Functional Programming

Shin-Cheng Mu

FLOLAC 2026

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, **case**, local definitions by **where** and **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

  $\boxed{\mathbf{0}}$ P 0 holds;

  $\boxed{\mathbf{1}}$ 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]

  data Nat = 0 | $\mathbf{1}_+$ 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:

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

- Even addition can be defined inductively

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

- 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:

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

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

0 | goal | $P\,0$.

| proof | For all n, we reason:

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

1 | assume | $P\,m$.

| goal | $P\,(\mathbf{1}_+\,m)$.

proof For all n, we reason:

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

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:
  0. 0 is in Nat;
  1. if n is in Nat, so is $\mathbf{1}_+\ n$;
  2. there is no other Nat.
- If we define a function F from sets to sets: $F\ X = \{0\} \cup \{\mathbf{1}_+\ n \mid n \in X\}$, 0. and 1. above means that $F\ Nat \subseteq Nat$. That is, Nat is a prefixed-point of F.
- 2. 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]

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

- 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 $List\ a$ is the *smallest* set such that

0. [ ] is in $List\ a$;
1. 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:

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

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

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

[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

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

- 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

  [0] $P\ [\,]$ holds;

  [1] 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, $P\ [1, 2, 3]$?

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

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.

[0] **[goal]** P [].

[proof] For all ys and zs, we reason:

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

[1] **[assume]** $P\ xs$

[goal] $P\ (x : xs)$

proof For all ys and zs, we reason:

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

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:

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

- Exercise: prove that $length$ distributes into $(\mathbin{+\!\!+})$:

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

Concatenation

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

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

- Compare with sum.
- Exercise: prove $sum \cdot concat = sum \cdot map\ sum$.

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

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

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

Take and Drop

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

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

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

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

TakeWhile and DropWhile

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

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

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

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

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

List Reversal

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

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

All Prefixes and Suffixes

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

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

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

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

Totality

- Structure of our definitions so far:

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

  - 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.

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

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

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

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

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

  - We do not want to define map, $filter$ again for $List^{+}\ a$. Thus we reuse $List\ a$ and pretend that we were talking about $List^{+}\ a$.
  - 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:

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

Zip

Another example:

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

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.

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

  - 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:

$$\begin{array}{l} f :: Int \to Int \\ f\ 0 = 0 \\ f\ n = f\ n\ . \end{array}$$

- 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:

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

- 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]

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

0 P Null holds, and;

1 for every x, t, and u, if $P\ t$ and $P\ u$ holds, $P\ (\mathsf{Node}\ x\ t\ u)$ 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 **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

stmt0 $P\ False$ holds, and

stmt1 $P\ True$ 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.

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

2 Red-Black Tree

- A self-balancing binary search tree, often used to represent sets.
- Supports $O(\log\ n)$-time searching, insertion, and deletion.
- One possible representation:

$$\begin{aligned} &\textbf{data}\ \mathsf{RBTree}\ a = \mathsf{E}\ | \\ &\qquad \mathsf{N}\ \mathsf{Color}\ (\mathsf{RBTree}\ a)\ a\ (\mathsf{RBTree}\ a)\ , \\ &\textbf{data}\ \mathsf{Color} \quad = \mathsf{R}\ |\ \mathsf{B}\ . \end{aligned}$$

Constraints

- It is a binary search tree.
  - In $\mathsf{N}\ _\ t\ x\ u$, every label in t is less than x, every label in u is more than x. The same holds for t and u.
- Each node is either colored red or black.
  - E is implicitly considered black.
- The root is black.
- Red nodes do not have red children.
- The number of black nodes from the root to each leaf is the same.

Searching

Searching in a red-black tree is just like that in a binary search tree:

$$\begin{aligned} &search :: \mathsf{Int} \to \mathsf{RBTree\ Int} \to \mathsf{Bool} \\ &search\ \mathsf{E} \qquad\qquad = \mathsf{False} \\ &search\ (\mathsf{N}\ t\ x\ u)\ |\ k < x = \ldots \\ &\qquad\qquad\qquad\quad |\ k == x = \ldots \\ &\qquad\qquad\qquad\quad |\ k > x = \ldots \end{aligned}$$

Exercise: what if we want to return the found element in a Maybe?

Insertion

- To insert a new element, perform a search to determine where to insert.
- The inserted node shall have color red.
- This would temporarily break the constraint that a red node shall not have a red children. We perform balancing upwards to restore the constraint. See the next slide.
- Finally we set the root to black.

Tree Balancing

- The re-balancing strategy is *not* unique.
- The strategy we will consider, shown in Figure 1, was presented by Okasaki [Oka99].
- Having only four rules, it is significantly simpler than those you'd find in most textbooks (which needs 8 rules or more)!
- Why?
- More will be discussed in the practicals.

3 Folds On Lists

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

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

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

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

This pattern is extracted and called $foldr$:

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

3.1 The Ubiquitous $foldr$

Replacing Constructors

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

- 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 $id = foldr\ (:)\ [\,]$.

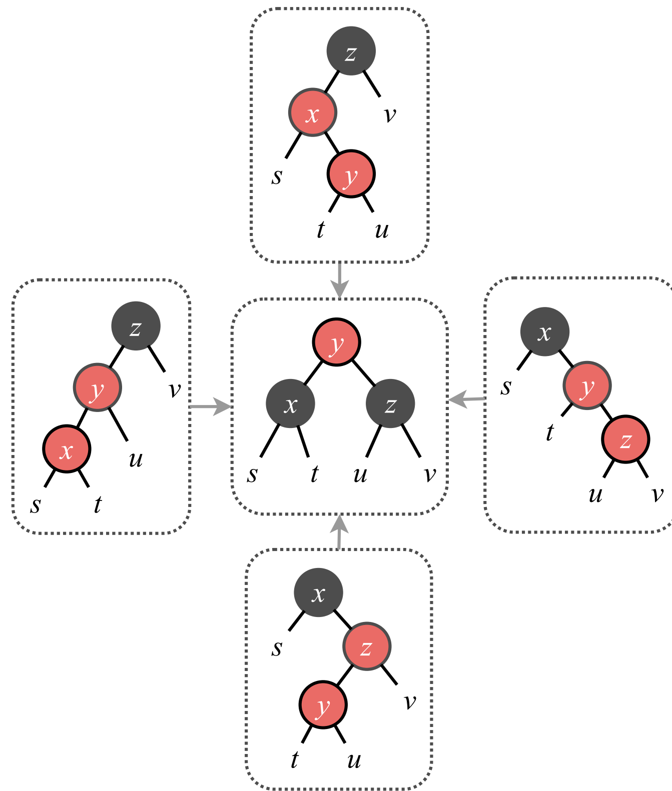

Figure 1: Red-black tree balancing [Oka99].

Some Trivial Folds on Lists

- Function max returns the least upper bound of elements in a list:

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

$max = foldr\ (\uparrow)\ {-\infty}.$

- This function is actually called $maximum$ in the standard Haskell Prelude, while max returns the maximum between its two arguments. For brevity, we denote the former by max and the latter by $(\uparrow)$.

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

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

$prod = foldr\ (\times)\ 1.$

- Function and returns the conjunction of a list:

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

$and = foldr\ (\wedge)\ true.$

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

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

$id = foldr\ (:)\ [\,].$

Some Functions We Have Seen...

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

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

- $concat = foldr\ (+\!\!+)\ [\,].$

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

Replacing Constructors

- Understanding $foldr$ from its type. Recall

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

- Types of the two constructors: $[\,] :: [a]$, and $(:) :: a \to [a] \to [a]$.

- *foldr* replaces the constructors:

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

3.2 The Fold-Fusion Theorem

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 1 ($foldr$-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:

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

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:

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

  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:

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

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 $steepsum$, for example, can be seen as fusing:

$$steepsum \cdot id \;=\; steepsum \cdot foldr\ (:)\ [\,].$$

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

3.3 More Useful Functions Defined as Folds

Longest Prefix

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

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

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

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

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

All Prefixes

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

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

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

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

All Suffixes

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

$$\begin{array}{lcl} tails\ [\,] &=& [[\,]], \\ tails\ (x : xs) &=& \mathbf{let}\ (ys : yss) \;=\; tails\ xs \\ && \mathbf{in}\ (x : ys) : ys : yss. \end{array}$$

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

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

Scan

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

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

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

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

4 Folds on Other Algebraic Datatypes

- 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:

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

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

$$\begin{array}{lcl} foldN &::& (a \to a) \to a \to Nat \to a \\ foldN\ f\ e\ 0 &=& e \\ foldN\ f\ e\ (\mathbf{1}_{+}\ n) &=& f\ (foldN\ f\ e\ n)\ . \end{array}$$

Examples of $foldN$

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

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

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

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

- $even = foldN\ not\ True$.

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

Fold-Fusion for Natural Numbers

Theorem 2 ($foldN$-Fusion)**.** Given $f :: a \to a$, $e :: a$, $h :: a \to b$, and $g :: b \to b$, we have:

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

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

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

Folds on Trees

- Example: internally labelled binary tree:

$$\begin{aligned} \mathbf{data}\ \mathsf{ITree}\ a = {} & \mathsf{Null} \\ \mid {} & \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a) \ . \end{aligned}$$

- Fold for ITree:

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

Folds on Trees

- Example: externally labelled binary tree:
- Some datatypes for trees:

$$\begin{aligned} \mathbf{data}\ \mathsf{ETree}\ a = {} & \mathsf{Tip}\ a \\ \mid {} & \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a) \ . \end{aligned}$$

- Fold for ETree:

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

Some Simple Functions on Trees

- To compute the size of an ITree:

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

- To sum up labels in an ETree:

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

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

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

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

5 Program Derivation

5.1 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:

$$\begin{array}{l}\mathbf{let}\ xs = [1,2,3] \\ \quad ys = [4,5] \\ \quad zs = xs \mathbin{+\!\!+} ys \\ \mathbf{in}\ \ldots body \ldots\end{array}$$

- The *body* may have access to all three values. Thus $+\!\!+$ 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:

$$\begin{array}{l}\mathbf{let}\ xs = [1..100] \\ \quad ys = update\ xs\ 10\ 20 \\ \mathbf{in}\ \ldots body \ldots\end{array}$$

- 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:

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

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

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

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.

5.2 Expand/Reduce Transformation

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: $sumsq = sum \cdot map\ square$.
- 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:

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

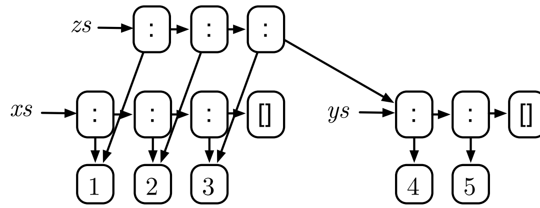

Figure 2: How (++) allocates new (:) cells in the heap.

Alternative Definition for *sumsq*

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

$$sumsq\ [\,] = 0$$
$$sumsq\ (x : xs) = square\ x + sumsq\ xs$$

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

$$f\ [\,] = 0$$
$$f\ (x : xs) = square\ x + f\ xs$$

- 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.

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

Example: Computing Polynomial

Given a list $as = [a_0, a_1, a_2 \ldots a_n]$ 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

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

where *iterate* can be defined by

$$iterate :: (a \to a) \to a \to \mathsf{List}\ a$$
$$iterate\ f\ x = x : map\ f\ (iterate\ f\ x)\ .$$

Iterating a List

To get some intuition about *iterate* let us try expanding it:

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

Zipping with a Binary Operator

While *iterate* generate a list, it is immediately truncated by *zipWith*:

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

Running the Specification

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

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

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

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

Zip-Map Exchange

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

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

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

We can do a formal proof if we want.

Distributivity

In the 5th step we used the property $sum \cdot map\ (\times x) = (\times x) \cdot sum$. 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:

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

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.

5.3 Tupling

Steep Lists

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

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

- 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

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

- 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:

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

Synthesised Program

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

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

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.

5.4 Accumulating Parameters

Reversing a List

- The function $\mathit{reverse}$ 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?

5.4.1 Fast List Reversal

Introducing an Accumulating Parameter

- Let us consider a generalisation of $\mathit{reverse}$. Define:

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

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

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

Reversing a List, Base Case

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

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

Reversing a List, Inductive Case

Case $x : xs$:

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

Linear-Time List Reversal

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

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

- A generalisation of $\mathit{reverse}$ is easier to implement than $\mathit{reverse}$ 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 $\mathit{reverse}\ xs = \mathit{revcat}\ xs\ [\,]$.
  - Hmm... it is like a *loop*, isn't it?

5.4.2 Tail Recursion and Loops

Tracing Reverse

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

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

$$\begin{aligned}
& \mathit{xs}, \mathit{ys} \leftarrow \mathit{XS}, [\,]; \\
& \mathbf{while}\ \mathit{xs} \neq [\,]\ \mathbf{do} \\
& \qquad \mathit{xs}, \mathit{ys} \leftarrow (\mathit{tail}\ \mathit{xs}), (\mathit{head}\ \mathit{xs} : \mathit{ys}); \\
& \mathbf{return}\ \mathit{ys}
\end{aligned}$$

Tail Recursion

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

$$\begin{aligned}
f\ x_1\ \ldots\ x_n &= \{\text{base case}\} \\
f\ x_1\ \ldots\ x_n &= f\ x_1'\ \ldots\ x_n'
\end{aligned}$$

- 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. $\mathit{reverse}\ \mathit{xs}$), 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:

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

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

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

5.5 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].

6 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?

Assume that we have a programming language supporting all these effects. Write an interpreter for this language, in a purely functional language.

Such an "interpreter" is actually a mathematical model of the language, offering a *semantics* for the language as well as the effects.

Example: A Pure Expression Evalulator

Our running example: consider the following type of expressions:

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

How to evaluate an expression?

$$\begin{aligned} &eval :: \mathsf{Expr} \rightarrow \mathsf{Int} \\ &eval\ (\mathsf{Num}\ n) \quad = n \\ &eval\ (\mathsf{Neg}\ e) \quad = -(eval\ e) \\ &eval\ (\mathsf{Add}\ e_1\ e_2) = eval\ e_1 + eval\ e_2 \ . \end{aligned}$$

6.1 A Language For Effects

- We are going to define a language that supports side effect. For now, imagine that is is represented by some data structure.
- The type $m\ a$ denotes an effectful program which, when executed, may compute a pure value of type a after possibly performing some effects.
- A program having type $m\ a$ in Haskell can be executed by an interpreter. For now, imagine that it is a function $run :: m\ a \rightarrow a$ (we will extend this later).
- The type m is written in small letters, meaning that it is a type variable: we have not yet decided what it is.

Effects Are Marked by Types

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

- a denotes a pure value;
- $m\ a$ is an *effectful* computation that may return a value of type a.

Syntax for the Language

- Let us define a datatype for the syntax tree of the language. How should an effectful language look like?
- One might imagine a language having a syntax like this:

$$\begin{aligned} &x \leftarrow comp_1; \\ &y \leftarrow comp_2; \\ &return\ (f\ x\ y) \ . \end{aligned}$$

  where $comp_1$ and $comp_2$ are computations that may be effectful, the value they compute are given to variables x and y, and the value of the entire computation is $f\ x\ y$.
- What about the types?
  - If $x :: a$ and $y :: b$, we must have $comp_1 :: m\ a$ and $comp_2 :: m\ b$.
  - If $f :: a \rightarrow b \rightarrow c$, the entire program has type $m\ c$.
  - The operator $return$ has type $c \rightarrow m\ c$.
- And what about the scope?
  - x can be used in line 2 and 3, but not in $comp_1$.
  - y can be used in line 3, but not in $comp_1$ and $comp_2$.

The Syntax in Haskell

- In Haskell, if we have the following "constructors" (the operator $(\gg\!=)$ is pronounced "bind"):

$$\begin{aligned} &return :: a \rightarrow m\ a \\ &(\gg\!=) \quad :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \ , \end{aligned}$$

- we can build the program above by:

$$\begin{aligned} &comp_1 \gg\!= (\lambda x \rightarrow \\ &comp_2 \gg\!= (\lambda y \rightarrow \\ &return\ (f\ x\ y))) \ . \end{aligned}$$

- Haskell allows us to omit the parentheses:

$$\begin{aligned} &comp_1 \gg\!= \lambda x \rightarrow \\ &comp_2 \gg\!= \lambda y \rightarrow \\ &return\ (f\ x\ y) \ . \end{aligned}$$

- For now, we see the program above as data (a syntax tree), even though we write $return$ in small letters.

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

Bind $(\gg=)$ is a bit like the "semicolon" in the syntax. It is also like an enhanced function application:

- $f\ x$, where f :: A → 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 → 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.

Monad and Monad Laws

- If we intend to let $return$ and $(\gg=)$ mean what we said, they should satisfy the following laws:

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

- For now, think of the equality above as "performing the same effects and return the same value(s)".
- A type m and two operators $return$ and $(\gg=)$ satisfying the laws above is called a *monad*.
  - There are other, equivalent ways to define monads.
  - Some people are scared off by this mathematical jargon. We wish to show in this lecture that monads need that be that hard.
- The laws are called *monad laws*.

The do Notation

To simplify and encourage the use of monads, Haskell provides a more concise notation, enclosed in the keyword **do**. We can actually write:

$$\begin{array}{l}\mathbf{do}\ x \leftarrow comp_1\\ \quad\ \ y \leftarrow comp_2\\ \quad\ \ return\ (f\ x\ y)\ \ .\end{array}$$

Expression Evaluation, Monadically

Back to our $eval$ example.

Instead of letting $eval$ be a function Expr → Int, let it *return a program that compute an* Int.

$$\begin{array}{ll}eval :: \mathsf{Expr} \rightarrow m\ \mathsf{Int} & \\ eval\ (\mathsf{Num}\ n) & = return\ n\\ eval\ (\mathsf{Neg}\ e) & = eval\ e \gg= \lambda v \rightarrow return\ (-v)\\ eval\ (\mathsf{Add}\ e_0\ e_1) & = eval\ e_0 \gg= \lambda v_0 \rightarrow\\ & \quad\ eval\ e_1 \gg= \lambda v_1 \rightarrow\\ & \quad\ return\ (v_0 + v_1)\ \ .\end{array}$$

This is for possible extension later.

Note: this is not yet valid Haskell — some information is missing in the type.

Expression Evaluation In do Notation

Equivalently, in **do**-notation:

$$\begin{array}{ll}eval :: \mathsf{Expr} \rightarrow m\ \mathsf{Int} & \\ eval\ (\mathsf{Num}\ n) & = return\ n\\ eval\ (\mathsf{Neg}\ e) & = \mathbf{do}\ v \leftarrow eval\ e\\ & \qquad\ \ return\ (-v)\\ eval\ (\mathsf{Add}\ e_0\ e_1) & = \mathbf{do}\ v_0 \leftarrow eval\ e_0\\ & \qquad\ \ v_1 \leftarrow eval\ e_1\\ & \qquad\ \ return\ (v_0 + v_1)\ \ .\end{array}$$

Which do you prefer?

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:

$$\begin{array}{ll}\mathbf{do}\ \{e\} & = e\\ \mathbf{do}\ \{e; es\} & = e \gg= \backslash_ \rightarrow \mathbf{do}\ \{es\}\\ \mathbf{do}\ \{x \leftarrow e; es\} & = e \gg= \lambda x \rightarrow \mathbf{do}\ \{es\}\\ \mathbf{do}\ \{\mathbf{let}\ x = e; es\} & = \mathbf{let}\ x = e\ \mathbf{in}\ \mathbf{do}\ \{es\}\ \ .\end{array}$$

Monad Laws with do Notation

In this course, I prefer to see the $(\gg=)$ operator for reasoning. However, use of **do** notation is predominant in practical monadic programs.

The monad laws, for example, are written:

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

Which do you prefer?

6.2 Exceptions

What if we have division in Expr?

$$\textbf{data}\ \mathsf{Expr} = \mathsf{Num\ Int} \mid \mathsf{Neg\ Expr} \mid \mathsf{Add\ Expr\ Expr} \mid \mathsf{Div\ Expr\ Expr}\ .$$

Division by zero should raise an exception.

Operators for Exception

We imagine having two more operators:

$$\begin{aligned} &fail\ \ \ :: m\ a\ ,\\ &catch :: m\ a \to m\ a \to m\ a\ .\end{aligned}$$

- $fail$ raises an exception. In our current application it simply means division by zero.
- $catch\ m\ hdl$ runs m and returns it value if it succeeds. If m fails, it runs hdl.

Note that they are still just the syntax of a program. The programs are not executed yet.

Evaluation with Failure

Now we define $eval$ by:

$$\begin{aligned} &eval\ (\mathsf{Num}\ n) &&= return\ n\\ &eval\ (\mathsf{Neg}\ e) &&= eval\ e \gg\!\!= \lambda v \to return\ (-v)\\ &eval\ (\mathsf{Add}\ e_0\ e_1) &&= eval\ e_0 \gg\!\!= \lambda v_0 \to\\ & &&\quad eval\ e_1 \gg\!\!= \lambda v_1 \to\\ & &&\quad return\ (v_0 + v_1)\\ &eval\ (\mathsf{Div}\ e_0\ e_1) &&= eval\ e_1 \gg\!\!= \lambda v_1 \to\\ & &&\quad\ \ \textbf{if}\ v_1 == 0\ \textbf{then}\ fail\\ & &&\quad\ \ \textbf{else}\ eval\ e_0 \gg\!\!= \lambda v_0 \to\\ & &&\quad\quad\quad return\ (v_0\ `div`\ v_1)\ .\end{aligned}$$

Consecutive Product

How to multiply a sequence of numbers?

$$\begin{aligned} &prod :: \mathsf{List\ Int} \to \mathsf{Int}\\ &prod\ [\,] &&= 1\\ &prod\ (x : xs) &&= x \times prod\ xs\ .\end{aligned}$$

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

$$\begin{aligned} &work :: \mathsf{List\ Int} \to m\ \mathsf{Int}\\ &work\ [\,] &&= return\ 1\\ &work\ (0 : xs) &&= fail\\ &work\ (x : xs) &&= work\ xs \gg\!\!= \lambda y \to\\ & &&\quad return\ (x \times y)\ ,\end{aligned}$$

$$\begin{aligned} &fastprod :: \mathsf{List\ Int} \to m\ \mathsf{Int}\\ &fastprod\ xs = catch\ (work\ xs)\ (return\ 0)\ .\end{aligned}$$

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 $catch$.

Laws Regarding Exceptions

- For exceptions, we may want the following properties:

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

- Note that means $catch$ and $fail$ form a *monoid*.
- And this is how $catch$ and $fail$ interact with $return$ and $(\gg\!\!=)$.

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

- Looks reasonable... what about when $catch$ meets $(\gg\!\!=)$? Is it reasonable to demand that

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

  Unfortunately no. See the practicals.

6.3 Reader

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

$$\textbf{data}\ \mathsf{Expr} = \mathsf{Num\ Int} \mid \mathsf{Neg\ Expr} \mid \mathsf{Add\ Expr\ Expr} \mid \mathsf{Var\ Name} \mid \mathsf{Let\ Name\ Expr\ Expr}\ ,$$

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:

$$\textbf{type}\ \mathsf{Env} = [(\mathsf{Name}, \mathsf{Int})]\ .$$

We can also define a function $lookup :: \mathsf{Env} \to \mathsf{Maybe\ Int}$.

"Having an Environment" as an Effect

- "Having an environment" can be seen as an effect.
- This effect is more commonly called the "Reader" effect, meaning that we can read from an environment.
- We extend our effect language with two operators:

$$\begin{aligned} &ask :: m\ \mathsf{Env}\ ,\\ &local :: (\mathsf{Env} \rightarrow \mathsf{Env}) \rightarrow m\ a \rightarrow m\ a\ . \end{aligned}$$

- ask returns the envionment.
- if the current environment is env, $local\ f\ m$ performs m in a local environment $f\ env$.

Evaluating an Expression Given an Environment

The usual cases stay the same.

$$\begin{aligned} &eval :: \mathsf{Expr} \rightarrow m\ \mathsf{Int}\\ &eval\ (\mathsf{Num}\ n) \quad\ = return\ n\\ &eval\ (\mathsf{Neg}\ e) \quad\ \ = eval\ e \gg\!= \lambda v \rightarrow return\ (-v)\\ &eval\ (\mathsf{Add}\ e_0\ e_1) = eval\ e_0 \gg\!= \lambda v_0 \rightarrow\\ &\qquad\qquad\qquad\quad\ eval\ e_1 \gg\!= \lambda v_1 \rightarrow\\ &\qquad\qquad\qquad\quad\ return\ (v_0 + v_1)\ . \end{aligned}$$

Cases relevant to variables and **let**:

$$\begin{aligned} &eval\ (\mathsf{Var}\ x) =\\ &\quad ask \gg\!= \lambda env \rightarrow\\ &\quad \mathbf{case}\ lookup\ env\ x\ \mathbf{of}\ \mathsf{Just}\ v \rightarrow return\ v\\ &eval\ (\mathsf{Let}\ x\ e_0\ e_1) =\\ &\quad eval\ e_0 \gg\!= \lambda v \rightarrow\\ &\quad local\ ((x, v){:})\ (eval\ e_1)\ . \end{aligned}$$

Laws Regarding Readers

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

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

Secondly, regarding $local$:

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

Finally, when $local$ meets ask:

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

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: exception, and reader.

6.4 Implementing the Exception and Reader Effect

Separation of Concerns

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

Independently, we design interpreters for the effect and implement the methods, ensuring that they satisfy the laws.

Running a Program With Exceptions

- Assume that exception is the only effect. How do you run a program?
- Let us use the Maybe type! If a program $m\ a$ successfully computes some x, we return Just x. Otherwise we return Nothing.

$$\begin{aligned} &run :: m\ a \rightarrow \mathsf{Maybe}\ a\\ &run\ (return\ x) = \mathsf{Just}\ x\\ &run\ (mx \gg\!= f) \ \ = \mathbf{case}\ run\ m\ \mathbf{of}\\ &\qquad\qquad\qquad\qquad \mathsf{Just}\ x \rightarrow f\ x\\ &\qquad\qquad\qquad\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing}\\ &run\ fail \qquad\quad\ \ = \mathsf{Nothing}\\ &run\ (catch\ mx\ hd) = \ldots \end{aligned}$$

Skip the Intermediate Layer

- But we immediately see that the intermediate layer is redundant. What if we just let m = Maybe?
- The operators can be implemented by:

$$\begin{aligned} &return\ x \qquad\ = \mathsf{Just}\ x\ ,\\ &\mathsf{Just}\ x \quad \gg\!= f = f\ x\\ &\mathsf{Nothing} \gg\!= f = \mathsf{Nothing}\ ,\\ &fail \qquad\qquad\ = \mathsf{Nothing}\ ,\\ &catch\ (\mathsf{Just}\ x)\ hdl = \mathsf{Just}\ x\\ &catch\ \mathsf{Nothing}\ hdl = \mathsf{Nothing}\ . \end{aligned}$$

- Then we simply have $run = id$.
- Of course, we are the obliged to show that the implementation above satisfise all the laws we demanded.

Evaluating a Program Needing an Environment

- Now assume that reader is the only effect.
- To execute a program needing an environment, you need an environment. Therefore, run has type:

$$run :: m\ a \rightarrow \mathsf{Env} \rightarrow a\ .$$

- That is, run takes a program, and an environment, and computes a result having type a.
- Or, the result of executing a program $m\ a$ is a function $\mathsf{Env} \rightarrow a$.
- Again, what if we let $m\ a = \mathsf{Env} \rightarrow a$?
- The operators can be implemented by:

$$\begin{array}{l}
return :: a \rightarrow m\ a \\
return\ x\ env \quad = x\ , \\
(\ggg) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
(mx \ggg f)\ env \quad = \mathbf{let}\ y = mx\ env\ \mathbf{in}\ f\ y\ env\ , \\
ask :: m\ \mathsf{Env} \\
ask\ env \qquad\quad = env\ , \\
local :: (\mathsf{Env} \rightarrow \mathsf{Env}) \rightarrow m\ a \rightarrow m\ a \\
local\ f\ mx\ env = mx\ (f\ env)\ .
\end{array}$$

Evaluating a Program with Exception and Reader

- What about a program that uses both exception and reader?
- To run a program, we need an environment. And we return a Maybe.

$$run :: m\ a \rightarrow \mathsf{Env} \rightarrow \mathsf{Maybe}\ a\ .$$

- Again, we can simply let $m\ a = \mathsf{Env} \rightarrow \mathsf{Maybe}\ a$.
- Try implementing $return$, $(\ggg)$, ask, $local$, $fail$, and $catch$?

Combining Effects

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, when operators of reader meet $fail$:

$$\begin{array}{r}
ask \ggg \lambda v \rightarrow fail = fail\ , \\
local\ f\ fail = fail\ .
\end{array}$$

When operators of reader meet $catch$:

$$\begin{array}{l}
ask \ggg \lambda v \rightarrow catch\ mx\ my = \\
\quad catch\ (ask \ggg \lambda v \rightarrow mx) \\
\qquad\quad (ask \ggg \lambda v \rightarrow my)\ , \\
local\ f\ (catch\ mx\ my) \\
\quad catch\ (local\ f\ mx)\ (local\ f\ my)\ .
\end{array}$$

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.

6.5 The Haskell Reality...

- The code we presented so far are *not* actual, executable Haskell code.
- Haskell uses *type classes* allow several datatypes to share the same names of operators.
- You do not need to understand type classes to understand monads and effects. It is actually easier to do without them when we calculate on paper.
- In an actual program, however, it is convenient using the type class mechanism.

The Monad **Typeclass**

Generally speaking, a type constructor m and two operators $return$ and $(\ggg)$ constitute a *monad*:

$$\begin{array}{l}
\mathbf{class}\ \mathsf{Monad}\ m\ \mathbf{where} \\
\quad return :: a \rightarrow m\ a \\
\quad (\ggg) \quad :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\ .
\end{array}$$

Note: this is still a simplification. In the standard Haskell, Monad is a subclass of Applicative... but we will avoid introducing the whole infrastructure here.

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:

$$\begin{array}{l} \mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadFail}\ m\ \mathbf{where} \\ \quad fail :: m\ a \\ \mathbf{class}\ \mathsf{MonadFail}\ m \Rightarrow \mathsf{MonadCatch}\ m\ \mathbf{where} \\ \quad catch :: m\ a \rightarrow m\ a \rightarrow m\ a \end{array}$$

Instance Declaration for Maybe

Somewhere in the library, one may declare that Maybe is instances of the classes above:

$$\begin{array}{l} \mathbf{instance}\ \mathsf{Monad}\ \mathsf{Maybe}\ \mathbf{where} \\ \quad return\ x = \ldots \\ \quad mx \gg\!\!= f \quad = \ldots \\ \mathbf{instance}\ \mathsf{MonadFail}\ \mathsf{Maybe}\ \mathbf{where} \\ \quad fail = \mathsf{Nothing}\ \ , \\ \mathbf{instance}\ \mathsf{MonadFail}\ \mathsf{Maybe}\ \mathbf{where} \\ \quad catch = \ldots \end{array}$$

A Type Class For Readers

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

$$\begin{array}{l} \mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadReader}\ env\ m\ \mathbf{where} \\ \quad ask\ \ :: m\ env \\ \quad local :: (env \rightarrow env) \rightarrow m\ a \rightarrow m\ a\ \ , \end{array}$$

where env is the type of environments — now a parameter, to be adapted to different use cases.

Instance Declaration

Similarly, we let

$$\mathbf{type}\ \mathsf{Reader}\ env\ a = env \rightarrow a\ \ .$$

Ideally, we wish to define

$$\begin{array}{l} \mathbf{instance}\ \mathsf{Monad}\ (\mathsf{Reader}\ env)\ \mathbf{where} \ldots \\ \mathbf{instance}\ \mathsf{MonadReader}\ (\mathsf{Reader}\ env)\ \mathbf{where} \\ \quad ask\ env \qquad\quad = env \\ \quad local\ f\ mx\ env = mx\ (f\ env)\ \ . \end{array}$$

Instance Declaration... In Reality

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

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

$$\begin{array}{l} \mathbf{data}\ \mathsf{Reader}\ env\ a = \mathsf{Rdr}\ (env \rightarrow a) \\ \mathbf{instance}\ \mathsf{Monad}\ (\mathsf{Reader}\ e)\ \mathbf{where} \\ \quad return\ x \qquad = \mathsf{Rdr}\ (\lambda env \rightarrow x) \\ \quad \mathsf{Rdr}\ mx \gg\!\!= f = \mathsf{Rdr}\ (\lambda env \rightarrow f\ (mx\ env)\ env)\ \ . \end{array}$$

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.

Class Constraints in Types

The actual type of $eval$ will be

$$\begin{array}{l} eval :: (\mathsf{Monad}\ m, \mathsf{MonadReader}\ m, \mathsf{MonadFail}\ m) \Rightarrow \\ \quad \mathsf{Expr} \rightarrow m\ \mathsf{Int}\ \ , \end{array}$$

meaning that we can use any datatype that supports the operators of Monad, MonadReader, and MonadFail.

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].

6.6 State

Recall that in $x \leftarrow m$ in the **do**-notation, x is *not* a mutable variable. That is covered by the *state* effect.

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

- $get :: m\ s$ retrieves the value of the mutable variable;
- $put :: s \rightarrow m\ ()$ assigns a value to the mutable variable.

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 ($\gg\!\!=$):

$$\begin{array}{l} (\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b \\ mx \gg my = mx \gg\!\!= \backslash_ \rightarrow my\ \ . \end{array}$$

It is like a "semicolon" in imperative programs.

[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.

Laws For get **and** put

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

$$\textbf{get-put} \quad get \gg\!= put = return\ () \ ,$$
$$\textbf{put-get} \quad put\ e \gg get = put\ e \gg return\ e \ ,$$
$$\textbf{put-put} \quad put\ e_0 \gg put\ e_1 = put\ e_1 \ ,$$

and a law similar to that of ask:

$$\textbf{get-get} \quad get \gg\!= \lambda v_0 \rightarrow get \gg\!= \lambda v_1 \rightarrow f\ v_0\ v_1 = get \gg\!= \lambda v \rightarrow f\ v\ v$$

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 the state effect?

"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 $s \rightarrow (a, s)$.

- 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

Let $m\ a = s \rightarrow (a, s)$ (for some fixed s). How do we implement the monad operators for state? Try it yourself before checking the answers below..!

$$return :: a \rightarrow m\ a$$
$$return\ x\ s = (x, s) \ ,$$
$$(\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$
$$(mx \gg\!= k)\ s_0 = \mathbf{let}\ (y, s_1) = mx\ s_0\ \mathbf{in}\ k\ y\ s_1 \ .$$

Implementing get **and** put

And how do we implement get and put?

$$get :: m\ s$$
$$get\ s = (s, s) \ ,$$
$$put :: s \rightarrow m\ ()$$
$$put\ s_1\ s_0 = ((), s_1) \ .$$

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...

$$\mathbf{data}\ \mathsf{State}\ s\ a = \mathsf{St}\ (s \rightarrow (a, s)) \ .$$
$$\mathbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ s)\ \mathbf{where}$$
$$return\ x = \mathsf{St} \ldots$$
$$\mathsf{St}\ mx \gg\!= k = \mathsf{St} \ldots$$

Try finishing them yourself!

6.7 Nondeterminism

A pure function maps an input to exactly one output.

Not producing an output is an effect.

Possibly producing more than one output is also an effect.

A Type Class for Nondeterminism

Recall:

$$\mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadFail}\ m\ \mathbf{where}$$
$$fail :: m\ a$$

We define:

$$\mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadAlt}\ m\ \mathbf{where}$$
$$(\langle|\rangle) :: m\ a \rightarrow m\ a \rightarrow m\ a$$
$$\mathbf{class}\ (\mathsf{MonadFail}\ m, \mathsf{MonadAlt}\ m) \Rightarrow \mathsf{MonadNondet}\ m\ \mathbf{where}$$

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

Laws for Nondeterminism

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

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

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

$$\textbf{idempotency:} \quad mx \langle|\rangle mx = mx \ ,$$
$$\textbf{commutivity:} \quad mx \langle|\rangle my = my \langle|\rangle mx \ .$$

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

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

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 commutativity.

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 → [(a, s)] ;
type StNd s a = s → ([a], s) .
```

What are their differences?

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

6.8 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/`.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[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

Command	Description
⟨*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

Command	Description
: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