Functional Programming

Shin-Cheng Mu

FLOLAC 2020

0 What's This All About?

Before we dive into technical details, it is worth spending some time pondering the main ideas we are going to cover in this course, and why they matter at all.

So, what's this all about?

0.1 The Isle of Knights and Knaves

- On a remote isle there live two kinds of people:
  - the *knights* always tell the truth, while
  - the *knaves* always lie.
  - Everyone on the isle is either a knight or a knave.
- You are at the entrance of a cave. Legend has it that the deep in the cave there buries a huge amount of gold... or a dragon that may swallow you alive. You see an old man. How do you form a question to know which is the case?

Warming Up

- With two islanders, A and B:
- A says: "if you ask B whether he is a knight, he would say 'Yes'.".
- What can you infer about A and B?

Exhaustive Enumeration?

- What matters more is how you solved the problem.
- Most people would exhaustively enumerate all possibilities.
  - "Suppose that A is a knight..."

Equivalence

- Abbreviate "A is a knight" to A.
- As a convention (among certain circles), we write logical equivalence, that is, "if and only if", or equality on booleans, as $(\equiv)$.
- Suppose that A said some sentence P. If A is a knight, P must be True. Otherwise P must be False.
- Thus, "A said P" can be denoted by $\mathsf{A} \equiv \mathsf{P}$.

Warming Up...

- $\mathsf{A} \equiv \mathsf{A}$ is always True.
  - Indeed, any person would say he/she is a knight.
- "A says: 'B is a knight'."
  - $\mathsf{A} \equiv \mathsf{B}$.
  - A and B are of the same kind.
- A says: "if you ask B whether he is a knight, he would say 'Yes'."

$$
\begin{aligned}
& \mathsf{A} \equiv (\mathsf{B} \equiv \mathsf{B}) \\
= {} & \mathsf{A} \equiv \mathsf{True} \\
= {} & \mathsf{A} \;\; .
\end{aligned}
$$

  - Thus we know that A is a knight. Nothing can be said about B.
- A says: "B and I are of the same kind!"

$$
\begin{aligned}
& \mathsf{A} \equiv (\mathsf{A} \equiv \mathsf{B}) \\
= {} & \quad \{ (\equiv) \text{ is associative } \} \\
& (\mathsf{A} \equiv \mathsf{A}) \equiv \mathsf{B} \\
= {} & \mathsf{True} \equiv \mathsf{B} \\
= {} & \mathsf{B} \;\; .
\end{aligned}
$$

- Thus we know that B is a knight. Nothing can be said about A.
- In fact, not many people know that $(\equiv)$ is associative.

Back to the Cave...

- Goal: design a question Q such that A answers Yes iff. there is gold in the cave.
- "A answers Yes to question Q" is also written $A \equiv Q$.
- Let G denote "there is gold in the cave."
- "A answers Yes to Q iff. there is gold in the cave."

$$\begin{aligned} & (A \equiv Q) \equiv G \\ = & (Q \equiv A) \equiv G \\ = & Q \equiv (A \equiv G) \ . \end{aligned}$$

- So the question is "Is 'You are a knight' equivalent to 'there is gold in the cave'?"

0.2 Abstraction

How Was the Problem Solved?

1. Turn the problem into mathematical formulae.
2. And then calculate, using the rules associated with the operators.

- The first step, called "abstraction", is harder.
- The second step is much easier, because we *let the symbols do the work!*
  - Well-designed symbols relieve us of the mental burden.
  - Recall how you calculate, say 17×24?
- Why does that concern us?

A Programming Language is a Symbolic, Formal System

- Because *a programming language is an abstract model, and a collections of symbols and their related rules, to relieve us of the mental burden of programming.*
- Abstraction: a programming language models the real world, while throws away some "unimportant parts".
- A formal system: a collection of symbols, and some rules to manipulate them.
  - We hope that a programming language is well-designed, such that it helps us to program.

Abstraction

- "What are the three most important factors in real estate?"
  - Location, location, and location.
- "What are the three most important factors in a programming language?"
  - Abstraction, abstraction, and abstraction — Paul Hudak.
- Abstration: the process of
  - extracting the underlying essence of a mathematical concept,
  - removing any dependence on real world objects with which it might originally have been connected, and
  - generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent phenomena.

Algebra

- "Mary had twice as many apples as John had. Mary found that half of her apples are rotten and thus throws them away. John ate one of his apples. Still, Mary has twice as many apples as John has. How many apples did they originally have?"

$$\begin{aligned} m &= 2 \cdot j, \\ m/2 &= 2 \cdot (j - 1). \end{aligned}$$

Abstraction

- From "Mary had twice as many apples as John..." to "$m = 2 \cdot j$":
  - extracted: values, and their relationships.
  - dropped: time, causality, ...
- What if time and causality turn out to be important? We need another abstraction.
  - Perhaps a stronger logic/algebra.

Not One, but Many Logics

- Propositional logic.
- (First-order) predicate logic: for all, exists...
- Modal logic: describing time and order.

- Separation logic: sharing of resources.
- Descriptive logic: concepts, and relationship between concepts.
- Each (or, some) logic corresponds to a type system in a programming language.

Abstraction in Imperative Programming Languages

- Abstraction of control structures: for-loops, while-loops...
- Procedure abstraction.
- Data abstraction: user-defined datatypes, instead of bits and bytes...
- What algebraic laws do they satisfy? Hmm... not many, unfortunately.

Abstractions of Other Paradigms

- Object-oriented programming: everything is an object!
- Functional programming: everything is a function!
- Logic programming: "computation = controlled deduction!" "algorithm = logic + control!"

A Language is an Abstraction

- A programming language is an abstract view toward computation, with attention on aspects the designers care about.
- To learn a language is to learn its view.
- Alan Perlis: *"A language that doesn't affect the way you think about programming, is not worth knowing."*
- In this term I hope you will see something that affects the way you think about programming.

0.3 Algebraic Manipulation

- What qualifies as a good abstraction?
- Our point of view: one that gives us more properties to manipulate with.

Algebraic Properties of Programs?

The following two programs are equivalent.

- ```
  s = 0; m = 0;
  for (i=0; i<=N; i++) s = a[i] + s;
  for (i=0; i<=N; i++) m = a[i] + m;
  ```
- ```
  s = 0; m = 0;
  for (i=0; i<=N; i++) {
     s = a[i] + s;
     m = a[i] + m;
  }
  ```

Is that easily seen? Can we transform one to another? Does the equivalence still hold if we replace the assignment by other statements?

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \Rightarrow 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \Rightarrow 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \Rightarrow 4$
- Not trivial. However, there is a linear time algorithm.
- $\begin{array}{cccccccccl} -1 & 3 & 1 & -4 & -1 & 1 & 2 & -1 & & \\ 3 & 4 & 1 & 0 & 2 & 3 & 2 & 0 & 0 & (\mathit{up} + \mathit{right}) \uparrow 0 \\ 4 & 4 & 3 & 3 & 3 & 3 & 2 & 0 & 0 & \mathit{up} \uparrow \mathit{right} \end{array}$
- The specification:

  $$\mathit{max}\ \{\mathit{sum}\ (i, j) \mid 0 \leqslant i \leqslant j \leqslant \mathrm{N}\}\ ,$$

  where $\mathit{sum}\ (i, j) = a\,[i] + a\,[i+1] \ldots + a\,[j-1]$.
  - *What* we want the program to do.
  - One can imagine a program using three nested loops.
- The program:

  ```
  s = 0; m = 0;
  for (i=0; i<=N; i++) {
     s = max(0, a[j]+s);
     m = max(m, s);
  }
  ```
  - *How* to do it.
- They do not look like each other at all!
- Moral: programs that appear "simple" might not be that simple after all!

Let the Symbols Do the Work!
"...the designer of the program had better regard the program as a sophisticated formula. And we also know that there is only one trustworthy way of designing a sophisticated formula, viz., derivation by means of symbol manipulation. We have to let the symbols do the work."
— E.W.Dijkstra, The next forty years. 14 June 1989.

Programming, and Programming Languages

- Correctness: that the behaviour of a program is allowed by the specification.
- Semantics: defining "behaviours" of a program.
- Programming: to code up a correct program!
- Thus the job of a programming language is to help the programmer to program,
  - either by making it easy to check that whether a program is correct,
  - or by ensuring that programmers may only construct correct programs, that is, disallowing the very construction of incorrect programs!

0.4 Reviews

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 nonterminating functions, infinite data structures, for now we will only consider its total, finite fragment.
- That is, we temporarily
  - consider only finite data structures,
  - demand that functions terminate for all value in its input type, and
  - provide guidelines to construct such functions.
- Infinite datatypes and non-termination 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 $\rightarrow$ 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 $\rightarrow$ 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]

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

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

A Proof Generator

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

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

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 $(\times)$?

[1] Not a real Haskell definition.

A Value Generator

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

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

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:

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

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]

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

- 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 $\perp$ involved.

Set-Theoretically Speaking...
The type *List a* is the *smallest* set such that

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

[2]Not a real Haskell definition.
[3]What does that mean? Other courses in FLOLAC might cover semantics in more detail.

Inductively Defined Functions on Lists

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

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

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

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

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

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

Appending is Associative

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

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

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

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

We have thus proved $P\ [\,]$.

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

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

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

Do We Have To Be So Formal?

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

Length

- The function *length* defined inductively:

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

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

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

- 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.

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

Take and Drop

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

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

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

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

TakeWhile and DropWhile

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

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

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

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

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

List Reversal

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

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

All Prefixes and Suffixes

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

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

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

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

Totality

- Structure of our definitions so far:

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

  - 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.

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

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

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

- 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:

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

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.

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

  - 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:

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

- 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:

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

- on which we may inductively define functions:

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

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
  1. P Null holds, and;
  2. for every x, t, and u, if $P\ t$ and $P\ u$ holds, $P\ (\mathrm{Node}\ x\ t\ u)$ holds.
- Exercise: prove that for all n and t, $minT\ (mapT\ (n+)\ t) = n + minT\ t$. That is, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.

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

Induction Principle for Other Types

- Recall that **data** *Bool* = *False* | *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: **data** *List a* = [ ] | *a* : *List a*.
- Our representation of lists is biased. The left most element can be fetched immediately.
  - Thus. (:), *head*, and *tail* are constant-time operations, while *init* and *last* takes linear-time.
- In most implementations, the list is represented as a linked-list.

List Concatenation Takes Linear Time

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

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

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

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

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

Full Persistency

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

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

- 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:

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

- 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.

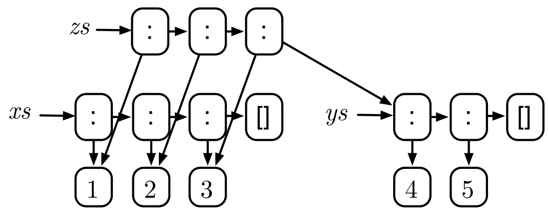

Figure 1: How (++) allocates new (:) cells in the heap.

Another Linear-Time Operation

- Taking all but the last element of a list:

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

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

$$\begin{array}{l} \quad \mathit{init}\ (1 : 2 : 3 : 4 : [\,]) \\ = 1 : \mathit{init}\ (2 : 3 : 4 : [\,]) \\ = 1 : 2 : \mathit{init}\ (3 : 4 : [\,]) \\ = 1 : 2 : 3 : \mathit{init}\ (4 : [\,]) \\ = 1 : 2 : 3 : [\,] \end{array}$$

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 $O(t)$, $\mathit{map}\ f$ takes time $O(n \times t)$ to complete. Similarly with *filter p*.

  - In a lazy setting, $\mathit{map}\ f$ 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 $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map\ square}$.

- The spec. builds an intermediate list. Can we eliminate it?

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

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Alternative Definition for *sumsq*

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

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

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

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

- 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 $as = [a_0, a_1, a_2 \ldots a_n]$ and $x :: \mathrm{Int}$, 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 *iterate* can be defined by

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

Iterating a List

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

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

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

Zipping with a Binary Operator

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

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

Running the Specification

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

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

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

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

Zip-Map Exchange

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

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

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

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:

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

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

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

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

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

Deriving for the Non-Empty Case

For the case for non-empty inputs:

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

Synthesised Program

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

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

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.

2.3.1 Accumulating Parameters

Reversing a List

- The function *reverse* is defined by:

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

- 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?

2.3.2 Fast List Reversal

Introducing an Accumulating Parameter

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

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

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

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

Reversing a List, Base Case

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

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

Reversing a List, Inductive Case

Case $x : \mathit{xs}$:

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

Linear-Time List Reversal

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

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

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

2.3.3 Tail Recursion and Loops

Tracing Reverse

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

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

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

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. *reverse xs*), we introduce a generalisation with an extra parameter, e.g.:

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

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

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

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

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

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

3 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].

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.

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

[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