

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

- $\mathit{dropWhile}\ p\ xs$ drops the prefix from xs.

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

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

List Reversal

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

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

All Prefixes and Suffixes

- $\mathit{inits}\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$

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

- $\mathit{tails}\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$

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

Totality

- Structure of our definitions so far:

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

  - 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.

1.2.3 Other Patterns of Induction

Variations with the Base Case

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

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

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

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

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

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

  - We do not want to define map, filter again for $\mathit{List}^{+}\ a$. Thus we reuse $\mathit{List}\ a$ 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:

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

Zip

Another example:

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

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. $f\ (x : \mathit{xs}) = ..f\ \mathit{xs}..$). 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.

$$\begin{array}{ll} msort & :: List\ Int \to List\ Int \\ msort\ [\,] & = [\,] \\ msort\ [x] & = [x] \\ msort\ xs & = merge\ (msort\ ys)\ (msort\ zs)\ \ , \\ \quad \mathbf{where} & n = length\ xs\ `div`\ 2 \\ & ys = take\ n\ xs \\ & zs = drop\ n\ xs\ \ . \end{array}$$

  - 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:

$$\begin{array}{ll} f & :: Int \to Int \\ f\,0 & = 0 \\ f\ n & = f\ n\ \ . \end{array}$$

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

1.3 User Defined Inductive Datatypes

Internally Labelled Binary Trees

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

$$\mathbf{data}\ ITree\ a\ =\ \mathsf{Null} \mid \mathsf{Node}\ a\ (ITree\ a)\ (ITree\ a)\ \ ,$$

- on which we may inductively define functions:

$$\begin{array}{lcl} sumT & :: & ITree\ Nat \to Nat \\ sumT\ \mathsf{Null} & = & 0 \\ sumT\ (\mathsf{Node}\ x\ t\ u) & = & x + sumT\ t + sumT\ u\ \ . \end{array}$$

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

1. $minT :: Tree\ Nat \to Nat$, which computes the minimal element in a tree.
2. $mapT :: (a \to b) \to Tree\ a \to Tree\ b$, 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
  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, $minT\ (mapT\ (n+)\ t) = n + minT\ t$. That is, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.

Induction Principle for Other Types

- Recall that $\mathbf{data}\ Bool = False \mid True$. 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.

2 Program Derivation

2.1 Some Comments on Efficiency

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: $\mathbf{data}\ List\ a = [\,] \mid a : List\ a$.
- 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 $(\mathbin{+\!\!+})$:

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

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

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

- $(\mathbin{+\!\!+})$ 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.

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.

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

$$\begin{aligned} \textbf{let}\ & xs = [1..100] \\ & ys = update\ xs\ 10\ 20 \\ \textbf{in}\ & \ldots body \ldots \end{aligned}$$

- 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:

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

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

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

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 $O(n)$, where n is the length of the list.

- 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.

2.2 Expand/Reduce Transformation

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?

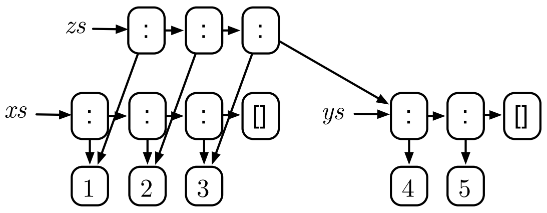

Figure 1: How $(+\!\!+)$ allocates new $(:)$ cells in the heap.

- The input is either empty or not. When it is empty:

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

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 $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map}\ \mathit{square}$, we have proved that

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

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

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

- 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.

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.

Example: Computing Polynomial

Given a list $\mathit{as} = [a_0, a_1, a_2 \ldots a_n]$ and x :: Int, the aim is to compute:

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

This can be specified by

$$\mathit{poly}\ x\ \mathit{as} = \mathit{sum}\ (\mathit{zipWith}\ (\times)\ \mathit{as}\ (\mathit{iterate}\ (\times x)\ 1))\ \ ,$$

where $\mathit{iterate}$ can be defined by

$$\begin{aligned}
& \mathit{iterate} :: (a \to a) \to a \to \text{List}\ a \\
& \mathit{iterate}\ f\ x = x : \mathit{map}\ f\ (\mathit{iterate}\ f\ x)\ \ .
\end{aligned}$$

Iterating a List

To get some intuition about $\mathit{iterate}$ let us try expanding it:

[5]Unless introduced in disciplined ways. For example, through a monad.

$$
\begin{aligned}
& \mathit{iterate}\ f\ x \\
= \quad & \{ \text{ definition of } \mathit{iterate} \} \\
& x : \mathit{map}\ f\ (\mathit{iterate}\ f\ x) \\
= \quad & \{ \text{ definition of } \mathit{map} \} \\
& x : \mathit{map}\ f\ (x : \mathit{map}\ f\ (\mathit{iterate}\ f\ x)) \\
= \quad & \{ \mathit{map} \text{ fusion } \} \\
& x : f\ x : \mathit{map}\ (f \cdot f)\ (\mathit{iterate}\ f\ x) \\
= \quad & \{ \text{ definitions of } \mathit{iterate} \text{ and } \mathit{map} \} \\
& x : f\ x : f\ (f\ x) : \mathit{map}\ (f \cdot f)\ (\mathit{map}\ f\ (\mathit{iterate}\ f\ x)) \\
= \quad & \{ \mathit{map} \text{ fusion } \} \\
& x : f\ x : f\ (f\ x) : \mathit{map}\ (f \cdot f \cdot f)\ (\mathit{iterate}\ f\ x) \quad \ldots
\end{aligned}
$$

Zipping with a Binary Operator

While $\mathit{iterate}$ generate a list, it is immediately truncated by $\mathit{zipWith}$:

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

Running the Specification

Try expanding $\mathit{poly}\ x\ [a, b, c, d]$, we get

$$
\begin{aligned}
& \mathit{poly}\ x\ [a, b, c, d] \\
= \ & \mathit{sum}\ (\mathit{zipWith}\ (\times)\ [a, b, c, d]\ (\mathit{iterate}\ (\times x)\ 1)) \\
= \quad & \{ \text{ expanding } \mathit{iterate} \} \\
& \mathit{sum}\ (\mathit{zipWith}\ (\times)\ [a, b, c, d] \\
& \quad (1 : (1 \times x) : (1 \times x \times x) : (1 \times x \times x \times x) : \\
& \quad \mathit{map}\ (\times x)^4\ (\mathit{iterate}\ (\times x)\ 1))) \\
= \ & a + b \times x + c \times x \times x + d \times x \times x \times x \quad .
\end{aligned}
$$

where f^4 denotes $f \cdot f \cdot f \cdot f$.

As the list gets longer, we get more $(\times x)$ accumulating. Can we do better?

The main calculation

$$
\begin{aligned}
& \mathit{poly}\ x\ (a : \mathit{as}) \\
= \quad & \{ \text{ definition of } \mathit{poly} \} \\
& \mathit{sum}\ (\mathit{zipWith}\ (\times)\ (a : \mathit{as})\ (\mathit{iterate}\ (\times x)\ 1)) \\
= \quad & \{ \text{ definition of } \mathit{iterate} \} \\
& \mathit{sum}\ (\mathit{zipWith}\ (\times)\ (a : \mathit{as}) \\
& \quad (1 : \mathit{map}\ (\times x)\ (\mathit{iterate}\ (\times x)\ 1))) \\
= \quad & \{ \text{ definitions of } \mathit{zipWith} \text{ and } \mathit{sum} \} \\
& a + \mathit{sum}\ (\mathit{zipWith}\ (\times)\ \mathit{as} \\
& \quad (\mathit{map}\ (\times x)\ (\mathit{iterate}\ (\times x)\ 1))) \\
= \quad & \{ \text{ see below } \} \\
& a + \mathit{sum}\ (\mathit{map}\ (\times x)\ (\mathit{zipWith}\ (\times) \\
& \quad \mathit{as}\ (\mathit{iterate}\ (\times x)\ 1))) \\
= \quad & \{ \mathit{sum} \cdot \mathit{map}\ (\times x) = (\times x) \cdot \mathit{sum} \} \\
& a + (\mathit{sum}\ (\mathit{zipWith}\ (\times)\ \mathit{as}\ (\mathit{iterate}\ (\times x)\ 1))) \times x \\
= \quad & \{ \text{ definition of } \mathit{poly} \} \\
& a + (\mathit{poly}\ x\ \mathit{as}) \times x \quad .
\end{aligned}
$$

Zip-Map Exchange

In the 4th step we used the property $\mathit{zipWith}\ (\times)\ \mathit{as} \cdot \mathit{map}\ (\times x) = \mathit{map}\ (\times x) \cdot \mathit{zipWith}\ (\times)\ \mathit{as}$.

It applies to any operator $(\otimes)$ that is associative. For an intuitive understanding:

$$
\begin{aligned}
& \mathit{zipWith}\ (\otimes)\ [a, b, c]\ (\mathit{map}\ (\otimes x)\ [d, e, f]) \\
= \ & [a \otimes (d \otimes x), b \otimes (e \otimes x), c \otimes (f \otimes x)] \\
= \quad & \{ \text{ associativity: } m \otimes (n \otimes k) = (m \otimes n) \otimes k \} \\
& [(a \otimes d) \otimes x, (b \otimes e) \otimes x, (c \otimes f) \otimes x] \\
= \ & \mathit{map}\ (\otimes x)\ (\mathit{zipWith}\ (\otimes)\ [a, b, c]\ [d, e, f]) \quad .
\end{aligned}
$$

We can do a formal proof if we want.

Distributivity

In the 5th step we used the property $\mathit{sum} \cdot \mathit{map}\ (\times x) = (\times x) \cdot \mathit{sum}$. For that we need distributivity between addition and multiplication.

We used that law to push sum to the right.

This is the crucial property that allows us to speed up poly: we are allowed to factor out common $(\times x)$.

Computing Polynomial

To conclude, we get:

$$
\begin{aligned}
& \mathit{poly}\ x\ [\,] = 0 \\
& \mathit{poly}\ x\ (a : \mathit{as}) = a + (\mathit{poly}\ \mathit{as}) \times x \quad ,
\end{aligned}
$$

which uses a linear number of $(\times)$.

Let the Symbols Do the Work!

How do we know what laws to use or to assume?

By observing the form of the expressions. Let the symbols do the work.

2.3 Tupling

Steep Lists

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

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

- 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

$$\begin{aligned} &steepsum &&:: List\ Int \to (Bool \times Int)\\ &steepsum\ xs &&= (steep\ xs, sum\ xs),\end{aligned}$$

- 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:

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

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.

2.4 Accumulating Parameters

Reversing a List

- The function $reverse$ is defined by:

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

- E.g. $reverse\ [1, 2, 3, 4] = ((([\,] +\!\!+ [4]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] = [4, 3, 2, 1]$.
- 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?

2.4.1 Fast List Reversal

Introducing an Accumulating Parameter

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

$$\begin{aligned} &revcat &&:: List\ a \to List\ a \to List\ a\\ &revcat\ xs\ ys &&= reverse\ xs +\!\!+ ys.\end{aligned}$$

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

$$reverse\ xs = revcat\ xs\ [\,].$$

Reversing a List, Base Case

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

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

Reversing a List, Inductive Case

Case $x : xs$:

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

Linear-Time List Reversal

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

$$\begin{array}{ll} revcat\ [\,]\ ys & = ys \\ revcat\ (x : xs)\ ys & = revcat\ xs\ (x : ys). \end{array}$$

- 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 $reverse\ xs = revcat\ xs\ [\,]$.
  - Hmm... it is like a *loop*, isn't it?

2.4.2 Tail Recursion and Loops

Tracing Reverse

$$\begin{array}{ll} & reverse\ [1,2,3,4] \\ = & revcat\ [1,2,3,4]\ [\,] \\ = & revcat\ [2,3,4]\ [1] \\ = & revcat\ [3,4]\ [2,1] \\ = & revcat\ [4]\ [3,2,1] \\ = & revcat\ [\,]\ [4,3,2,1] \\ = & [4,3,2,1] \end{array}$$

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

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

Tail Recursion

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

$$\begin{array}{ll} f\ x_1\ \ldots\ x_n & = \{\text{base case}\} \\ f\ x_1\ \ldots\ x_n & = f\ x'_1\ \ldots\ x'_n \end{array}$$

- 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.:

$$revcat\ xs\ ys = reverse\ xs +\!\!+\ ys.$$

- 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:

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

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

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

2.5 Conclusions

Conclusions

- Let the symbols do the work!
  - Algebraic manipulation helps us to separate the more mechanical parts of reasoning, from the parts that needs real innovation.
- For more examples of fun program calculation, see Bird [Bir10].
- For a more systematic study of algorithms using functional program reasoning, see Bird and Gibbons [BG20].

3 Monads and 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 *mutable* variable,
- raising an exception,
- file/terminal I/O,
- asking for the current time,
- tossing a coin / generating a random number,
- partialty (possible failure),
- non-determinism,
- non-termination... and many more.

How to talk about all these effects?

Hint: in *functional* programming, everything is a function!

Modelling Effects

- Given

  data Maybe a = Return a | Exception Msg ,

  a function mapping A to B, possibly raising an exception, can be modelled by A → Maybe B.
- Given a global *mutable* variable of type S, a computation that may modify the variable can be modelled by a function S → S.
- A computation that emits a value of type B while modifying the said variable is modelled by S → (B, S).
- What about a function that maps A to B and may possibly modify the said variable of type S?

Example: A Pure Expression Evalulator

Consider the following type of expressions:

```
data Expr = Num Int | Neg Expr | Add Expr Expr .
```

How to evaluate an expression?

```
eval :: Expr → Int
eval (Num n)      = n
eval (Neg e)      = −(eval e)
eval (Add e1 e2)  = eval e1 + eval e2 .
```

3.1 Exceptions

What if we have division?

```
data Expr = Num Int | Neg Expr | Add Expr Expr
          | Div Expr Expr .
```

Division by zero should raise an exception.

We use the datatype Maybe in the Haskell Prelude:

```
data Maybe a = Just a | Nothing .
```

A value of type Maybe a might contain an a, or nothing.

What about letting *eval* return Maybe Int?

Hmm... let's try.

```
eval :: Expr → Maybe Int
eval (Num n)     = Just n
eval (Neg e)     =
  case eval e of
    Just v   → Just (−v)
    Nothing → Nothing
eval (Add e0 e1) =
  case eval e0 of
    Just v0  → case eval e1 of
                 Just v1  → Just (v0 + v1)
                 Nothing → Nothing
    Nothing → Nothing

eval (Div e0 e1) =
  case eval e1 of
    Just v1  → if v1 == 0 then Nothing
               else case eval e0 of
                 Just v0 → Just (div v0 v1)
                 Nothing → Nothing
    Nothing → Nothing
```

Hmmm.. a bit repetitive, isn't it?

Needing some Abstraction...

The idea is to represent a function A $\rightarrow$ B that may fail (that is, a partial function) by a total function A $\rightarrow$ Maybe B.

It works!

It is just rather tedious because we suddenly have lots of repetitive details to take care of.

This is when we need some abstraction.

Return and Bind for Maybe

Observing the repetitive pattern, if we define

$$\begin{array}{l}
(\gg\!=) : \text{Maybe}\ a \rightarrow (a \rightarrow \text{Maybe}\ b) \rightarrow \text{Maybe}\ b \\
mx \gg\!= f = \textbf{case}\ mx\ \textbf{of} \\
\quad \text{Just}\ x \quad \rightarrow f\ x \\
\quad \text{Nothing} \rightarrow \text{Nothing}\ \ ,
\end{array}$$

(where $(\gg\!=)$ is pronounced "bind") or equivalently,

$$\begin{array}{l}
(\gg\!=) : \text{Maybe}\ a \rightarrow (a \rightarrow \text{Maybe}\ b) \rightarrow \text{Maybe}\ b \\
\text{Just}\ x \quad \gg\!= f = f\ x \\
\text{Nothing} \gg\!= f = \text{Nothing}\ \ ,
\end{array}$$

and, for reasons to be clear later, let $return$ = Just...

The function $eval$ can be abbreviated to:

$$\begin{array}{lll}
eval\ (\text{Num}\ n) & = & return\ n \\
eval\ (\text{Neg}\ e) & = & eval\ e\ \gg\!= \lambda v \rightarrow return\ (-v) \\
eval\ (\text{Add}\ e_0\ e_1) & = & eval\ e_0 \gg\!= \lambda v_0 \rightarrow \\
 & & eval\ e_1 \gg\!= \lambda v_1 \rightarrow \\
 & & return\ (v_0 + v_1) \\
eval\ (\text{Div}\ e_0\ e_1) & = & eval\ e_1 \gg\!= \lambda v_1 \rightarrow \\
 & & \quad \textbf{if}\ v_1 == 0\ \textbf{then}\ \text{Nothing} \\
 & & \quad \textbf{else}\ eval\ e_0 \gg\!= \lambda v_0 \rightarrow \\
 & & \qquad return\ (v_0\ `div`\ v_1)\ \ .
\end{array}$$

Effects Are Marked by Types

Notice how we mark the existence of side effects by types.

- a denotes a pure value;
- while Maybe a is an *effectful* computation that may return a value of type a, or fail.

The principle applies to other effects in this lecture. Each effect will be represented by a type.

Monads, Generally Speaking

Maybe is just one instance. Generally speaking, a type constructor m and two operators $return$ and $(\gg\!=)$ constitute a *monad*:

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

That's not all — $return$ and $(\gg\!=)$ are supposed to satisfy some properties, to be discussed later.

Monads Denote Computation

Let m be a monad. We often use $m\ a$ to denote a *computation* that, if executed, might yield a result of type a.

Executing the computation may incur some side effects.

- Failing to return a result is a side effect.
- We will see examples of other side effects.

For $e :: a$, $return\ e :: m\ a$ denotes a computation that simply returns e, with no side effects.

What Is This $(\gg\!=)$ All About?

Bind $(\gg\!=)$ is like an enhanced function application:

- $f\ x$, where f :: A $\rightarrow$ B and x :: A, applies f to x.
- Recall that $m\ a$ denotes a *computation* that may yield a value of type a, while also incurs some side effects.
- $p \gg\!= f$, where f :: A $\rightarrow$ m B and p :: m A, also "applies" f to p. However, evaluating p might incur some side effects. If the computation succeeds, we may extract some value of type A, which is passed to f, which in turn yields a computation m B.

Failure and Catching

The idea of exception handling is not tied to the datatype Maybe. To be more general, let

$$fail = \text{Nothing}\ \ .$$

We can also define

$$\begin{array}{l}
catch :: \text{Maybe}\ a \rightarrow \text{Maybe}\ a \rightarrow \text{Maybe}\ a \\
catch\ (\text{Just}\ x)\ hdl = \text{Just}\ x \\
catch\ \text{Nothing}\ hdl = hdl\ \ .
\end{array}$$

Consecutive Product

How to multiply a sequence of numbers?

$$\begin{array}{lll}
prod\ [\,] & = & 1 \\
prod\ (x : xs) & = & x \times prod\ xs\ \ .
\end{array}$$

Hmm... can we stop early when there is a zero?

$$\begin{array}{lll}
work\ [\,] & = & return\ 1 \\
work\ (0 : xs) & = & fail \\
work\ (x : xs) & = & work\ xs \gg\!= \lambda y \rightarrow \\
 & & return\ (x \times y)\ \ , \\
\end{array}$$

$$fastprod\ xs = catch\ (work\ xs)\ (return\ 0)\ \ .$$

How do we show that $fastprod$ is "correct"?

We want to prove that, for all xs,

$$fastprod\ xs = return\ (prod\ xs)\ \ .$$

Hmm... we need to know some more properties of $return$, $(\gg\!=)$, $fail$ and $catch$.

3.2 Environments

For the next example we want to allow expressions to have **let**-defined local variables, e.g.

$$\textbf{let } x = 3 \textbf{ in } x + (\textbf{let } x = 4 \textbf{ in } x) + (-x)$$

should evaluate to $3 + 4 + (-3) = 4$.

Extending Expr

To represent such expressions we extend the Expr type

```
data Expr = Num Int | Neg Expr | Add Expr Expr
          | Var Name | Let Name Expr Expr ,
```

where **type** Name = String.

The previous expression can be represented by:

```
Let "x" (Num 3)
  (Add (Add "x" (Let "x" (Num 4) (Var "x")))
    (Neg (Var "x"))) .
```

Environment

So, what is the value of $x + 2$?

We don't know, unless we know the value of x.

An *environment* is a mapping from variables to values. For now, we denote it by:

```
type Env = [(Name, Int)] .
```

We can also define a function *lookup* :: Env → Maybe Int.

The *meaning* of an expression is Env → Int.

Evaluating an Expression Given an Environment

Now our *eval* converts an Expr to Env → Int.

```
eval :: Expr → Env → Int
eval (Num n)     env = n
eval (Neg e)     env = −(eval e env)
eval (Add e0 e1) env =
  eval e0 env + eval e1 env .
```

Looking up Variables

When we encounter a variable, we look up the environment:

```
eval (Var x) env =
  case lookup env x of Just v → v .
```

Wait, what if *lookup* returns Nothing?

To be discussed later. For now, we just let *eval* (truly) fail.

Extending Environment

To evaluate **let** $x = e_0$ **in** e_1, we evaluate e_0, and evaluate e_1 in an *extended* environment:

```
eval (Let x e0 e1) env =
  let v = eval e0 env
  in  eval e1 ((x, v) : env) .
```

Env → a is a Monad

Again, that works. However, manually passing the environment around can be error-prone. We can hide the details in a monad — called a *Reader* monad by convention.

Define **type** Reader e $a = e \to a$, and

```
return :: a → Reader a
return x env = x ,
(>>=) :: Reader a → (a → Reader b) → Reader b
(mx >>= f) env = f (mx env) env .
```

Evaluation in a Reader Monad

Now we redefine *eval* using *return* and (>>=). The first three cases are:

```
eval :: Expr → Reader Int
eval (Num n)     = return n
eval (Neg e)     = eval e  >>= λv → return (−v)
eval (Add e0 e1) = eval e0 >>= λv0 →
                   eval e1 >>= λv1 →
                   return (v0 + v1) .
```

Exactly the same as that in Maybe monad! We have indeed discovered a common pattern.

Retrieving and Updating the Environment

For the next two cases we define two *methods*.

Function *ask* returns the envionment:

```
ask :: Reader Env
ask env = env ,
```

while *local* updates the environment:

```
local :: (Env → Env) → Reader a → Reader a
local f p env = p (f env) .
```

***eval* continued...**

We may then define:

```
eval (Var x) =
  ask >>= λenv →
  case lookup env x of Just v → return v
eval (Let x e0 e1) =
  eval e0 >>= λv →
  local ((x, v):) (eval e1) .
```

A Slight Generalisation

For ease of discussion we have assumed that the type of the environment is fixed to Env.

We can generalize Env to a parameter. That is,

$$\mathbf{type}\ \text{Reader}\ e\ a = e \to a$$

What are the types of $return$, $(\gg\!=)$, ask, and $local$? What about their implementations?

In Reality...

To *overload* the same symbols ($return$ and $(\gg\!=)$) for both Maybe and Reader, they have to be declared instances of type class Monad.

However, the type system of Haskell has a limitation that type synonyms cannot be instances of type classes.

Instance Declaration

Therefore we have to wrap $e \to a$ in a **data** definition: [6]

```
data Reader e a = Rdr (e → a)
instance Monad (Reader e) where
  return x      = Rdr (λenv → x)
  Rdr mx >>= f  = Rdr (λenv → f (mx env) env)  .
```

Functor and Applicative Instances

Furthermore, a monad is also an instance of some other useful mathematical structures, such as an *applicative functor*, that are also useful in Haskell.

To reflect that, Haskell wants us to declare:

```
instance Functor (Reader e) where
  fmap = liftM
instance Applicative (Reader e) where
  pure = return
  (⟨*⟩) = ap
```

You need to do so for every monad you define.

We cannot cover the details in this lecture, unfortunately.

What About Missing Variables?

But wait... what if $lookup$ cannot find the variable?

Our program has to return an exception.

Therefore we actually need a monad that allows *two* effects.

Hmmm... what about the following type:

$$\mathbf{type}\ \text{RE}\ e\ a = e \to \text{Maybe}\ a\ \ ?$$

How do we implement the following functions?

[6] In fact people generally use a **newtype** in this case, which is different from **data** in subtle ways. But we choose not to complicate the matter.

```
return :: a → RE e a
(>>=)  :: RE e a → (a → RE e b) → RE e b
fail   :: RE e a
catch  :: RE e a → RE e a → RE e a
ask    :: RE e e
local  :: (e → e) → RE e a → RE e a
```

3.3 A Top-Down View of Monads

So far we have seen two examples of talking about effects using monads.

- Exception is modelled by the type Maybe, with two methods $fail$ and $catch$.
- Reading from a context is modelled by the type Reader, with methods ask and $local$.
- Wait... we actually have *three* examples: we needed a monad having both effects, and both their methods.

What about the general pattern?

A Top-Down Approach

So far we have been taking a bottom-up approach to comprehend monads. We start with writing programs without monads, spot the repetitive patterns, and notice that there is a monad to be discovered. We then construct the specific monad, and eventually construct a monadic version of the program.

A complimentary, top-down approach is also useful: we start with writing monadic programs, assuming the effects we want and properties they should satisfy. Then we thinking about how to implement the monad that does satisfy the properties.

To Talk About An Effect

Say you want to model an effect (or the combination of some effects) in your program.

- Think about what operations (methods) this effect may need,
- and what properties they should satisfy.
- Create a datatype modelling this effect.
- Implement its $return$ and $(\gg\!=)$,
- and its methods, such that all properties are indeed satisfied.

Monad Laws

$return$ and $(\gg\!=)$ cannot be implemented arbitrarily. To model computations property, it is demanded that they satisfy the following *monad laws*:

$$\begin{array}{lr}
\textbf{left identity} & return\ x \gg\!= k = k\ x\ \ , \\
\textbf{right identity} & mx \gg\!= return = mx\ \ , \\
\textbf{associativity} & (mx \gg\!= k_1) \gg\!= k_2 = \\
 & mx \gg\!= (\lambda x \rightarrow k_1\ x \gg\!= k_2)\ \ .
\end{array}$$

A Type Class For Failable Monads

Now assume that we want the monad to support $fail$ and $catch$. To be abstract, we do not assume what datatype it is, but declare types that support these operators as classes:

```
class Monad m ⇒ MonadFail m where
  fail :: m a
class MonadFail m ⇒ MonadCatch m where
  catch :: m a → m a → m a
```

Note On The Use Of Type Classes

Two things to note:

- Type classes are *not* necessary to talk about monads! We use type class just to make it clear that our discussions are not tied to a particular implementation.
- Our type classes are different from those in standard Haskell Prelude, but closer to that of Gibbons and Hinze [GH11].

Laws Regarding Exceptions

For exceptions, we may want the following properties:

$$\begin{array}{r}
catch\ fail\ h = h\ \ , \\
catch\ mx\ fail = m\ \ , \\
catch\ mx\ (catch\ h\ h') = catch\ (catch\ m\ h)\ h'\ \ .
\end{array}$$

Note that means $catch$ and $fail$ form a *monoid*.

And this is how $catch$ and $fail$ interact with $return$ and $(\gg\!=)$.

$$\begin{array}{r}
catch\ (return\ x)\ h = return\ x\ \ \ , \\
fail \gg\!= f = fail\ \ \ ,
\end{array}$$

Looks reasonable... what about when $catch$ meets $(\gg\!=)$? Do we have

$$catch\ mx\ h \gg\!= f = catch\ (mx \gg\!= f)\ (h \gg\!= f)\ \ ?$$

Unfortunately no. See the practicals.

A Type Class For Readers

Any monad m that supports ask and $local$ is in the class MonadReader:

```
class Monad m ⇒ MonadReader e m where
  ask   :: m e
  local :: (e → e) → m a → m a
```

Laws Regarding Readers

For readers, we may want the following properties. Firstly, regarding ask,

$$\begin{array}{l}
ask \gg\!= \lambda v \rightarrow return\ e = return\ e\ \ , \\
ask \gg\!= \lambda v_0 \rightarrow ask \gg\!= \lambda v_1 \rightarrow f\ v_0\ v_1 = \\
\quad ask \gg\!= \lambda v \rightarrow f\ v\ v\ \ .
\end{array}$$

Secondly, regarding $local$:

$$\begin{array}{ll}
local\ g\ (return\ e) = return\ e\ \ , & \\
local\ g\ (p \gg\!= f) & = local\ g\ p \gg\!= (local\ g \cdot f)\ \ .
\end{array}$$

Finally, when $local$ meets ask:

$$local\ g\ ask = ask \gg\!= (return \cdot g)\ \ .$$

Separation of Concerns

The laws are used to reason about monadic programs — assuming that the monad exists and obey the laws.

Independently, we design a datatype for the monad and implement the methods, ensuring that they satisfy the laws.

Alternative Implementations

Maybe is not the only implementation of MonadFail and MonadCatch, and Reader is not the only implementation of MonadReader.

Whatever the implementation is, it should satisfy the relevant laws.

Combining Effects

Can one monad implementation be instances of MonadFail, MonadCatch, and MonadReader?

Yes. When a monad implements multiple effects, it is supposed to

- satisfy laws of all the effects, as well as
- "tensor" of these effects, meaning that operators from each effects commute with each other.

Having chosen some effects, how do we implement a monad that meets the above requirements?

Commutivity of Effects

For example,

$$\begin{aligned} ask \gg\!= \lambda v \to \mathit{fail} &= \mathit{fail} \quad , \\ \mathit{local}\ f\ \mathit{fail} &= \mathit{fail} \quad , \end{aligned}$$

$$\begin{aligned} & ask \gg\!= \lambda v \to catch\ mx\ my = \\ & \quad catch\ (ask \gg\!= \lambda v \to mx) \\ & \qquad\quad (ask \gg\!= \lambda v \to my) \quad , \\ & local\ f\ (catch\ mx\ my) \\ & \quad catch\ (local\ f\ mx)\ (local\ f\ my) \quad . \end{aligned}$$

Composing Monads?

It is known that monads are difficult to compose, in the sense that once two monads are implemented, it is hard to combine them to form a monad having both their effects.

People have come up with various methods, e.g. *monad transformers*[LHJ95], and *effect handling*[KI15], which we cannot cover in this course.

However, properties of effects are easy to compose: just take the union (and "tensor") of all their properties.

3.4 Interlude: Alternative Notations

The do Notation

To simplify and encourage the use of monads, Haskell provides a more concise notation, enclosed in the keyword **do**. For example:

$$\begin{aligned} & eval :: \mathsf{Expr} \to \mathsf{Reader\ Int} \\ & eval\ (\mathsf{Num}\ n) \quad\ = return\ n \\ & eval\ (\mathsf{Neg}\ e) \quad\ \ = \mathbf{do}\ v \leftarrow eval\ e \\ & \qquad\qquad\qquad\qquad\quad return\ (-v) \\ & eval\ (\mathsf{Add}\ e_0\ e_1) = \mathbf{do}\ v_0 \leftarrow eval\ e_0 \\ & \qquad\qquad\qquad\qquad\quad v_1 \leftarrow eval\ e_1 \\ & \qquad\qquad\qquad\qquad\quad return\ (v_0 + v_1) \\ & eval\ (\mathsf{Var}\ x) \qquad = \mathbf{do}\ env \leftarrow ask \\ & \quad \mathbf{case}\ lookup\ env\ x\ \mathbf{of}\ \mathsf{Just}\ v \to return\ v \\ & eval\ (\mathsf{Let}\ x\ e_0\ e_1) = \mathbf{do}\ v \leftarrow eval\ e_0 \\ & \quad local\ ((x, v){:})\ (eval\ e_1) \quad . \end{aligned}$$

Not Assignments!

It gives you an impression that you were writing an imperative program. Doesn't $v \leftarrow eval\ e$ look like "assign the value of $eval\ e$ to variable e?

In fact, $v \leftarrow eval\ e$ is closer to **let** in nature: it declares a new local variable v, whose scope extends to the end of the **do**-block. It can be shadowed by other bindings, like in **let**.

Translation

To be more precise, this is how a program using **do** is translated to monadic operators:

$$\begin{aligned} & \mathbf{do}\ \{e\} && = e \\ & \mathbf{do}\ \{e; es\} && = e \gg\!= \backslash\_ \to \mathbf{do}\ \{es\} \\ & \mathbf{do}\ \{x \leftarrow e; es\} && = e \gg\!= \lambda x \to \mathbf{do}\ \{es\} \\ & \mathbf{do}\ \{\mathbf{let}\ x = e; es\} && = \mathbf{let}\ x = e\ \mathbf{in}\ \mathbf{do}\ \{es\} \quad . \end{aligned}$$

Monad Laws with do Notation

In this course we do not use **do** a lot, since I prefer to see the $(\gg\!=)$ operator for reasoning. However, use of **do** notation is predominant in practical monadic programs.

It is certainly possible to reason about programs using **do** notation. The monad laws, for example, are written:

$$\begin{aligned} & \mathbf{do}\ \{y \leftarrow return\ x; k\ y\} = \mathbf{do}\ \{k\ x\} \quad , \\ & \mathbf{do}\ \{x \leftarrow mx; return\ x\} = \mathbf{do}\ \{mx\} \quad , \\ & \mathbf{do}\ \{x \leftarrow mx; y \leftarrow k_1\ x; k_2\ y\} = \\ & \quad \mathbf{do}\ \{y \leftarrow \mathbf{do}\ \{x \leftarrow mx; k_1\ x\}; k_2\ y\} \quad . \end{aligned}$$

Which do you prefer?

Functor and Applicative, Again

To another extreme, we have operators that makes monadic programs more like expressions.

$$\begin{aligned} & (\langle\$\rangle) :: ... \Rightarrow (a \to b) \to m\ a \to m\ b \\ & f\ \langle\$\rangle\ mx = mx \gg\!= \lambda x \to return\ (f\ x) \quad , \\ & (\langle*\rangle) :: ... \Rightarrow m\ (a \to b) \to m\ a \to m\ b \\ & mf\ \langle*\rangle\ mx = mf \gg\!= \lambda f \to mx \gg\!= \lambda x \to \\ & \qquad\qquad\qquad return\ (f\ x) \quad . \end{aligned}$$

They are related to the Functor and Applicative class, but we do not go into the details.

With them, $eval$ can be defined as:

$$\begin{aligned} & eval\ (\mathsf{Num}\ n) \quad\ = return\ n \\ & eval\ (\mathsf{Neg}\ e) \quad\ \ = (0-)\ \langle\$\rangle\ eval\ e \\ & eval\ (\mathsf{Add}\ e_0\ e_1) = (+)\ \langle\$\rangle\ eval\ e_0\ \langle*\rangle\ eval\ e_1 \\ & \ldots \end{aligned}$$

3.5 State

In the *state* effect, we have *one*, *anonymous* global mutable variable of type s, and two methods:

- get retrieves the value of the mutable variable;
- $put\ v$ assigns the value v to the mutable variable.

A Type Class For States

Any monad m that supports get and put is in the class MonadState:

$$\begin{aligned} & \mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadState}\ s\ m\ \mathbf{where} \\ & \quad get :: m\ s \\ & \quad put :: s \to m\ () \end{aligned}$$

State, get, and put

If "a program that returns a value of type a while having access to a mutable variable of type s" is represented by a type State s a, the two methods have type:

$$\begin{aligned}&get :: \mathsf{State}\ s\ s\\&put :: s \to \mathsf{State}\ s\ ()\ .\end{aligned}$$

The monad operators have types:

$$\begin{aligned}&return :: a \to \mathsf{State}\ s\ a\\&(\gg\!=)\ :: \mathsf{State}\ s\ a \to (a \to \mathsf{State}\ s\ b) \to \mathsf{State}\ s\ b\ .\end{aligned}$$

The "Semicolon"

Note that put returns (), since put itself does not yield information. We use it merely for its side effect.

The value returned by put can be discarded.

Since it happens a lot we define a variation of $(\gg\!=)$:

$$\begin{aligned}&(\gg) :: \mathsf{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b\\&mx \gg my = mx \gg\!= \backslash\_ \to my\ .\end{aligned}$$

It is like a "semicolon" in imperative programs.

Laws For get and put

It should be reasonable to demand that they satisfy the following laws:

$$\begin{aligned}&\textbf{get-put} && get \gg\!= put = return\ ()\ ,\\&\textbf{put-get} && put\ e \gg get = put\ e \gg return\ e\ ,\\&\textbf{put-put} && put\ e_0 \gg put\ e_1 = put\ e_1\ ,\end{aligned}$$

and a law similar to that of ask:

$$\begin{aligned}\textbf{get-get}\quad & get \gg\!= \lambda v_0 \to\\& \quad get \gg\!= \lambda v_1 \to f\ v_0\ v_1 =\\& get \gg\!= \lambda v \to f\ v\ v\end{aligned}$$

Reasoning about Stateful Programs

With these laws we can already reason about programs that manipulate states.

See the practicals.

Implementation of State

And how do we implement State?

"A program that returns a value of type a while being able to read from and write to a mutable variable of type s" can be represented by a function:

$$\textbf{type}\ \mathsf{State}\ s = s \to (a, s)\ .$$

- Like Reader, the function takes the initial value of the variable as its input;
- unlike Reader, it returns not just an a, but also *the new value of the mutable variable.*

Implementing $return$ and $(\gg\!=)$ for State

How do we implement the monad operators for State? Try it yourself before checking the answers below..!

$$\begin{aligned}&return :: a \to \mathsf{State}\ s\ a\\&return\ x\ s = (x, s)\ ,\\&(\gg\!=) :: \mathsf{State}\ s\ a \to (a \to \mathsf{State}\ s\ b) \to \mathsf{State}\ s\ b\\&(mx \gg\!= k)\ s_0 = \textbf{let}\ (y, s_1) = mx\ s_0\\&\qquad\qquad\qquad\ \ \textbf{in}\ k\ y\ s_1\ .\end{aligned}$$

Implementing get and put

And how do we implement get and put?

$$\begin{aligned}&get :: \mathsf{State}\ s\ s\\&get\ s = (s, s)\ ,\\&put :: s \to \mathsf{State}\ s\ ()\\&put\ s_1\ s_0 = ((), s_1)\ .\end{aligned}$$

But of course, it is only one of the possible implementations.

In Reality...

Again, due to the limitation of Haskell's type system, the real code is not that sleek...

$$\begin{aligned}&\textbf{data}\ \mathsf{State}\ s\ a = \mathsf{St}\ (s \to (a, s))\ .\\&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ s)\ \textbf{where}\\&\quad return\ x \quad\ \ = \mathsf{St}\ ...\\&\quad \mathsf{St}\ mx \gg\!= k = \mathsf{St}\ ...\end{aligned}$$

Try finishing them yourself!

3.6 Nondeterminism

A pure function maps an input to exactly one output.

Not producing an output is an effect (MonadFail).

Possibly producing more than one output is also an effect.

A Type Class for Nondeterminism

Recall:

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadFail}\ m\ \textbf{where}\\&\quad fail :: m\ a\end{aligned}$$

Let $mx \langle|\rangle my$ denote "either mx or my".

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadAlt}\ m\ \textbf{where}\\&\quad (\langle|\rangle) :: m\ a \to m\ a \to m\ a\\&\textbf{class}\ (\mathsf{MonadFail}\ m, \mathsf{MonadAlt}\ m) \Rightarrow\\&\quad \mathsf{MonadNondet}\ m\ \textbf{where}\end{aligned}$$

Laws for Nondeterminism

What laws can we demand? We usually want fail and $(\langle|\rangle)$ to form a monoid:

$$\mathit{fail} \langle|\rangle mx = mx = mx \langle|\rangle \mathit{fail} \ ,$$
$$(mx \langle|\rangle my) \langle|\rangle mz = mx \langle|\rangle (my \langle|\rangle mz) \ .$$

We probably wish that the order and numbers of choices we make do not matter:

idempotency: $mx \langle|\rangle mx = mx \ ,$
commutivity: $mx \langle|\rangle my = my \langle|\rangle mx \ .$

Bind distributes into $(\langle|\rangle)$ ... from both directions!

$$(mx \langle|\rangle my) \gg= f = (mx \gg= f) \langle|\rangle (my \gg= f) \ ,$$
$$mx \langle|\rangle (\lambda x \to f\ x \langle|\rangle g\ x) = (mx \gg= f) \langle|\rangle (mx \gg= g) \ .$$

However, most implementations do not satisfy all the laws.

Lists as MonadNondet

If nondeterminism is the only effect we need, ideally, we can implement nondeterminism by sets.

Unfortunately we do not have real sets in Haskell. One can simulate nondeterminism using lists.

```
instance MonadFail List where
  fail = [ ]
instance MonadAlt List where
  (⟨|⟩) = (++)
instance MonadNondet List where
```

Note: in fact, Haskell does not allow us to say List here. We have to use [ ].

However, the implementation above does not satisfy idempotency and commutivity.

An "honest" representation of sets requires Eq a and incurs certain overhead. As a trade off, we may keep using List as long as we remember that the results we get are only correct "modulo idempotency and commutivity".

Nondeterminism and State

Can we have a monad that support both nondeterminism and state? It is supposed to satisfy laws of both effects.

There are at least two possible implementations:

type StNd $s\ a = s \to [(a, s)]$;
type StNd $s\ a = s \to ([a], s)$.

What are their differences?

The latter does not satisy the second distributivity law. However, there are cases when we find it useful.

3.7 Some Final Words

There are a lot about monads we cannot cover here.

- Other alternative, equivalent definition of monads (e.g. in terms of fmap, return, and join), and their roles in category theory.
- Relationship with other important constructs, e.g. applicative functors, monoids...

Our presentation of monads is a simplified one, different from that in Haskell Prelude, which is more tightly coupled with the "big picture."

Currently, combining and managing effects is still an active research area.

Our focus is on proving properties of monadic programs. However, this is a relatively undeveloped field.

References

[BG20] Richard S. Bird and Jeremy Gibbons. *Algorithm Design with Haskell.* Cambridge University Press, 2020.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[Bir10] Richard S. Bird. *Pearls of Functional Algorithm Design.* Cambridge University Press, 2010.

[GH11] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In Olivier Danvy, editor, *International Conference on Functional Programming*, pages 2–14. ACM Press, 2011.

[Hut16] Graham Hutton. *Programming in Haskell, 2nd Edition.* Cambridge University Press, 2016.

[KI15] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In John H Reppy, editor, *Symposium on Haskell*, pages 94–105. ACM Press, 2015.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011. Available online at `http://learnyouahaskell.com/`.

[OSG98] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly, 1998. Available online at `http://book.realworldhaskell.org/`.

A GHCi Commands

⟨*statement*⟩	evaluate/run ⟨*statement*⟩
`:`	repeat last command
`:\{\n ..lines.. \n:\}\n`	multiline command
`:add [*]<module> ...`	add module(s) to the current target set
`:browse[!] [[*]<mod>]`	display the names defined by module `<mod>` (`!`: more details; `*`: all top-level names)
`:cd <dir>`	change directory to `<dir>`
`:cmd <expr>`	run the commands returned by `<expr>::IO String`
`:ctags[!] [<file>]`	create tags file for Vi (default: `"tags"`) (`!`: use regex instead of line number)
`:def <cmd> <expr>`	define command `:<cmd>` (later defined command has precedence, `::<cmd>` is always a builtin command)
`:edit <file>`	edit file
`:edit`	edit last module
`:etags [<file>]`	create tags file for Emacs (default: `"TAGS"`)
`:help, :?`	display this list of commands
`:info [<name> ...]`	display information about the given names
`:issafe [<mod>]`	display safe haskell information of module `<mod>`
`:kind <type>`	show the kind of `<type>`
`:load [*]<module> ...`	load module(s) and their dependents
`:main [<arguments> ...]`	run the main function with the given arguments
`:module [+/-] [*]<mod> ...`	set the context for expression evaluation
`:quit`	exit GHCi
`:reload`	reload the current module set
`:run function [<arguments> ...]`	run the function with the given arguments
`:script <filename>`	run the script `<filename>`
`:type <expr>`	show the type of `<expr>`
`:undef <cmd>`	undefine user-defined command `:<cmd>`
`:!<command>`	run the shell command `<command>`

Commands for debugging

`:abandon`	at a breakpoint, abandon current computation
`:back`	go back in the history (after `:trace`)
`:break [<mod>] <l> [<col>]`	set a breakpoint at the specified location
`:break <name>`	set a breakpoint on the specified function
`:continue`	resume after a breakpoint
`:delete <number>`	delete the specified breakpoint
`:delete *`	delete all breakpoints
`:force <expr>`	print `<expr>`, forcing unevaluated parts
`:forward`	go forward in the history (after `:back`)
`:history [<n>]`	after :trace, show the execution history
`:list`	show the source code around current breakpoint
`:list identifier`	show the source code for `<identifier>`
`:list [<module>] <line>`	show the source code around line number `<line>`
`:print [<name> ...]`	prints a value without forcing its computation
`:sprint [<name> ...]`	simplifed version of :print
`:step`	single-step after stopping at a breakpoint
`:step <expr>`	single-step into `<expr>`
`:steplocal`	single-step within the current top-level binding
`:stepmodule`	single-step restricted to the current module
`:trace`	trace after stopping at a breakpoint

`:trace <expr>`	evaluate `<expr>` with tracing on (see `:history`)

Commands for changing settings

`:set <option> ...`	set options
`:seti <option> ...`	set options for interactive evaluation only
`:set args <arg> ...`	set the arguments returned by `System.getArgs`
`:set prog <progname>`	set the value returned by `System.getProgName`
`:set prompt <prompt>`	set the prompt used in GHCi
`:set editor <cmd>`	set the command used for `:edit`
`:set stop [<n>] <cmd>`	set the command to run when a breakpoint is hit
`:unset <option> ...`	unset options

Options for `:set` and `:unset`

`+m`	allow multiline commands
`+r`	revert top-level expressions after each evaluation
`+s`	print timing/memory stats after each evaluation
`+t`	print type after evaluation
`-<flags>`	most GHC command line flags can also be set here (eg. `-v2`, `-fglasgow-exts`, etc). For GHCi-specific flags, see User's Guide, Flag reference, Interactive-mode options.

Commands for displaying information

`:show bindings`	show the current bindings made at the prompt
`:show breaks`	show the active breakpoints
`:show context`	show the breakpoint context
`:show imports`	show the current imports
`:show modules`	show the currently loaded modules
`:show packages`	show the currently active package flags
`:show language`	show the currently active language flags
`:show <setting>`	show value of `<setting>`, which is one of [`args`, `prog`, `prompt`, `editor`, `stop`]
`:showi language`	show language flags for interactive evaluation