Functional Programming 2

Ralf Hinze

Radboud University

February 2017

Part 0

Overview

0.0 Outline

Contents

Organization

Literature

0.1 Recap: Functional Programming 1

- equational style of programming
- expression-oriented style of programming
- computation by evaluation
- datatypes (static aspects) and functions (dynamic aspects)
- static typing
- universal polymorphism: parametric polymorphism
- ad-hoc polymorphism: type classes
- higher-order functions
- equational reasoning

0.1 Contents: Functional Programming 2

0.2 Organizational matters

- *your goal:* obtain a good grade
- (*my goal:* show you the beauty of FP)
- *how to achieve your goal:*
  - make good use of me i.e. attend the lectures and the tutorials
    tutorials: Q&A and worked example
  - make good use of the teaching assistants:
    Mart and Ward
  - obtain at least a sufficient grade for at least 5 practical sets
    - work and submit in pairs
    - submission: Friday night
    - redo possible within two weeks if insufficient
  - pass the final exam
- *a gentle request and a suggestion:*
  keep the use of electronic devices to a minimum;
  make notes using pencil and paper

0.3 Literature

- Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, No Starch Press, 2011.
- Richard Bird, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.
- Paul Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Graham Hutton, *Programming in Haskell (2nd Edition)*, Cambridge University Press, 2016.
- Bryan O'Sullivan, John Goerzen, Don Stewart, *Real World Haskell*, O'Reilly Media, 2008.
- Simon Thompson, *Haskell: The Craft of Functional Programming (3rd Edition)*, Addison-Wesley Professional, 2011.

Part 1

Lazy evaluation

1.0 Outline

1.1 Evaluation orders

- different evaluation orders are possible
- it matters which strategy we choose
  - applicative-order evaluation
  - normal-order evaluation
  - lazy evaluation

1.1 Different evaluation orders

- recall different evaluation orders from before (the function *square* is defined $square\ x = x * x$):

$$
\begin{array}{ll}
 & square\ (3 + 4) \\
\Longrightarrow & \quad \{\ \text{definition of } +\ \} \\
 & square\ 7 \\
\Longrightarrow & \quad \{\ \text{definition of } square\ \} \\
 & 7 * 7 \\
\Longrightarrow & \quad \{\ \text{definition of } *\ \} \\
 & 49
\end{array}
$$

$$
\begin{array}{ll}
 & square\ (3 + 4) \\
\Longrightarrow & \quad \{\ \text{definition of } square\ \} \\
 & (3 + 4) * (3 + 4) \\
\Longrightarrow & \quad \{\ \text{definition of } +\ \} \\
 & 7 * (3 + 4) \\
\Longrightarrow & \quad \{\ \text{definition of } +\ \} \\
 & 7 * 7 \\
\Longrightarrow & \quad \{\ \text{definition of } *\ \} \\
 & 49
\end{array}
$$

- not two different answers
- but sometimes no answer at all, see next slide!

1.1 Non-terminating evaluations

- consider script

$$\begin{aligned}
&\mathit{three} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{three}\ _ = 3\\[1ex]
&\mathit{infinity} :: \mathit{Integer}\\
&\mathit{infinity} = 1 + \mathit{infinity}
\end{aligned}$$

- two different evaluation orders:

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

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

- not all evaluation orders terminate, which order to choose?

1.1 Applicative-order evaluation

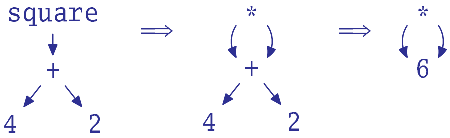
- to reduce the application $f\ e$:
  - first reduce e to normal form
  - then expand definition of f and continue reducing
- simple and obvious
- easy to implement
- may not terminate when other evaluation orders would

1.1 Normal-order evaluation

- to reduce the application $f\ e$:
  - expand definition of f, substituting e
  - reduce result of expansion
- avoids non-termination, if any evaluation order will
- may involve repeating work

1.1 A third way: lazy evaluation

- like normal-order evaluation, but instead of copying arguments we share them



- terms are directed graphs, not trees; *graph reduction*
- best of both worlds: evaluates argument only when needed, so terminating, but never evaluates argument more than once, so efficient

1.1 Lazy evaluation via let

- equivalently, expand application to **let** expression

$$
\begin{array}{ll}
 & \mathit{square}\ (3 + 4) \\
\Longrightarrow & \quad \{\ \text{application}\ \} \\
 & \mathbf{let}\ x = 3 + 4\ \mathbf{in}\ \mathit{square}\ x \\
\Longrightarrow & \quad \{\ \text{definition of}\ \mathit{square}\ \} \\
 & \mathbf{let}\ x = 3 + 4\ \mathbf{in}\ x * x \\
\Longrightarrow & \quad \{\ \text{reduce argument}\ \} \\
 & \mathbf{let}\ x = 7\ \mathbf{in}\ x * x \\
\Longrightarrow & \quad \{\ \text{substitute}\ \} \\
 & 7 * 7 \\
\Longrightarrow & \quad \{\ \text{definition of}\ *\ \} \\
 & 49
\end{array}
$$

- sharing is expressed using **let**-expressions

1.1 Undefined and strictness

- some expressions denote no normal value (e.g. *infinity*, $1 / 0$)
- for simplicity (every syntactically well-formed expression denotes a value), introduce special value *undefined* (sometimes written '$\perp$')
- in evaluating such an expression, evaluator may hang or may give error message
- can apply functions to $\perp$; *strict* functions (*square*) give $\perp$ as a result, *non-strict* functions (*three*) may give some non-$\perp$ value
- the function f is strict iff $f\ \perp = \perp$

1.1 Normal forms

- recall, an expression is in *normal form* (NF) when it cannot be reduced any further
- an expression is in *weak head normal form* (WHNF) if it is a lambda expression, or if it is a constructor applied to zero or more arguments
  - e.g. $\backslash n \to 2 * 3 + n$
  - e.g. $f\,x : map\,f\,xs$
  - e.g. $(1 + 2, 1 - 2)$
- an expression in normal form is in weak head normal form, but converse does not hold

1.1 Demand-driven evaluation

- pattern-matching may trigger reduction of arguments to WHNF

$$head\,[1\,..\,1000000] = head\,(1 : [1 + 1\,..\,1000000]) = 1$$

- patterns matched top to bottom, left to right

$$\begin{aligned} &False \mathbin{\&\&} x = False \\ &True \mathbin{\&\&} x = x \end{aligned}$$

- guards may also trigger reduction

$$\begin{aligned} f\,z \mid\ & fst\,z > 0 && = fst\,z \\ \mid\ & otherwise && = snd\,z \end{aligned}$$

- local definitions not reduced until needed

$$g\,x = (x \neq 0 \mathbin{\&\&} y < 10)\ \mathbf{where}\ y = 1 / x$$

1.1 A pipeline

- the outermost function drives the evaluation

$$
\begin{aligned}
& \mathit{foldl}\ (+)\ 0\ (\mathit{map\ square}\ [1\,..\,1000]) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 0\ (\mathit{map\ square}\ (1 : [2\,..\,1000])) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 0\ (1 : \mathit{map\ square}\ [2\,..\,1000]) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 1\ (\mathit{map\ square}\ [2\,..\,1000]) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 1\ (\mathit{map\ square}\ (2 : [3\,..\,1000])) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 1\ (4 : \mathit{map\ square}\ [3\,..\,1000]) \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 5\ (\mathit{map\ square}\ [3\,..\,1000]) \\
\Longrightarrow\quad & \ldots \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 14\ (\mathit{map\ square}\ [4\,..\,1000]) \\
\Longrightarrow\quad & \ldots \\
\Longrightarrow\quad & 333833500
\end{aligned}
$$

- *note:* the list $[1\,..\,1000]$ never exists all at once

1.1 Demand-driven programming

- lazy evaluation has useful implications for program design
- many computations can be thought of as *pipelines*
- expressed with lazy evaluation, intermediate data structures need not exist all at once
- same effect requires major program surgery in most languages

> **Slogan:** lazy evaluation allows new and better means of modularizing programs

- (but that realization does not help so much in other languages)

1.2 Efficiency

- measure time taken by number of reduction steps
- measure space usage by maximum expression size
- *garbage collection* reclaims discarded space

1.2 Simplifications

- time measure is an approximation, because we ignore time to find redexes
- space measure also an approximation (sharing!)
- e.g. to evaluate and print $[1 \mathinner{..} 1000]$ does not take 1000 units of space
- on the other hand, *space leaks* may surprise

  $$\mathit{numbers} = [1 \mathinner{..} 1000]$$

  evaluating and printing *numbers* leaves a pointer, prevents garbage collection
- space occupied by script may grow with use

1.2 Strictness

- recall summing a list (simplified)

$$
\begin{aligned}
& \mathit{foldl}\ (+)\ 0\ [1\,..\,100] \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 1\ [2\,..\,100] \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ 3\ [3\,..\,100] \\
\Longrightarrow\quad & \ldots
\end{aligned}
$$

- this is a white lie; additions are not forced yet

$$
\begin{aligned}
& \mathit{foldl}\ (+)\ 0\ [1\,..\,100] \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ (0+1)\ [2\,..\,100] \\
\Longrightarrow\quad & \mathit{foldl}\ (+)\ ((0+1)+2)\ [3\,..\,100] \\
\Longrightarrow\quad & \ldots
\end{aligned}
$$

- linear space usage, unnecessarily
- what to do about it?

1.2 Forcing evaluation with *seq*

- judicious mix of lazy and eager evaluation to force additions (safe, because $+$ is strict in both arguments)
- the primitive $seq\ a\ b$ reduces a to WHNF, then returns b

$$
\begin{aligned}
&strict :: (a \to b) \to (a \to b)\\
&strict\ f\ a = seq\ a\ (f\ a)
\end{aligned}
$$

- $strict\ f$ is a strict function i.e. $strict\ f\ \bot = \bot$
- same reductions (on strict functions), but in different order

$$
\begin{aligned}
& && succ\ (succ\ (8 * 5)) \\
&\Longrightarrow && succ\ (8 * 5) + 1 \\
&\Longrightarrow && ((8 * 5) + 1) + 1 \\
&\Longrightarrow && (40 + 1) + 1 \\
&\Longrightarrow && 41 + 1 \\
&\Longrightarrow && 42
\end{aligned}
\qquad
\begin{aligned}
& && strict\ succ\ (strict\ succ\ (8 * 5)) \\
&\Longrightarrow && strict\ succ\ (strict\ succ\ 40) \\
&\Longrightarrow && strict\ succ\ (40 + 1) \\
&\Longrightarrow && strict\ succ\ 41 \\
&\Longrightarrow && 41 + 1 \\
&\Longrightarrow && 42
\end{aligned}
$$

1.2 A strict variant of *foldl*

- now try $\mathit{sfoldl}\ (+)\ 0\ [1\,..\,100]$, where

$$
\begin{aligned}
&\mathit{sfoldl} :: (\mathit{ans} \to a \to \mathit{ans}) \to \mathit{ans} \to ([a] \to \mathit{ans})\\
&\mathit{sfoldl}\ (\triangleleft)\ e = \mathit{loop}\ e\\
&\quad \mathbf{where}\ \mathit{loop}\ a\ [\,] \qquad\ \ = a\\
&\qquad\qquad\ \ \mathit{loop}\ a\ (x : \mathit{xs}) = \mathit{strict}\ \mathit{loop}\ (a \triangleleft x)\ \mathit{xs}
\end{aligned}
$$

1.3 Case study: postage in Fremont

You are a postal worker in Fremont. Given postage denominations, 1, 10, 21, 34, 70, and 100,

dispense a given amount to customer using smallest number of stamps.

- a greedy approach doesn't work:

$$\begin{aligned} \text{greedy:} \quad & 140 = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1 \\ \text{optimal:} \quad & 140 = 70 + 70 \end{aligned}$$

- for simplicity, let us assume that we are only interested in the total number of stamps

1.3 Postage: a recursive implementation

- a naive recursive implementation

$$
\begin{aligned}
&\mathit{stamps} :: [\,\mathit{Integer}\,] \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{stamps}\ \mathit{ds}\ n = f\ n \\
&\quad \mathbf{where}\ f\ 0 = 0 \\
&\qquad\qquad\ f\ i\ \ = \mathit{minimum}\,[\,f\,(i - d) + 1 \mid d \leftarrow \mathit{ds}, d \leqslant i\,]
\end{aligned}
$$

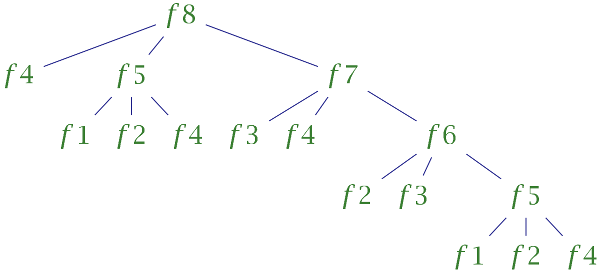
- why naive?

$$
\begin{aligned}
&\rangle\rangle\rangle\rangle\quad \mathit{stamps}\,[\,4, 3, 1\,]\ 6 \\
&2 \\
&\rangle\rangle\rangle\rangle\quad \mathit{stamps}\,[\,100, 70, 34, 21, 10, 1\,]\ 140 \\
&\ldots
\end{aligned}
$$

- the second call is answered by a looong wait

1.3 Naive recursion: analysis

- recursion tree of $f = stamps\,[4, 3, 1]$:



- very slow: exponential running-time
- *problem:* solutions to sub-problems are computed over and over again, e.g. $f\,5$

1.3 Dynamic programming

- *idea of dynamic programming:* replace a function that computes data by a look-up table that contains data
- we trade space for time: we decrease the running-time at the cost of increased space consumption
- candidates for look-up tables
  - lists: linear running time of look-up $\Theta(i)$
  - arrays: constant running time of look-up $\Theta(1)$
- (the term "programming" refers to the method of finding an optimal program, in the sense of a schedule for logistics)

1.3 Interlude: lazy functional arrays

- the library *Data.Array* provides *lazy* functional arrays
- an array is a finite mapping from indices to values

$$\textbf{type}\ \mathit{Array}\ \mathit{ix}\ \mathit{val}$$
$$\textbf{class}\ \mathit{Ix}\ \mathit{ix}$$

$$\begin{aligned} \mathit{array} &:: (\mathit{Ix}\ \mathit{ix}) \Rightarrow (\mathit{ix}, \mathit{ix}) \to [\,(\mathit{ix}, \mathit{val})\,] \to \mathit{Array}\ \mathit{ix}\ \mathit{val} \\ (!) &:: (\mathit{Ix}\ \mathit{ix}) \Rightarrow \mathit{Array}\ \mathit{ix}\ \mathit{val} \to \mathit{ix} \to \mathit{val} \end{aligned}$$

- elements of many types may serve as indices e.g. tuples of indices yield multi-dimensional arrays
- the function *array* (l, u) lazily constructs an array from a list of index/value pairs with indices within bounds (l, u)
- the operator ! is array indexing
- (but no update operation: Haskell is pure)

1.3 Postage: dynamic programming

- we simply (!) replace recursive calls by table look-ups

$$
\begin{aligned}
&\mathit{stamps} :: [\mathit{Integer}] \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{stamps}\ \mathit{ds}\ n \quad = \mathit{memof}\ !\ n \\
&\quad \textbf{where}\ f\ 0 \quad = 0 \\
&\qquad\qquad f\ i \quad = \mathit{minimum}\ [\mathit{memof}\ !\ (i - d) + 1 \mid d \leftarrow \mathit{ds}, d \leqslant i] \\
&\qquad\qquad \mathit{memof} = \mathit{array}\ (0, n)\ [(i, f\ i) \mid i \leftarrow [0 \mathinner{..} n]]
\end{aligned}
$$

- lazy evaluation at work: look-up table is filled in a demand-driven fashion
- linear running time $\Theta(d \cdot n)$ where d is the number of denominations and n is the target denomination

$$
\begin{aligned}
&\rangle\rangle\rangle\rangle \quad \mathit{stamps}\ [4, 3, 1]\ 6 \\
&2 \\
&\rangle\rangle\rangle\rangle \quad \mathit{stamps}\ [100, 70, 34, 21, 10, 1]\ 140 \\
&2
\end{aligned}
$$

1.4 Infinite data structures

- demand-driven evaluation means that programs can manipulate *infinite* data structures
- whole structure is not evaluated at once (fortunately)
- because of laziness, finite result can be obtained from (finite prefix of) infinite data structure
- any recursive datatype has infinite elements, but we will consider only lists

1.4 Infinite lists

- $ones = 1 : ones$
- $[\,n\,..\,] = [\,n, n + 1, n + 2, \ldots\,]$
- $[\,n, n + k\,..\,] = [\,n, n + k, n + 2 * k, \ldots\,]$
- $repeat\; n = n : repeat\; n$
- $iterate\; f\; x = x : iterate\; f\; (f\; x)$
- $fibs = 0 : 1 : zipWith\; (+)\; fibs\; (tail\; fibs)$

1.4 No magic

- can apply functions to infinite data structures

$$\mathit{filter}\ \mathit{even}\ [1\,..]=[2,4,6,8...]$$

- can return finite results

$$\mathit{takeWhile}\ ({<}10)\ [1\,..]=[1,2,3,4,5,6,7,8,9]$$

- note that these do not always behave like infinite sets in maths

$$\mathit{filter}\ ({<}10)\ [1\,..]=[1,2,3,4,5,6,7,8,9$$

- to interrupt, ctrl-C

1.4 What does it mean?

- essential idea is that infinite data structure is *limit* of series of *approximations*
- e.g. infinite list

$$[1, 2, 3, 4, 5, \ldots$$

is limit of series of approximations

$$\begin{array}{l} \perp \\ 1 : \perp \\ 1 : 2 : \perp \\ 1 : 2 : 3 : \perp \\ \ldots \end{array}$$

where $\perp$ represents the “lack of information”

1.4 Primes

- recall bounded sequences of primes

$$\begin{aligned} &\mathit{primes}\ m = [\, n \mid n \leftarrow [\,1\,..\,m\,], \mathit{divisors}\ n == [\,1, n\,]\,] \\ &\mathit{divisors}\ n = [\, d \mid d \leftarrow [\,1\,..\,n\,], n \mathbin{`\mathit{mod}`} d == 0\,] \end{aligned}$$

- infinite sequence of primes

$$\mathit{primes} = [\, n \mid n \leftarrow [\,1\,..\,], \mathit{divisors}\ n == [\,1, n\,]\,]$$

- much more efficient version: *sieve of Eratosthenes*

$$\begin{aligned} &\mathit{primes} = \mathit{sieve}\ [\,2\,..\,]\ \textbf{where} \\ &\quad \mathit{sieve}\ (x : \mathit{xs}) = x : \mathit{sieve}\ [\, y \mid y \leftarrow \mathit{xs}, y \mathbin{`\mathit{mod}`} x \neq 0\,] \end{aligned}$$

1.4 What's the point?

- better abstraction: some real-world entities are infinite
- better modularity: separation of concerns, reuse of components
- fun!

1.4 Summary

- two principal evaluation strategies:
  - applicative order: efficient, but may not terminate
  - normal order: avoids non-termination if possible, but work possibly replicated
- lazy evaluation: best of both worlds
- enables infinite data structures
- better modularity: creation and traversal of structures can be cleanly separated (eg game trees)

Part 2

Imperative programming

2.0 Outline

2.1 Separation of Church and state

- a pure functional language such as Haskell is *referentially transparent*
- expressions do not have side-effects
- remember: the sole purpose of an expression is to denote a value
- but what about state-changing computations (e.g. printing to the console or writing to the file system?)
- how to incorporate these into Haskell?

2.1 Gedankenexperiment

- imagine you are a language designer
- how would you incorporate an outputting computation?

$$putStr :: String \to ()$$

- what's the value and what's the effect of

$$\mathbf{let}\ x = putStr\ \texttt{"ha"}\ \mathbf{in}\ [x, x]$$

- and of this one?

$$[putStr\ \texttt{"ha"}, putStr\ \texttt{"ha"}]$$

- if we noticed different effects, then we would no longer be able to replace equals by equals!
- side-effects and lazy evaluation are not natural bedfellows

2.1 To-do lists and procrastination

- *idea:* $putStr$ `"ha"` has *no* effect at all
- introduce a new type of I/O computations

$$putStr :: String \to IO\ ()$$

- $IO\ a$ is type of computation that may do I/O, then returns an element of type a
- $IO\ a$ can be seen as the type of a *to-do list*
- to-do list *vs* actually doing something
- recording an I/O computation *vs* executing an I/O computation
- $main$ has type $IO\ ()$
- *only* the to-do list that is bound to $main$ is executed

2.1 Interpreting strings

- if evaluator evaluates non-*IO* type, prints value; otherwise, performs computation
- strings as values get displayed as strings:

```
⟫⟫ "Hello,\nWorld"
"Hello,\nWorld"
```

- *putStr* turns a string into an outputting computation:

```
⟫⟫ putStr "Hello,\nWorld"
Hello,
World
```

2.2 The I/O interface

- $IO\ a$ is an abstract datatype of I/O computations
- $return$ turns a value into an I/O computation that has no effect

$$return :: a \to IO\ a$$

- $m \gg n$ first executes m and then n

$$(\gg) :: IO\ a \to IO\ b \to IO\ b$$

- $m \gg= n$ additionally feeds the result of the first computation into the second

$$(\gg=) :: IO\ a \to (a \to IO\ b) \to IO\ b$$

2.2 I/O operations

- console I/O

$$\begin{aligned}
&\mathit{putStr} &&:: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{putStrLn} &&:: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{getLine} &&:: \mathit{IO}\ \mathit{String}
\end{aligned}$$

- console I/O via *Show* and *Read*

$$\begin{aligned}
&\mathit{print} &&:: (\mathit{Show}\ a) \Rightarrow a \to \mathit{IO}\ ()\\
&\mathit{readLn} &&:: (\mathit{Read}\ a) \Rightarrow \mathit{IO}\ a
\end{aligned}$$

- file I/O

$$\mathbf{type}\ \mathit{FilePath} = \mathit{String}$$

$$\begin{aligned}
&\mathit{writeFile} &&:: \mathit{FilePath} \to \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{readFile} &&:: \mathit{FilePath} \to \mathit{IO}\ \mathit{String}
\end{aligned}$$

- many, many more ...

2.2 Example: console I/O

- a simple interactive program

$$
\begin{aligned}
&\mathit{welcome} :: \mathit{IO}\ ()\\
&\mathit{welcome}\\
&\quad = \mathit{putStr}\ \texttt{"Please enter your name.\textbackslash n"} \gg\\
&\qquad \mathit{getLine} \gg\!= \backslash s \rightarrow\\
&\qquad \mathit{putStr}\ (\texttt{"Welcome "} +\!\!+\ s +\!\!+ \texttt{"!\textbackslash n"})
\end{aligned}
$$

- remember: $\backslash s \rightarrow \ldots$ is a lambda expression, an anonymous function

2.2 Example: console I/O

- the same program using **do** notation

```
welcome :: IO ()
welcome
  = do putStr "Please enter your name.\n"
       s ← getLine
       putStr ("Welcome " ++ s ++ "!\n")
```

- syntax: layout-sensitive versus braces and semicolons

```
welcome :: IO ()
welcome
  = do {putStr "Please enter your name.\n";
        s ← getLine; putStr ("Welcome " ++ s ++ "!\n")}
```

2.2 Do notation

- special syntactic sugar for expressions of type $IO\ a$
- inspired by (in fact, a generalization of) list comprehensions

$$
\begin{aligned}
\textbf{do}\ \{m\} &= m \\
\textbf{do}\ \{x \leftarrow m; ms\} &= m \gg= \lambda x \rightarrow \textbf{do}\ \{ms\} \\
\textbf{do}\ \{m; ms\} &= m \gg= \lambda\_ \rightarrow \textbf{do}\ \{ms\} \\
\textbf{do}\ \{\textbf{let}\ ds; ms\} &= \textbf{let}\ ds\ \textbf{in}\ \textbf{do}\ \{ms\}
\end{aligned}
$$

  where x can appear free in ms
- "a generator" (pronounce "x is drawn from m")

$$x \leftarrow m$$

- note that m has type $IO\ a$, whereas x has type a

2.2 Example: file I/O

- processing a file

$$
\begin{aligned}
&\mathit{processFile} :: \mathit{FilePath} \to (\mathit{String} \to \mathit{String}) \to \mathit{FilePath} \to \mathit{IO}\ ()\\
&\mathit{processFile}\ \mathit{inFile}\ f\ \mathit{outFile}\\
&\quad = \mathbf{do}\ s \leftarrow \mathit{readFile}\ \mathit{inFile}\\
&\qquad\qquad \mathbf{let}\ s' = f\ s\\
&\qquad\qquad \mathit{writeFile}\ \mathit{outFile}\ s'
\end{aligned}
$$

2.2 I/O computations as first-class citizens

- we can freely mix I/O computations with, say, lists

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ ()\\
&\mathit{main} = \mathit{sequence}\ [\mathit{print}\ i \mid i \leftarrow [0\,..\,9]]
\end{aligned}
$$

- don't forget the list design pattern

$$
\begin{aligned}
&\mathit{sequence} :: [\mathit{IO}\ ()] \to \mathit{IO}\ ()\\
&\mathit{sequence}\ [\,] \quad\quad\ \ = \mathit{return}\ ()\\
&\mathit{sequence}\ (a : \mathit{as}) = a \gg \mathit{sequence}\ \mathit{as}
\end{aligned}
$$

(the predefined version of *sequence* is more general)

- I/O computations are first-class citizens!
- Haskell is the world's finest imperative language!

2.2 Composition of effectful functions

- pure functions can be chained with function composition $\circ$
- effectful functions can be chained with

$$
\begin{aligned}
&(\odot) :: (b \to IO\ c) \to (a \to IO\ b) \to (a \to IO\ c)\\
&(f \odot g)\ x = g\ x \gg\!= f
\end{aligned}
$$

- turning a pure into an effectful function

$$
\begin{aligned}
&lift :: (a \to b) \to (a \to IO\ b)\\
&lift\ f\ x = return\ (f\ x)
\end{aligned}
$$

- example

$$
\begin{aligned}
&processFile :: FilePath \to (String \to String) \to FilePath \to IO\ ()\\
&processFile\ outFile\ f\\
&\quad = writeFile\ outFile \odot lift\ f \odot readFile
\end{aligned}
$$

```
                                    Jcccc,        ,d$$$b,
                                    J$$$$$$c     ,d$$$$$$,
                                    $$$$$$$$$c,,c$$$$P$$$$,
                                   J$$$$$$$$$$$$$$$$$$3$$$$,
                                   $$$$$$$$$$$$$$$$$$$F$$$$$$c=
                                  J$$$$$$$$$$$$$?$$$$$F  """
                               zcd$$$$$$$$$$$$$Fb3$$$"
                            ,c=cc$$$$$$$$$$??$$$c
                         ,z$"   ‘$$$$$$$$$??$.?$$b
                        d$$",d$$c$$$$$$P"  ?$Fd$$$$
                      ,$$$,dP" "$$$$$$$$$$c ‘$$$$$$r
                     z$$$$P"=$c $$$$$$P""?$, $$$$$$$
            .,,ccc,,4$$$$$’  ‘$ $$$$"-cc  $$,$$$$$$$
        ,c$$$$$$$$$$$$$$$$ ,,d$,$$P   $$FJ$$$$$$$E3F
       ,$$$$$$$$$$$$$$$$$$L"$$’$$$ ,zd$$ $$$$$$$$???-
       $$$$$$$$$$$$$$$$P""??cc$$$$ $$$F,$$$$$$$$$$$$$$$$ccc,.
       $$$$$$$$$$$$$$P d$$$$bc3$$$bc,cd$$$$$$$$$$$$$$$$$$$$$$$bc.
        $$$$$$$$$$$$$F:$$$$$$$$$P?$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$b.
        ‘$$$$$$$$$$$$b.?$$$$$$$$$c"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$c
          ?$$$$$$$$$$$ c,?$$$$$$$",$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$h
           ‘$$$$$$$$$"d$$$cc,,,,c $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$F
            ‘?$$$$$$$z$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$F
              ?$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$?$$$$$$$$$P"
               $$$$$$$$$?$$$$$$$’d$$$$$$$$$$$$$$$$$F’,ccccccc,"’’
               ‘$$$$$$$F "$$P$$P,$$$$$$$$$$$$$$$P",$$$$$$$$$$$$$bc.
                $$$$$P$$$$$$ $$b$$$$$$$$$$$$P?",$$$$$$$$$$$$$$$$$$$$c
    .,,,,.      ‘$$$$$,"???",$$$$$$$$$$$$CCh$’J$$$$$$$$$$$$$$$$$$$$$$$b.
,c$$$$$$$$$$c, $ $$$$$$$$$$$$$$$$$$$$$$$$$$$’d$$$$$$$$$$$$$$$$$$$$$$$$$$$c
$$$$$$$$$$$$$$ $ ?$$$$$$$$$$$$$$$$$$$$$$$$$$’c$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$ $L‘$$$$$$$$$$$$$$$$$$$$$$$$$$,$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$ $$,‘$$$$$$$$Fl$$$$$$$$$$$$$L$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$ $$$c,?$$$P"l$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$,‘$$$$$cccd$$$$$?$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$ ?$$$$$$$$$$$",$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$)$$$$$$$$$,"$$$$$$P",c$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$FJ$$$$$$$$$$b,,..,,d$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

2.3 Case study: Haskinator

Think about a real or fictional character . . . I will try to guess who it is.

$$iGuessTheCelebrity :: IO\ ()$$

Think of number between l and r . . . I will try to guess the number.

$$iGuessTheNumber :: Integer \to Integer \to IO\ ()$$

2.3 A game tree

Goal: separate the game logic from the underlying data.

$$\textbf{data}\ \mathit{Tree}\ a\ b = \mathit{Tip}\ a \mid \mathit{Node}\ b\ (\mathit{Tree}\ a\ b)\ (\mathit{Tree}\ a\ b)$$
$$\quad\textbf{deriving}\ (\mathit{Show})$$

The type is parametric in the type of labels of external nodes (i.e. tips) and in the type of labels of internal nodes.

$$\begin{aligned}
&\mathit{bimap} :: (a1 \to a2) \to (b1 \to b2) \to (\mathit{Tree}\ a1\ b1 \to \mathit{Tree}\ a2\ b2)\\
&\mathit{bimap}\ f\ _\ (\mathit{Tip}\ a) \quad\quad\; = \mathit{Tip}\ (f\ a)\\
&\mathit{bimap}\ f\ g\ (\mathit{Node}\ b\ l\ r) = \mathit{Node}\ (g\ b)\ (\mathit{bimap}\ f\ g\ l)\ (\mathit{bimap}\ f\ g\ r)
\end{aligned}$$

The function *bimap* is a binary variant of *fmap*.

2.3 The game logic

$$
\begin{aligned}
&\mathit{guess} :: \mathit{Tree}\ \mathit{String}\ \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{guess}\ (\mathit{Tip}\ s)\\
&\quad = \mathit{putStrLn}\ s\\
&\mathit{guess}\ (\mathit{Node}\ q\ l\ r)\\
&\quad = \mathbf{do}\ b \leftarrow \mathit{yesOrNo}\ q\\
&\qquad\qquad \mathbf{if}\ b\ \mathbf{then}\\
&\qquad\qquad\quad \mathit{guess}\ l\\
&\qquad\qquad \mathbf{else}\\
&\qquad\qquad\quad \mathit{guess}\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{yesOrNo} :: \mathit{String} \to \mathit{IO}\ \mathit{Bool}\\
&\mathit{yesOrNo}\ \mathit{question}\\
&\quad = \mathbf{do}\ \mathit{putStrLn}\ \mathit{question}\\
&\qquad\qquad \mathit{answer} \leftarrow \mathit{getLine}\\
&\qquad\qquad \mathit{return}\ (\mathit{map}\ \mathit{toLower}\ \mathit{answer}\ \text{`}\mathit{isPrefixOf}\text{`}\ \texttt{"yes"})\\
&\qquad\qquad\quad \text{— empty answer means “yes”}
\end{aligned}
$$

2.3 I guess the celebrity

```
iGuessTheCelebrity
  = do putStrLn ("Think of a celebrity.")
       guess (bimap (\s → s ++ "!") (\q → q ++ "?") celebrity)

celebrity :: Tree String String
celebrity
  = Node "Female"
      (Node "Actress"
        (Tip "Angelina Jolie")
        (Tip "Adele"))
      (Node "Actor"
        (Tip "Brad Pitt")
        (Tip "Steve Hackett"))
```

2.3 I guess the number

```
iGuessTheNumber l r
  = do putStrLn ("Think of number between " ++
                 show l ++ " and " ++ show r ++ ".")
       guess (bimap (\n → show n ++ "!")
                    (\m → "<= " ++ show m ++ "?")
                    (nest l r))

nest :: Integer → Integer → Tree Integer Integer
nest l r
  | l == r     = Tip l
  | otherwise = Node m (nest l m) (nest (m + 1) r)
  where m = (l + r) `div` 2
```

2.4 Other I/O goodies

- the *IO* type offers a lot more
  - exception handling
  - threads
  - updatable variables (aka references or pointers)
  - updatable arrays
  - ...
- Haskell's sin bin

2.4 References

- remember referential transparency
- updatable variables live in the *IO* world
- excerpt of the interface

$$
\begin{aligned}
&\mathbf{type}\ \mathit{IORef}\ a \\
&\mathit{newIORef}\ \ :: a \to \mathit{IO}\ (\mathit{IORef}\ a) \\
&\mathit{readIORef}\ :: \mathit{IORef}\ a \to \mathit{IO}\ a \\
&\mathit{writeIORef} :: \mathit{IORef}\ a \to a \to \mathit{IO}\ ()
\end{aligned}
$$

- *newIORef* creates a new *IORef* and initializes
- *readIORef* reads the value of an IORef
- *writeIORef* writes a new value into an IORef

2.4 References: examples

- copying "variables"

$$
\begin{aligned}
&copy :: IORef\ a \to IORef\ a \to IO\ () \qquad \text{— } x := y \\
&copy\ x\ y = \mathbf{do}\ b \leftarrow readIORef\ y \\
&\qquad\qquad\qquad writeIORef\ x\ b
\end{aligned}
$$

- swapping "variables"

$$
\begin{aligned}
&swap :: IORef\ a \to IORef\ a \to IO\ () \\
&swap\ x\ y = \mathbf{do}\ a \leftarrow readIORef\ x \qquad \text{— } x, y := y, x \\
&\qquad\qquad\qquad b \leftarrow readIORef\ y \\
&\qquad\qquad\qquad writeIORef\ x\ b \\
&\qquad\qquad\qquad writeIORef\ y\ a
\end{aligned}
$$

2.4 Singly-linked lists

- *IORef*s are first class citizens; they can mix and mingle
- singly-linked lists

$$\textbf{type}\ \mathit{ListRef\ elem} = \mathit{IORef}\ (\mathit{List\ elem})$$
$$\textbf{data}\ \mathit{List\ elem} = \mathit{Nil} \mid \mathit{Cons\ elem}\ (\mathit{ListRef\ elem})$$

- the two types are mutually recursive
- likewise, operations on singly-linked lists often come in pairs defined by mutual recursion
- (no *null* pointer; Tony Hoare's billion-dollar mistake)

2.4 Linked lists: length

- the length of a singly-linked list (definition style)

$$
\begin{aligned}
&\mathit{length} :: \mathit{ListRef}\ \mathit{elem} \to \mathit{IO}\ \mathit{Integer}\\
&\mathit{length}\ \mathit{ref} = \mathbf{do}\ \{\mathit{list} \leftarrow \mathit{readIORef}\ \mathit{ref}; \mathit{length}'\ \mathit{list}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{length}' :: \mathit{List}\ \mathit{elem} \to \mathit{IO}\ \mathit{Integer}\\
&\mathit{length}'\ \mathit{Nil} &&= \mathit{return}\ 0\\
&\mathit{length}'\ (\mathit{Cons}\ x\ \mathit{next}) &&= \mathbf{do}\ \{n \leftarrow \mathit{length}\ \mathit{next}; \mathit{return}\ (n+1)\}
\end{aligned}
$$

2.4 Linked lists: length

- alternative definition (expression style)

$$
\begin{array}{l}
\mathit{length} :: \mathit{ListRef}\ \mathit{elem} \to \mathit{IO}\ \mathit{Integer} \\
\mathit{length}\ \mathit{ref} = \\
\quad \mathbf{do}\ \mathit{list} \leftarrow \mathit{readIORef}\ \mathit{ref} \\
\qquad \mathbf{case}\ \mathit{list}\ \mathbf{of} \\
\qquad\quad \mathit{Nil} \qquad\qquad\quad \to \mathit{return}\ 0 \\
\qquad\quad \mathit{Cons}\ x\ \mathit{next} \to \mathbf{do}\ \{ n \leftarrow \mathit{length}\ \mathit{next}; \mathit{return}\ (n+1) \}
\end{array}
$$

- note: layout-sensitive syntax and syntax using braces and semicolons can be freely mixed

2.4 Linked lists: concatenation

- rear of a list (last reference cell)

$$
\begin{array}{l}
\mathit{rear} :: \mathit{ListRef}\ \mathit{elem} \rightarrow \mathit{IO}\ (\mathit{ListRef}\ \mathit{elem}) \\
\mathit{rear}\ \mathit{ref} = \\
\quad \mathbf{do}\ \mathit{list} \leftarrow \mathit{readIORef}\ \mathit{ref} \\
\qquad \mathbf{case}\ \mathit{list}\ \mathbf{of} \\
\qquad\quad \mathit{Nil} \qquad\qquad\quad \rightarrow \mathit{return}\ \mathit{ref} \\
\qquad\quad \mathit{Cons}\ a\ \mathit{next} \rightarrow \mathit{rear}\ \mathit{next}
\end{array}
$$

- concatenating two singly-linked lists

$$
\begin{array}{l}
\mathit{append} :: \mathit{ListRef}\ \mathit{elem} \rightarrow \mathit{ListRef}\ \mathit{elem} \rightarrow \mathit{IO}\ () \\
\mathit{append}\ \mathit{xref}\ \mathit{yref} = \\
\quad \mathbf{do}\ \mathit{ref} \leftarrow \mathit{rear}\ \mathit{xref} \\
\qquad \mathit{copy}\ \mathit{ref}\ \mathit{yref}
\end{array}
$$

2.4 Linked lists: a puzzle

- what's printed?

$$
\begin{array}{l}
\mathit{puzzle} = \\
\quad \mathbf{do}\ x \leftarrow \mathit{fromList}\,[\,0\,..\,14\,] \\
\qquad\ \ y \leftarrow \mathit{fromList}\,[\,15\,..\,19\,] \\
\qquad\ \ \mathit{append}\ x\ y \\
\qquad\ \ \mathit{n1} \leftarrow \mathit{length}\ x \\
\qquad\ \ \mathit{print}\ \mathit{n1} \\
\qquad\ \ \mathit{append}\ x\ y \\
\qquad\ \ \mathit{n2} \leftarrow \mathit{length}\ x \\
\qquad\ \ \mathit{print}\ \mathit{n2}
\end{array}
$$

2.5 Summary

- "lazy makes you pure"
- I/O computations are first-class citizens!
- Haskell is the world's finest imperative language
- in general, try to minimize the I/O part of your program

Part 3

Applicative functors and monads

3.0 Outline

3.1 Manifest interfaces

- functions have manifest interfaces
- (**manifest**. Clearly revealed to the eye, mind, or judgement; open to view or comprehension; obvious.)
- this is both a blessing and a curse
- blessing: definitions can be read and understood in isolation
- curse: details cannot be brushed under the carpet

3.1 An evaluator

- recall the datatype of expressions

```
infixl 6 :+:
infixl 7 :*:

data Expr
   = Lit Integer          — a literal
   | Expr :+: Expr        — addition
   | Expr :*: Expr        — multiplication
   | Div Expr Expr        — integer division
```

- small extension: integer division

```
good, bad :: Expr
good = Div (Lit 7) (Div (Lit 4) (Lit 2))
bad  = Div (Lit 7) (Div (Lit 2) (Lit 4))
```

3.1 The vanilla evaluator

- recall the evaluation function

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}\\
&\mathit{eval}\ (\mathit{Lit}\ i) &&= i\\
&\mathit{eval}\ (e1 :\!+\!: e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2\\
&\mathit{eval}\ (e1 :\!*\!: e2) &&= \mathit{eval}\ e1 * \mathit{eval}\ e2\\
&\mathit{eval}\ (\mathit{Div}\ e1\ e2) &&= \mathit{div}\ (\mathit{eval}\ e1)\ (\mathit{eval}\ e2)
\end{aligned}
$$

- example evaluations:

$$
\begin{aligned}
&\rangle\rangle\rangle\quad \mathit{eval}\ \mathit{good}\\
&3\\
&\rangle\rangle\rangle\quad \mathit{eval}\ \mathit{bad}\\
&\mathit{Exception} : \mathit{divide\ by\ zero}
\end{aligned}
$$

3.1 Exception handling

- evaluation may fail, because of division by zero
- let's handle the exceptional behaviour:

```
evalE :: Expr → Maybe Integer
evalE (Lit i)        = Just i
evalE (Div e1 e2) =
  case evalE e1 of
    Nothing → Nothing
    Just v1  →
      case evalE e2 of
        Nothing                  → Nothing
        Just v2 | v2 == 0        → Nothing   — division by zero
                | otherwise → Just (div v1 v2)
```

- (other cases omitted for reasons of space)

3.1 Counting evaluation steps

- we could instrument the evaluator to count evaluation steps:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Counter}\ a = (a, \mathit{Int})\\
&\mathit{evalC} :: \mathit{Expr} \to \mathit{Counter}\ \mathit{Integer}\\
&\mathit{evalC}\ (\mathit{Lit}\ i) = (i, 1)\\
&\mathit{evalC}\ (\mathit{Div}\ e1\ e2) = \mathbf{let}\ (v1, n1) = \mathit{evalC}\ e1\\
&\qquad\qquad\qquad\qquad\quad\ \ (v2, n2) = \mathit{evalC}\ e2\\
&\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \ (\mathit{div}\ v1\ v2, 1 + n1 + n2)
\end{aligned}$$

- (other cases omitted for reasons of space)

3.1 Ugly!

- none of the two extensions is difficult
- but each is rather awkward, and obscures the previously clear structure
- how can we simplify the presentation?
- what do they have in common?

3.1 Ugly!

- none of the two extensions is difficult
- but each is rather awkward, and obscures the previously clear structure
- how can we simplify the presentation?
- what do they have in common?
- both evaluators have type $Expr \to F\ Integer$ where F specifies the *computational effect*

3.1 The applicative type class

- a type class for computations

$$\textbf{infixl}\ 4\ \ \langle\ast\rangle$$

$$\begin{array}{l} \textbf{class}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\ \quad \mathit{pure} :: a \to f\ a \\ \quad (\langle\ast\rangle) :: f\ (a \to b) \to f\ a \to f\ b \end{array}$$

- *pure* turns a value into a pure computation that has no effect
- $\langle\ast\rangle$ is function application where both function and argument are obtained as results of computations

3.1 The applicative type class—continued

- there are also one-sided versions of $\langle * \rangle$—useful if a computation is *only* executed for its effect

$$\textbf{infixl}\ 4\ \langle *, *\rangle$$

$$(\langle *) :: f\, a \to f\, b \to f\, a$$
$$(*\rangle) :: f\, a \to f\, b \to f\, b$$

$$a \langle * \ b = pure\ (\backslash x\ y \to x)\ \langle * \rangle\ a\ \langle * \rangle\ b$$
$$a *\rangle\ b = pure\ (\backslash x\ y \to y)\ \langle * \rangle\ a\ \langle * \rangle\ b$$

- in addition to these generic operations, each instance of *Applicative* also provides operations for the *effect-specific behaviour*

3.1 Evaluator, applicative style

- recall the vanilla evaluator

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}\\
&\mathit{eval}\ (\mathit{Lit}\ i) &&= i\\
&\mathit{eval}\ (e1 :+: e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2\\
&\mathit{eval}\ (e1 :*: e2) &&= \mathit{eval}\ e1 * \mathit{eval}\ e2\\
&\mathit{eval}\ (\mathit{Div}\ e1\ e2) &&= \mathit{div}\ (\mathit{eval}\ e1)\ (\mathit{eval}\ e2)
\end{aligned}
$$

- same evaluator in an applicative style

$$
\begin{aligned}
&\mathit{evalA} :: (\mathit{Applicative}\ f) \Rightarrow \mathit{Expr} \to f\ \mathit{Integer}\\
&\mathit{evalA}\ (\mathit{Lit}\ i) &&= \mathit{pure}\ i\\
&\mathit{evalA}\ (e1 :+: e2) &&= \mathit{pure}\ (+)\ \langle * \rangle\ \mathit{evalA}\ e1\ \langle * \rangle\ \mathit{evalA}\ e2\\
&\mathit{evalA}\ (e1 :*: e2) &&= \mathit{pure}\ (*)\ \langle * \rangle\ \mathit{evalA}\ e1\ \langle * \rangle\ \mathit{evalA}\ e2\\
&\mathit{evalA}\ (\mathit{Div}\ e1\ e2) &&= \mathit{pure}\ \mathit{div}\ \langle * \rangle\ \mathit{evalA}\ e1\ \langle * \rangle\ \mathit{evalA}\ e2
\end{aligned}
$$

- two changes compared to the vanilla evaluator
  - prefix: $(+)\ a\ b$ instead of $a + b$
  - application made explicit: $\mathit{pure}\ f\ \langle * \rangle\ a\ \langle * \rangle\ b$ instead of $f\ a\ b$
- still pure, but much easier to extend

3.1 Recovering the vanilla evaluator

- meet the identity functor

$$\textbf{newtype}\ Id\ a = I\ \{\ fromI :: a\}$$

$$\begin{aligned}
&\textbf{instance}\ Functor\ Id\ \textbf{where}\\
&\quad fmap\ f\ (I\ x) = I\ (f\ x)\\
&\textbf{instance}\ Applicative\ Id\ \textbf{where}\\
&\quad pure\ a \qquad\quad = I\ a\\
&\quad I\ f \langle * \rangle\ I\ x \quad = I\ (f\ x)
\end{aligned}$$

- *pure* is the identity and $\langle * \rangle$ is function application
- example evaluation:

$$\rangle\rangle\rangle\rangle \quad fromI\ (evalA\ good)$$
$$3$$

3.1 The counter instance

- counters instantiate the functor and applicative class:

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{Counter}\ a = C\ \{\ \mathit{fromC} :: (a, \mathit{Int})\ \}\\
&\quad \mathbf{deriving}\ (\mathit{Show})\\
&\mathbf{instance}\ \mathit{Functor}\ \mathit{Counter}\ \mathbf{where}\\
&\quad \mathit{fmap}\ f\ (C\ (a, n)) = C\ (f\ a, n)\\
&\mathbf{instance}\ \mathit{Applicative}\ \mathit{Counter}\ \mathbf{where}\\
&\quad \mathit{pure}\ a = C\ (a, 0)\\
&\quad C\ (f, m)\ \langle * \rangle\ C\ (x, n) = C\ (f\ x, m + n)
\end{aligned}
$$

- the effect-specific behaviour is to increment the count:

$$
\begin{aligned}
&\mathit{tick} :: \mathit{Counter}\ ()\\
&\mathit{tick} = C\ ((), 1)
\end{aligned}
$$

3.1 Counting evaluator, applicative style

- to integrate *tick* we use $\langle *$

$$
\begin{aligned}
&\mathit{evalC} :: \mathit{Expr} \to \mathit{Counter\ Integer} \\
&\mathit{evalC}\ (\mathit{Lit}\ i) &&= (\mathit{pure}\ i)\ \langle *\ \mathit{tick} \\
&\mathit{evalC}\ (e1 :+: e2) &&= (\mathit{pure}\ (+)\ \langle * \rangle\ \mathit{evalC}\ e1\ \langle * \rangle\ \mathit{evalC}\ e2)\ \langle *\ \mathit{tick} \\
&\mathit{evalC}\ (e1 :*: e2) &&= (\mathit{pure}\ (*)\ \langle * \rangle\ \mathit{evalC}\ e1\ \langle * \rangle\ \mathit{evalC}\ e2)\ \langle *\ \mathit{tick} \\
&\mathit{evalC}\ (\mathit{Div}\ e1\ e2) &&= (\mathit{pure}\ \mathit{div}\ \langle * \rangle\ \mathit{evalC}\ e1\ \langle * \rangle\ \mathit{evalC}\ e2)\ \langle *\ \mathit{tick}
\end{aligned}
$$

- *tick* is only called for its effect, not its value
- example evaluation:

$$
\begin{aligned}
&\rangle\rangle\rangle\ \ \mathit{fromC}\ (\mathit{evalC}\ \mathit{good}) \\
&(3, 5)
\end{aligned}
$$

3.2 The exception instance

- exceptions instantiate the functor and applicative class

$$
\begin{array}{l}
\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \\
\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{fmap}\ f\ (\mathit{Nothing}) & = & \mathit{Nothing} \\
\mathit{fmap}\ f\ (\mathit{Just}\ a) & = & \mathit{Just}\ (f\ a)
\end{array} \\
\textbf{instance}\ \mathit{Applicative}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{pure}\ a = \mathit{Just}\ a \\
\quad \begin{array}{lclcl}
\mathit{Nothing} & \langle * \rangle & \mathit{Nothing} & = & \mathit{Nothing} \\
\mathit{Nothing} & \langle * \rangle & \mathit{Just}\ x & = & \mathit{Nothing} \\
\mathit{Just}\ f & \langle * \rangle & \mathit{Nothing} & = & \mathit{Nothing} \\
\mathit{Just}\ f & \langle * \rangle & \mathit{Just}\ x & = & \mathit{Just}\ (f\ x)
\end{array}
\end{array}
$$

3.2 Exception handling evaluator

- but how to modify the interpreter?

$$evalA\ (Div\ e1\ e2) = pure\ div\ \langle * \rangle\ evalA\ e1\ \langle * \rangle\ evalA\ e2$$

- *div* can check whether its second argument is zero, but it cannot raise an exception as it is a pure function
- we need an additional combinator that applies an impure function ($a \to Maybe\ b$) to an impure argument ($Maybe\ a$)

3.2 The monad type class

- the required combinator is a method of a sub-class of *Applicative*

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Applicative}\ m) \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg)\ \ :: m\ a \to m\ b \to m\ b \\
\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b \\
\quad m \gg n = m \gg\!= \backslash\_ \to n
\end{array}
$$

- (have you seen the combinators before?)
- (*monad* is a term from category theory, purloined from philosophy)

3.2 Levels of impurity ⋆

- notice the difference between *fmap*, $\langle * \rangle$, and $=\!\!\ll$ (the combinator $\gg\!\!=$ with arguments interchanged)

$$
\begin{aligned}
\mathit{fmap} &:: (a \to b) \quad\;\; \to f\,a \to f\,b \\
(\langle * \rangle) &:: f\,(a \to b) \to f\,a \to f\,b \\
(=\!\!\ll) &:: (a \to f\,b) \to f\,a \to f\,b
\end{aligned}
$$

- first argument of
  - *fmap*: a pure function
  - $\langle * \rangle$: an impure computation that yields a pure function
  - $=\!\!\ll$: an impure function
- $=\!\!\ll$ is strictly more powerful than $\langle * \rangle$, which in turn is more powerful than *fmap*, more later

3.2 The exception instance

- exceptions instantiate the monad class:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \\
&\mathbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \mathbf{where} \\
&\quad \mathit{return}\ a \qquad\quad = \mathit{Just}\ a \\
&\quad \mathit{Nothing} \gg\!= \_ = \mathit{Nothing} \\
&\quad \mathit{Just}\ a \quad\ \gg\!= f = f\ a
\end{aligned}
$$

- the effect-specific behaviour is to terminate the execution:

$$
\begin{aligned}
&\mathit{halt} :: \mathit{Maybe}\ a \\
&\mathit{halt} = \mathit{Nothing}
\end{aligned}
$$

3.2 Exception handling evaluator

- using $=\!\!\ll$ we can guard the second argument of *div*

$$
\begin{aligned}
&evalE :: Expr \to Maybe\ Integer\\
&evalE\ (Lit\ i) &&= pure\ i\\
&evalE\ (e1 :\!+\!: e2) &&= pure\ (+)\ \langle * \rangle\ evalE\ e1\ \langle * \rangle\ evalE\ e2\\
&evalE\ (e1 :\!*\!: e2) &&= pure\ (*)\ \langle * \rangle\ evalE\ e1\ \langle * \rangle\ evalE\ e2\\
&evalE\ (Div\ e1\ e2) &&= pure\ div\ \langle * \rangle\ evalE\ e1\ \langle * \rangle\ (guard =\!\!\ll evalE\ e2)
\end{aligned}
$$

$$
\begin{aligned}
&guard :: Integer \to Maybe\ Integer\\
&guard\ n = \mathbf{if}\ n == 0\ \mathbf{then}\ halt\ \mathbf{else}\ pure\ n
\end{aligned}
$$

- example evaluations:

$$
\begin{aligned}
&\rangle\rangle\rangle\rangle\quad evalE\ good\\
&Just\ 3\\
&\rangle\rangle\rangle\rangle\quad evalE\ bad\\
&Nothing
\end{aligned}
$$

3.2 Original evaluator, monadically

- we can also write the interpreter in a monadic style

$$
\begin{aligned}
&evalM :: (Monad\ m) \Rightarrow Expr \to m\ Integer \\
&evalM\ (Lit\ i) \qquad\quad = return\ i \\
&evalM\ (Div\ e1\ e2) = evalM\ e1 \gg\!= \backslash n1 \to \\
&\qquad\qquad\qquad\qquad\quad evalM\ e2 \gg\!= \backslash n2 \to \\
&\qquad\qquad\qquad\qquad\quad return\ (n1\ \text{`}div\text{`}\ n2)
\end{aligned}
$$

- (other cases omitted for reasons of space)

3.2 Original evaluator, using **do** notation

- we can also use **do**-notation for *Monad* instances

$$
\begin{aligned}
&\mathit{evalM} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Expr} \rightarrow m\ \mathit{Integer} \\
&\mathit{evalM}\ (\mathit{Lit}\ i) \qquad\quad = \mathbf{do}\ \mathit{return}\ i \\
&\mathit{evalM}\ (\mathit{Div}\ e1\ e2) = \mathbf{do}\ n1 \leftarrow \mathit{evalM}\ e1 \\
&\qquad\qquad\qquad\qquad\qquad\quad n2 \leftarrow \mathit{evalM}\ e2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (n1\ \text{`}\mathit{div}\text{`}\ n2)
\end{aligned}
$$

- (other cases omitted for reasons of space)
- imperative look'n'feel

3.2 Exceptional evaluator, monadically

- the monadic version is equally easy to extend

$$
\begin{array}{l}
\mathit{evalE} :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Integer} \\
\mathit{evalE}\ (\mathit{Lit}\ i) \qquad\quad = \mathbf{do}\ \mathit{return}\ i \\
\mathit{evalE}\ (\mathit{Div}\ e1\ e2) = \mathbf{do}\ n1 \leftarrow \mathit{evalE}\ e1 \\
\qquad\qquad\qquad\qquad\qquad\quad n2 \leftarrow \mathit{evalE}\ e2 \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\ n2 == 0\ \mathbf{then}\ \mathit{halt} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{return}\ (n1\ \text{`}\mathit{div}\text{`}\ n2)
\end{array}
$$

- (other cases omitted for reasons of space)

3.2 The IO monad

- monads like applicative functors form

  an abstract datatype of computations

- we have already encountered a monad: *IO*
- computations in general may have *effects*: I/O, exceptions, mutable state, non-determinism etc
- applicative functors and monads are a mechanism for cleanly incorporating such impure features in a pure setting
- there's no magic to monads in general: all monads are just plain data, implementing a particular interface
- but there is one magic monad: the *IO* monad
- its implementation is hard-wired in Haskell

  data *IO a* = ...
  instance *Monad IO* **where** ...

3.3 Case study: Monty Hall problem

> Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

- probabilistic programming
- two strategies: stick to original choice, or switch choice
- strategies as programs

3.3 The probability monad

- discrete probability distribution (probability mass function)

$$\textbf{type}\ \mathit{Prob} = \mathit{Rational}$$
$$\textbf{newtype}\ \mathit{Dist}\ \mathit{event} = D\ \{\mathit{fromD} :: [(\mathit{event}, \mathit{Prob})]\}$$

- *invariant:* probabilities of a distribution *dist* sum up to 1

$$\mathit{sum}\ [p \mid (e, p) \leftarrow \mathit{fromD}\ \mathit{dist}] == 1$$

- (ideally, each event occurs exactly once; *exercise:* define $\mathit{norm} :: (\mathit{Ord}\ \mathit{event}) \Rightarrow \mathit{Dist}\ \mathit{event} \to \mathit{Dist}\ \mathit{event}$)
- uniform distribution

$$\begin{aligned}
&\mathit{uniform} :: [\mathit{event}] \to \mathit{Dist}\ \mathit{event}\\
&\mathit{uniform}\ \mathit{es} = D\ [(e, 1 \mathbin{\%} n) \mid e \leftarrow \mathit{es}]\\
&\quad \textbf{where}\ n = \mathit{genericLength}\ \mathit{es}
\end{aligned}$$

3.3 The probability monad

- functor, applicative, and monad instances

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{Dist}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (D\ d) = D\ [\,(f\ e, p) \mid (e, p) \leftarrow d\,]\\
&\textbf{instance}\ \mathit{Applicative}\ \mathit{Dist}\ \textbf{where}\\
&\quad \mathit{pure}\ a \qquad\quad\ = D\ [\,(a, 1)\,]\\
&\quad D\ \mathit{fd}\ \langle * \rangle\ D\ \mathit{xd} = D\ [\,(f\ x, p * q) \mid (f, p) \leftarrow \mathit{fd}, (x, q) \leftarrow \mathit{xd}\,]\\
&\textbf{instance}\ \mathit{Monad}\ \mathit{Dist}\ \textbf{where}\\
&\quad \mathit{return}\ a \qquad = D\ [\,(a, 1)\,]\\
&\quad D\ \mathit{xd} \gg\!= k \qquad = D\ [\,(y, p * q) \mid (x, p) \leftarrow \mathit{xd}, (y, q) \leftarrow \mathit{fromD}\ (k\ x)\,]
\end{aligned}
$$

- *exercise:* is the invariant always satisfied?

3.3 Roll the dice

- a fair dice

$$dice = uniform\,[\,1\,..\,6\,]$$

- sum of two dice (applicative and monadic style)

$$rollA \;=\; pure\;(+)\;\langle*\rangle\;dice\;\langle*\rangle\;dice$$
$$rollM = \mathbf{do}\;\{\,a \leftarrow dice; b \leftarrow dice; return\;(a+b)\,\}$$

- roll the dice

$\rangle\rangle\rangle\rangle$ $rollA$
$D\,[\,(2, 1\,\%\,36), (3, 1\,\%\,36), (4, 1\,\%\,36), (5, 1\,\%\,36), (6, 1\,\%\,36),$
$(7, 1\,\%\,36), (3, 1\,\%\,36), \ldots$
$\rangle\rangle\rangle\rangle$ $norm\;it$
$D\,[\,(2, 1\,\%\,36), (3, 1\,\%\,18), (4, 1\,\%\,12), (5, 1\,\%\,9), (6, 5\,\%\,36), (7, 1\,\%\,6),$
$(8, 5\,\%\,36), (9, 1\,\%\,9), (10, 1\,\%\,12), (11, 1\,\%\,18), (12, 1\,\%\,36)\,]$

3.3 Back to Monty Hall

- we model the game show as follows

$$\textbf{data}\ \mathit{Outcome} = \mathit{Win} \mid \mathit{Lose}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show})$$
$$\textbf{data}\ \mathit{Door} = \mathit{No1} \mid \mathit{No2} \mid \mathit{No3}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Enum})$$
$$\mathit{doors} = [\mathit{No1} \mathinner{..} \mathit{No3}]$$

- host hides the car behind one of the doors; you pick one

$$\begin{aligned}&\mathit{hide}, \mathit{pick} :: \mathit{Dist}\ \mathit{Door}\\&\mathit{hide} = \mathit{uniform}\ \mathit{doors}\\&\mathit{pick} = \mathit{uniform}\ \mathit{doors}\end{aligned}$$

- host teases you by opening one of the doors

$$\mathit{tease}\ h\ p = \mathit{uniform}\ (\mathit{doors} \mathbin{\backslash\backslash} [h, p])$$

- the two strategies

$$\begin{aligned}&\mathit{stick}, \mathit{switch} :: \mathit{Door} \to \mathit{Door} \to \mathit{Dist}\ \mathit{Door}\\&\mathit{stick}\ \ \ p\ t = \mathit{return}\ p\\&\mathit{switch}\ p\ t = \mathit{return}\ (\mathit{head}\ (\mathit{doors} \mathbin{\backslash\backslash} [p, t]))\end{aligned}$$

3.3 Back to Monty Hall

- whole game parametrized by strategy

$$
\begin{array}{l}
\mathit{play} :: (\mathit{Door} \to \mathit{Door} \to \mathit{Dist}\ \mathit{Door}) \to \mathit{Dist}\ \mathit{Outcome} \\
\mathit{play}\ \mathit{strategy} = \\
\quad \mathbf{do}\ h \leftarrow \mathit{hide} \qquad\qquad \text{— host hides the car behind door } h \\
\qquad\quad p \leftarrow \mathit{pick} \qquad\qquad \text{— you pick door } p \\
\qquad\quad t \leftarrow \mathit{tease}\ h\ p \qquad\ \text{— host teases you with door } t\ (\neq h, p) \\
\qquad\quad s \leftarrow \mathit{strategy}\ p\ t \quad \text{— you choose, based on } p \text{ and } t \\
\qquad\quad \mathit{return}\ (\mathbf{if}\ s == h\ \mathbf{then}\ \mathit{Win}\ \mathbf{else}\ \mathit{Lose})
\end{array}
$$

- you win iff your choice s equals h

$$
\begin{array}{l}
\rangle\rangle\rangle\rangle \quad \mathit{norm}\ (\mathit{play}\ \mathit{stick}) \\
D\ [\ (\mathit{Win}, 1 \% 3), (\mathit{Lose}, 2 \% 3)\ ] \\
\rangle\rangle\rangle\rangle \quad \mathit{norm}\ (\mathit{play}\ \mathit{switch}) \\
D\ [\ (\mathit{Win}, 2 \% 3), (\mathit{Lose}, 1 \% 3)\ ]
\end{array}
$$

- switching doubles (!) your chance of winning

3.4 Applicative functor laws ⋆

- instances of *Applicative* are required to satisfy the applicative functor laws

$$\begin{aligned}
&\mathit{pure}\ \mathit{id} \langle * \rangle v &&= v \\
&\mathit{pure}\ (\circ) \langle * \rangle u \langle * \rangle v \langle * \rangle w &&= u \langle * \rangle (v \langle * \rangle w) \\
&\mathit{pure}\ f \langle * \rangle \mathit{pure}\ x &&= \mathit{pure}\ (f\ x) \\
&u \langle * \rangle \mathit{pure}\ x &&= \mathit{pure}\ (\backslash f \to f\ x) \langle * \rangle u
\end{aligned}$$

- identity, composition, pure computations can be combined, pure computations can be interchanged with impure ones

3.4 Monad laws ⋆

- instances of *Monad* are required to satisfy the monad laws

$$
\begin{aligned}
m \gg\!= \lambda a \to return\ a &= m \\
return\ a \gg\!= \lambda b \to f\ b &= f\ a \\
(m \gg\!= \lambda a \to f\ a) \gg\!= \lambda b \to g\ b &= m \gg\!= \lambda a \to (f\ a \gg\!= \lambda b \to g\ b)
\end{aligned}
$$

- or, expressed in terms of *return* and $\odot$

$$
\begin{aligned}
f \odot return &= f \\
return \odot f &= f \\
(f \odot g) \odot h &= f \odot (g \odot h)
\end{aligned}
$$

- (so monads are intimately related to monoids)

3.4 Interdefinability ⋆

- applicative functor implies functor

$$\textbf{instance}\ (\mathit{Applicative}\ m) \Rightarrow \mathit{Functor}\ m\ \textbf{where}$$
$$\quad \mathit{fmap}\ f\ \mathit{mx} = \mathit{pure}\ f \langle * \rangle\ \mathit{mx}$$

- (the instance declaration is not legal Standard Haskell)
- (snappier: $\mathit{fmap} = \mathit{liftA}$, where liftA is provided by the standard library *Control.Applicative*)

3.4 Interdefinability ⋆

- monad implies functor and applicative functor

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Functor}\ m\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ \mathit{mx} = \textbf{do}\ \{\, x \leftarrow \mathit{mx}; \mathit{return}\ (f\ x)\,\}\\
&\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Applicative}\ m\ \textbf{where}\\
&\quad \mathit{pure}\ a \qquad\quad = \mathit{return}\ a\\
&\quad \mathit{mf}\ \langle * \rangle\ \mathit{mx} \ = \textbf{do}\ \{\, f \leftarrow \mathit{mf}; x \leftarrow \mathit{mx}; \mathit{return}\ (f\ x)\,\}
\end{aligned}
$$

- (the instance declarations are not legal Standard Haskell)
- (snappier: $\mathit{fmap} = \mathit{liftM}$ and $(\langle * \rangle) = \mathit{ap}$, where *liftM* and *ap* are provided by the standard library *Control.Monad*)

3.4 Interdefinability: pragmatics ⋆

- recall: *Ord* is a sub-class of *Eq*
- standard approach: provide an instance of *Eq*, then provide an instance of *Ord*, possibly using *Eq*
- surprisingly, we can also turn things upside down:

```
instance Eq T where
  a == b = compare a b == EQ
instance Ord T where ...
```

- the *Ord* instance is used to define the *Eq* instance!

3.4 Interdefinability: pragmatics ⋆

- *Monad* is a sub-class of *Applicative*, which in turn is a sub-class of *Functor*
- standard approach: provide an instance of *Functor*, then provide an instance of *Applicative*, and finally of *Monad*
- again, we can turn things upside down:

```
instance Functor M where
  fmap = liftM
instance Applicative M where
  pure  = return
  (<*>) = ap
instance Monad M where ...
```

- the *Monad* instance is used to define *Applicative* and *Functor* instances!

3.5 Abstraction, abstraction, abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- applicative functors and monads allow you to abstract over patterns of computations (effects)
- a small hierarchy in order of expressiveness:
  - functor
  - applicative functor
  - monad
- Haskell allows you to implement your own computational effect or combination of effects (how cool is this?)
- two levels of computations
  - application independent: *Applicative* and *Monad* instance
  - application dependent: e.g. *eval*
- datatypes for/as computations: *Maybe*, [ ] etc

Part 4

Type and class system extensions

4.0 Outline

4.0 Recap: strong typing and polymorphism

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types

4.0 Recap: strong typing and polymorphism

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- Haskell is *statically typed*: type checking occurs before run-time (after syntax checking)
- type checking guarantees that type errors cannot occur

4.0 Recap: strong typing and polymorphism

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- Haskell is *statically typed*: type checking occurs before run-time (after syntax checking)
- type checking guarantees that type errors cannot occur
- type safety and flexibility are in tension
- *polymorphism* partially releases the tension
- parametric polymorphism: same code for all types
- ad-hoc polymorphism: different code for different types

4.0 Overview

- we take a closer look at some features of Standard Haskell
  - kinds
  - nested datatypes
- we also discuss some extensions of the type and class system
  - type families
  - rank-2 types
- we take an example-driven approach

4.0 Types of types

- *Int* is a type
- *Maybe* is a unary type constructor
- the function space "→" is a binary type constructor
- types and type constructors also possess types, called *kinds*

$$
\begin{array}{ll}
\mathit{Int} & :: * \\
\mathit{Maybe} & :: * \to * \\
[\,] & :: * \to * \\
(,) & :: * \to * \to * \\
(\to) & :: * \to * \to *
\end{array}
$$

- $*$ is the kind of types
- binary type constructors are curried!

4.1 Random-access lists

- lists: extension is cheap, but indexing is slow

$\rangle\rangle\rangle\rangle$ **let** $n = 99999 :: \mathit{Int}$
$\rangle\rangle\rangle\rangle$ **let** $x = \mathit{map}\ \mathit{square}\ [n, n-1 \mathinner{..} 0]$
$\rangle\rangle\rangle\rangle$ $\mathit{sum}\ x$
333328333350000
(0.08 *secs*, 27, 416, 896 *bytes*)
$\rangle\rangle\rangle\rangle$ $\mathit{sum}\ [x \mathbin{!!} i \mid i \leftarrow [0 \mathinner{..} n]]$
333328333350000
(23.32 *secs*, 28, 217, 600 *bytes*)
$\rangle\rangle\rangle\rangle$ **let** $s = \mathit{fromList}\ x$
$\rangle\rangle\rangle\rangle$ $\mathit{sum}\ [s \mathbin{!} i \mid i \leftarrow [0 \mathinner{..} n]]$
333328333350000
(0.28 *secs*, 221, 007, 960 *bytes*)

- goal: efficient sequence type, extensible, fast indexing

4.1 Natural numbers and lists

- a datatype for unary numbers (Peano numbers)

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat} \\
&(+) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
&\mathit{Zero} \quad\ \ + n = n \\
&\mathit{Succ}\ m + n = \mathit{Succ}\ (m + n)
\end{aligned}
$$

- Haskell's list datatype

$$
\begin{aligned}
&\textbf{data}\ \mathit{List}\ \mathit{elem} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{elem}\ (\mathit{List}\ \mathit{elem}) \\
&(+\!\!+) :: \mathit{List}\ \mathit{elem} \to \mathit{List}\ \mathit{elem} \to \mathit{List}\ \mathit{elem} \\
&\mathit{Nil} \qquad\quad +\!\!+\ n = n \\
&\mathit{Cons}\ a\ m +\!\!+\ n = \mathit{Cons}\ a\ (m +\!\!+\ n)
\end{aligned}
$$

- same structure with the elements witnessing the number

4.1 Natural numbers and lists

- removing the witnesses

$$
\begin{aligned}
&\mathit{length} :: \mathit{List}\ \mathit{elem} \to \mathit{Nat}\\
&\mathit{length}\ (\mathit{Nil}) = \mathit{Zero}\\
&\mathit{length}\ (\mathit{Cons}\ a\ n) = \mathit{Succ}\ (\mathit{length}\ n)
\end{aligned}
$$

- adding the witnesses

$$
\begin{aligned}
&\mathit{replicate} :: \mathit{elem} \to \mathit{Nat} \to \mathit{List}\ \mathit{elem}\\
&\mathit{replicate}\ a\ (\mathit{Zero}) = \mathit{Nil}\\
&\mathit{replicate}\ a\ (\mathit{Succ}\ n) = \mathit{Cons}\ a\ (\mathit{replicate}\ a\ n)
\end{aligned}
$$

4.1 Numerical representations

- lists are modelled after the unary numbers
- operations on lists are modelled after the corresponding operations on unary numbers

unary numbers	**lists**
meaning	length
zero	nil
increment	cons
decrement	tail
addition	concatenation
comparison	indexing
number system	**container type**

- the list datatype is a so-called *numerical representation*
- can we repeat the exercise with binary numbers?
- binary numbers are more efficient than unary numbers

4.1 Recap: binary numbers

- a datatype for binary numbers

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Binary}\\
&\quad = \mathit{Nil}\\
&\quad \mid \mathit{Zero}\ \mathit{Binary}\\
&\quad \mid \mathit{One}\ \ \mathit{Binary}
\end{aligned}
$$

- meaning of a binary number (semantics)

$$
\begin{aligned}
&\mathit{natural} :: \mathit{Binary} \to \mathit{Int}\\
&\mathit{natural}\ (\mathit{Nil}) = 0\\
&\mathit{natural}\ (\mathit{Zero}\ n) = 0 + 2 * \mathit{natural}\ n\\
&\mathit{natural}\ (\mathit{One}\ \ n) = 1 + 2 * \mathit{natural}\ n
\end{aligned}
$$

4.1 Binary numbers and random-access lists

- a datatype for binary numbers

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Binary} & & \\
\quad = \mathit{Nil} & & \text{— } \mathit{natural}\,(\mathit{Nil}) = 0 \\
\quad \mid \mathit{Zero}\ \mathit{Binary} & & \text{— } \mathit{natural}\,(\mathit{Zero}\ n) = 0 + 2 * \mathit{natural}\ n \\
\quad \mid \mathit{One}\ \ \mathit{Binary} & & \text{— } \mathit{natural}\,(\mathit{One}\ n) = 1 + 2 * \mathit{natural}\ n
\end{array}
$$

- sequence type modelled after binary numbers (failed attempt)

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Sequ}\ \mathit{elem} \\
\quad = \mathit{Nil} \\
\quad \mid \mathit{Zero}\qquad\quad (\mathit{Sequ}\ \mathit{elem}) \\
\quad \mid \mathit{One}\ \mathit{elem}\ (\mathit{Sequ}\ \mathit{elem})
\end{array}
$$

- doesn't work: tails must contain twice as many elements

4.1 Binary numbers and random-access lists

- a datatype for binary numbers

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Binary}\\
&\quad = \mathit{Nil} && \text{— } \mathit{natural}\,(\mathit{Nil}) = 0\\
&\quad \mid \mathit{Zero}\ \mathit{Binary} && \text{— } \mathit{natural}\,(\mathit{Zero}\ n) = 0 + 2 * \mathit{natural}\ n\\
&\quad \mid \mathit{One}\ \ \mathit{Binary} && \text{— } \mathit{natural}\,(\mathit{One}\ n) = 1 + 2 * \mathit{natural}\ n
\end{aligned}
$$

- if N is a container type for storing exactly n elements, then $\mathit{Pair}\ N$ allows us to store exactly $2 * n$ elements

$$\mathbf{type}\ \mathit{Pair}\ \mathit{elem} = (\mathit{elem}, \mathit{elem})$$

- sequence type modelled after binary numbers

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Sequ}\ \mathit{elem}\\
&\quad = \mathit{Nil}\\
&\quad \mid \mathit{Zero}\qquad\quad (\mathit{Sequ}\ (\mathit{Pair}\ \mathit{elem}))\\
&\quad \mid \mathit{One}\ \mathit{elem}\ (\mathit{Sequ}\ (\mathit{Pair}\ \mathit{elem}))
\end{aligned}
$$

4.1 Regular versus nested datatypes

- all of the datatypes we have seen before are *regular*
- occurrences of the declared type on the right-hand side of the defining equation are copies of the left-hand side
- *Sequ* is a *nested datatype*
- occurrences on the right-hand side appear with different instances of the accompanying type parameter(s)

$$
\begin{array}{l}
\rangle\rangle\rangle\rangle \quad \mathit{foldr}\ \mathit{cons}\ \mathit{Nil}\ [1\,..\,7] \\
\mathit{One}\ 1\ (\mathit{One}\ (2,3)\ (\mathit{One}\ ((4,5),(6,7))\ \mathit{Nil})) \\
\rangle\rangle\rangle\rangle \quad \mathit{cons}\ 0\ \mathit{it} \\
\mathit{Zero}\ (\mathit{Zero}\ (\mathit{Zero}\ (\mathit{One}\ (((0,1),(2,3)),((4,5),(6,7)))\ \mathit{Nil})))
\end{array}
$$

- the elements are integers, pairs of integers, pairs of pairs of integers, . . .

4.1 Meaning and size

- meaning of a binary number (semantics)

$$
\begin{aligned}
&\mathit{natural} :: \mathit{Binary} \to \mathit{Int} \\
&\mathit{natural}\ (\mathit{Nil}) \quad\quad\ \ = 0 \\
&\mathit{natural}\ (\mathit{Zero}\ n) = 0 + 2 * \mathit{natural}\ n \\
&\mathit{natural}\ (\mathit{One}\ \ n) = 1 + 2 * \mathit{natural}\ n
\end{aligned}
$$

- size of a random-access list

$$
\begin{aligned}
&\mathit{size} :: \mathit{Sequ}\ \mathit{elem} \to \mathit{Int} \\
&\mathit{size}\ (\mathit{Nil}) \quad\quad\quad = 0 \\
&\mathit{size}\ (\mathit{Zero}\ \ \ n) = 0 + 2 * \mathit{size}\ n \\
&\mathit{size}\ (\mathit{One}\ a\ n) = 1 + 2 * \mathit{size}\ n
\end{aligned}
$$

- the function *size* uses *polymorphic recursion*: the recursive call is at type *Sequ* (*Pair elem*), not *Sequ elem*
- the type signature is mandatory!

4.1 Increment and cons

- incrementing a binary number

$$
\begin{aligned}
&\mathit{succ} :: \mathit{Binary} \to \mathit{Binary}\\
&\mathit{succ}\ (\mathit{Nil}) &&= \mathit{One}\ \mathit{Nil}\\
&\mathit{succ}\ (\mathit{Zero}\ n) &&= \mathit{One}\ n\\
&\mathit{succ}\ (\mathit{One}\ \ n) &&= \mathit{Zero}\ (\mathit{succ}\ n)
\end{aligned}
$$

- extending a random-access list (cons)

$$
\begin{aligned}
&\mathit{cons} :: \mathit{elem} \to \mathit{Sequ}\ \mathit{elem} \to \mathit{Sequ}\ \mathit{elem}\\
&\mathit{cons}\ a\ (\mathit{Nil}) &&= \mathit{One}\ a\ \mathit{Nil}\\
&\mathit{cons}\ a\ (\mathit{Zero}\ n) &&= \mathit{One}\ a\ n\\
&\mathit{cons}\ a1\ (\mathit{One}\ a2\ n) &&= \mathit{Zero}\ (\mathit{cons}\ (a1, a2)\ n)
\end{aligned}
$$

4.1 Comparison and indexing

- comparison: is a binary number above a given natural?

$$
\begin{array}{lll}
(\succ) :: \mathit{Binary} \to \mathit{Int} \to \mathit{Bool} & & \text{— } (b \succ i) == (\mathit{natural}\ b > i) \\
\mathit{Nil} \succ n & = \mathit{False} & \\
\mathit{Zero}\ b \succ n & = b \succ (n \text{ `div` } 2) & \\
\mathit{One}\ b \succ 0 & = \mathit{True} & \\
\mathit{One}\ b \succ (n+1) & = b \succ (n \text{ `div` } 2) &
\end{array}
$$

- indexing (only defined for indices smaller than the size)

$$
\begin{array}{ll}
(!) :: \mathit{Sequ}\ \mathit{elem} \to \mathit{Int} \to \mathit{elem} & \\
\mathit{Nil}\ !\ n & = \mathit{error}\ \texttt{"(!): index out of bounds"} \\
\mathit{Zero}\ b\ !\ n & = b\ !\ (n \text{ `div` } 2)\ !\text{`}\ (n \text{ `mod` } 2) \\
\mathit{One}\ a\ b\ !\ 0 & = a \\
\mathit{One}\ a\ b\ !\ (n+1) & = b\ !\ (n \text{ `div` } 2)\ !\text{`}\ (n \text{ `mod` } 2)
\end{array}
$$

$$
\begin{array}{ll}
(a1, \_)\ !\text{`}\ 0 = & a1 \\
(\_, a2)\ !\text{`}\ 1 = & a2
\end{array}
$$

4.1 Numerical representations

- random-access lists are modelled after the binary numbers
- operations on random-access lists are modelled after the corresponding operations on binary numbers

binary numbers	**random-access lists**
meaning	size
zero	nil
increment	cons
decrement	tail
comparison	indexing
conversion: from unary	conversion: from lists
conversion: to unary	conversion: to lists
number system	**container type**

- what about addition?
- other numerical representations: binomial heaps

4.2 Type classes

- a method of a type class can be seen as a *family* of functions
- e.g. the family of equality functions

$$\begin{aligned}(==) &:: \mathit{Char} \to \mathit{Char} \to \mathit{Bool}\\ (==) &:: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\end{aligned}$$

- is captured by

$$\begin{array}{ll}\textbf{class } \mathit{Eq}\ a & \textbf{where } (==) :: a \to a \to \mathit{Bool}\\ \textbf{instance } \mathit{Eq}\ \mathit{Char} & \textbf{where} \ldots\\ \textbf{instance } \mathit{Eq}\ \mathit{Int} & \textbf{where} \ldots\end{array}$$

4.2 Type families

- how to capture families where two types vary?

$$\begin{aligned} &\mathit{insert} :: \mathit{Bool} \to \mathit{BitVector} \to \mathit{BitVector} \\ &\mathit{insert} :: \mathit{Int} \to \mathit{SearchTree} \to \mathit{SearchTree} \end{aligned}$$

- for each element type there is a dedicated set type

4.2 Type families

- how to capture families where two types vary?

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Bool} \to \mathit{BitVector} \to \mathit{BitVector}\\
&\mathit{insert} :: \mathit{Int} \to \mathit{SearchTree} \to \mathit{SearchTree}
\end{aligned}
$$

- for each element type there is a dedicated set type
- *type families* come to the rescue

$$
\begin{aligned}
&\mathbf{type\ family}\ \mathit{Set}\ \mathit{elem} :: *\\
&\mathbf{type\ instance}\ \mathit{Set}\ \mathit{Bool} = \mathit{BitVector}\\
&\mathbf{type\ instance}\ \mathit{Set}\ \mathit{Int} = \mathit{SearchTree}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Elem}\ \mathit{elem}\ \mathbf{where}\ \mathit{insert} :: \mathit{elem} \to \mathit{Set}\ \mathit{elem} \to \mathit{Set}\ \mathit{elem}\\
&\mathbf{instance}\ \mathit{Elem}\ \mathit{Bool}\ \mathbf{where}\ \ldots\\
&\mathbf{instance}\ \mathit{Elem}\ \mathit{Int}\ \mathbf{where}\ \ldots
\end{aligned}
$$

- *Set* can be seen as a function on types

4.3 Case study: C's printf

- a *type-safe* version of C's *printf* in Haskell

  ⟩⟩⟩⟩ *printf* ("I am " & *D* & " years old.") 51
  "I am 51 years old."
  ⟩⟩⟩⟩ *printf* ("I am " & *D* & " " & *S* & " old.") 1 "year"
  "I am 1 year old."
  ⟩⟩⟩⟩ *fmt* = "Color " & *S* & ", Number " & *D* & ", Float " & *F*
  ⟩⟩⟩⟩ *printf fmt* "purple" 4711 3.1415
  "Color purple, Number 4711, Float 3.1415"

- quite amazingly, *printf* takes a variable number of arguments, depending on the format directive

4.3 Format directives

- different format directives have different types!

```
⟩⟩⟩⟩ "Color " & S & ", Number " & D & ", Float " & F
("Color ", (S, (", Number ", (D, (", Float ", F)))))
⟩⟩⟩⟩ :type D
D
⟩⟩⟩⟩ D & "" & S
(D, ("", S))
⟩⟩⟩⟩ :type D & "" & S
D & "" & S :: (D, (String, S))
```

- non-trivial format directives are nested pairs; & is simply an infix operator for pairing

4.3 Format directives

- a domain-specific language (DSL) for format directives

$$\begin{aligned}
&\mathbf{data}\ D = D\ \mathbf{deriving}\ (Show)\\
&\mathbf{data}\ F\ = F\ \mathbf{deriving}\ (Show)\\
&\mathbf{data}\ S\ = S\ \mathbf{deriving}\ (Show)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{infixr}\ 4\ \&\\
&(\&) :: a \to b \to (a, b)\\
&a\ \&\ b = (a, b)
\end{aligned}$$

- D, F, S, (D, D), $(D, (S, F))$, etc are *singleton types*: each type contains exactly one element (ignoring $\bot$)

4.3 The type of *printf*

- the type of *printf* depends on the type of the format directive

$$
\begin{array}{llll}
printf :: D & \to Int \to & & String \\
printf :: F & \to Double \to & & String \\
printf :: String & \to & & String \\
printf :: (D, F) & \to Int \to Double \to & & String \\
printf :: ((D, F), (String, D)) & \to Int \to Double \to Int \to & & String
\end{array}
$$

- a "functorial" view of *printf*'s type

$$
\begin{array}{lll}
D & \to (Int \to) & String \\
F & \to (Double \to) & String \\
String & \to Id & String \\
(D, F) & \to ((Int \to) \circ (Double \to)) & String \\
((D, F), (String, D)) & \to ((Int \to) \circ (Double \to) \circ Id \circ (Int \to)) & String
\end{array}
$$

- so the type of *printf* is $dir \to Arg\ dir\ String$
- (the type operator section $(a \to) = (\to)\ a$ is not legal Haskell)

4.3 Interlude: functors

- recall the identity functor

$$\begin{array}{l}\textbf{newtype}\ \mathit{Id}\ a = I\ \{\mathit{fromI} :: a\}\\ \textbf{instance}\ \mathit{Functor}\ \mathit{Id}\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (I\ x) = I\ (f\ x)\end{array}$$

- functors compose

$$\begin{array}{l}\textbf{newtype}\ (f \circ g)\ a = C\ \{\mathit{fromC} :: f\ (g\ a)\}\\ \textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (f \circ g)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (C\ x) = C\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ x)\end{array}$$

- recall that $(a \rightarrow) = (\rightarrow)\ a$ is a functor

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ a)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ g = f \circ g\end{array}$$

4.3 The type of *printf*—continued

- the argument type of *printf* depends on the type of the format directive

$$
\begin{aligned}
&\textbf{type family } \mathit{Arg}\ \mathit{dir} :: * \to * \\
&\textbf{type instance } \mathit{Arg}\ D = (\to)\ \mathit{Int} \\
&\textbf{type instance } \mathit{Arg}\ F = (\to)\ \mathit{Double} \\
&\textbf{type instance } \mathit{Arg}\ S = (\to)\ \mathit{String} \\
&\textbf{type instance } \mathit{Arg}\ \mathit{String} = \mathit{Id} \\
&\textbf{type instance } \mathit{Arg}\ (a, b) = \mathit{Arg}\ a \circ \mathit{Arg}\ b
\end{aligned}
$$

- for example

$$
\begin{aligned}
\mathit{Arg}\ (D, (\mathit{String}, S))\ x &= ((\to)\ \mathit{Int} \circ \mathit{Id} \circ (\to)\ \mathit{String})\ x \\
&= ((\mathit{Int} \to) \circ (\mathit{String} \to))\ x \\
&= \mathit{Int} \to (\mathit{String} \to x)
\end{aligned}
$$

4.3 Towards *printf*

- we first define a helper function

$$
\begin{array}{l}
\mathbf{class}\ (\mathit{Functor}\ (\mathit{Arg}\ \mathit{dir})) \Rightarrow \mathit{Format}\ \mathit{dir}\ \mathbf{where} \\
\quad \mathit{format} :: \mathit{dir} \to \mathit{Arg}\ \mathit{dir}\ \mathit{String} \\
\mathbf{instance}\ \mathit{Format}\ D\ \mathbf{where} \\
\quad \mathit{format}\ D = \mathit{show} \\
\mathbf{instance}\ \mathit{Format}\ F\ \mathbf{where} \\
\quad \mathit{format}\ F = \mathit{show} \\
\mathbf{instance}\ \mathit{Format}\ S\ \mathbf{where} \\
\quad \mathit{format}\ S = \mathit{id} \\
\mathbf{instance}\ \mathit{Format}\ \mathit{String}\ \mathbf{where} \\
\quad \mathit{format}\ s = I\ s \\
\mathbf{instance}\ (\mathit{Format}\ \mathit{dir1}, \mathit{Format}\ \mathit{dir2}) \Rightarrow \mathit{Format}\ (\mathit{dir1}, \mathit{dir2})\ \mathbf{where} \\
\quad \mathit{format}\ (d1, d2) = C\ (\mathit{format}\ d1 \diamond \mathit{format}\ d2)
\end{array}
$$

4.3 Composition of formatters

- composing formatters ($\triangleright$ is a flipped variant of *fmap*)

$$(\diamond) :: (Functor\ f, Functor\ g) \Rightarrow f\ String \to g\ String \to f\ (g\ String)$$
$$f \diamond g = f \triangleright \backslash s \to g \triangleright \backslash t \to s +\!\!+ t$$
$$(\triangleright) :: (Functor\ f) \Rightarrow f\ a \to (a \to b) \to f\ b$$
$$x \triangleright h = fmap\ h\ x$$

- let's inspect the types:

$$\begin{array}{ll}
g & :: G\ String \\
(\backslash t \to s +\!\!+ t) & :: String \to String \\
g \triangleright (\backslash t \to s +\!\!+ t) & :: G\ String \\
f & :: F\ String \\
(\backslash s \to g \triangleright \backslash t \to s +\!\!+ t) & :: String \to G\ String \\
f \triangleright (\backslash s \to g \triangleright \backslash t \to s +\!\!+ t) & :: F\ (G\ String)
\end{array}$$

4.3 Getting rid of newtypes

- unfortunately, we are not quite there
- we have:

$$\rangle\rangle\rangle\rangle \quad : \textbf{type}\ \mathit{format}\ (D\ \&\ F)$$
$$\mathit{format}\ (D\ \&\ F) :: (((\to)\ \mathit{Int}) \circ ((\to)\ \mathit{Double}))\ \mathit{String}$$

- we want:

$$\rangle\rangle\rangle\rangle \quad : \textbf{type}\ \mathit{format}\ (D\ \&\ F)$$
$$\mathit{format}\ (D\ \&\ D) :: \mathit{Int} \to (\mathit{Double} \to \mathit{String})$$

- the **newtypes** get in the way
- solution: type cast

$$\rangle\rangle\rangle\rangle \quad : \textbf{type}\ \mathit{cast}\ (\mathit{format}\ (D\ \&\ F))$$
$$\mathit{cast}\ (\mathit{format}\ (D\ \&\ D)) :: \mathit{Int} \to (\mathit{Double} \to \mathit{String})$$

4.3 Getting rid of newtypes—continued

- the final implementation of *printf*

$$\mathit{printf} :: (\mathit{Format}\ \mathit{dir}, \mathit{Cast}\ (\mathit{Arg}\ \mathit{dir}\ \mathit{String})) \Rightarrow \mathit{dir} \to U\ (\mathit{Arg}\ \mathit{dir}\ \mathit{String})$$
$$\mathit{printf}\ d = \mathit{cast}\ (\mathit{format}\ d)$$

- the method *cast* transforms an element of a newtype *new* into the underlying type *U new*

$$\begin{array}{l}\textbf{class}\ \mathit{Cast}\ \mathit{new}\ \textbf{where}\\ \quad \textbf{type}\ U\ \mathit{new}\\ \quad \mathit{cast} :: \mathit{new} \to U\ \mathit{new}\end{array}$$

- *U* is an associated type family (a family associated to a class)

4.3 Getting rid of newtypes—continued

- "recursive" elimination of newtypes

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cast}\ \mathit{String}\ \textbf{where} \\
\quad \textbf{type}\ U\ \mathit{String} = \mathit{String} \\
\quad \mathit{cast} = \mathit{id} \\
\textbf{instance}\ (\mathit{Cast}\ b) \Rightarrow \mathit{Cast}\ (a \to b)\ \textbf{where} \\
\quad \textbf{type}\ U\ (a \to b) = a \to U\ b \\
\quad \mathit{cast}\ f = \backslash a \to \mathit{cast}\ (f\ a) \\
\textbf{instance}\ (\mathit{Cast}\ a) \Rightarrow \mathit{Cast}\ (\mathit{Id}\ a)\ \textbf{where} \\
\quad \textbf{type}\ U\ (\mathit{Id}\ a) = U\ a \\
\quad \mathit{cast}\ (I\ a) = \mathit{cast}\ a \\
\textbf{instance}\ (\mathit{Cast}\ (f\ (g\ a))) \Rightarrow \mathit{Cast}\ ((f \circ g)\ a)\ \textbf{where} \\
\quad \textbf{type}\ U\ ((f \circ g)\ a) = U\ (f\ (g\ a)) \\
\quad \mathit{cast}\ (C\ a) = \mathit{cast}\ a
\end{array}
$$

4.4 A little game ⋆

- recall the game from FP1: I give you a type, you give me a function of that type

$$\forall a\ .\ a \to a \to a$$

- how many total functions are there of this type?

4.4 A little game ⋆

- recall the game from FP1: I give you a type, you give me a function of that type

$$\forall a\ .\ a \to a \to a$$

- how many total functions are there of this type?
- only two!
- hence the type is isomorphic to the type *Bool*

4.4 Booleans as functions ⋆

- the Booleans can be represented as functions

$$\textbf{type}\ \mathit{Boolean} = \forall a\ .\ a \to a \to a$$

- the $\forall$ makes explicit that these functions are polymorphic
- type of an if-then-else
- *idea:* the Booleans act as conditionals

$$\begin{aligned}
&\mathit{false}, \mathit{true} :: \mathit{Boolean}\\
&\mathit{false} = \backslash x\ y \to y\\
&\mathit{true}\ \ = \backslash x\ y \to x
\end{aligned}$$

- read $\mathit{false}\ e1\ e2$ as **if** false **then** $e1$ **else** $e2$; likewise, read $\mathit{true}\ e1\ e2$ as **if** true **then** $e1$ **else** $e2$

4.4 Booleans as functions—continued ⋆

- negation, conjunction, and disjunction

$$
\begin{aligned}
&\mathit{not} :: \mathit{Boolean} \to \mathit{Boolean} \\
&\mathit{not}\ b = b\ \mathit{false}\ \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&(\&\&), (||) :: \mathit{Boolean} \to \mathit{Boolean} \to \mathit{Boolean} \\
&a\ \&\&\ b = a\ b\ \mathit{false} \\
&a\ ||\ b \quad = a\ \mathit{true}\ b
\end{aligned}
$$

- *not*, &&, and || take polymorphic functions as arguments and return polymorphic functions as results
- some example evaluations:

```
⟩⟩⟩⟩ (false && true) "yes" "no"
"no"
⟩⟩⟩⟩ (false || true) "yes" "no"
"yes"
```

4.4 Natural numbers as functions ⋆

- the natural numbers can be represented as functions, via repeated composition

$$\textbf{type}\ \mathit{Natural} = \forall a\ .\ (a \to a) \to (a \to a)$$

- the representation of n takes a function and an initial value, and applies the function n times to the initial value
- *idea:* the natural numbers act as for-loops (*bounded* iteration)

$$\begin{aligned}
&\mathit{zero} :: \mathit{Natural}\\
&\mathit{zero}\ f = \mathit{id}\\
&\mathit{succ} :: \mathit{Natural} \to \mathit{Natural}\\
&\mathit{succ}\ n\ f = f \circ n\ f
\end{aligned}$$

- read $n\ f\ a$ as `x = a ; for (i = 0; i < n; i++) x = f x; return x`
- these are called *Church numerals*

4.4 Naturals as functions—continued ⋆

- addition is a sequence of two for-loops

$$\begin{array}{l}\textbf{infixl}\ 6\ .\!+\\ (.\!+) :: \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural}\\ (m\ .\!+\ n)\ f = m\ f \circ n\ f\end{array}$$

- multiplication is given by two nested for-loops

$$\begin{array}{l}\textbf{infixl}\ 7\ .\!*\\ (.\!*) :: \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural}\\ (m\ .\!*\ n)\ f = m\ (n\ f)\end{array}$$

- exponentiation (mysterious?)

$$\begin{array}{l}\textbf{infixr}\ 8\ .\hat{}\\ (.\hat{}) :: \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural}\\ m\ .\hat{}\ n = n\ m\end{array}$$

4.4 Naturals as functions—continued ⋆

- some example evaluations:

```
⟩⟩⟩⟩ two = succ (succ zero)
⟩⟩⟩⟩ (two .+ succ two) ('|':) ""
"|||||"
⟩⟩⟩⟩ (two .∗ succ two) ('|':) ""
"||||||"
⟩⟩⟩⟩ (two .^ two .^ two) ('|':) ""
"||||||||||||||||"
⟩⟩⟩⟩ (two two two) ('|':) ""
"||||||||||||||||"
```

- (difficult: predecessor and subtraction)

4.5 Summary

- nested datatypes capture structural invariants
- dependent type: a type that depends on a value
- type families mimic dependent types via singleton types
- polymorphic functions are first-class citizens

Part 5

Duality: folds and unfolds

5.0 Outline

5.1 Duality: fold revisited

- so far we have focused on *consumers* (this seems to be close to the spirit of the time)
- *producers* are important too
- producers (unfolds) are *dual* to consumers (folds)
- to exhibit the duality we first re-define *foldr*

5.1 Fold re-defined

- a *non-recursive* “variant” of the list datatype

$$\textbf{data}\ \mathit{List}\ \mathit{elem}\ \mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{elem}\ \mathit{list}$$

- one layer of a list
- *foldr* reformulated

$$\begin{aligned}
&\mathit{fold} :: (\mathit{List}\ \mathit{elem}\ \mathit{ans} \to \mathit{ans}) \to ([\mathit{elem}] \to \mathit{ans})\\
&\mathit{fold}\ \mathit{alg} = \mathit{consume}\\
&\quad \textbf{where}\ \mathit{consume}\ [\,] \qquad\ \ = \mathit{alg}\ \mathit{Nil}\\
&\qquad\qquad\ \mathit{consume}\ (x : \mathit{xs}) = \mathit{alg}\ (\mathit{Cons}\ x\ (\mathit{consume}\ \mathit{xs}))
\end{aligned}$$

5.1 Examples of fold

- summing a list of numbers

$$
\begin{aligned}
&sum :: (Num\ a) \Rightarrow [\,a\,] \to a\\
&sum = fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\quad Nil \qquad\ \to 0\\
&\qquad\qquad\qquad\qquad\quad Cons\ a\ b \to a + b)
\end{aligned}
$$

- *map* can be expressed as a fold

$$
\begin{aligned}
&map :: (a \to b) \to ([\,a\,] \to [\,b\,])\\
&map\ f = fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\quad Nil \qquad\ \to [\,]\\
&\qquad\qquad\qquad\qquad\quad Cons\ a\ x' \to f\ a : x')
\end{aligned}
$$

5.1 Unfold

- folds consume lists
- *dually*, unfolds produce or generate lists

$$
\begin{array}{l}
\mathit{unfold} :: (\mathit{state} \to \mathit{List}\ \mathit{elem}\ \mathit{state}) \to (\mathit{state} \to [\mathit{elem}]) \\
\mathit{unfold}\ \mathit{coalg} = \mathit{produce} \\
\quad \mathbf{where}\ \mathit{produce}\ x = \mathbf{case}\ \mathit{coalg}\ x\ \mathbf{of} \\
\qquad\qquad\qquad \begin{array}{ll} \mathit{Nil} & \to [\,] \\ \mathit{Cons}\ a\ x' & \to a : \mathit{produce}\ x' \end{array}
\end{array}
$$

- think of *produce*'s argument as a state
- relation to OO iterators?

5.1 Examples of unfold

- $[\,m\,..\,n\,]$ aka $enumFromTo\ m\ n$

$$
\begin{array}{l}
enumFromTo :: (Num\ a, Ord\ a) \Rightarrow a \to a \to [\,a\,] \\
enumFromTo\ m\ n \\
\quad = unfold\ (\backslash i \to \mathbf{if}\ i > n\ \mathbf{then}\ Nil \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ Cons\ i\ (i+1))\ m
\end{array}
$$

- map can also be expressed as an unfold

$$
\begin{array}{l}
map :: (a \to b) \to ([\,a\,] \to [\,b\,]) \\
map\ f = unfold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \quad\ \ \to Nil \\
\qquad\qquad\qquad\qquad\qquad a : x' \to Cons\ (f\ a)\ x')
\end{array}
$$

5.1 Sorting by insertion

- given

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow \mathit{List}\ a\ [a] \to [a] \\
&\mathit{insert}\ \mathit{Nil} = [\,] \\
&\mathit{insert}\ (\mathit{Cons}\ x\ [\,]) = [x] \\
&\mathit{insert}\ (\mathit{Cons}\ x\ (y : ys)) \\
&\quad \mid x \leqslant y = x : y : ys \\
&\quad \mid \mathit{otherwise} = y : \mathit{insert}\ (\mathit{Cons}\ x\ ys)
\end{aligned}
$$

  we have

$$
\begin{aligned}
&\mathit{insertionSort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\
&\mathit{insertionSort} = \mathit{fold}\ \mathit{insert}
\end{aligned}
$$

- focus on the input (more later)

5.1 Sorting by selection

- dually, given

$$
\begin{array}{l}
select :: (Ord\ a) \Rightarrow [a] \rightarrow List\ a\ [a] \\
select\ [\,] \qquad\quad = Nil \\
select\ (x : xs) = \mathbf{case}\ select\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\quad Nil \qquad\qquad\quad \rightarrow Cons\ x\ [\,] \\
\qquad\qquad\qquad\quad Cons\ y\ ys \\
\qquad\qquad\qquad\qquad \mid x \leqslant y \qquad\quad \rightarrow Cons\ x\ (y : ys) \\
\qquad\qquad\qquad\qquad \mid otherwise \rightarrow Cons\ y\ (x : ys)
\end{array}
$$

we have

$$
\begin{array}{l}
selectionSort :: (Ord\ a) \Rightarrow [a] \rightarrow [a] \\
selectionSort = unfold\ select
\end{array}
$$

- focus on the output (more later)

5.1 Duality

- *unfold* is *dual* to *fold*

$$
\begin{aligned}
\mathit{fold} &:: (\mathit{List}\ a\ b \to b) \to ([a] \to b) \\
\mathit{unfold} &:: (b \to \mathit{List}\ a\ b) \to (b \to [a])
\end{aligned}
$$

- *fold* reduces a list to a value
- *fold*'s argument is a so-called *algebra*
- an algebra reduces a single 'layer' (step function)
- *unfold* grows a list from a seed
- *unfold*'s argument is a so-called *coalgebra*
- a coalgebra creates a single 'layer' (step function)

5.2 Generic programming

- many of the higher-order operators on lists generalize to other datatypes
- *map*
- *fold*
- *unfold*
- in fact, possible to give a single *generic* definition of each
- *map* using **deriving** (*Functor*)

5.2 Recursive datatypes and their base functors

- recall the recursive datatype of expressions

$$\begin{aligned} &\mathbf{data}\ \mathit{Expr} \\ &\quad = \mathit{Lit}\ \ \mathit{Integer} \\ &\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ &\quad \mid \mathit{Mul}\ \ \mathit{Expr}\ \mathit{Expr} \end{aligned}$$

- *base functor:* abstract away from recursive components

$$\begin{aligned} &\mathbf{data}\ \mathit{EXPR}\ \mathit{expr} \\ &\quad = \mathit{LIT}\ \ \mathit{Integer} \\ &\quad \mid \mathit{ADD}\ \mathit{expr}\ \mathit{expr} \\ &\quad \mid \mathit{MUL}\ \ \mathit{expr}\ \mathit{expr} \end{aligned}$$

- one layer of an expression
- *note:* Expr and $\mathit{EXPR}\ \mathit{Expr}$ are isomorphic

5.2 Base functor

- the base functor is a functor

```
instance Functor EXPR where
  fmap f (LIT i)        = LIT i
  fmap f (ADD e1 e2)    = ADD (f e1) (f e2)
  fmap f (MUL e1 e2)    = MUL (f e1) (f e2)
```

- *fmap f* applies *f* to the “recursive components”

5.2 Tying and untying the recursive knot

- relating base functor and recursive datatype

$$
\begin{array}{l}
\mathbf{class}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Base}\ f\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Rec}\ f :: * \\
\quad \mathit{inn} :: f\ (\mathit{Rec}\ f) \to \mathit{Rec}\ f \quad \text{— tying the recursive knot} \\
\quad \mathit{out} :: \mathit{Rec}\ f \to f\ (\mathit{Rec}\ f) \quad \text{— untying the recursive knot}
\end{array}
$$

- *idea:* the types $\mathit{Rec}\ f$ and $f\ (\mathit{Rec}\ f)$ are isomorphic: $\mathit{out} \circ \mathit{inn} = \mathit{id}$ and $\mathit{inn} \circ \mathit{out} = \mathit{id}$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Base}\ \mathit{EXPR}\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Rec}\ \mathit{EXPR} = \mathit{Expr} \\
\quad \mathit{inn}\ (\mathit{LIT}\ i) \qquad\quad = \mathit{Lit}\ i \\
\quad \mathit{inn}\ (\mathit{ADD}\ e1\ e2) = \mathit{Add}\ \ e1\ e2 \\
\quad \mathit{inn}\ (\mathit{MUL}\ e1\ e2) = \mathit{Mul}\ \ e1\ e2 \\
\quad \mathit{out}\ (\mathit{Lit}\ i) \qquad\quad = \mathit{LIT}\ i \\
\quad \mathit{out}\ (\mathit{Add}\ \ e1\ e2) = \mathit{ADD}\ e1\ e2 \\
\quad \mathit{out}\ (\mathit{Mul}\ \ e1\ e2) = \mathit{MUL}\ e1\ e2
\end{array}
$$

5.2 Generic fold and unfold

- fold and unfold given by

$$
\begin{aligned}
&\mathit{fold} :: (\mathit{Base}\ f) \Rightarrow (f\ a \to a) \to (\mathit{Rec}\ f \to a)\\
&\mathit{fold}\ \mathit{alg} = \mathit{consume}\\
&\quad \mathbf{where}\ \mathit{consume} = \mathit{alg} \circ \mathit{fmap}\ \mathit{consume} \circ \mathit{out}\\
&\mathit{unfold} :: (\mathit{Base}\ f) \Rightarrow (a \to f\ a) \to (a \to \mathit{Rec}\ f)\\
&\mathit{unfold}\ \mathit{coalg} = \mathit{produce}\\
&\quad \mathbf{where}\ \mathit{produce} = \mathit{inn} \circ \mathit{fmap}\ \mathit{produce} \circ \mathit{coalg}
\end{aligned}
$$

- note the duality: much clearer in generic presentation

$$
\begin{aligned}
\mathit{fold} &:: (\mathit{Base}\ f) \Rightarrow (f\ a \to a) \to (\mathit{Rec}\ f \to a)\\
\mathit{unfold} &:: (\mathit{Base}\ f) \Rightarrow (a \to f\ a) \to (a \to \mathit{Rec}\ f)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{consume} &= \mathit{alg} \circ \mathit{fmap}\ \mathit{consume} \circ \mathit{out}\\
\mathit{produce} &= \mathit{inn} \circ \mathit{fmap}\ \mathit{produce} \circ \mathit{coalg}
\end{aligned}
$$

5.2 Examples of fold

- evaluating an expression

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}\\
&\mathit{eval} = \mathit{fold}\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad \mathit{LIT}\ i \qquad\quad\ \to i\\
&\qquad\qquad\qquad \mathit{ADD}\ v1\ v2 \to v1 + v2\\
&\qquad\qquad\qquad \mathit{MUL}\ v1\ v2 \to v1 * v2)
\end{aligned}
$$

5.2 Example: binary trees

- recursive datatype of binary trees and its base functor

$$\textbf{data}\ \mathit{Tree\ elem} = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree\ elem})\ \mathit{elem}\ (\mathit{Tree\ elem})$$
$$\textbf{data}\ \mathit{TREE\ elem\ tree} = \mathit{EMPTY} \mid \mathit{NODE\ tree\ elem\ tree}$$

- instance of *Functor* and *Base*

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ (\mathit{TREE\ elem})\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{EMPTY}) = \mathit{EMPTY} \\
\quad \mathit{fmap}\ f\ (\mathit{NODE}\ l\ a\ r) = \mathit{NODE}\ (f\ l)\ a\ (f\ r) \\
\textbf{instance}\ \mathit{Base}\ (\mathit{TREE\ elem})\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Rec}\ (\mathit{TREE\ elem}) = \mathit{Tree\ elem} \\
\quad \mathit{inn}\ (\mathit{EMPTY}) = \mathit{Empty} \\
\quad \mathit{inn}\ (\mathit{NODE}\ l\ a\ r) = \mathit{Node}\ l\ a\ r \\
\quad \mathit{out}\ (\mathit{Empty}) = \mathit{EMPTY} \\
\quad \mathit{out}\ (\mathit{Node}\ l\ a\ r) = \mathit{NODE}\ l\ a\ r
\end{array}$$

5.2 Example: binary trees

- examples of folds: measures on trees

$$
\begin{aligned}
&size :: Tree\ elem \to Integer\\
&size = fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad EMPTY \qquad\quad \to 0\\
&\qquad\qquad\qquad NODE\ sl\ a\ sr \to sl + 1 + sr)
\end{aligned}
$$

$$
\begin{aligned}
&depth :: Tree\ elem \to Integer\\
&depth = fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad EMPTY \qquad\quad \to 0\\
&\qquad\qquad\qquad NODE\ dl\ a\ dr \to 1 + dl\ `max`\ dr)
\end{aligned}
$$

- example of an unfold: growing a tree

$$
\begin{aligned}
&create :: Integer \to Tree\ ()\\
&create = unfold\ (\backslash n \to \mathbf{case}\ n\ \mathbf{of}\\
&\qquad\qquad\qquad 0 \qquad\quad \to EMPTY\\
&\qquad\qquad\qquad m + 1 \to NODE\ k\ ()\ (m - k)\\
&\qquad\qquad\qquad\quad \mathbf{where}\ k = m\ `div`\ 2)
\end{aligned}
$$

5.2 Example: binary trees

- inserting an element into a binary search tree (failed attempt)

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \to \mathit{Tree}\ \mathit{elem}\\
&\mathit{insert}\ a = \mathit{fold}\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{EMPTY} \to \mathit{Node}\ \mathit{Empty}\ a\ \mathit{Empty}\\
&\qquad\qquad\qquad\qquad \mathit{NODE}\ \mathit{al}\ b\ \mathit{ar}\\
&\qquad\qquad\qquad\qquad\quad \mid a \leqslant b \qquad\quad\ \to \mathit{Node}\ \mathit{al}\ b\ \ ??\\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \to \mathit{Node}\ ??\ b\ \mathit{ar})
\end{aligned}
$$

- *problem:* the original sub-trees are not available

5.2 Primitive recursion

- meet *fold*'s mate

$$
\begin{aligned}
&para :: (Base\ f) \Rightarrow (f\ (Rec\ f \times a) \to a) \to (Rec\ f \to a)\\
&para\ alg = consume\\
&\quad \mathbf{where}\ consume = alg \circ fmap\ (id \vartriangle consume) \circ out
\end{aligned}
$$

- the algebra is additionally provided with the original sub-components (*para* eats its argument and keeps it too)

$$
\begin{aligned}
&\mathbf{data}\ a \times b = a :@\ b\\
&(\vartriangle) :: (x \to a) \to (x \to b) \to (x \to a \times b)\\
&(f \vartriangle g)\ x = f\ x :@\ g\ x
\end{aligned}
$$

- (*para* is short for paramorphism—from the Greek $\pi\alpha\rho\alpha$ meaning “beside”, “next to”, or “alongside”—aka primitive recursion or iteration)

5.2 Example: binary trees

- inserting an element into a binary search tree

$$
\begin{array}{l}
\mathit{insert} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \\
\mathit{insert}\ a = \mathit{para}\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad \mathit{EMPTY} \to \mathit{Node}\ \mathit{Empty}\ a\ \mathit{Empty} \\
\qquad\qquad \mathit{NODE}\ (l :@\ \mathit{al})\ b\ (r :@\ \mathit{ar}) \\
\qquad\qquad\quad \mid a \leqslant b \qquad\quad \to \mathit{Node}\ \mathit{al}\ b\ \ r \\
\qquad\qquad\quad \mid \mathit{otherwise} \to \mathit{Node}\ l\ \ b\ \mathit{ar})
\end{array}
$$

- *l* is the left sub-tree; *al* is the solution for the left sub-tree i.e. the sub-tree *l* with *a* added
- relies on lazy evaluation (why?)

5.2 Primitive co-recursion

- the recursion scheme *para* also has a dual

$$
\begin{aligned}
&apo :: (Base\ f) \Rightarrow (a \to f\,(Rec\ f + a)) \to (a \to Rec\ f)\\
&apo\ coalg = produce\\
&\quad \textbf{where}\ produce = inn \circ fmap\ (id \triangledown produce) \circ coalg
\end{aligned}
$$

- the coalgebra signals whether to stop or to continue

$$
\begin{aligned}
&\textbf{data}\ a + b = Stop\ a \mid Go\ b\\
&(\triangledown) :: (a \to x) \to (b \to x) \to (a + b \to x)\\
&(f \triangledown g)\ (Stop\ a) = f\ a\\
&(f \triangledown g)\ (Go\ \ \ b) = g\ b
\end{aligned}
$$

- two recursion schemes for the price of one!
- (*apo* is short for apomorphism—from the Greek $\alpha\pi o$, meaning “away from” or “separate”—aka primitive co-recursion or co-iteration)

5.2 Example: binary trees

- inserting an element into a binary search tree revisited

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \to \mathit{Tree}\ \mathit{elem}\\
&\mathit{insert}\ a = \mathit{apo}\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad \mathit{Empty} \to \mathit{NODE}\ (\mathit{Stop}\ \mathit{Empty})\ a\ (\mathit{Stop}\ \mathit{Empty})\\
&\qquad\qquad\qquad \mathit{Node}\ l\ b\ r\\
&\qquad\qquad\qquad\quad \mid a \leqslant b \qquad\quad\ \to \mathit{NODE}\ (\mathit{Go}\quad l)\ b\ (\mathit{Stop}\ r)\\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} \to \mathit{NODE}\ (\mathit{Stop}\ l)\ b\ (\mathit{Go}\quad r))
\end{aligned}
$$

- we continue in one branch and stop in the other
- does not rely on lazy evaluation

5.3 Case study: a duality of sorts

- insertion sort is dual to selection sort
- insertion sort is a fold
- selection sort is an unfold
- *next:* insertion itself is an unfold (actually an apo)
- *next:* selection itself is a fold (actually a para)

5.3 Type-directed programming

- let's derive the types of the algebras and coalgebras
- let $L = [a]$ and $F = \mathit{List}\ a$ for some type a

$$\begin{aligned}
&\mathit{insertionSort} = \mathit{fold}\ a && :: L \to L \\
&a = \mathit{unfold}\ c && :: F\ L \to L \\
&c = \mathit{swap} \circ \mathit{fmap}\ \mathit{out} && :: F\ L \to F\ (F\ L) \\
&\mathit{swap} && :: F\ (F\ L) \to F\ (F\ L)
\end{aligned}$$

- dually

$$\begin{aligned}
&\mathit{selectionSort} = \mathit{unfold}\ c' && :: L \to L \\
&c' = \mathit{fold}\ a' && :: L \to F\ L \\
&a' = \mathit{fmap}\ \mathit{inn} \circ \mathit{swap} && :: F\ (F\ L) \to F\ L \\
&\mathit{swap} && :: F\ (F\ L) \to F\ (F\ L)
\end{aligned}$$

5.3 Swapping elements

- the *algorithmic core* of swap-based sorts

$$
\begin{aligned}
&\mathit{swap} :: (\mathit{Ord}\ a) \Rightarrow \mathit{List}\ a\ (\mathit{List}\ a\ x) \rightarrow \mathit{List}\ a\ (\mathit{List}\ a\ x)\\
&\mathit{swap}\ \mathit{Nil} && = \mathit{Nil}\\
&\mathit{swap}\ (\mathit{Cons}\ a\ \mathit{Nil}) && = \mathit{Cons}\ a\ \mathit{Nil}\\
&\mathit{swap}\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ x))\\
&\quad \mid a \leqslant b && = \mathit{Cons}\ a\ (\mathit{Cons}\ b\ x)\\
&\quad \mid \mathit{otherwise} && = \mathit{Cons}\ b\ (\mathit{Cons}\ a\ x)
\end{aligned}
$$

- swaps two adjacent elements that are out of order
- note that *swap* is polymorphic in x
- the benefit of polymorphism: only one sensible definition

5.3 Two naive sorting algorithms

- naive insertion sort (why naive?) is dual bubble sort

$$
\begin{aligned}
&\mathit{naiveInsertionSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{naiveInsertionSort} = \mathit{fold}\ (\mathit{unfold}\ (\mathit{swap} \circ \mathit{fmap}\ \mathit{out}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bubbleSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{bubbleSort} = \mathit{unfold}\ (\mathit{fold}\ (\mathit{fmap}\ \mathit{inn} \circ \mathit{swap}))
\end{aligned}
$$

- fold of an unfold *versus* unfold of a fold
- two algorithms for the price of one!

5.3 Swapping elements revisited

- the definition of insertion sort is naive because insert always traverses the ordered list to its very end
- the variant below stops in one branch

$$
\begin{aligned}
&\mathit{swop} :: (\mathit{Ord}\ a) \Rightarrow \mathit{List}\ a\ (x \times \mathit{List}\ a\ x) \to \mathit{List}\ a\ (x + \mathit{List}\ a\ x)\\
&\mathit{swop}\ \mathit{Nil} = \mathit{Nil}\\
&\mathit{swop}\ (\mathit{Cons}\ a\ (x :@\ \mathit{Nil})) = \mathit{Cons}\ a\ (\mathit{Stop}\ x)\\
&\mathit{swop}\ (\mathit{Cons}\ a\ (x :@\ \mathit{Cons}\ b\ x'))\\
&\quad \mid a \leqslant b \quad = \mathit{Cons}\ a\ (\mathit{Stop}\ x)\\
&\quad \mid \mathit{otherwise} = \mathit{Cons}\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ x'))
\end{aligned}
$$

- swop is short for swap'n'stop

5.3 Two sorting algorithms

- insertion sort is dual selection sort

$$\begin{aligned}
&\mathit{insertionSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{insertionSort} = \mathit{fold}\ (\mathit{apo}\ (\mathit{swop} \circ \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out})))
\end{aligned}$$

$$\begin{aligned}
&\mathit{selectionSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{selectionSort} = \mathit{unfold}\ (\mathit{para}\ (\mathit{fmap}\ (\mathit{id} \triangledown \mathit{inn}) \circ \mathit{swop}))
\end{aligned}$$

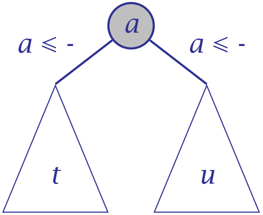
- fold of an apomorphism *versus* unfold of a paramorphism
- two algorithms for the price of one

5.3 Heapsort and minglesort ⋆

- insertion sort and selection sort have a running time of $\Theta(n^2)$
- quadratic running-time is unavoidable if only adjacent elements are swapped
- efficient sorting algorithms typically involve some intermediate data structure e.g. a search tree or a heap
- let's take a closer look at *heap sort*
- two-phase algorithm:
  - first phase: create heap from unordered list
  - second phase: reduce heap to ordered list

5.3 Heap-ordered trees ⋆

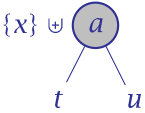
- a tree is *heap-ordered* if the element at each node is no larger than the elements at its children



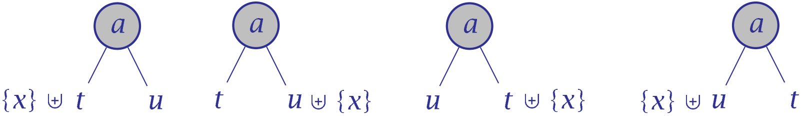
- thus, the element at the root is minimal
- the elements on any path from the root to a leaf are ordered
- note that there are no conditions on the relative order of elements between siblings

5.3 Inserting an element into a heap ⋆

- what are our options?



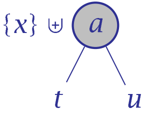
- the minimum, say a, must be kept at the root
- the maximum, say x, must be inserted recursively
- we have essentially four (!) choices:



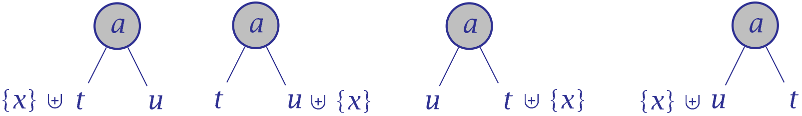
- which one is preferable?

5.3 Inserting an element into a heap ⋆

- what are our options?



- the minimum, say a, must be kept at the root
- the maximum, say x, must be inserted recursively
- we have essentially four (!) choices:



- which one is preferable?
- one of the two on the right: we always recursively insert into the left (right) subtree, but additionally *swap* the two subtrees

5.3 First phase: creating a heap ⋆

- base functor for binary heaps

$$\textbf{data}\ \mathit{Heap}\ \mathit{elem}\ \mathit{tree} = \mathit{Null} \mid \mathit{Heap}\ \mathit{elem}\ \mathit{tree}\ \mathit{tree}$$

- (*exercise:* define *Functor* and *Base* instances)
- *algorithmic core* of heap creation

$$\begin{aligned}
&\mathit{pile} :: (\mathit{Ord}\ a) \Rightarrow \mathit{List}\ a\ (x \times \mathit{Heap}\ a\ x) \to \mathit{Heap}\ a\ (x + \mathit{List}\ a\ x)\\
&\mathit{pile}\ \mathit{Nil} \qquad\qquad\qquad\qquad\quad\;\; = \mathit{Null}\\
&\mathit{pile}\ (\mathit{Cons}\ a\ (t :@\ \mathit{Null})) = \mathit{Heap}\ a\ (\mathit{Stop}\ t)\ (\mathit{Stop}\ t)\\
&\mathit{pile}\ (\mathit{Cons}\ a\ (t :@\ \mathit{Heap}\ b\ l\ r))\\
&\quad \mid a \leqslant b \qquad\;\; = \mathit{Heap}\ a\ (\mathit{Go}\ (\mathit{Cons}\ b\ r))\ (\mathit{Stop}\ l)\\
&\quad \mid \mathit{otherwise} = \mathit{Heap}\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ r))\ (\mathit{Stop}\ l)
\end{aligned}$$

5.3 Second phase: reducing a heap ⋆

- *algorithmic core* of heap reduction

$$
\begin{aligned}
&\mathit{sift} :: (\mathit{Ord}\ a) \Rightarrow \mathit{Heap}\ b\ (x \times \mathit{List}\ a\ x) \to \mathit{List}\ b\ (x + \mathit{Heap}\ a\ x)\\
&\mathit{sift}\ \mathit{Null} = \mathit{Nil}\\
&\mathit{sift}\ (\mathit{Heap}\ a\ (l :@\ \mathit{Nil})\ (r :@\ \_)) = \mathit{Cons}\ a\ (\mathit{Stop}\ r)\\
&\mathit{sift}\ (\mathit{Heap}\ a\ (l :@\ \_)\ (r :@\ \mathit{Nil})) = \mathit{Cons}\ a\ (\mathit{Stop}\ l)\\
&\mathit{sift}\ (\mathit{Heap}\ a\ (l :@\ \mathit{Cons}\ b\ l')\ (r :@\ \mathit{Cons}\ c\ r'))\\
&\quad \mid b \leqslant c = \mathit{Cons}\ a\ (\mathit{Go}\ (\mathit{Heap}\ b\ l'\ r))\\
&\quad \mid \mathit{otherwise} = \mathit{Cons}\ a\ (\mathit{Go}\ (\mathit{Heap}\ c\ l\ \ r'))
\end{aligned}
$$

5.3 Two efficient sorting algorithms ⋆

- heap sort is dual to “mingle sort” (a variant of merge sort)

$$
\begin{aligned}
&\mathit{heapSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{heapSort} = \mathit{unfold}\ (\mathit{para}\ (\mathit{fmap}\ (\mathit{id} \triangledown \mathit{inn}) \circ \mathit{sift})) \\
&\qquad\qquad \circ\ \mathit{fold}\ (\mathit{apo}\ (\mathit{pile} \circ \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mingleSort} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow [\mathit{elem}] \to [\mathit{elem}] \\
&\mathit{mingleSort} = \mathit{fold}\ (\mathit{apo}\ (\mathit{sift} \circ \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out}))) \\
&\qquad\qquad \circ\ \mathit{unfold}\ (\mathit{para}\ (\mathit{fmap}\ (\mathit{id} \triangledown \mathit{inn}) \circ \mathit{pile}))
\end{aligned}
$$

- two, well, actually four (!) algorithms for the price of one

5.4 Summary

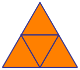
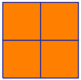
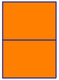
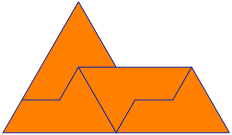
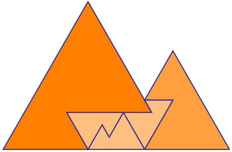
- producers are *dual* to consumers
- single *generic* definition of fold and unfold
- (duality much clearer in generic presentation)
- folds generalize to paramorphisms
- unfolds generalize to apomorphisms
- algorithmic duality: comparison-based sorting

Part 6

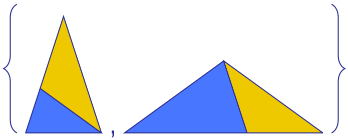
Case study: turtles and tesselations

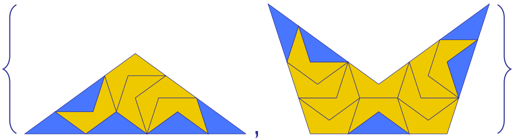
6.0 Outline

6.1 Reptiles

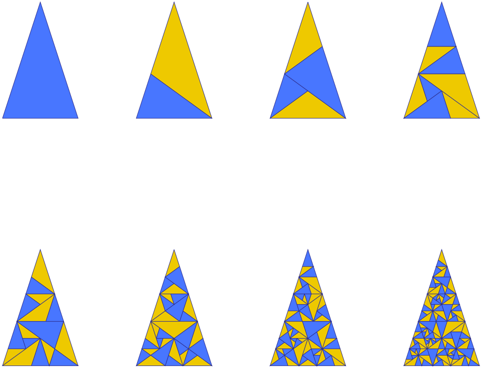






6.1 Setisets

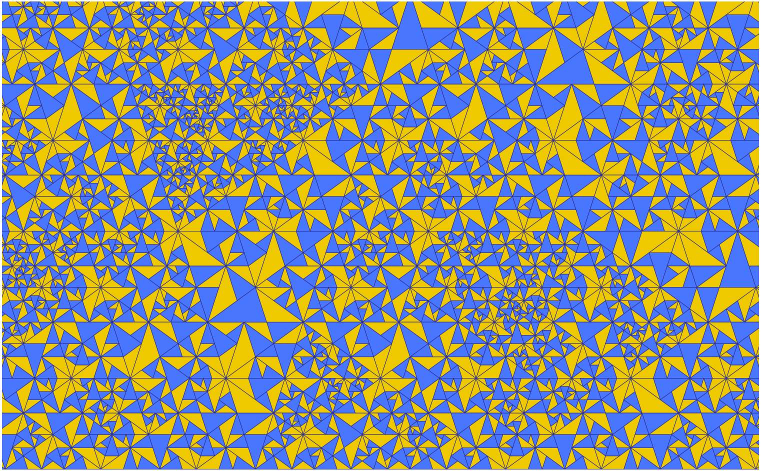




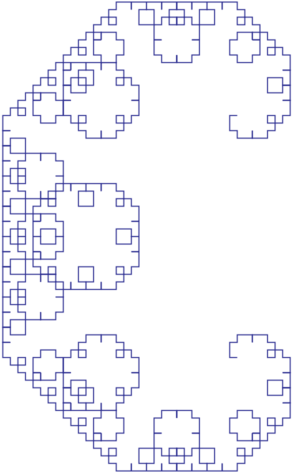
6.1 Self-tilings



6.1 Self-tilings



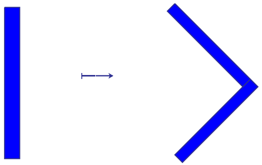
6.2 Lévy C Curve



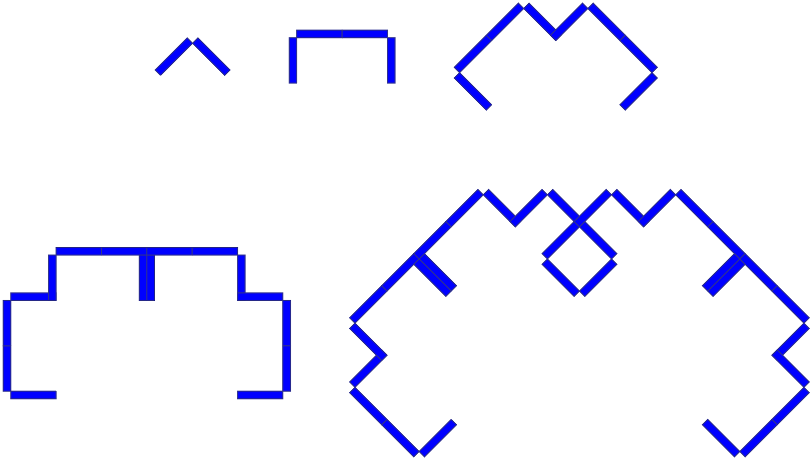
6.2 Lévy C Curve: Turtle Graphics

$$
\begin{aligned}
&\mathit{l\acute{e}vy} :: \mathit{Integer} \to \mathit{Program}\ \mathit{G8} \\
&\mathit{l\acute{e}vy}\ 0 \qquad\quad\ = \mathit{forward}\ 1 \\
&\mathit{l\acute{e}vy}\ (n+1) = \mathit{left}\ 1 \mathbin{;} \mathit{l\acute{e}vy}\ n \mathbin{;} \mathit{right}\ 2 \mathbin{;} \mathit{l\acute{e}vy}\ n \mathbin{;} \mathit{left}\ 1
\end{aligned}
$$

6.2 Lévy C Curve: Substitution Rule



6.2 Lévy C Curve



6.3 Turtle Graphics: Geometries

Turtle graphics is vector-based.
To avoid dealing with the nitty-gritty details of representing vectors, directions, and lengths, we introduce a type class:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Num}\ g, \mathit{Num}\ (\mathit{Dir}\ g), \mathit{Num}\ (\mathit{Len}\ g)) \Rightarrow \mathit{Geometry}\ g\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Dir}\ g \\
\quad \textbf{type}\ \mathit{Len}\ g \\
\quad \mathit{origin} :: g \\
\quad \mathit{polar} :: \mathit{Dir}\ g \to \mathit{Len}\ g \to g \\
\quad \mathit{cartesian} :: g \to (\mathit{Double}, \mathit{Double})
\end{array}
$$

We use tailor-made geometries for different types of reptiles: in n-gonia, G_n, the command $\mathit{left}\ 1$ instructs the turtle to turn left by $360° / n$ degrees. (We assume exact *real* arithmetic, making free use of algebraic numbers such as the golden ratio ϕ).

6.3 A DSL for Turtle Graphics

A turtle program is a list of commands (for technical reasons, we fuse the type of lists with the type of commands):

$$
\begin{array}{llll}
\textbf{data}\ \mathit{Program}\ g\ \mathit{obj} = & \mathit{Skip} & & \\
| & \mathit{Drop}\ \mathit{obj} & & (\mathit{Program}\ g\ \mathit{obj}) \\
| & \mathit{Forward} & (\mathit{Len}\ g) & (\mathit{Program}\ g\ \mathit{obj}) \\
| & \mathit{Left} & (\mathit{Dir}\ g) & (\mathit{Program}\ g\ \mathit{obj}) \\
| & \mathit{Fork} & (\mathit{Program}\ g\ \mathit{obj}) & (\mathit{Program}\ g\ \mathit{obj})
\end{array}
$$

Smart constructors for single commands:

$$
\begin{array}{l}
\mathit{drop} :: \mathit{obj} \to \mathit{Program}\ g\ \mathit{obj} \\
\mathit{drop}\ \mathit{obj} = \mathit{Drop}\ \mathit{obj}\ \mathit{Skip}
\end{array}
$$

$$
\begin{array}{l}
\mathit{forward} :: \mathit{Len}\ g \to \mathit{Program}\ g\ \mathit{obj} \\
\mathit{forward}\ a = \mathit{Forward}\ a\ \mathit{Skip}
\end{array}
$$

$\ldots$

6.3 Turtle Graphics: Sequencing

Sequencing (concatenation of two lists of commands):

$$
\begin{array}{lll}
\multicolumn{3}{l}{(;) :: \mathit{Program}\ g\ \mathit{obj} \to \mathit{Program}\ g\ \mathit{obj} \to \mathit{Program}\ g\ \mathit{obj}} \\
\mathit{Skip} & ; \mathit{cmd} = & \mathit{cmd} \\
\mathit{Drop}\ \mathit{obj}\ \ \mathit{cnt} & ; \mathit{cmd} = & \mathit{Drop}\ \mathit{obj}\ (\mathit{cnt} ; \mathit{cmd}) \\
\mathit{Forward}\ a\ \mathit{cnt} & ; \mathit{cmd} = & \mathit{Forward}\ a\ (\mathit{cnt} ; \mathit{cmd}) \\
\mathit{Left}\ \alpha\ \ \mathit{cnt} & ; \mathit{cmd} = & \mathit{Left}\ \alpha\ (\mathit{cnt} ; \mathit{cmd}) \\
\mathit{Fork}\ \mathit{cmd}_1\ \mathit{cnt} & ; \mathit{cmd} = & \mathit{Fork}\ \mathit{cmd}_1\ (\mathit{cnt} ; \mathit{cmd})
\end{array}
$$

6.3 Turtle Graphics: Substitution

First-order terms with variables form a monad with substitution acting as "bind":

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ (\mathit{Program}\ g)\ \textbf{where} \\
\quad \mathit{return}\ a = \mathit{drop}\ a \\
\quad \begin{array}{lll}
\mathit{Skip} & \gg\!= k = \mathit{Skip} & \\
(\mathit{Drop}\ a\ \ \mathit{cnt}) & \gg\!= k = k\ a\ ; & (\mathit{cnt} \gg\!= k) \\
(\mathit{Forward}\ r\ \mathit{cnt}) & \gg\!= k = \mathit{Forward}\ r & (\mathit{cnt} \gg\!= k) \\
(\mathit{Left}\ i\ \ \mathit{cnt}) & \gg\!= k = \mathit{Left}\ i & (\mathit{cnt} \gg\!= k) \\
(\mathit{Fork}\ \mathit{cmd}\ \ \mathit{cnt}) & \gg\!= k = \mathit{Fork}\ (\mathit{cmd} \gg\!= k) & (\mathit{cnt} \gg\!= k)
\end{array}
\end{array}
$$

A mapping from variables to terms is extended to a mapping from terms to terms (Kleisli extension):

$$
\begin{array}{l}
\mathit{ext} :: (a \to \mathit{Program}\ g\ b) \to (\mathit{Program}\ g\ a \to \mathit{Program}\ g\ b) \\
\mathit{ext}\ k\ m = m \gg\!= k
\end{array}
$$

6.4 Lévy C Curve Revisited

A setiset is given by

- a set of shapes and
- a collection of substitution rules, one for each shape.

Shapes are represented by elements of some datatype. Their semantics is specified by a mapping to turtle graphics:

$$\textbf{data}\ \mathit{Shape} = \mathit{Line}$$

$$\begin{aligned}&\mathit{line} :: \mathit{Shape} \rightarrow \mathit{Program}\ \mathit{G8}\ \mathit{Void}\\&\mathit{line}\ \mathit{Line} = \mathit{forward}\ 1\end{aligned}$$

A substitution rule is represented by a coalgebra:

$$\begin{aligned}&\mathit{lévy} :: \mathit{Shape} \rightarrow \mathit{Program}\ \mathit{G8}\ \mathit{Shape}\\&\mathit{lévy}\ \mathit{Line} = \mathit{left}\ 1\ ;\ \mathit{drop}\ \mathit{Line}\ ;\ \mathit{right}\ 2\ ;\ \mathit{drop}\ \mathit{Line}\ ;\ \mathit{left}\ 1\end{aligned}$$

6.4 Lévy C Curve Revisited

To create a picture, the substitution rule is repeatedly applied to a "start string", followed by an invocation of the semantic mapping:

$$ext\ line \mathbin{⟋} (ext\ lévy)^n \mathbin{⟋} drop\ Line$$

where $f \mathbin{⟋} a$ is right-associative function application and f^n denotes the n-fold self-composition of f.

6.4 Dragon Curve: Substitution Rules

6.4 Symmetry

We use coproducts to make symmetries explicit.

$$
\begin{aligned}
&\mathbf{data}\ a + b = L\ a \mid R\ b \\
&\mathbf{infix}\ 1\ \triangledown \\
&(\triangledown) :: (a \to x) \to (b \to x) \to (a + b \to x) \\
&(f \triangledown g)\ (L\ a) = f\ a \\
&(f \triangledown g)\ (R\ b) = g\ b
\end{aligned}
$$

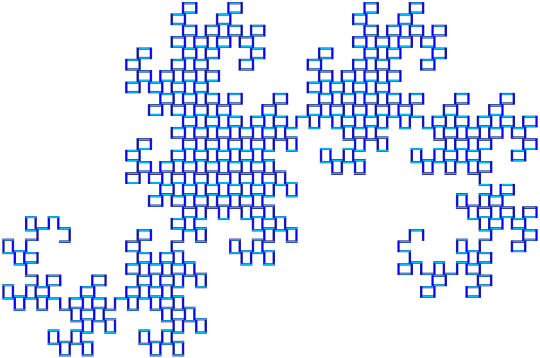
6.4 Dragon Curve

Only one substitution rule is programmed:

$$
\begin{aligned}
&\mathit{dragon} :: \mathit{Shape} \to \mathit{Program}\ \mathit{G8}\ (\mathit{Shape} + \mathit{Shape})\\
&\mathit{dragon}\ \mathit{Line} = \mathit{left}\ 1\ ; \mathit{drop}\ (\mathit{L}\ \mathit{Line})\ ; \mathit{right}\ 2\ ; \mathit{drop}\ (\mathit{R}\ \mathit{Line})\ ; \mathit{left}\ 1
\end{aligned}
$$

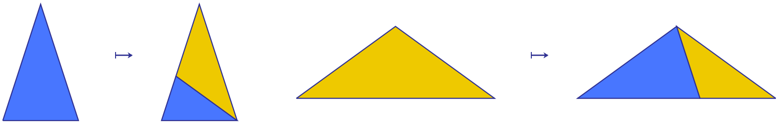
The coalgebra is then given by $\mathit{mirror} \cdot \mathit{dragon} \triangledown \mathit{dragon}$, where the transformation mirror changes left to right turns and vice versa: $\mathit{mirror}\ (\mathit{left}\ \alpha) = \mathit{left}\ (-\alpha)$.

6.4 Dragon Curve

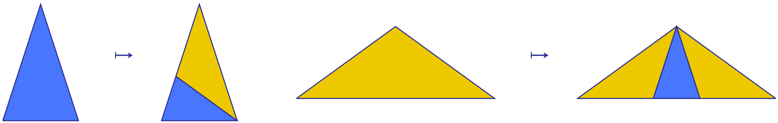


$$ext\ (line \triangledown line) \nearrow (ext\ (mirror \cdot dragon \triangledown dragon))^{10} \nearrow drop\ (R\ Line)$$
$$:: Program\ G8\ Void$$

6.5 Golden Triangles: Substitution Rules



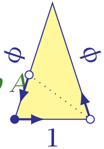
A closer inspection of the set reveals that it is irregular. We obtain a regular set if we sub-divide the larger copy of the A triangle:

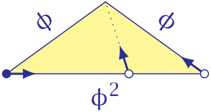


6.5 Golden Triangles: Substitution Rules

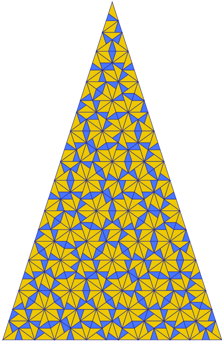
$$
\begin{aligned}
&\mathbf{data}\ \mathit{Robinson} = A \mid B\\[4pt]
&\mathit{robinson} :: \mathit{Robinson} \to \mathit{Program}\ G_{10}\ \mathit{Void}\\
&\mathit{robinson}\ A = \ \mathit{forward}\ 1 \mathbin{;} \mathit{left}\ 3 \mathbin{;} \mathit{forward}\ \phi \mathbin{;} \mathit{left}\ 4 \mathbin{;} \mathit{forward}\ \phi\\
&\mathit{robinson}\ B = \ \mathit{forward}\ \phi^2 \mathbin{;} \mathit{left}\ 4 \mathbin{;} \mathit{forward}\ \phi \mathbin{;} \mathit{left}\ 2 \mathbin{;} \mathit{forward}\ \phi
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{phidias} :: \mathit{Robinson} \to \mathit{Program}\ G_{10}\ \mathit{Robinson}\\
&\mathit{phidias}\ A\\
&\quad = \mathit{fork}\ (\mathit{left}\ 2 \mathbin{;} \mathit{forward}\ (1\ /\ \phi) \mathbin{;} \mathit{left}\ 5 \mathbin{;} \mathit{drop}\ A)\\
&\quad\ \mathbin{;}\ \mathit{fork}\ (\mathit{forward}\ 1 \mathbin{;} \mathit{left}\ 3 \mathbin{;} \mathit{drop}\ B)\\
&\mathit{phidias}\ B\\
&\quad = \mathit{fork}\ (\mathit{forward}\ \phi \mathbin{;} \mathit{left}\ 3 \mathbin{;} \mathit{phidias}\ A)\\
&\quad\ \mathbin{;}\ \mathit{fork}\ (\mathit{forward}\ \phi^2 \mathbin{;} \mathit{left}\ 4 \mathbin{;} \mathit{drop}\ B)
\end{aligned}
$$





6.5 Golden Triangles



$$\mathit{fmap}\ \mathit{robinson} \diagup (\mathit{scale}\ \phi \cdot \mathit{ext}\ \mathit{phidias})^7 \diagup \mathit{drop}\ A$$
$$\quad :: \mathit{Program}\ G_{10}\ (\mathit{Program}\ G_{10}\ \mathit{Void})$$

6.5 Sun and Star

We combine a triangle and its mirror image to form a kite or a dart:

$$
\begin{aligned}
&\mathit{kite}, \mathit{dart} :: \mathit{Program}\ G_{10}\ (\mathit{Robinson} + \mathit{Robinson})\\
&\mathit{kite} = \mathit{drop}\ (R\ A)\ ;\ \mathit{left}\ 4\ ;\ \mathit{drop}\ (L\ A)\ ;\ \mathit{right}\ 4\\
&\mathit{dart} = \mathit{drop}\ (R\ B)\ ;\ \mathit{left}\ 2\ ;\ \mathit{drop}\ (L\ B)\ ;\ \mathit{right}\ 2
\end{aligned}
$$

and then whirl the combined diagrams:

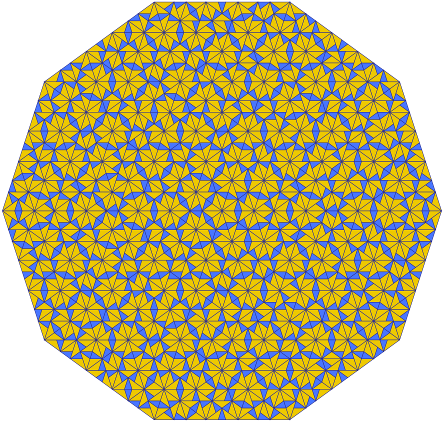
$$
\begin{aligned}
&\mathit{sun}, \mathit{star} :: \mathit{Program}\ G_{10}\ (\mathit{Robinson} + \mathit{Robinson})\\
&\mathit{sun} = \mathit{repeat}\ 5\ (\mathit{kite}\ ;\ \mathit{repeat}\ 2\ (\mathit{forward}\ 1\ ;\ \mathit{left}\ 1))\\
&\mathit{star} = \mathit{repeat}\ 5\ (\mathit{dart}\ ;\ \mathit{left}\ 2)
\end{aligned}
$$

To adapt *phidias* we make use of the coproduct of coalgebras:

$$
\begin{aligned}
&\mathbf{infix}\ 1\ \oplus\\
&(\oplus) :: (\mathit{Functor}\ f) \Rightarrow (a \to f\ a) \to (b \to f\ b) \to (a + b) \to f\ (a + b)\\
&f \oplus g = \mathit{fmap}\ L \cdot f \triangledown \mathit{fmap}\ R \cdot g
\end{aligned}
$$

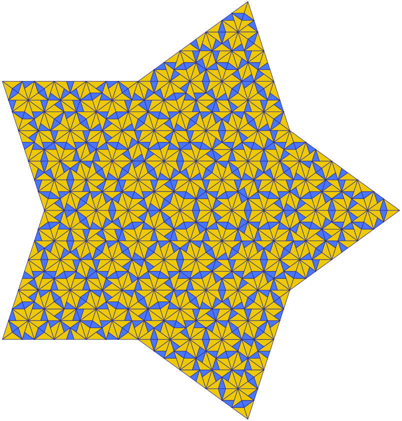
The coalgebra is then $\mathit{mirror} \cdot \mathit{phidias} \oplus \mathit{phidias}$.

6.5 Golden Triangles: Sun



$$\begin{aligned}&\mathit{fmap}\ (\mathit{mirror}\cdot\mathit{robinson}\mathbin{\triangledown}\mathit{robinson})\ \swarrow\\&\quad(\mathit{scale}\ \phi\cdot\mathit{ext}\ (\mathit{mirror}\cdot\mathit{phidias}\oplus\mathit{phidias}))^{6}\ \swarrow\ \mathit{sun}\\&\qquad ::\ \mathit{Program}\ G_{10}\ (\mathit{Program}\ G_{10}\ \mathit{Void})\end{aligned}$$

6.5 Golden Triangles: Star



$$\begin{array}{l} \mathit{fmap}\ (\mathit{mirror} \cdot \mathit{robinson} \mathbin{\triangledown} \mathit{robinson}) \mathbin{\diagup} \\ \quad (\mathit{scale}\ \phi \cdot \mathit{ext}\ (\mathit{mirror} \cdot \mathit{phidias} \oplus \mathit{phidias}))^{5} \mathbin{\diagup} \mathit{star} \\ \qquad :: \mathit{Program}\ G_{10}\ (\mathit{Program}\ G_{10}\ \mathit{Void}) \end{array}$$

6.6 Conclusion

- reptiles, setisets, and fractal curves
- free monads and coalgebras
- typed Lindenmayer systems

Part 7

Conclusion

7.0 Outline

Recap

7.1 Recap: Functional Programming 2

- lazy vs eager evaluation
- “lazy makes you pure”
- monadic approach to I/O
- abstract datatypes of computations (effects)
  - (functor)
  - applicative functor
  - monad
- nested datatypes capture structural invariants
- numerical representations
- type classes and type families
- generic programming: folds and unfolds

7.1 Thank you

Thanks for listening. It was good fun!