Functional Programming

Kung Chen and Shin-Cheng Mu

FLOLAC, 2016

A Quick Introduction to Haskell

- We will mostly learn some syntactical issues, but there are some important messages too.
- Most of the materials today are adapted from the book *Introduction to Functional Programming using Haskell* by Richard Bird. Prentice Hall 1998.
- References to more Haskell materials are on the course homepage.

Course Materials and Tools

- Course homepage: `http://flolac.iis.sinica.edu.tw/pl2015`
  - Announcements, slides, assignments, additional materials, etc.
- We will be using the Glasgow Haskell Compiler (GHC).
  - A Haskell compiler written in Haskell, with an interpreter that both interprets and runs compiled code.
  - Installation: the Haskell Platform: `http://hackage.haskell.org/platform/`

Function Definition

- A function definition consists of a type declaration, and the definition of its body:

$$\begin{array}{ll} \mathit{square} & :: \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{square}\ x & = x \times x \\ & \\ \mathit{smaller} & :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{smaller}\ x\ y & = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{array}$$

- The GHCi interpreter evaluates expressions in the loaded context:

$$\begin{array}{l} ?\ \mathit{square}\ 3768 \\ 14197824 \\ ?\ \mathit{square}\ (\mathit{smaller}\ 5\ (3+4)) \\ 25 \end{array}$$

1 Values and Evaluation

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3+4)$:

$$\begin{array}{ll} & \mathit{square}\ (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & \mathit{square}\ 7 \\ = & \{\ \text{definition of } \mathit{square}\ \} \\ & 7 \times 7 \\ = & \{\ \text{definition of } \times\ \} \\ & 49 \end{array}$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{array}{ll} & \mathit{square}\ (3+4) \\ = & \{\ \text{definition of } \mathit{square}\ \} \\ & (3+4) \times (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & 7 \times (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & 7 \times 7 \\ = & \{\ \text{definition of } \times\ \} \\ & 49 \end{array}$$

- In this sequence the rule for *square* is applied first. The final result stays the same.
- Do different evaluations orders always yield the same thing?

A Non-terminating Reduction

- Consider the following program:

$$\begin{aligned} &\mathit{three} &&:: \mathit{Int} \to \mathit{Int} \\ &\mathit{three}\ x &&= 3 \\ &\mathit{infinity} &&:: \mathit{Int} \\ &\mathit{infinity} &&= \mathit{infinity} + 1 \end{aligned}$$

- Try evaluating *three infinity*. If we simplify *infinity* first:

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = & \quad \{\ \text{definition of } \mathit{infinity}\ \} \\ & \mathit{three}\ (\mathit{infinity} + 1) \\ = & \ \mathit{three}\ ((\mathit{infinity} + 1) + 1) \ldots \end{aligned}$$

- If we start with simplifying *three*:

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = & \quad \{\ \text{definition of } \mathit{three}\ \} \\ & 3 \end{aligned}$$

Evaluation Order

- There can be many other evaluation orders. As we have seen, some terminates while some do not.
- *normal form*: an expression that cannot be reduced anymore.
  - 49 is in normal form, while 7×7 is not.
  - Some expressions do not have a normal form. E.g. *infinity*.
- A corollary of the *Church–Rosser theorem*: an expression has at most one normal form.
  - If two evaluation sequences both terminate, they reach the same normal form.

Evaluation Order

- Applicative order evaluation: starting with the innermost reducible expression (a redex).
- Normal order evaluation: starting with the outermost redex.
- If an expression has a normal form, normal order evaluation delivers it. Hence the name.
- For now you can imagine that Haskell uses normal order evaluation. A way to implement normal order evaluation is called *lazy evaluation.*

2 Functions

Mathematical Functions

- Mathematically, a function is a mapping between arguments and results.
  - A function $f :: A \to B$ maps each element in A to a unique element in B.
- In contrast, C "functions" are not mathematical functions:
  - `int y = 1; int f (x:int) { return ((y++) * x); }`
- Functions in Haskell have no such *side-effects*: (unconstrained) assignments, IO, etc.
- Why removing these useful features? We will talk about that later in this course.

2.1 Using Functions

Curried Functions

- Consider again the function *smaller*:

$$\begin{aligned} &\mathit{smaller} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ &\mathit{smaller}\ x\ y &&= \mathbf{if}\ x \le y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{aligned}$$

- We sometimes informally call it a function "taking two arguments".
- Usage: *smaller* 3 4.
- Strictly speaking, however, *smaller* is a function returning a function. The type should be bracketed as $\mathit{Int} \to (\mathit{Int} \to \mathit{Int})$.

Precedence and Association

- In a sense, all Haskell functions takes exactly one argument.
  - Such functions are often called *curried.*
- Type: $a \to b \to c = a \to (b \to c)$, not $(a \to b) \to c$.
- Application: $f\ x\ y = (f\ x)\ y$, not $f\ (x\ y)$.
  - *smaller* 3 4 means (*smaller* 3) 4.
  - *square square* 3 means (*square square*) 3, which results in a type error.
- Function application binds tighter than infix operators. E.g. *square* $3 + 4$ means $(\mathit{square}\ 3) + 4$.

Why Currying?

- It exposes more chances to reuse a function, since it can be partially applied.

$$\begin{aligned}
&\mathit{twice} && :: (a \to a) \to (a \to a)\\
&\mathit{twice}\ f\ x && = f\ (f\ x)\\
&\mathit{quad} && :: \mathit{Int} \to \mathit{Int}\\
&\mathit{quad} && = \mathit{twice}\ \mathit{square}
\end{aligned}$$

- Try evaluating $\mathit{quad}\ 3$:

$$\begin{aligned}
&\ \ \mathit{quad}\ 3\\
=&\ \ \mathit{twice}\ \mathit{square}\ 3\\
=&\ \ \mathit{square}\ (\mathit{square}\ 3)\\
=&\ \ \ldots
\end{aligned}$$

- Had we defined:

$$\begin{aligned}
&\mathit{twice} && :: (a \to a, a) \to a\\
&\mathit{twice}\ (f, x) && = f\ (f\ x)
\end{aligned}$$

  we would have to write

$$\begin{aligned}
&\mathit{quad} && :: \mathit{Int} \to \mathit{Int}\\
&\mathit{quad}\ x && = \mathit{twice}\ (\mathit{square}, x)
\end{aligned}$$

- There are situations where you'd prefer not to have curried functions. We will talk about coversion between curried and uncurried functions later.

2.2 Sectioning

Sectioning

- Infix operators are curried too. The operator $(+)$ may have type $\mathit{Int} \to \mathit{Int} \to \mathit{Int}$.
- Infix operator can be partially applied too.

$$\begin{aligned}
&(x\ \oplus)\ y && = x \oplus y\\
&(\oplus\ y)\ x && = x \oplus y
\end{aligned}$$

  - $(1\ +) :: \mathit{Int} \to \mathit{Int}$ increments its argument by one.
  - $(1.0\ /) :: \mathit{Float} \to \mathit{Float}$ is the "reciprocal" function.
  - $(/\ 2.0) :: \mathit{Float} \to \mathit{Float}$ is the "halving" function.

Infix and Prefix

- To use an infix operator in prefix position, surrounded it in parentheses. For example, $(+)\ 3\ 4$ is equivalent to $3 + 4$.
- Surround an ordinary function by back-quotes (not quotes!) to put it in infix position. E.g. 3 '*mod*' 4 is the same as $\mathit{mod}\ 3\ 4$.

Function Composition

- Functions composition:

$$\begin{aligned}
&(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)\\
&(f \cdot g)\ x = f\ (g\ x)
\end{aligned}$$

- E.g. another way to write *quad*:

$$\begin{aligned}
&\mathit{quad} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{quad}\ = \mathit{square} \cdot \mathit{square}
\end{aligned}$$

- Some important properties:
  - $\mathit{id} \cdot f = f = f \cdot \mathit{id}$, where $\mathit{id}\ x = x$.
  - $(f \cdot g) \cdot h = f \cdot (g \cdot h)$.

2.3 Definitions

Guarded Equations

- Recall the definition:

$$\begin{aligned}
&\mathit{smaller} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y && = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{aligned}$$

- We can also write:

$$\begin{aligned}
&\mathit{smaller} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y && \mid x \leq y = x\\
& && \mid x > y = y
\end{aligned}$$

- Equivalently,

$$\begin{aligned}
&\mathit{smaller} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y \mid x \leq y \qquad\quad = x\\
&\qquad\qquad\quad\ \mid \mathbf{otherwise} = y
\end{aligned}$$

- Helpful when there are many choices:

$$\begin{aligned}
&\mathit{signum} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{signum}\ x \mid x > 0 \quad\ = 1\\
&\qquad\qquad\ \mid x == 0 = 0\\
&\qquad\qquad\ \mid x < 0 \quad\ = -1
\end{aligned}$$

  Otherwise we'd have to write

$$\begin{aligned}
\mathit{signum}\ x\ \ = &\ \mathbf{if}\ x > 0\ \mathbf{then}\ 1\\
&\ \mathbf{else\ if}\ x == 0\ \mathbf{then}\ 0\ \mathbf{else}\ -1
\end{aligned}$$

λ Expressions

- Since functions are first-class constructs, we can also construct functions in expressions.
- A λ expression denotes an anonymous function.
  - $\lambda x \rightarrow e$: a function with argument x and body e.
  - $\lambda x \rightarrow \lambda y \rightarrow e$ abbreviates to $\lambda x\ y \rightarrow e$.
  - In ASCII, we write λ as \
- Yet another way to define *smaller*:

$$\begin{array}{lcl} \mathit{smaller} & :: & \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{smaller} & = & \lambda x\ y \rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{array}$$

- Why λs? Sometimes we may want to quickly define a function and use it only once.
- In fact, λ is a more primitive concept.

Local Definitions

There are two ways to define local bindings in Haskell.

- **let**-expression:

$$\begin{array}{lcl} f & :: & \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} \\ f\ x\ y & = & \mathbf{let}\ a = (x+y)/2 \\ & & \phantom{\mathbf{let}}\ b = (x+y)/3 \\ & & \mathbf{in}\ (a+1) \times (b+2) \end{array}$$

- **where**-clause:

$$\begin{array}{l} f \quad :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\ f\ x\ y \mid x \leq 10 = x + a \\ \qquad\ \mid x > 10 = x - a \\ \quad \mathbf{where}\ a = \mathit{square}\ (y+1) \end{array}$$

- **let** can be used in expressions (e.g. $1 + (\mathbf{let}..\mathbf{in}..)$), while **where** qualifies multiple guarded equations.

3 Types

Types

- The universe of values is partitioned into collections, called *types*.
- Some basic types: *Int*, *Float*, *Bool*, *Char*...
- Type "constructors": functions, lists, trees ... to be introduced later.
- Operations on values of a certain type might not make sense for other types. For example: *square square* 3.
- Strong typing: the type of a well-formed expression can be deducted from the constituents of the expression.
  - It helps you to detect errors.
  - More importantly, programmers may consider the types for the values being defined before considering the definition themselves, leading to clear and well-structured programs.

Polymorphic Types

- Suppose $\mathit{square} :: \mathit{Int} \rightarrow \mathit{Int}$ and $\mathit{sqrt} :: \mathit{Int} \rightarrow \mathit{Float}$.
  - $\mathit{square} \cdot \mathit{square} :: \mathit{Int} \rightarrow \mathit{Int}$
  - $\mathit{sqrt} \cdot \mathit{square} :: \mathit{Int} \rightarrow \mathit{Float}$
- The $(\cdot)$ operator has different types in the two expressions:
  - $(\cdot) :: (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow (\mathit{Int} \rightarrow \mathit{Int})$
  - $(\cdot) :: (\mathit{Int} \rightarrow \mathit{Float}) \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow (\mathit{Int} \rightarrow \mathit{Float})$
- To allow $(\cdot)$ to be used in many situations, we introduce type variables and let its type be: $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$.

Summary So Far

- Functions are essential building blocks in a Haskell program. They can be applied, composed, passed as arguments, and returned as results.
- Types sometimes guide you through the design of a program.
- Equational reasoning: let the symbols do the work!

Recommanded Textbooks

- *Introduction to Functional Programming using Haskell* [Bir98]. My recommended book. Covers equational reasoning very well.
- *Programming in Haskell* [Hut07]. A thin but complete textbook.

Online Haskell Tutorials

- *Learn You a Haskell for Great Good!* [Lip11], a nice tutorial with cute drawings!
- *Yet Another Haskell Tutorial* [DI02].
- *A Gentle Introduction to Haskell* by Paul Hudak, John Peterson, and Joseph H. Fasel: a bit old, but still worth a read. [HPF00]
- *Real World Haskell* [OSG98]. Freely available online. It assumes some basic knowledge of Haskell, however.

4 Simple Datatypes

4.1 Booleans

Booleans

The datatype *Bool* can be introduced with a *datatype declaration*:

$$\mathbf{data}\ \mathit{Bool}\ =\ \mathit{False} \mid \mathit{True}$$

(But you need not do so. The type *Bool* is already defined in the Haskell Prelude.)

Datatype Declaration

- In Haskell, a **data** declaration defines a new type.

$$\begin{array}{lll} \mathbf{data}\ \mathit{Type} & = & \mathit{Con}_1\ \mathit{Type}_{11}\ \mathit{Type}_{12} \ldots \\ & \mid & \mathit{Con}_2\ \mathit{Type}_{21}\ \mathit{Type}_{22} \ldots \\ & \mid & \vdots \end{array}$$

- The declaration above introduces a new type, *Type*, with several cases.
- Each case starts with a constructor, and several (zero or more) arguments (also types).
- Informally it means "a value of type *Type* is either a Con_1 with arguments Type_{11}, $\mathit{Type}_{12}\ldots$, or a Con_2 with arguments Type_{21}, $\mathit{Type}_{22}\ldots$"
- Types and constructors begin in capital letters.

Functions on Booleans

Negation:

$$\begin{array}{ll} \mathit{not} & :: \mathit{Bool} \to \mathit{Bool} \\ \mathit{not}\ \mathit{False} & = \mathit{True} \\ \mathit{not}\ \mathit{True} & = \mathit{False} \end{array}$$

- Notice the definition by *pattern matching*. The definition has two cases, because *Bool* is defined by two cases. The shape of the function follows the shape of its argument.

Functions on Booleans

Conjunction and disjunction:

$$\begin{array}{ll} (\&\&), (||) & :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\ \mathit{False}\ \&\&\ x & = \mathit{False} \\ \mathit{True}\ \&\&\ x & = x \\ \mathit{False}\ ||\ x & = x \\ \mathit{True}\ ||\ x & = \mathit{True} \end{array}$$

Functions on Booleans

Equality check:

$$\begin{array}{ll} (==), (\neq) & :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\ x == y & = (x\ \&\&\ y)\ ||\ (\mathit{not}\ x\ \&\&\ \mathit{not}\ y) \\ x \neq y & = \mathit{not}\ (x == y) \end{array}$$

- = is a definition, while == is a function.
- ≠ is written / = in ASCII.

Example

$$\begin{array}{ll} \mathit{leapyear} & :: \mathit{Int} \to \mathit{Bool} \\ \mathit{leapyear}\ y & = (y\ `\mathit{mod}`\ 4 == 0)\ \&\& \\ & \qquad (y\ `\mathit{mod}`\ 100 \neq 0\ ||\ y\ `\mathit{mod}`\ 400 == 0) \end{array}$$

- Note: y ‘*mod*‘ 100 could be written *mod* y 100. The backquotes turns an ordinary function to an infix operator.
- It's just personal preference whether to do so.

4.2 Characters

Characters

- You can think of *Char* as a big **data** definition:

$$\textbf{data}\ \mathit{Char}\ =\ \text{'}a\text{'} \mid \text{'}b\text{'} \mid \ldots$$

with functions:

$$\begin{aligned} &\mathit{ord} :: \mathit{Char} \to \mathit{Int} \\ &\mathit{chr} :: \mathit{Int} \to \mathit{Char} \end{aligned}$$

- Characters are compared by their order:

$$\begin{aligned} &\mathit{isDigit} && :: \mathit{Char} \to \mathit{Bool} \\ &\mathit{isDigit}\ x && = \text{'}0\text{'} \leq x \mathbin{\&\&} x \leq \text{'}9\text{'} \end{aligned}$$

Equality Check

- Of course, you can test equality of characters too:

$$(==) :: \mathit{Char} \to \mathit{Char} \to \mathit{Bool}$$

- $(==)$ is an *overloaded* name — one name shared by many different definitions of equalities, for different types:
  - $(==) :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$
  - $(==) :: (\mathit{Int}, \mathit{Char}) \to (\mathit{Int}, \mathit{Char}) \to \mathit{Bool}$
  - $(==) :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool}$ ...
- Haskell deals with overloading by a general mechanism called *type classes*. It is considered a major feature of Haskell.
- While the type class is an interesting topic, we might not cover much of it since it is orthogonal to the central message of this course.

4.3 Products

Tuples

- The polymorphic type (a, b) is essentially the same as the following declaration:

$$\textbf{data}\ \mathit{Pair}\ a\ b = \mathit{MkPair}\ a\ b$$

- Or, had Haskell allow us to use symbols:

$$\textbf{data}\ (a, b) = (a, b)$$

- Two projections:

$$\begin{aligned} &\mathit{fst} && :: (a, b) \to a \\ &\mathit{fst}\ (a, b) && = a \\ &\mathit{snd} && :: (a, b) \to b \\ &\mathit{snd}\ (a, b) && = b \end{aligned}$$

5 Functions on Lists

Lists in Haskell

- Traditionally an important datatype in functional languages.
- In Haskell, all elements in a list must be of the same type.
  - $[1, 2, 3, 4] :: [\mathit{Int}]$
  - $[\mathit{True}, \mathit{False}, \mathit{True}] :: [\mathit{Bool}]$
  - $[[1, 2], [], [6, 7]] :: [[\mathit{Int}]]$
  - $[] :: [a]$, the empty list (whose element type is not determined).
- *String* is an abbreviation for $[\mathit{Char}]$; "abcd" is an abbreviation of $[\text{'}a\text{'}, \text{'}b\text{'}, \text{'}c\text{'}, \text{'}d\text{'}]$.

List as a Datatype

- $[] :: [a]$ is the empty list whose element type is not determined.
- If a list is non-empty, the leftmost element is called its *head* and the rest its *tail*.
- The constructor $(:) :: a \to [a] \to [a]$ builds a list. E.g. in $x : \mathit{xs}$, x is the head and xs the tail of the new list.
- You can think of a list as being defined by

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

- $[1, 2, 3]$ is an abbreviation of $1 : (2 : (3 : []))$.

Head and Tail

- $\mathit{head} :: [a] \to a$. e.g. $\mathit{head}\ [1, 2, 3] = 1$.
- $\mathit{tail} :: [a] \to [a]$. e.g. $\mathit{tail}\ [1, 2, 3] = [2, 3]$.
- $\mathit{init} :: [a] \to [a]$. e.g. $\mathit{init}\ [1, 2, 3] = [1, 2]$.
- $\mathit{last} :: [a] \to a$. e.g. $\mathit{last}\ [1, 2, 3] = 3$.
- They are all partial functions on non-empty lists. e.g. $\mathit{head}\ [] = \bot$.
- $\mathit{null} :: [a] \to \mathit{Bool}$ checks whether a list is empty.

$$\begin{aligned} &\mathit{null}\ [] && = \mathit{True} \\ &\mathit{null}\ (x : \mathit{xs}) && = \mathit{False} \end{aligned}$$

5.1 List Generation

List Generation

- $[0..25]$ generates the list $[0,1,2..25]$.
- $[0,2..25]$ yields $[0,2,4..24]$.
- $[2..0]$ yields $[]$.
- The same works for all *ordered* types. For example *Char*:
  - $[\texttt{'}a\texttt{'}..\texttt{'}z\texttt{'}]$ yields $[\texttt{'}a\texttt{'},\texttt{'}b\texttt{'},\texttt{'}c\texttt{'}..\texttt{'}z\texttt{'}]$.
- $[1..]$ yields the *infinite* list $[1,2,3..]$.

List Comprehension

- Some functional languages provide a convenient notation for list generation. It can be defined in terms of simpler functions.
- e.g. $[x \times x \mid x \leftarrow [1..5], \mathit{odd}\ x] = [1,9,25]$.
- Syntax: $[e \mid Q_1, Q_2..]$. Each Q_i is either
  - a generator $x \leftarrow \mathit{xs}$, where x is a (local) variable or pattern of type a while xs is an expression yielding a list of type $[a]$, or
  - a guard, a boolean valued expression (e.g. $\mathit{odd}\ x$).
  - e is an expression that can involve new local variables introduced by the generators.

List Comprehension

Examples:

- $[(a,b) \mid a \leftarrow [1..3], b \leftarrow [1..2]] = [(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]$
- $[(a,b) \mid b \leftarrow [1..2], a \leftarrow [1..3]] = [(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]$
- $[(i,j) \mid i \leftarrow [1..4], j \leftarrow [i+1..4]] = [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]$
- $[(i,j) \mid \leftarrow [1..4], \mathit{even}\ i, j \leftarrow [i+1..4], \mathit{odd}\ j] = [(2,3)]$

5.2 Combinators on Lists

Two Modes of Programming

- Functional programmers switch between two modes of programming.
  - Inductive/recursive mode: go into the structure of the input data and recursively process it.
  - Combinatorial mode: compose programs using existing functions (combinators), process the input in stages.
- We will try the latter style today. However, that means we have to familiarise ourselves to a large collection of library functions.
- In the next lecture we will talk about how these library functions can be defined, in the former style.

Length and Indexing

- $(!!) :: [a] \rightarrow \mathit{Int} \rightarrow a$. List indexing starts from zero. e.g. $[1,2,3]!!0 = 1$.
- $\mathit{length} :: [a] \rightarrow \mathit{Int}$. e.g. $\mathit{length}\ [0..9] = 10$.

Append and Concatenation

- Append: $(+\!\!+) :: [a] \rightarrow [a] \rightarrow [a]$. In ASCII one types (++).
  - $[1,2] +\!\!+ [3,4,5] = [1,2,3,4,5]$
  - $[] +\!\!+ [3,4,5] = [3,4,5] = [3,4,5] +\!\!+ []$
- Compare with $(:) :: a \rightarrow [a] \rightarrow [a]$. It is a type error to write $[] : [3,4,5]$. $(+\!\!+)$ is defined in terms of $(:)$.
- $\mathit{concat} :: [[a]] \rightarrow [a]$.
  - e.g. $\mathit{concat}\ [[1,2],[],[3,4],[5]] = [1,2,3,4,5]$.
  - concat is defined in terms of $(+\!\!+)$.

Take and Drop

- $\mathit{take}\ n$ takes the first n elements of the list. For a definition:

$$
\begin{array}{ll}
\mathit{take} & :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
\mathit{take}\ 0\ \mathit{xs} & = [] \\
\mathit{take}\ (n+1)\ [] & = [] \\
\mathit{take}\ (n+1)\ (x:\mathit{xs}) & = x : \mathit{take}\ n\ \mathit{xs}
\end{array}
$$

- For example, $\mathit{take}\ 0\ \mathit{xs} = [\,]$
- $\mathit{take}\ 3$ "abcde" = "abc"
- $\mathit{take}\ 3$ "ab" = "ab"

- Working with infinite list: $\mathit{take}\ 5\ [1..] = [1,2,3,4,5]$. Thanks to normal order (lazy) evaluation.

- Dually, $\mathit{drop}\ n$ drops the first n elements of the list. For a definition:

$$\begin{aligned}
&\mathit{drop} && :: \mathit{Int} \to [a] \to [a] \\
&\mathit{drop}\ 0\ \mathit{xs} && = \mathit{xs} \\
&\mathit{drop}\ (n+1)\ [\,] && = [\,] \\
&\mathit{drop}\ (n+1)\ (x : \mathit{xs}) && = \mathit{drop}\ n\ \mathit{xs}
\end{aligned}$$

  - For example, $\mathit{drop}\ 0\ \mathit{xs} = \mathit{xs}$
  - $\mathit{drop}\ 3$ "abcde" = "cd"
  - $\mathit{drop}\ 3$ "ab" = ""

- $\mathit{take}\ n\ \mathit{xs} \mathbin{+\!\!+} \mathit{drop}\ n\ \mathit{xs} = \mathit{xs}$, as long as $n \not\equiv \bot$.

Map and λ

- $\mathit{map} :: (a \to b) \to [a] \to [b]$. e.g. $\mathit{map}\ (1+)\ [1,2,3,4,5] = [2,3,4,5,6]$.
- $\mathit{map}\ \mathit{square}\ [1,2,3,4] = [1,4,9,16]$.
- Every once in a while you may need a small function which you do not want to give a name to. At such moments you can use the λ notation:
  - $\mathit{map}\ (\lambda x \to x \times x)\ [1,2,3,4] = [1,4,9,16]$
  - In ASCII λ is written \.
- λ is an important primitive notion. We will talk more about it later.

Filter

- $\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a]$.
  - e.g. $\mathit{filter}\ \mathit{even}\ [2,7,4,3] = [2,4]$
  - $\mathit{filter}\ (\lambda n \to n\ \text{`}\mathit{mod}\text{`}\ 3 == 0)\ [3,2,6,7] = [3,6]$
- Application: count the number of occurrences of $'a'$ in a list:
  - $\mathit{length} \cdot \mathit{filter}\ ('a' ==)$
  - Or $\mathit{length} \cdot \mathit{filter}\ (\lambda x \to 'a' == x)$
- **Note** a list comprehension can always be translated into a combination of primitive list generators and *map*, *filter*, and *concat*.

Zip

- $\mathit{zip} :: [a] \to [b] \to [(a,b)]$
- e.g. zip "abcde" $[1,2,3] = [('a',1), ('b',2), ('c',3)]$
- The length of the resulting list is the length of the shorter input list.

Positions

- Exercise: define $\mathit{positions} :: \mathit{Char} \to \mathit{String} \to [\mathit{Int}]$, such that $\mathit{positions}\ x\ \mathit{xs}$ returns the positions of occurrences of x in xs. E.g. $\mathit{positions}\ 'o'$ "roodo" $= [1,2,4]$.
- $\mathit{positions}\ x\ \mathit{xs} = \mathit{map}\ \mathit{snd}\ (\mathit{filter}\ ((x ==) \cdot \mathit{fst})\ (\mathit{zip}\ \mathit{xs}\ [0..]))$
- Or, $\mathit{positions}\ x\ \mathit{xs} = \mathit{map}\ \mathit{snd}\ (\mathit{filter}\ (\lambda(y,i) \to x == y)\ (\mathit{zip}\ \mathit{xs}\ [0..]))$
- What if you want only the position of the *first* occurrence of x?

$$\begin{aligned}
&\mathit{pos} && :: \mathit{Char} \to \mathit{String} \to \mathit{Int} \\
&\mathit{pos}\ x\ \mathit{xs} && = \mathit{head}\ (\mathit{positions}\ x\ \mathit{xs})
\end{aligned}$$

  - Due to lazy evaluation (normal order evaluation), positions of the other occurrences are *not* evaluated!
  - **Note** For now, think of "lazy evaluation" as another (more popular) name for normal order evaluation. Some people distinguish them by saying that normal order evaluation is a mathematical idea while lazy evaluation is a way to implement normal order evaluation.

Morals of the Story

- Lazy evaluation helps to improve modularity.
  - List combinators can be conveniently reused. Only the relevant parts are computed.
- The combinator style encourages "wholemeal programming".
  - Think of aggregate data as a whole, and process them as a whole!

6 λ expressions

- $\lambda x \to e$ denotes a function whose argument is x and whose body is e.
- $(\lambda x \to e_1)\ e_2$ denotes the function $(\lambda x \to e_1)$ applied to e_2. It can be reduced to e_1 with its *free* occurrences of x replaced by e_2.
- E.g.

$$\begin{array}{ll} & (\lambda x \to x \times x)\ (3+4) \\ = & (3+4) \times (3+4) \\ = & 49 \ . \end{array}$$

- λ expression is a primitive and essential notion. Many other constructs can be seen as syntax sugar of λ's.
- For example, our previous definition of *square* can be seen as an abbreviation of

$$\begin{array}{l} \mathit{square} :: \mathit{Int} \to \mathit{Int} \\ \mathit{square} = \lambda x \to x \times x \ . \end{array}$$

  - Indeed, *square* is merely a value that happens to be a function, which is in turn given by a λ expression.
- λ's are like all values — they can appear inside an expression, be passed as parameters, returned as results, etc.
- In fact, it is possible to build a complete programming language consisting of only λ expressions and applications. Look up "λ calculus".
- $\lambda x \to \lambda y \to e$ is abbreviated to $\lambda x\, y \to e$.
- The following definitions are all equivalent:

$$\begin{array}{l} \mathit{smaller}\ x\ y = \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y \\ \mathit{smaller}\ x = \lambda y \to \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y \\ \mathit{smaller} = \lambda x \to \lambda y \to \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y \\ \mathit{smaller} = \lambda x\, y \to \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y \ . \end{array}$$

7 Fold on Lists

Replacing Constructors

- The function *foldr* is among the most important functions on lists.

$$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$$

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

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

- $\mathit{sum} = \mathit{foldr}\ (+)\ 0$.
- One can see that $\mathit{id} = \mathit{foldr}\ (:)\ [\,]$.

Some Trivial Folds on Lists

- Function *maximum* returns the maximum element in a list:
  - $\mathit{maximum} = \mathit{foldr}\ \mathit{max}\ {-\infty}$.
- Function *prod* returns the product of a list:
  - $\mathit{prod} = \mathit{foldr}\ (\times)\ 1$.
- Function *and* returns the conjunction of a list:
  - $\mathit{and} = \mathit{foldr}\ (\&\&)\ \mathit{True}$.
- Lets emphasise again that *id* on lists is a fold:
  - $\mathit{id} = \mathit{foldr}\ (:)\ [\,]$.

Some Slightly Complex Folds

- $\mathit{length} = \mathit{foldr}\ (\lambda x\ n \to 1 + n)\ 0$.
- $\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ \mathit{xs} \to f\ x : \mathit{xs})\ [\,]$.
- $\mathit{xs} +\!\!+ \mathit{ys} = \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs}$. Compare this with *id*!
- $\mathit{filter}\ p = \mathit{foldr}\ (\mathit{fil}\ p)\ [\,]$ where $\mathit{fil}\ p\ x\ \mathit{xs} =$ $\textbf{if}\ p\ x\ \textbf{then}\ (x : \mathit{xs})\ \textbf{else}\ \mathit{xs}$.

The Ubiquitous Fold

- In fact, *any* function that takes a list as its input can be written in terms of *foldr* — although it might not be always practical.
- With fold it comes one of the most important theorem in program calculation — the fold-fusion theorem. We might not have time to cover it, though.

8 Induction on Natural Numbers

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 will be discussed later in this course.

The So-Called "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 $\mathbb{N} \to \mathit{Bool}$.
  - 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 $\mathbb{N} \to \mathit{Bool}$ 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.

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]

$$\mathbf{data}\ \mathbb{N}\ =\ 0 \mid \mathbf{1+}\ \mathbb{N}\quad .$$

[1] Not a real Haskell definition.

- That is, any natural number is either 0, or $\mathbf{1+}\ n$ where n is a natural number.
- The type $\mathbb{N}$ is the *smallest* set such that
  1. 0 is in $\mathbb{N}$;
  2. if n is in $\mathbb{N}$, so is $\mathbf{1+}\ n$.
- Thus to show that P holds for all natural numbers, we only need to consider these two cases.
- 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).

Inductively Defined Functions

- Since the type $\mathbb{N}$ is defined by two cases, it is natural to define functions on $\mathbb{N}$ following the structure:

$$\begin{array}{ll} \mathit{exp} & :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \mathit{exp}\ b\ 0 & = 1 \\ \mathit{exp}\ b\ (\mathbf{1+}\ n) & = b \times \mathit{exp}\ b\ n \quad . \end{array}$$

- Even addition can be defined inductively

$$\begin{array}{ll} (+) & :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ 0 + n & = n \\ (\mathbf{1+}\ m) + n & = \mathbf{1+}\ (m + n) \quad . \end{array}$$

- Exercise: define $(\times)$?

Without the $n + k$ Pattern

- Unfortunately, newer versions of Haskell abandoned the "$n + k$ pattern" used in the previous slide. And there is not a built-in type for $\mathbb{N}$. Instead we have to write:

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

- For the purpose of this course, the pattern $1 + n$ reveals the correspondence between $\mathbb{N}$ 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 $\mathbb{N}$, 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 $m :: \mathbb{N}$, where $P\,m \equiv exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$.

Case $m := 0$:

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

Proof by Induction

Case $m := \mathbf{1}+\ m$:

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

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 $\mathbb{N}$).
- 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, $\mathbb{N}$ 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 $\mathbb{N}$ as given.

9 Induction on Lists

Inductively Defined Lists

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

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

- Every list is built from the base case $[\,]$, with elements added by $(:)$ one by one: $[1,2,3] = 1 : (2 : (3 : [\,]))$.
- The type $[a]$ is the *smallest* set such that
  1. $[\,]$ is in $[a]$;
  2. if xs is in $[a]$ and x is in a, $x : xs$ is in $[a]$ as well.
- But what about infinite lists?
  - For now let's consider finite lists only, as having infinite lists make the *semantics* much more complicated. [3]
  - In fact, all functions we talk about today are total functions. No $\perp$ involved.

Inductively Defined Functions on Lists

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

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

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

[2] Not a real Haskell definition.

[3] What does that mean? We will talk about it later.

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

9.1 Append, and Some of Its Properties

List Append

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

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

- 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.
- The type $[a]$ is the smallest set such that
  1. $[\,]$ is in $[a]$;
  2. if xs is in $[a]$ and x is in a, $x : xs$ is in $[a]$ as well.
- To prove that some property P holds for all finite lists, we show that
  1. $P\ [\,]$ holds;
  2. $P\ (x : xs)$ holds, provided that $P\ xs$ holds.

Appending is Associative

To prove that $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$. Case $xs := [\,]$:

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

Appending is Associative

Case $xs := x : xs$:

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

Length

- The function *length* defined inductively:

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

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

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

Concatenation

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

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

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

9.2 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*.

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

Take and Drop

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

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

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

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

TakeWhile and DropWhile

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

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

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

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

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

List Reversal

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

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

All Prefixes and Suffixes

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

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

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

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

Totality

- Structure of our definitions so far:

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

  - Both the empty and the non-empty cases are covered, guaranteeing there is a matching clause for all inputs.
  - The recursive call is made on a “smaller” argument, guranteeing termination.
- Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

9.3 Other Patterns of Induction

Variations with the Base Case

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

$$
\begin{array}{ll}
\mathit{fib} & :: \mathbb{N} \to \mathbb{N} \\
\mathit{fib}\ 0 & = 0 \\
\mathit{fib}\ 1 & = 1 \\
\mathit{fib}\ (2 + n) & = \mathit{fib}\ (1 + n) + \mathit{fib}\ n \quad .
\end{array}
$$

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

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

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

$$
\mathbf{data}\ [a]^{+} \;=\; \mathit{Singleton}\ a \mid a : [a]^{+} \quad .
$$

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

Lexicographic Induction

- It also occurs often that we perform *lexicographic induction* on multiple arguments: some arguments decrease in size, while others stay the same.
- E.g. the function *merge* merges two sorted lists into one sorted list:

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

Zip

Another example:

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

Non-Structural Induction

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

Mergesort

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

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

  - What if we omit the case for $[x]$?
- If all cases are covered, and all recursive calls are applied to smaller arguments, the program defines a total function.

A Non-Terminating Definition

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

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

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

10 User Defined Inductive Datatypes

Internally Labelled Binary Trees

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

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

- on which we may inductively define functions:

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

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

1. $minT :: Tree\ \mathbb{N} \to \mathbb{N}$, 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 $\mathbb{N}$? [4]

Induction Principle for *Tree*

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

Induction Principle for Other Types

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

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

References

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

[DI02] Hal Daume III. Yet another haskell tutorial. `http://en.wikibooks.org/wiki/Haskell/YAHT`, 2002.

[HPF00] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell, version 98. `http://www.haskell.org/tutorial/`, 2000.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

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

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

A GHCi Commands

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

Commands for debugging

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

:steplocal	single-step within the current top-level binding
:stepmodule	single-step restricted to the current module
:trace	trace after stopping at a breakpoint
:trace <expr>	evaluate <expr> with tracing on (see :history)

Commands for changing settings

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

Options for :set and :unset

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

Commands for displaying information

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