Functional Programming

Shin-Cheng Mu

FLOLAC 2022

0 To Begin With...

Prerequisites

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

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

Recommanded Textbooks

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

1 Definition and Proof by Induction

Total Functional Programming

- The next few lectures concerns inductive definitions and proofs of datatypes and programs.
- While Haskell provides allows one to define non-terminating functions, infinite data structures, for now we will only consider its total, finite fragment.
- That is, we temporarily
  - consider only finite data structures,
  - demand that functions terminate for all value in its input type, and
  - provide guidelines to construct such functions.
- Infinite datatypes and non-termination can be modelled with more advanced theory, which we cannot cover in this course.

1.1 Induction on Natural Numbers

Recalling "Mathematical Induction"

- Let P be a predicate on natural numbers.
  - What is a predicate? Such a predicate can be seen as a function of type Nat $\to$ 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 $\to$ Bool for predicates.
- We've all learnt this principle of proof by induction: to prove that P holds for all natural numbers, it is sufficient to show that
  - $P\,0$ holds;
  - $P\,(1+n)$ holds provided that $P\,n$ does.

1.1.1 Proof by Induction

Proof by Induction on Natural Numbers

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

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

- That is, any natural number is either 0, or $\mathbf{1}_+\ n$ where n is a natural number.

[1] Not a real Haskell definition.

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

A Proof Generator

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

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

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 $(\times)$?

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]

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

- 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 $\bot$ 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.

Inductively Defined Functions on Lists

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

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

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

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

[2] Not a real Haskell definition.

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

1.2.1 Append, and Some Of Its Properties

List Append

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

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

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

Proof by Structural Induction on Lists

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

For a Particular List...

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

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

Appending is Associative

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

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

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

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

We have thus proved $P\ []$.

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

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

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

Do We Have To Be So Formal?

- In our style of proof, every step is given a reason. Do we need to be so pedantic?
- Being formal *helps* you to do the proof:
  - In the proof of $exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n$, 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:

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

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

$$length\ (xs +\!\!+ ys) = length\ xs + length\ ys$$

Concatenation

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

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

- 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.

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

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

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

TakeWhile and DropWhile

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

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

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

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

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

List Reversal

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

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

All Prefixes and Suffixes

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

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

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

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

Totality

- Structure of our definitions so far:

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

  - Both the empty and the non-empty cases are covered, guaranteeing there is a matching clause for all inputs.
  - The recursive call is made on a "smaller" argument, guranteeing termination.

- Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

1.2.3 Other Patterns of Induction

Variations with the Base Case

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

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

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

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

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

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

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

Lexicographic Induction

- It also occurs often that we perform *lexicographic induction* on multiple arguments: some arguments decrease in size, while others stay the same.

- E.g. the function $merge$ merges two sorted lists into one sorted list:

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

Zip

Another example:

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

Non-Structural Induction

- In most of the programs we've seen so far, the recursive call are made on direct sub-components of the input (e.g. $f\ (x : xs) = ..f\ xs..$). This is called *structural induction*.
  - It is relatively easy for compilers to recognise structural induction and determine that a program terminates.

- In fact, we can be sure that a program terminates if the arguments get "smaller" under some (well-founded) ordering.

Mergesort

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

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

  - What if we omit the case for $[x]$?

- If all cases are covered, and all recursive calls are applied to smaller arguments, the program defines a total function.

A Non-Terminating Definition

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

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

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

1.3 User Defined Inductive Datatypes

Internally Labelled Binary Trees

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

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

- on which we may inductively define functions:

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

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

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

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

Induction Principle for $Tree$

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

Induction Principle for Other Types

- Recall that $\textbf{data}\ Bool = False \mid True$. Do we have an induction principle for $Bool$?
- To prove a predicate P on $Bool$ holds for all booleans, it is sufficient to show that
  1. $P\ False$ holds, and
  2. $P\ True$ holds.
- Well, of course.
- What about $(A \times B)$? How to prove that a predicate P on $(A \times B)$ is always true?
- One may prove some property P_1 on A and some property P_2 on B, which together imply P.
- That does not say much. But the "induction principle" for products allows us to extract, from a proof of P, the proofs P_1 and P_2.
- *Every inductively defined datatype comes with its induction principle.*
- We will come back to this point later.

2 Program Derivation

2.1 Some Comments on Efficiency

Data Representation

- So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
- Take lists for example. Recall the definition: $\textbf{data}\ List\ a\ =\ [\,] \mid a : List\ a$.
- Our representation of lists is biased. The left most element can be fetched immediately.

- Thus. $(:)$, $head$, and $tail$ are constant-time operations, while $init$ and $last$ takes linear-time.

- In most implementations, the list is represented as a linked-list.

List Concatenation Takes Linear Time

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

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

- 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:

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

- 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:

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

- To allow access to both xs and ys in $body$, the $update$ operation has to duplicate the entire array.

- Thus people have invented some smart data structure to do so, in around $O(\log n)$ time.

- On the other hand, $update$ may simply overwrite xs if we can somehow make sure that *nobody* other than ys uses xs.

- Both are advanced topics, however.

Another Linear-Time Operation

- Taking all but the last element of a list:

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

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

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

Sum, Map, etc

- Functions like sum, $maximum$, etc. needs to traverse through the list once to produce a result. So their running time is definitely $O(n)$, where n is the length of the list.

- If f takes time $O(t)$, $map\ f$ takes time $O(n \times t)$ to complete. Similarly with $filter\ p$.

  - In a lazy setting, $map\ f$ produces its first result in $O(t)$ time. We won't need lazy features for now, however.

2.2 Expand/Reduce Transformation

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $sumsq = sum \cdot map\ square$.

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

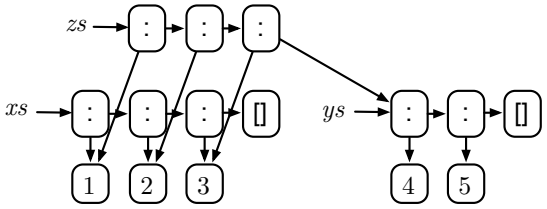

Figure 1: How $(+\!\!+)$ allocates new $(:)$ cells in the heap.

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

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Alternative Definition for $sumsq$

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

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

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

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

- 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 $x :: \mathsf{Int}$, the aim is to compute:

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

This can be specified by

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

where $iterate$ can be defined by

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

Iterating a List

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

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

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

Zipping with a Binary Operator

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

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

Running the Specification

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

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

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

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

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:

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

We can do a formal proof if we want.

Distributivity

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

We used that law to push sum to the right.

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

Computing Polynomial

To conclude, we get:

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

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:

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

- The definition above, if executed directly, is an $O(n^2)$ program. Can we do better?

- Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

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

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

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

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

Deriving for the Non-Empty Case

For the case for non-empty inputs:

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

Synthesised Program

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

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

Being Quicker by Doing More?

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

2.4 Accumulating Parameters

Reversing a List

- The function $reverse$ is defined by:

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

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

2.4.1 Fast List Reversal

Introducing an Accumulating Parameter

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

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

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

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

Reversing a List, Base Case

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

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

Reversing a List, Inductive Case

Case $x : xs$:

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

Linear-Time List Reversal

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

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

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

2.4.2 Tail Recursion and Loops

Tracing Reverse

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

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

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

Tail Recursion

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

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

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

Accumulating Parameters

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

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

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

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

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

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

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

2.5 Conclusions

Conclusions

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

3 Folds On Lists

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

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

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

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

This pattern is extracted and called *foldr*:

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

3.1 The Ubiquitous *foldr*

Replacing Constructors

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

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

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

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

Some Trivial Folds on Lists

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

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

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

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

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

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

- Function and returns the conjunction of a list:

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

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

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

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

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

Some Functions We Have Seen...

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

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

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

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

Replacing Constructors

- Understanding *foldr* from its type. Recall

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

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

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

3.2 The Fold-Fusion Theorem

Why Folds?

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

The Fold-Fusion Theorem

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

Theorem 1 (*foldr*-Fusion). Given $f :: a \to b \to b$, $e :: b$, $h :: b \to c$, and $g :: a \to c \to c$, we have:

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

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

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

Tracing an Example

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

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

Sum of Squares, Again

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

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

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

Sum of Squares, without Folds

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

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

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

More on Folds and Fold-fusion

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

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

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

3.3 More Useful Functions Defined as Folds

Longest Prefix

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

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

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

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

- 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:

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

4 Folds on Other Algebraic Datatypes

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

Fold on Natural Numbers

- Recall the definition:

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

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

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

Examples of $foldN$

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

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

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

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

- $even = foldN\ not\ True$.

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

Fold-Fusion for Natural Numbers

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

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

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

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

Folds on Trees

- Example: internally labelled binary tree:

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

- Fold for ITree:

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

Folds on Trees

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

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

- Fold for ETree:

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

Some Simple Functions on Trees

- To compute the size of an ITree:

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

- To sum up labels in an ETree:

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

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

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

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

5 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.

Specifying Maximum Segment Sum

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

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

- Therefore, mss is specified by:

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

The Derivation!

We reason:

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

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

Maximum Prefix Sum

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

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

The fold fusion works because:

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

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

Maximum Prefix Sum, 2nd Fold Fusion

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

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

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

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

Back to Maximum Segment Sum

We reason:

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

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 $step\ x\ (m,y) = ((0 \uparrow (x+y)) \uparrow m, 0 \uparrow (x+y))$.

6 Red-Black Tree

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

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

Constraints

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

Searching

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

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

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.

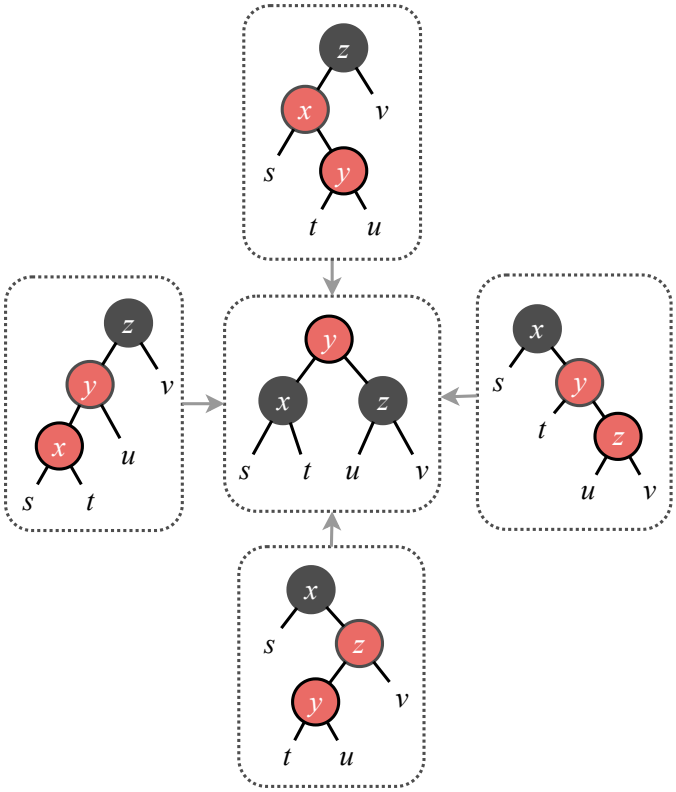

Figure 2: Red-black tree balancing [Oka99].

Tree Balancing

- The re-balancing strategy is *not* unique.
- The strategy we will consider, shown in Figure 2, was presented by Okasaki [Oka99].
- Having only four rules, it is significantly simpler than those you'd find in most textbooks (which needs 8 rules or more)!
- Why?
- More will be discussed in the practicals.

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/`.

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

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

A GHCi Commands

⟨*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 and

`+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