Functional Programming

Shin-Cheng Mu

FLOLAC 2022

Academia Sinica

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.

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

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.

Recalling "Mathematical Induction"

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

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

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

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

[1] Not a real Haskell definition.

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

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

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

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

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

- Even addition can be defined inductively

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

- Exercise: define $(\times)$?

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.

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

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

Proving and programming are very similar activities.

Without the $n + k$ Pattern

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We have thus proved $P\,0$.

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

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

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

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

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

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

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

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

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

Proof by Induction

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

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

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

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

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

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

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

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

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

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

- 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.

- 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.

- 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

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

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

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

- 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."

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.

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

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

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

[2] Not a real Haskell definition.

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.

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

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.

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

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

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

List Append

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

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

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

Proof by Structural Induction on Lists

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

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

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

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

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

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

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

We have thus proved $P\ []$.

Appending is Associative

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

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

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 $xs +\!\!+ (ys +\!\!+ zs)$.
- By being formal we can work on the *form*, not the *meaning*. Like how we solved the knight/knave problem
- Being formal actually makes the proof easier!
- *Make the symbols do the work.*

Length

- The function *length* defined inductively:

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

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

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

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

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

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

Definition by Induction/Recursion

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

Filter

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

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

Take and Drop

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

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

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

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

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

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

- *dropWhile p xs* drops the prefix from *xs*.

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

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

List Reversal

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

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

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

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

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

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

Totality

- Structure of our definitions so far:

$$f\,[\,] \quad = \ldots$$
$$f\,(x : xs) = \ldots f\,xs \ldots$$

  - 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.

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

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

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

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

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

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

  - 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:

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

Zip

Another example:

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

Non-Structural Induction

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

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

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

  - 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.

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

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

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

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

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

- on which we may inductively define functions:

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

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

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

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

Induction Principle for *Tree*

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

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

Induction Principle for *Tree*

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

- 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

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

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

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

Program Derivation

- 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.

- Recall $(++)$:

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

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

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

List Concatenation Takes Linear Time

- Recall $(++)$:

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

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

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

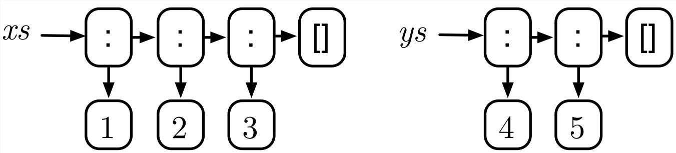
- $(++)$ runs in time proportional to the length of its left argument.

Full Persistency

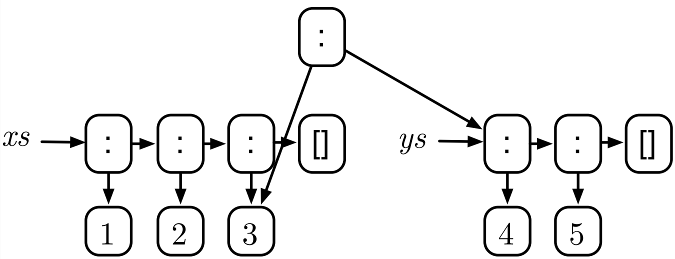
- Compound data structures, like simple values, are just values, and thus must be *fully persistent*.
- That is, in the following code:

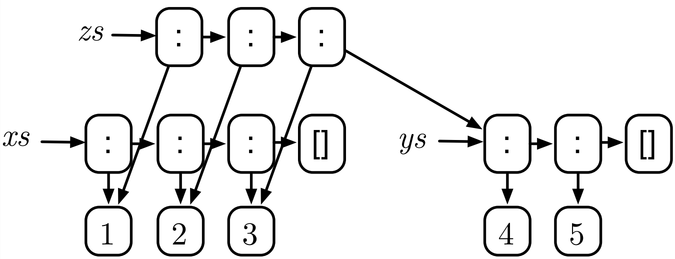
$$\begin{aligned}\textbf{let } & xs = [1,2,3] \\ & ys = [4,5] \\ & zs = xs \mathbin{+\!\!+} ys \\ \textbf{in } & \ldots body \ldots\end{aligned}$$

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



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


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



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

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

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

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

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

Another Linear-Time Operation

- Taking all but the last element of a list:

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

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

Another Linear-Time Operation

- Taking all but the last element of a list:

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

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

- Taking all but the last element of a list:

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

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

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

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

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

$$sumsq\ [\,]$$

Sum of Squares

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

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

Sum of Squares

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

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

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, 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:

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

- 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:

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

- Consider the case when the input is not empty:

  $$sumsq\ (x : xs)$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Alternative Definition for *sumsq*

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

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

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

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

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

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.

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

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

This can be specified by

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

where *iterate* can be defined by

$$\begin{array}{l} iterate :: (a \to a) \to a \to \mathsf{List}\ a \\ iterate\ f\ x = x : map\ f\ (iterate\ f\ x) \quad . \end{array}$$

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

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

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

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

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?

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

Zip-Map Exchange

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

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

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

We can do a formal proof if we want.

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 $(\times x)$.

To conclude, we get:

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

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

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

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

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.

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

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

- 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,

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

For the case for non-empty inputs:

$$steepsum\ (x : xs)$$

For the case for non-empty inputs:

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

For the case for non-empty inputs:

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

Deriving for the Non-Empty Case

For the case for non-empty inputs:

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

For the case for non-empty inputs:

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

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

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

Being Quicker by Doing More?

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

- The function *reverse* is defined by:

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

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

- 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 [ ]:

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

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

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

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

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

Reversing a List, Base Case

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

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

Case $x : xs$:

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

Case $x : xs$:

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

Case $x : xs$:

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

Case $x : xs$:

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

Case $x : xs$:

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

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

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

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

Tracing Reverse

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

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

$\mathit{xs}, \mathit{ys} \leftarrow \mathit{XS}, [\,];$

while $\mathit{xs} \neq [\,]$ **do**

$\quad \mathit{xs}, \mathit{ys} \leftarrow (\mathit{tail}\ \mathit{xs}), (\mathit{head}\ \mathit{xs} : \mathit{ys});$

return ys

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

$$f\, x_1 \; \dots \; x_n = \{\text{base case}\}$$
$$f\, x_1 \; \dots \; x_n = f\, x'_1 \; \dots \; x'_n$$

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

Accumulating Parameters

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

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

- 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.

- 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 $ssp\ xs\ n =$.
- Initialisation: $sumsq\ xs =$.
- Construct ssp:

- Recall the "sum of squares" problem:

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

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

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

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

- 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:

- 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:

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

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 (2010).
- For a more systematic study of algorithms using functional program reasoning, see Bird and Gibbons (2020).

Folds On Lists

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

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

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

This pattern is extracted and called *foldr*:

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

Replacing Constructors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- 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:

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

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

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

- 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:

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

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

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

$$\mathit{and} = \mathit{foldr}\ (\wedge)\ \mathit{true}.$$

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

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

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

Some Functions We Have Seen...

.

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

- $concat =$.

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

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

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

- $concat =$.

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

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

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

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

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

Replacing Constructors

- Understanding *foldr* from its type. Recall

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

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

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

Why Folds?

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

Why Folds?

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

Why Folds?

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

- 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.

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

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

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

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

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

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

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

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

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

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

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

Tracing an Example

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

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

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

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

Sum of Squares, Again

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

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

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

Sum of Squares, without Folds

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

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

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

- 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, $tail :: [a] \rightarrow [a]$ is not a fold!

Longest Prefix

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

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

- 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:

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

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

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

All Suffixes

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

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

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

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

Scan

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

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

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

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

Folds 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.

- Recall the definition:

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

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

$$foldN \quad :: \quad \rightarrow Nat \rightarrow a$$

Fold on Natural Numbers

- Recall the definition:

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

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

$$foldN \quad :: (a \to a) \to a \to Nat \to a$$

- Recall the definition:

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

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

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

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

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

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

Examples of *foldN*

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

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

$$\cdot$$

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

$$\cdot$$

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

Examples of *foldN*

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

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

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

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

.

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

Examples of *foldN*

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

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

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

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

- $even = foldN\ not\ True$.

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

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

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

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

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

- Example: internally labelled binary tree:

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

- Fold for ITree:

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

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

$$\textbf{data}\ \mathsf{ETree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a) \quad .$$

- Fold for ETree:

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

- To compute the size of an ITree:

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

- To sum up labels in an ETree:

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

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

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

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

Maximum Segment Sum

- The *maximum segment sum* is a classical problem, often used to demonstrate the effectness of program derivation.
- Given: a list of numbers — positive, zero, or negative.
- Compute: the maximum possible sum of a consecutive segment of the list.

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

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

- Therefore, *mss* is specified by:

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

We reason:

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

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

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

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

Maximum Prefix Sum

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

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

The fold fusion works because:

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

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

Maximum Prefix Sum, 2nd Fold Fusion

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

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

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

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

We reason:

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

Maximum Segment Sum in Linear Time!

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

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

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

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:

$$
\begin{aligned}
&\mathbf{data}\ \mathrm{RBTree}\ a = \mathrm{E} \mid \\
&\qquad\quad \mathrm{N}\ \mathrm{Color}\ (\mathrm{RBTree}\ a)\ a\ (\mathrm{RBTree}\ a)\quad , \\
&\mathbf{data}\ \mathrm{Color} \qquad = \mathrm{R} \mid \mathrm{B}\quad .
\end{aligned}
$$

- It is a binary search tree.
  - In 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 in a red-black tree is just like that in a binary search tree:

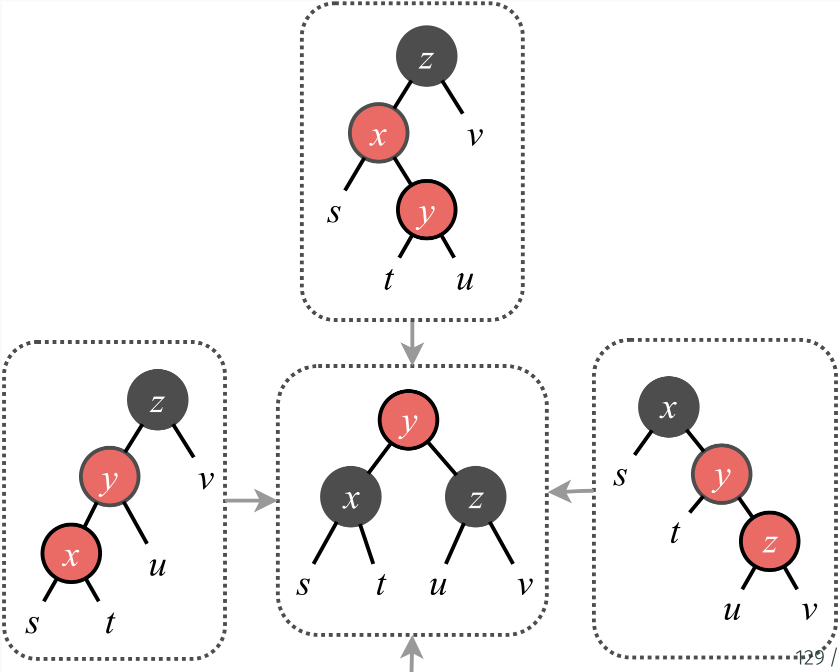
$$
\begin{aligned}
&\mathit{search} :: \mathrm{Int} \to \mathrm{RBTree\ Int} \to \mathrm{Bool} \\
&\mathit{search}\ \mathrm{E} \quad\quad\quad\quad = \mathrm{False} \\
&\mathit{search}\ (\mathrm{N}\ t\ x\ u) \mid k < x = \ldots \\
&\quad\quad\quad\quad\quad\quad\quad \mid k == x = \ldots \\
&\quad\quad\quad\quad\quad\quad\quad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

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.

- The re-balancing strategy is *not* unique.
- The strategy we will consider, shown in the next slide, was presented by Okasaki [**?**].
- 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.


z
x
v
s
y
t
u
z
y
v
x
u
s
t
y
x
z
s
t
u
v
x
s
y
t
z
u
v