Introduction to Functional Programming

Ralf Hinze

University of Oxford

June 2012

The slides are partly based on

- Richard Bird's textbook "Introduction to Functional Programming using Haskell (2nd Edition)", and
- Jeremy Gibbons' FPR slides.

Thanks to both of you!

Part 0

Course aims and objectives

0.0 Outline

Aims

Motivation

Contents

What's it all about?

Literature

0.1 Aims

- *functional programming* is *programming with values*: *value-oriented programming*
- no ‘actions’, no side-effects — a radical departure from ordinary (imperative or OO) programming
- surprisingly, it is a powerful (and fun!) paradigm
- better ways of gluing programs together: *component-oriented programming*
- ideas are applicable in ordinary programming languages too; aim to introduce you to the ideas, to improve your day-to-day programming
- (I don’t expect you all to start using functional languages)

0.2 Motivation

LISP is worth learning [because of] the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot.

Eric S. Raymond, American computer programmer (1957–)
How to Become a Hacker
www.catb.org/~esr/faqs/hacker-howto.html

You can never understand one language until you understand at least two.

Ronald Searle, British artist (1920–2011)

0.3 Contents

Aims

Motivation

Contents

What’s it all about?

Literature

0.4 Expressions vs statements

- in ordinary programming languages the world is divided into a world of statements and a world of expressions
- statements:
  - x:=E, s1 ; s2, **while** b **do** s
  - evaluation order is important

    i:=i+1 ; a:=a∗i ≠ *a:=a∗i ; i:=i+1*

- expressions:
  - eg a+b∗c, a and not b
  - evaluation order is unimportant (assuming no side-effects): in (2∗a∗y+b) ∗ (2∗a∗y+c), evaluate either parenthesis first (or both together!)

0.4 Optimizations

- useful optimizations:
  - reordering:

```
    x:=0 ; p ; if x#0 then ... end
=   x:=0 ; if x#0 then ... end ; p
=   x:=0 ; p
```

  - common subexpression elimination:

```
    z := (2*a*y+b)*(2*a*y+c)
=   t := 2*a*y ; z := (t+b)*(t+c)
```

  - parallel execution: evaluate subexpressions concurrently
- most optimizations require *referential transparency*
  - all that matters about the expression is its value
  - follows from 'no side effects'
  - ...which follows from 'no :='
  - with assignments, side-effect-freeness is very hard to check

0.4 Programming with expressions

- expressions are much shorter and simpler than the corresponding statements
- eg compare using expression:

```
z := (2*a*y+b)*(2*a*y+c)
```

with not using expressions:

```
ac := 2; ac *= a; ac *= y; ac += b; t := ac;
ac := 2; ac *= a; ac *= y; ac += c; ac *= t;
z := ac
```

- but in order to discard statements, the expression language must be extended
- functional programming is *programming with an extended expression language*

0.4 Comparison with ‘ordinary’ programming

- insertion sort
- quicksort

0.4 Insertion sort

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

$$
\begin{aligned}
&\mathit{insert}\ a\ [\,] &&= [a]\\
&\mathit{insert}\ a\ (b : \mathit{xs}) &&\\
&\quad \mid a \leqslant b &&= a : b : \mathit{xs}\\
&\quad \mid \mathit{otherwise} &&= b : \mathit{insert}\ a\ \mathit{xs}
\end{aligned}
$$

```
PROCEDURE InsertSort(VAR a:ArrayT);
VAR i, j: CARDINAL;
    t: ElementT;
BEGIN
  FOR i := 2 TO Size DO
    (* a[1..i-1] already sorted *)
    t := a[i];
    j := i;
    WHILE (j > 1) AND (a[j−1] > t) DO
      a[j] := a[j−1]; j := j−1
    END;
    a[j] := t
  END
END InsertSort;
```

0.4 Quicksort

$$
\begin{aligned}
&\mathit{quickSort}\ [\,] &&= [\,]\\
&\mathit{quickSort}\ (x : \mathit{xs}) &&= \mathit{quickSort}\ \mathit{littles} \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{quickSort}\ \mathit{bigs}\\
&\quad \mathbf{where}\ \mathit{littles} &&= [\,a \mid a \leftarrow \mathit{xs}, a < x\,]\\
&\qquad\qquad\ \mathit{bigs} &&= [\,a \mid a \leftarrow \mathit{xs}, a \geqslant x\,]
\end{aligned}
$$

```
void quicksort(int a[], int l, int r)
{
  if (r > l)
    {
      int i = l; int j = r;
      int p = a[(l + r) / 2];
      for (;;) {
        while (a[i] < p) i++;
        while (a[j] > p) j--;
        if (i > j) break;
        swap(&a[i++], &a[j--]);
      };
      quicksort(a, l, j);
      quicksort(a, i, r);
    }
}
```

0.5 Literature

- Richard Bird, *Introduction to Functional Programming using Haskell (2nd Edition)*, Prentice Hall, 1998.
- Paul Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Graham Hutton, *Programming in Haskell*, Cambridge University Press, 2007.
- Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, No Starch Press, 2011.
- 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

Programming with expressions and values

1.5 Outline

1.6 Calculators

- functional programming is like using a pocket calculator
- user enters in expression, the system evaluates and prints result
- interactive 'read-eval-print' loop
- powerful mechanism for defining new functions
- we can calculate not only with numbers, but also with lists, trees, pictures, music ...

1.6 Scripts and sessions

- we will use *GHCi*, an interactive version of the *Glasgow Haskell Compiler*, a popular implementation of the standard lazy functional programming language *Haskell*
- program is a collection of modules
- a module is a collection of definitions: a *script*
- running a program consists of loading script and evaluating expressions: a *session*
- a standalone program includes a ‘main’ expression
- scripts may or may not be *literate* (emphasis on comments)

1.6 An illiterate script

```
-- compute the square of an integer
square :: Integer -> Integer
square x = x * x

-- smaller of two arguments
smaller :: (Integer, Integer) -> Integer
smaller (x, y) = if x < y then x else y
```

1.6 A literate script

```
The following function squares an integer.

> square :: Integer -> Integer
> square x = x * x

This one takes a pair of integers as an argument,
and returns the smaller of the two as a result.
For example,

  smaller (3, 4) = 3

> smaller :: (Integer, Integer) -> Integer
> smaller (x, y)  =  if x < y then x else y
```

1.6 Layout

- elegant and unobtrusive syntax
- structure obtained by layout, not punctuation
- all definitions in same scope must start in the same column
- indentation from start of definition implies continuation

```
smaller :: (Integer, Integer) → Integer
smaller (x, y)
  = if
      x < y
    then
      x
    else
      y
```

- blank lines around code in literate script!
- use spaces, not tabs!

1.6 A session

$$? \, 42$$
$$42$$
$$? \, 6 * 7$$
$$42$$
$$? \, \mathit{square}\ 7 - \mathit{smaller}\ (3, 4) - \mathit{square}\ (\mathit{smaller}\ (2, 3))$$
$$42$$
$$? \, \mathit{square}\ 1234567890$$
$$1524157875019052100$$

1.7 Evaluation

- interpreter evaluates expression by reducing to simplest possible form
- reduction is rewriting using meaning-preserving simplifications: replacing equals by equals

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

- expression 49 cannot be reduced any further: *normal form*
- *applicative order* evaluation: reduce arguments before expanding function definition (call by value, eager evaluation)

1.7 Alternative evaluation orders

- other evaluation orders are possible:

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

- final result is the same: if two evaluation orders terminate, both yield the same result (*confluence*)
- *normal order* evaluation: expand function definition before reducing arguments (call by need, lazy evaluation)

1.7 Non-terminating evaluations

- consider script

$$
\begin{aligned}
&\mathit{three} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{three}\ \_ = 3
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{infinity} :: \mathit{Integer}\\
&\mathit{infinity} = 1 + \mathit{infinity}
\end{aligned}
$$

- two different evaluation orders:

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

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

- not all evaluation orders terminate, even on the same expression; Haskell uses lazy evaluation

1.7 Values

- in FP, as in maths, the sole purpose of an expression is to denote a value
- other characteristics (time to evaluate, number of characters, etc) are irrelevant
- values may be of various kinds: numbers, truth values, characters, tuples, lists, functions, etc
- important to distinguish *abstract value* (the number 42) from concrete representation (the characters '4' and '2', the string "XLII", the bitsequence 0000000000101010)
- evaluator prints *canonical representation* of value
- some values have no canonical representation (eg functions), some have only infinite ones (eg π)

1.7 Undefined

- some expressions denote no normal value (eg *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, *nonstrict* functions (*three*) may give some non-$\perp$ value

1.8 Functions

- naturally, FP is a matter of functions
- script defines *functions* (*square*, *smaller*)
- (script actually defines *values*; indeed, in FP functions are values)
- function transforms (one or more) arguments into result
- *deterministic*: same arguments always give same result
- may be *partial*: result may sometimes be $\perp$
- eg cosine, square root; distance between two cities; compiler; text formatter; process controller

1.8 Function types

- *type declaration* in script specifies type of function
- eg $square :: Integer \rightarrow Integer$
- in general, $f :: A \rightarrow B$ indicates that function f takes arguments of type A and returns results of type B
- *apply* function to argument: $f\ x$
- sometimes parentheses are necessary: $square\ (3 + 4)$ (function application is an operator, binding more tightly than $+$)
- be careful not to confuse the function f with the value $f\ x$

1.8 Lambda

- notation for anonymous functions
- eg $\lambda x \rightarrow x * x$ as another way of writing *square*
- eg $\lambda a\ b \rightarrow a$ (which we'll call *const* later)
- ASCII '\' is nearest equivalent to Greek λ
- from Church's λ-calculus theory of computability (1941)

1.8 Declaration vs expression style

- Haskell is a committee language
- Haskell supports two different programming styles
- *declaration style*: using equations, patterns and expressions

$$quad :: Integer \to Integer$$
$$quad\ x = square\ x * square\ x$$

- *expression style*: emphasising the use of expressions

$$quad :: Integer \to Integer$$
$$quad = \lambda x \to square\ x * square\ x$$

- expression style is often more flexible
- experienced programmers use both simultaneously

1.8 Extensionality

- two functions are equal ($f = g$) if they give equal results for all arguments ($f\,x = g\,x$ for every x of the right type)
- this is why the two definitions of $quad$ (see previous slide) are equivalent
- the important thing about a function is its mapping from arguments to results
- other properties (eg how a mapping is described) are irrelevant
- eg these two functions are equal, as well:

$$\begin{aligned}
&double, double' :: Integer \rightarrow Integer \\
&double\ x\ \ = x + x \\
&double'\ x = 2 * x
\end{aligned}$$

1.8 Currying

- replace single structured argument by several simpler ones

$$\begin{aligned} &add :: (Integer, Integer) \to Integer \\ &add\ (x, y) = x + y \end{aligned}$$

$$\begin{aligned} &add' :: Integer \to (Integer \to Integer) \\ &add'\ x\ y = x + y \end{aligned}$$

- useful for reducing number of parentheses
- *add* takes a pair of *Integer*s and returns an *Integer*
- *add'* takes an *Integer* and returns a function of type $Integer \to Integer$
- eg $add'\ 3$ is a function; $(add'\ 3)\ 4$ reduces to 7
- can be written just $add'\ 3\ 4$ (see why shortly)

1.8 Operators

- functions with alphabetic names are *prefix*: $f\ 3\ 4$
- functions with symbolic names are *infix*: $3 + 4$
- make an alphabetic name infix by enclosing in backquotes: $17\ `mod`\ 10$
- make symbolic operator prefix (and curried) by enclosing it in parentheses: $(+)\ 3\ 4$
- thus, $add' = (+)$
- extend notion to include one argument too: *sectioning*
- eg $(1/)$ is the reciprocal function, (>0) is the positivity test

1.8 Associativity

- why operators at all? why not prefix notation?
- there is a problem of ambiguity:

  $$x \otimes y \otimes z$$

  what does this mean: $(x \otimes y) \otimes z$ or $x \otimes (y \otimes z)$?
- sometimes it doesn't matter, eg addition

  $$(x + y) + z = x + (y + z)$$

  the operator $+$ is associative
- *recommendation*: use infix notation *only* for associative operators
- the operator $+$ has also a neutral element

  $$x + 0 = x = 0 + x$$

- 0 and $+$ form a monoid (more later)

1.8 Association

- some operators are not associative ($-$, $/$, $\uparrow$)
- to disambiguate without parentheses, operators may *associate* to the left or to the right
- eg subtraction associates to the left: $5 - 4 - 2 = -1$
- function application associates to the left: $f\ a\ b$ means $(f\ a)\ b$
- function type operator associates to the right:
  $Integer \to Integer \to Integer$ means
  $Integer \to (Integer \to Integer)$
- not to be confused with *associativity*, when adjacent occurrences of same operator are unambiguous anyway

1.8 Precedence

- association does not help when operators are mixed
- to disambiguate without parentheses, there is a notion of *precedence* (binding power)
- eg $*$ has higher precedence (binds more tightly) than $+$

$$\textbf{infixl}\ 7\ *$$
$$\textbf{infixl}\ 6\ +$$

- function application can be seen as an operator, and has the highest precedence, so $\mathit{square}\ 3 + 4 = 13$

1.8 Composition

- glue functions together with *function composition*
- defined as follows:

$$
\begin{aligned}
&(\circ) :: (\mathit{Integer} \to \mathit{Integer}) \to (\mathit{Integer} \to \mathit{Integer}) \\
&\qquad\quad \to (\mathit{Integer} \to \mathit{Integer}) \\
&(f \circ g)\ x = f\ (g\ x)
\end{aligned}
$$

- eg function $\mathit{square} \circ \mathit{double}$ takes 3 to 36
- associative, so parentheses not needed in $f \circ g \circ h$
- (actually has a different type; explained later)

1.9 Definitions

- we've seen some simple definitions of functions so far
- can also define other kinds of values:

$$\mathit{name} :: \mathit{String}$$
$$\mathit{name} = \texttt{"Ralf"}$$

- all so far have had an identifier (and perhaps formal parameters) on the left, and an expression on the right
- other forms possible: conditional, pattern-matching and local definitions
- also recursive definitions (later sections)

1.9 Conditional definitions

- earlier definition of *smaller* used a *conditional expression*:

$$\begin{array}{l} \mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{smaller}\ (x, y) = \mathbf{if}\ x < y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{array}$$

- could also use *guarded equations*:

$$\begin{array}{l} \mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{smaller}\ (x, y) \\ \quad \mid x < y = x \\ \quad \mid x \geqslant y = y \end{array}$$

- each *clause* has a *guard* and an *expression* separated by $=$
- last guard can be *otherwise* (synonym for *True*)
- especially convenient with three or more clauses
- *declaration style*: guard; *expression style*: **if** ... **then** ... **else**...

1.9 Pattern matching

- define function by several equations
- arguments on lhs not just variables, but *patterns*
- patterns may be *variables* or *constants* (or *constructors*, later)
- eg

$$
\begin{aligned}
&day :: Integer \to String\\
&day\ 1 = \texttt{"Saturday"}\\
&day\ 2 = \texttt{"Sunday"}\\
&day\ \_ = \texttt{"Weekday"}
\end{aligned}
$$

- also *wildcard pattern* _
- evaluate by reducing argument to normal form, then applying first matching equation
- result is $\perp$ if argument has no normal form, or no equation matches

1.9 Local definitions

- repeated subexpression can be captured in a *local definition*

$$
\begin{aligned}
&qroots :: (Float, Float, Float) \to (Float, Float)\\
&qroots\ (a, b, c) = ((-b - sd) \mathbin{/} (2 * a), (-b + sd) \mathbin{/} (2 * a))\\
&\quad \mathbf{where}\ sd = sqrt\ (b * b - 4 * a * c)
\end{aligned}
$$

- scope of 'where' clause extends over whole right-hand side
- multiple local definitions can be made:

$$
\begin{aligned}
&demo :: Integer \to Integer \to Integer\\
&demo\ x\ y = (a + 1) * (b + 2)\\
&\quad \mathbf{where}\ a = x - y\\
&\qquad\qquad\ b = x + y
\end{aligned}
$$

  (nested scope, so layout rule applies here too: all definitions must start in same column)

- in conjunction with guarded equations, the scope of a **where** clause covers all guard clauses

1.9 let-expressions

- a **where** clause is syntactically attached to an equation
- also: definitions local to an expression

$$
\begin{aligned}
&\mathit{demo} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{demo}\ x\ y = \mathbf{let}\ a = x - y \\
&\qquad\qquad\qquad\quad\ b = x + y \\
&\qquad\qquad\quad\ \mathbf{in}\ (a + 1) * (b + 2)
\end{aligned}
$$

- *declaration style*: **where**; *expression style*: **let ... in...**
- **let**-expressions are more flexible than **where** clauses

1.10 The art of functional programming

- a problem is given by an expression
- a solution is a value
- a solution is obtained by evaluating an expression to a value
- a program introduces vocabulary to express problems and specifies rules for evaluating expressions
- the art of functional programming: finding rules
- Haskell has a very simple computational model
- as in primary school: replacing equals by equals
- we can calculate not only with numbers, but also with lists, trees, pictures, music ...

Part 2

Types and polymorphism

2.10 Outline

2.11 Strong typing

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur
- Haskell is *statically typed*: type checking occurs before runtime (after syntax checking)
- experience shows well-typed programs are likely to be correct
- Haskell can *infer types*: determine the most general type of each expression
- wise to specify (some) types anyway, for documentation and redundancy

2.12 Simple types

- booleans
- characters
- strings
- numbers

2.12 Booleans

- type *Bool* (note: type names capitalized)
- two constants, *True* and *False* (note: constructor names capitalized)
- eg definition by pattern-matching

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

- and &&, or ||, both strict in first argument

$$
\begin{aligned}
&(\&\&) :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
&\mathit{False}\ \&\&\ \_ = \mathit{False} \\
&\mathit{True}\ \&\&\ x = x
\end{aligned}
$$

- comparisons $==$, $\neq$, orderings $<$, $\leqslant$ etc

2.12 Boole design pattern

- every type comes with a design pattern
- *task:* define a function $f :: Bool \to S$;
- *step 1:* solve the problem for $False$

$$f\ False = \ \ldots$$

- *step 2:* solve the problem for $True$

$$\begin{aligned} f\ False &= \ \ldots \\ f\ True &= \ \ldots \end{aligned}$$

- (exercise: define your own conditional)

2.12 Characters

- type *Char*
- constants in single quotes: `'a'`, `'?'`
- special characters escaped: `'\''`, `'\n'`
- ASCII coding: $Data.Char.ord :: Char \to Int$,
  $Data.Char.chr :: Int \to Char$
- comparison and ordering, as before

2.12 Strings

- type *String*
- (actually defined in terms of *Char*; see later)
- constants in double quotes: `"Hello"`
- comparison and (lexicographic) ordering
- concatenation $+\!\!+$
- monadic *putStr* to print formatted text
- display function $show :: Integer \to String$ (actually more general than this; see later)

2.12 Numbers

- fixed-size (32-bit) integers *Int*
- arbitrary-precision integers *Integer*
- single- and double-precision floats *Float*, *Double*
- others too: rationals, complex numbers, ...
- comparisons and ordering
- $+$, $-$, $*$, $\uparrow$
- *abs*, *negate*
- $/$, *div*, *mod*, *quot*, *rem*
- etc
- operations are overloaded (more later)

2.13 Enumerations

- mechanism for declaring new types

  data *Day* = *Mon* | *Tue* | *Wed* | *Thu* | *Fri* | *Sat* | *Sun*

- eg *Bool* is not built in (although **if** … **then** … **else** syntax is):

  data *Bool* = *False* | *True*

- types may even be parameterized and recursive! (more later)

2.14 Tuples

- pairing types: eg $(\mathit{Char}, \mathit{Integer})$
- values in the same syntax: `('a',440)`
- selectors fst, snd
- definition by pattern matching:

$$\mathit{fst}\ (x, \_) = x$$

- nested tuples: $(\mathit{Integer}, (\mathit{Char}, \mathit{Bool}))$
- triples, etc: $(\mathit{Integer}, \mathit{Char}, \mathit{Bool})$
- nullary tuple $()$
- comparisons, (lexicographic) ordering

2.15 Polymorphism

- what is the type of *fst*?
- applicable at different types: $\mathit{fst}\ (1,2)$, $\mathit{fst}\ (\texttt{'a'}, \mathit{True})$, ...
- what about strong typing?
- *fst* is *polymorphic* — it works for *any* type of pairs:

$$\mathit{fst} :: (a, b) \to a$$

- a, b here are *type variables* (uncapitalized)
- values can be polymorphic too: $\bot :: a$
- regain principal types for all expressions

2.15 A little game

- here is a little game: I give you a type, you give me a function of that type
  - $Int \to Int$
  - $a \to a$
  - $(Int, Int) \to Int$
  - $(a, a) \to a$
  - $(a, b) \to a$
  - $[a] \to [a]$
- polymorphic functions: flexible to use, hard to define
- polymorphism is a property of an algorithm

2.16 Type synonyms

- alternative names for types
- brevity, clarity, documentation
- eg

$$\mathbf{type}\ \mathit{Card} = (\mathit{Rank}, \mathit{Suit})$$

- cannot be recursive
- just a ‘macro’: no new type

2.17 Type classes

- what about numeric operations?
- $(+) :: Integer \to Integer \to Integer$
- $(+) :: Float \to Float \to Float$
- cannot have $(+) :: a \to a \to a$ (too general)
- the solution is *type classes* (sets of types)
- eg the type class *Num* is a set of numeric types; includes *Integer*, *Float*, etc
- now $(+) :: (Num\ a) \Rightarrow (a \to a \to a)$
- *ad hoc polymorphism* (different code for different types), as opposed to *parametric polymorphism* (same code for all types)

2.17 Some standard type classes

- *Eq*: $==$, $\neq$
- *Ord*: $<$ etc, *min* etc
- *Enum*: *succ*, ..
- *Bounded*: *minBound*, *maxBound*
- *Show*: $show :: a \to String$
- *Num*: $+$, $*$ etc
- *Real* (ordered numeric types)
- *Integral*: *div* etc
- *Fractional*: $/$ etc
- *Floating*: *exp* etc
- more later

2.17 Derived type classes

- new **data** types are not automatically instances of useful type classes
- possible to install as instances:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Gender} = \mathit{Female} \mid \mathit{Male}\\
&\mathbf{instance}\ \mathit{Eq}\ \mathit{Gender}\ \mathbf{where}\\
&\quad \mathit{Female} == \mathit{Female} = \mathit{True}\\
&\quad \mathit{Female} == \mathit{Male} \quad\ = \mathit{False}\\
&\quad \mathit{Male} \quad == \mathit{Female} = \mathit{False}\\
&\quad \mathit{Male} \quad == \mathit{Male} \quad\ = \mathit{True}
\end{aligned}
$$

- (default definition of $\neq$ obtained for free from $==$, more later)
- tedious for simple cases, which can be derived automatically:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Gender} = \mathit{Female} \mid \mathit{Male}\\
&\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Enum}, \mathit{Bounded}, \mathit{Show}, \mathit{Read})
\end{aligned}
$$

2.18 Type-driven program development

- types are a vital part of any program
- types are not an afterthought
- first specify the type of a function
- its definition is then driven by the type

$$f :: T \to U$$

- f consumes a T value: dictates case analysis
- f produces a U value: dicates use of constructors
- type safety and flexibility are in tension
- polymorphism partially releases the tension

Part 3

Lists

3.0 Outline

3.1 List notation

- lists are central to functional programming (cf LISP!)
- sequences of elements of the same type
- enclosed in square brackets, comma-separated: $[1,2,3]$, $[\,]$
- the type of lists with elements of type a is $[a]$
- strings are just lists of characters: `['H','e','l','l','o']`

  $$\mathbf{type}\ \mathit{String} = [\mathit{Char}]$$

  but with special syntax `"Hello"`
- list elements can be any type:

  $$\begin{array}{ll} [1,2,3] & :: [\mathit{Integer}] \\ [[1,2],[\,],[3]] & :: [[\mathit{Integer}]] \\ [(+),(*)] & :: [\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}] \end{array}$$

3.2 Some library functions

- exploring the library *Data.List*

  import *Data.List*

- $concat :: [[a]] \to [a]$ eg $concat\ [[1,2],[\,],[3]] = [1,2,3]$
- $length :: [a] \to Int$ eg $length\ [1,2,3] = 3$
- $reverse :: [a] \to [a]$ eg `reverse "ralf" = "flar"`
- $map :: (a \to b) \to ([a] \to [b])$ eg $map\ (+1)\ [1,2,3] = [2,3,4]$
- $lines :: String \to [String]$ eg
  `lines "a\nbc\nd" = ["a","bc","d"]`
- $unlines :: [String] \to String$ eg
  `unlines ["a","bc","d"] = "a\nbc\nd\n"`
- $tails :: [a] \to [[a]]$ eg
  `tails "ralf" = ["ralf","alf","lf","f",""]`

3.2 How to solve it?

- write down the type (what's the input?, what's the output?)
- can you solve it using existing vocabulary?
- use function application and function composition
- some exercises: given a string (a list of characters)
  - remove newlines
  - count the number of lines
  - flip text upside down
  - flip text from left to right
  - determine the list of all substrings

3.2 Solutions

- remove newlines

$$unwrap :: String \to String$$
$$unwrap = concat \circ lines$$

- count the number of lines

$$countLines :: String \to Int$$
$$countLines = length \circ lines$$

- flip text upside down

$$upsideDown :: String \to String$$
$$upsideDown = unlines \circ reverse \circ lines$$

- flip text from left to right

$$leftRight :: String \to String$$
$$leftRight = unlines \circ map\ reverse \circ lines$$

3.2 Solutions continued

- determine the list of all prefixes (actually, also defined in the library: *inits*)

$$
\begin{aligned}
&\mathit{suffixes}, \mathit{prefixes} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{suffixes} = \mathit{tails} \\
&\mathit{prefixes} = \mathit{map}\ \mathit{reverse} \circ \mathit{tails} \circ \mathit{reverse}
\end{aligned}
$$

- determine the list of all substrings

$$
\begin{aligned}
&\mathit{substrings} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{substrings} = \mathit{concat} \circ \mathit{map}\ \mathit{prefixes} \circ \mathit{suffixes}
\end{aligned}
$$

3.3 List constructors

- a list is either
  - empty, written $[\,]$
  - or consists of an element x followed by a list xs, written $x : xs$
- every finite list can be built up from $[\,]$ using $:$
- eg $[1, 2, 3] = 1 : (2 : (3 : [\,])) = 1 : 2 : 3 : [\,]$
- $[\,]$ and $:$ are called *constructors*

3.3 Type of list constructors

- *nil*: the empty list

$$[\,] :: [a]$$

- *cons*: function for prefixing an element onto a list

$$(:) :: a \to [a] \to [a]$$

- $[\,]$ and $:$ are polymorphic
- puzzle: is $[\,] : [\,]$ well-typed? what about $[\,] : ([\,] : [\,])$ and $([\,] : [\,]) : [\,]$?

3.4 Pattern matching

- constructors are exhaustive
- to define function over lists, it suffices to consider the two cases $[\,]$ and $:$
- eg to test if list is empty

$$
\begin{aligned}
&\mathit{null} :: [\,a\,] \rightarrow \mathit{Bool}\\
&\mathit{null}\ [\,] \qquad = \mathit{True}\\
&\mathit{null}\ (\_ : \_) = \mathit{False}
\end{aligned}
$$

  (why is this different from $(==[\,])$?)
- eg to return first element of non-empty list

$$
\begin{aligned}
&\mathit{head} :: [\,a\,] \rightarrow a\\
&\mathit{head}\ (x : \_) = x
\end{aligned}
$$

3.4 Case analysis

- cases can also be analysed using a **case**-expression

$$
\begin{aligned}
&\mathit{null} :: [\,a\,] \to \mathit{Bool} \\
&\mathit{null}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad\qquad [\,] \qquad\ \to \mathit{True} \\
&\qquad\qquad (\_ : \_) \to \mathit{False}
\end{aligned}
$$

- *declaration style*: equation using patterns; *expression style*: **case**-expression using patterns

3.4 Recursive definitions

- definitions by pattern-matching can be recursive too
- natural as the type is also recursively defined
- eg sum of a list of integers

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

- eg length of a list of elements

$$
\begin{aligned}
&\mathit{length} :: [\,a\,] \to \mathit{Int} \\
&\mathit{length}\ [\,] \qquad\quad = 0 \\
&\mathit{length}\ (\_ : xs) = 1 + \mathit{length}\ xs
\end{aligned}
$$

3.4 List design pattern

- remember: every type comes with a design pattern
- *task:* define a function $f :: [P] \to S$
- *step 1:* solve the problem for the empty list

$$f\,[\,] = \ldots$$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for xs at hand;
  extend the intermediate solution to a solution for $x : xs$

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

  you have to program only a *step*
- put on your problem-solving glasses

3.5 Some list operations

- append: $[1,2,3] \mathbin{+\!\!+} [4,5] = [1,2,3,4,5]$

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

- concatenation: $concat\ [[1,2],[\,],[3]] = [1,2,3]$

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

- reverse: $reverse\ [1,2,3] = [3,2,1]$

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

(exercise: complexity? improve!)

- is a list ordered?

$$
\begin{aligned}
&\mathit{ordered} :: (\mathit{Ord}\ a) \Rightarrow [a] \to \mathit{Bool}\\
&\mathit{ordered}\ [\,] &&= \mathit{True}\\
&\mathit{ordered}\ [x] &&= \mathit{True}\\
&\mathit{ordered}\ (x1 : x2 : xs) &&= x1 \leqslant x2 \mathbin{\&\&} \mathit{ordered}\ (x2 : xs)
\end{aligned}
$$

- we distinguish three cases
- zip: eg $\mathit{zip}\ [1, 2, 3]\ \texttt{"ab"} = [(1, \texttt{'a'}), (2, \texttt{'b'})]$

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

- we pattern match on both arguments

3.6 List comprehensions

- two useful operators on lists: *map* and *filter*
- list comprehensions provide a convenient syntax for expressions involving *map*, *filter*, *concat*
- analogous to a database query language
- useful for constructing new lists from old lists

3.6 Map

- applies given function to every element of given list
- eg $\mathit{map}\ \mathit{square}\ [1,2,3] = [1,4,9]$
- eg $\mathit{map}\ \mathit{succ}$ `"HAL"` $=$ `"IBM"`
- definition

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to ([a] \to [b]) \\
&\mathit{map}\ \_\ [\,] \qquad\quad = [\,] \\
&\mathit{map}\ f\ (x : xs) = f\ x : \mathit{map}\ f\ xs
\end{aligned}
$$

- another eg: $\mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1\,..\,10])$
- (special syntax $[m\,..\,n]$ for enumerations)

3.6 Filter

- returns sublist of the argument whose elements satisfy given predicate
- eg *filter isDigit* `"more4u2say"` = `"42"`
- eg $(\mathit{sum} \circ \mathit{map}\ \mathit{square} \circ \mathit{filter}\ \mathit{odd})\ [1\,..\,5] = 35$
- definition

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

3.6 Comprehensions

- special convenient syntax for list-generating expressions
- eg $\mathit{sum}\,[\mathit{square}\;x \mid x \leftarrow [1\,..\,5], \mathit{odd}\;x]$
- formally, a comprehension $[e \mid \mathit{Qs}]$ for expression e and non-empty comma-separated sequence of qualifiers Qs
- qualifier may be *generator* (of the form $x \leftarrow \mathit{xs}$) or *guard* (a boolean expression)

3.6 Examples of comprehensions

- eg primes up to a given bound

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

- eg database query

$$
\begin{aligned}
&\mathit{overdue} = \\
&\quad [\,(\mathit{nm}, \mathit{ad}) \mid (\mathit{id}, \mathit{nm}, \mathit{ad}) \leftarrow \mathit{names}, \\
&\qquad\qquad\qquad (\mathit{id}', \mathit{dt}, \_) \leftarrow \mathit{invoices}, \mathit{id} == \mathit{id}', \\
&\qquad\qquad\qquad \mathit{dt} < \mathit{today}\,]
\end{aligned}
$$

- eg Quicksort

$$
\begin{aligned}
&\mathit{quicksort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\
&\mathit{quicksort}\ [\,] = [\,] \\
&\mathit{quicksort}\ (x : xs) = \mathit{quicksort}\ [\, y \mid y \leftarrow xs, y < x\,] \\
&\qquad\qquad +\!\!+\ [x]\ +\!\!+ \\
&\qquad\qquad \mathit{quicksort}\ [\, y \mid y \leftarrow xs, y \geqslant x\,]
\end{aligned}
$$

3.6 Another point of view

- list comprehension is 'really' a form of nested loop
- eg $[\, f\ b \mid a \leftarrow x, b \leftarrow g\ a, p\ b \,]$ is related to

```
foreach a in x do
  foreach b in g a do
    if p b then
      yield f b
```

3.6 Advanced: semantics by translation

- generator iterates over list, binding new variable

$$[\, e \mid x \leftarrow xs, Qs \,] = concat\ (map\ (\lambda x \rightarrow [\, e \mid Qs \,])\ xs)$$

- guard prunes collection

$$[\, e \mid p, Qs \,] = \mathbf{if}\ p\ \mathbf{then}\ [\, e \mid Qs \,]\ \mathbf{else}\ [\,]$$

- empty qualifier list generates a singleton

$$[\, e \mid \,] = [\, e \,]$$

- eg

$$\begin{aligned}
& [\, x * x \mid x \leftarrow [\, 1 \,..\, 5 \,], odd\ x \,] \\
= \quad & concat\ (map\ (\lambda x \rightarrow [\, x * x \mid odd\ x \,])\ [\, 1 \,..\, 5 \,]) \\
= \quad & concat\ (map\ (\lambda x \rightarrow \mathbf{if}\ odd\ x\ \mathbf{then}\ [\, x * x \,]\ \mathbf{else}\ [\,])\ [\, 1 \,..\, 5 \,]) \\
= \quad & concat\ [\, [\, 1 * 1 \,], [\,], [\, 3 * 3 \,], [\,], [\, 5 * 5 \,] \,] \\
= \quad & [\, 1, 9, 25 \,]
\end{aligned}$$

3.7 Case study: Google's map-reduce

- let's explore Google's map-reduce API
- *idea:* do something uniform across a huge collection of data (in parallel) and then combine the results
- if we use lists to model huge collections of data, then the first step is simply an application of *map*
- it remains to define a reduction: collapsing a list of values into a single value

3.7 Reduction

- example: $reduce\ 0\ (+)\ [\,] = 0$,
  $reduce\ 0\ (+)\ [4, 7, 1, 1] = 4 + 7 + 1 + 1$
- definition

$$
\begin{aligned}
&reduce :: m \to (m \to m \to m) \to ([m] \to m)\\
&reduce\ \epsilon\ (\otimes) = crush\\
&\quad \textbf{where}\ crush\ [\,] = \epsilon\\
&\qquad\qquad crush\ (x : xs) = x \otimes crush\ xs
\end{aligned}
$$

- assumption: ϵ and $\otimes$ form a monoid ie $\otimes$ is associative with ϵ as its neutral element (why?)
- *reduce* is another higher-order function (more later)

3.7 Applications of reduce

- numbers
  - $\mathit{reduce}\ 0\ (+)$
  - $\mathit{reduce}\ 1\ (*)$
  - $\mathit{reduce}\ \mathit{maxBound}\ \mathit{min}$
  - $\mathit{reduce}\ \mathit{minBound}\ \mathit{max}$
- Booleans
  - $\mathit{reduce}\ \mathit{True}\ (\wedge)$
  - $\mathit{reduce}\ \mathit{False}\ (\vee)$
  - $\mathit{reduce}\ \mathit{True}\ (==)$
  - $\mathit{reduce}\ \mathit{False}\ (\neq)$
- $\mathit{reduce}\ [\,]\ (+\!\!+)$
- $\mathit{reduce}\ \mathit{id}\ (\circ)$

3.7 Map-reduce

- map-reduce simply combines *map* with *reduce*

$$mapReduce :: m \to (m \to m \to m) \to (a \to m) \to ([a] \to m)$$
$$mapReduce\ \epsilon\ (\otimes)\ f = reduce\ \epsilon\ (\otimes) \circ map\ f$$

- the art of map-reduce is to find a suitable monoid!

3.7 Applications of map-reduce

- exact search eg
  $member$ `"lisa" ["anja","lisa","flo","ralf"]` $= True$

$$member :: String \to ([String] \to Bool)$$
$$member\ s = mapReduce\ False\ (\vee)\ (==\ s)$$

- substring search eg
  $search$ `"is" ["anja","lisa","flo","ralf"]` $= True$

$$search :: String \to ([String] \to Bool)$$
$$search\ s = member\ s \circ mapReduce\ [\,]\ (+\!\!+)\ substrings$$

- ranking webpages

$$\textbf{type}\ Rank = Int$$

$$best :: String \to ([String] \to Rank)$$
$$best\ s = mapReduce\ minBound\ max\ (rank\ s)$$

$$rank :: String \to String \to Rank \quad \text{-- Google's secret}$$

3.7 Decorating monoids

- of course, we usually want to see the highest-ranked webpage (*best* only returns the rank)
- *idea:* pair the webpages with their rank

$$\textbf{type}\ \mathit{RankedPage} = (\mathit{String}, \mathit{Rank})$$

$$\mathit{best}' :: \mathit{String} \to ([\mathit{String}] \to \mathit{RankedPage})$$
$$\mathit{best}'\ s = \mathit{mapReduce}\ \mathit{minBound}'\ \mathit{max}'\ (\lambda x \to (x, \mathit{rank}\ s\ x))$$

- *minBound'* and *max'* thread the information around

$$\mathit{minBound}' :: \mathit{RankedPage}$$
$$\mathit{minBound}' = (\texttt{"<<not found>>"}, \mathit{minBound})$$

$$\mathit{max}' :: \mathit{RankedPage} \to \mathit{RankedPage} \to \mathit{RankedPage}$$
$$\begin{aligned}&\mathit{max}'\ (s, m)\ (t, n)\\ &\quad \mid m \geqslant n \quad\;\;\, = (s, m)\\ &\quad \mid \mathit{otherwise} = (t, n)\end{aligned}$$

3.8 Summary: How to solve it?

- write down the type (what's the input?, what's the output?)
- can you solve the problem using existing vocabulary?
- if not, define new vocabulary
- use the list design pattern
- remember: you only have to solve a step
- can you solve the step using existing vocabulary?
- if not, define new vocabulary (identify a subproblem)
- solve the subproblem in the same manner

Part 4

Algebraic datatypes

4.0 Outline

4.1 New datatypes

- we've seen **type** synonyms for existing types
- we've also seen enumerations as new **data** types
- **data** is *much* more general than this
- product and sum datatypes
- polymorphic datatypes
- recursive datatypes

4.2 Product and sum datatypes

- constructors of enumerated types are constants (Mon); constructors may be functions too
- eg people with names and ages

$$\begin{array}{l}\textbf{type}\ \mathit{Name} = \mathit{String}\\ \textbf{type}\ \mathit{Age} = \mathit{Int}\\ \textbf{data}\ \mathit{Person} = \mathit{P}\ \mathit{Name}\ \mathit{Age}\end{array}$$

- then $P :: \mathit{Name} \to \mathit{Age} \to \mathit{Person}$
- such *constructor functions* do not simplify, they are in normal form; moreover, they can be used in pattern-matching

$$\begin{array}{l}\mathit{showPerson} :: \mathit{Person} \to \mathit{String}\\ \mathit{showPerson}\ (P\ n\ a) = \texttt{"Name: "} \mathbin{+\!\!+} n \mathbin{+\!\!+} \texttt{", Age: "} \mathbin{+\!\!+} \mathit{show}\ a\end{array}$$

- safer than type synonyms, and can have their own type classes (eg specialized equality)

$$\textbf{type}\ \mathit{Person} = (\mathit{Name}, \mathit{Age})$$

4.2 Sum types

- datatypes can have multiple *variants*

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Suit} &&= \mathit{Spades} \mid \mathit{Hearts} \mid \mathit{Diamonds} \mid \mathit{Clubs} \\
&\mathbf{data}\ \mathit{Rank} &&= \mathit{Faceless}\ \mathit{Integer} \mid \mathit{Jack} \mid \mathit{Queen} \mid \mathit{King} \\
&\mathbf{data}\ \mathit{Card} &&= \mathit{Card}\ \mathit{Rank}\ \mathit{Suit} \mid \mathit{Joker}
\end{aligned}
$$

- so a *Rank* is *either* of the form *Faceless n* for some *n*, *or* a constant *Jack*, *Queen* or *King*

4.2 Temperatures

- another example

$$\textbf{data}\ \mathit{Temp} = \mathit{Cels}\ \mathit{Float} \mid \mathit{Fahr}\ \mathit{Float}$$
$$\quad\textbf{deriving}\ (\mathit{Show})$$

- define our own equality function

$$\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Temp}\ \textbf{where} \\
\quad \mathit{Cels}\ x == \mathit{Cels}\ y = x == y \\
\quad \mathit{Fahr}\ x == \mathit{Fahr}\ y = x == y \\
\quad \mathit{Cels}\ x == \mathit{Fahr}\ y = x * 1.8 == y - 32.0 \\
\quad \mathit{Fahr}\ x == \mathit{Cels}\ y = \mathit{Cels}\ y == \mathit{Fahr}\ x
\end{array}$$

4.3 Parametric datatypes

- constructors may be polymorphic functions
- then datatype is parametric

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

- eg $\mathit{Just}\ 13 :: \mathit{Maybe\ Int}$
- so $\mathit{Just} :: a \to \mathit{Maybe}\ a$, $\mathit{Nothing} :: \mathit{Maybe}\ a$
- useful for modelling exceptions

$$\begin{aligned} &\mathit{head'} :: [a] \to \mathit{Maybe}\ a \\ &\mathit{head'}\ [\,] \qquad = \mathit{Nothing} \\ &\mathit{head'}\ (x : \_) = \mathit{Just}\ x \end{aligned}$$

- similarly, sum datatype

$$\mathbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

4.4 Recursive datatypes

- datatypes may be recursive too
- arithmetic expressions
- natural numbers
- lists
- binary trees
- general trees

4.4 Arithmetic expressions

- datatype of arithmetic expressions

```
data Expr = Lit Integer | Add Expr Expr | Mul Expr Expr
```

- an arithmetic expressions is either a literal, or two expressions added together, or two multiplied
- constructor names may be operators (starting with ':')

```
infixl 7 :*:
infixl 6 :+:
data Expr
  = Lit Integer      -- a literal
  | Expr :+: Expr    -- addition
  | Expr :*: Expr    -- multiplication
  deriving (Show)
```

4.4 Constructing expressions

- constructing expressions

$$
\begin{aligned}
&\mathit{expr1}, \mathit{expr2} :: \mathit{Expr}\\
&\mathit{expr1} = (\mathit{Lit}\ 4 :\!*\!: \mathit{Lit}\ 7) :\!+\!: (\mathit{Lit}\ 11)\\
&\mathit{expr2} = (\mathit{Lit}\ 4 :\!+\!: \mathit{Lit}\ 7) :\!*\!: (\mathit{Lit}\ 11)
\end{aligned}
$$

- note the difference between *syntax*

$$
\begin{aligned}
&?\ \mathit{Lit}\ 4 :\!+\!: \mathit{Lit}\ 7 :\!*\!: \mathit{Lit}\ 11\\
&\mathit{Lit}\ 4 :\!+\!: \mathit{Lit}\ 7 :\!*\!: \mathit{Lit}\ 11
\end{aligned}
$$

- and *semantics*

$$
\begin{aligned}
&?\ 4 + 7 * 11\\
&81
\end{aligned}
$$

4.4 *Expr* design pattern

- recursive definitions by pattern-matching

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

- the evaluator essentially replaces syntax ($:+:$ and $:*:$) by semantics ($+$ and $*$)

4.4 *Expr* design pattern

- remember: every datatype comes with a design pattern
- *task:* define a function $f :: Expr \to S$
- *step 1:* solve the problem for literals

$$f\,(Lit\;n) = \;\ldots$$

- *step 2:* solve the problem for addition;
  assume that you already have the solution for x and y at hand;
  extend the intermediate solution to a solution for $x :+: y$

$$\begin{aligned} f\,(Lit\;n) &= \;\ldots \\ f\,(x :+: y) &= \;\ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \end{aligned}$$

  you have to program only a *step*
- *step 2:* do the same for $x :*: y$

$$\begin{aligned} f\,(Lit\;n) &= \;\ldots \\ f\,(x :+: y) &= \;\ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \\ f\,(x :*: y) &= \;\ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \end{aligned}$$

4.4 Naturals

- *Peano* definition of natural numbers (non-negative integers)

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

- every natural is either *Zero* or the *Succ*essor of a natural
- eg $\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))$ corresponds to 3
- extraction

$$\begin{aligned}
&\mathit{nat2int} :: \mathit{Nat} \to \mathit{Integer} \\
&\mathit{nat2int}\ \mathit{Zero} \quad\quad = 0 \\
&\mathit{nat2int}\ (\mathit{Succ}\ n) = 1 + \mathit{nat2int}\ n
\end{aligned}$$

- addition

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

(does this look familiar?)

4.4 Peano design pattern

- remember: every datatype comes with a design pattern
- *task:* define a function $f :: Nat \to S$
- *step 1:* solve the problem for $Zero$

$$f\ Zero = \ \ldots$$

- *step 2:* solve the problem for $Succ\ n$;
  assume that you already have the solution for n at hand;
  extend the intermediate solution to a solution for $Succ\ n$

$$\begin{aligned} f\ Zero &= \ \ldots \\ f\ (Succ\ n) &= \ \ldots n \ldots f\ n \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses
- (exercise: nth power)

4.4 Lists

- built-in type of lists is not special (has only special syntax)

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

- eg $[1, 2, 3]$ or $1 : 2 : 3 : [\,]$ corresponds to $\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))$
- recursive definitions by pattern-matching

$$\begin{aligned}
&\mathit{mapList} :: (a \to b) \to (\mathit{List}\ a \to \mathit{List}\ b)\\
&\mathit{mapList}\ _\ \mathit{Nil} && = \mathit{Nil}\\
&\mathit{mapList}\ f\ (\mathit{Cons}\ x\ xs) && = \mathit{Cons}\ (f\ x)\ (\mathit{mapList}\ f\ xs)
\end{aligned}$$

4.4 List design pattern

- remember: every datatype comes with a design pattern
- *task:* define a function $f :: List\ P \to S$
- *step 1:* solve the problem for the empty list

$$f\ Nil = \ \ldots$$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for xs at hand;
  extend the intermediate solution to a solution for $Cons\ x\ xs$

$$\begin{aligned} f\ Nil &= \ \ldots \\ f\ (Cons\ x\ xs) &= \ \ldots x \ldots xs \ldots f\ xs \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses

4.4 Binary trees

- externally-labelled binary trees

$$\textbf{data}\ \mathit{Btree}\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Btree}\ a)\ (\mathit{Btree}\ a)$$

- eg $\mathit{Bin}\ (\mathit{Tip}\ 1)\ (\mathit{Bin}\ (\mathit{Tip}\ 2)\ (\mathit{Tip}\ 3))$
- eg size (number of elements)

$$\begin{aligned} &\mathit{size} :: \mathit{Btree}\ a \to \mathit{Int} \\ &\mathit{size}\ (\mathit{Tip}\ \_) \quad = 1 \\ &\mathit{size}\ (\mathit{Bin}\ t\ u) = \mathit{size}\ t + \mathit{size}\ u \end{aligned}$$

4.4 General trees

- internally-labelled trees with arbitrary branching (*rose trees*)

$$\textbf{data}\ \mathit{Gtree}\ a = \mathit{Branch}\ a\ [\mathit{Gtree}\ a]$$

- eg
  $\mathit{Branch}\ 1\ [\mathit{Branch}\ 2\ [\,], \mathit{Branch}\ 3\ [\mathit{Branch}\ 4\ [\,]], \mathit{Branch}\ 5\ [\,]]$
- eg given available moves $m :: \mathit{Pos} \to [\mathit{Pos}]$, generate game tree

$$\begin{aligned} &\mathit{gametree} :: (\mathit{Pos} \to [\mathit{Pos}]) \to (\mathit{Pos} \to \mathit{Gtree}\ \mathit{Pos}) \\ &\mathit{gametree}\ m\ p = \mathit{Branch}\ p\ (\mathit{map}\ (\mathit{gametree}\ m)\ (m\ p)) \end{aligned}$$

4.5 Case study: compiler construction

- let's implement a compiler that translates arithmetic expressions into stack machine code and
- a virtual machine that executes stack machine code

$$\begin{aligned}&\mathit{compile}\ (\mathit{Lit}\ 4 :\!*\!: (\mathit{Lit}\ 7 :\!+\!: \mathit{Lit}\ 11))\\ &\quad = \mathit{Push}\ 4 :\hat{}: \mathit{Push}\ 7 :\hat{}: \mathit{Push}\ 11 :\hat{}: \mathit{Add} :\hat{}: \mathit{Mul}\end{aligned}$$

- when executed, the stack grows and shrinks

$$\begin{aligned}&[\,]\\ &4 : [\,]\\ &7 : 4 : [\,]\\ &11 : 7 : 4 : [\,]\\ &18 : 4 : [\,]\\ &22 : [\,]\end{aligned}$$

- we also show the correctness of compiler and VM

4.5 Warm-up: showing expressions

- *showExpr* maps an expression to its string representation

$$
\begin{aligned}
&\mathit{showExpr} :: \mathit{Expr} \to \mathit{String} \\
&\mathit{showExpr}\ (\mathit{Lit}\ i) \\
&\quad = \mathit{show}\ i \\
&\mathit{showExpr}\ (\mathit{e1} :\!+\!: \mathit{e2}) \\
&\quad = \texttt{"("} +\!\!+ \mathit{showExpr}\ \mathit{e1} +\!\!+ \texttt{" + "} +\!\!+ \mathit{showExpr}\ \mathit{e2} +\!\!+ \texttt{")"} \\
&\mathit{showExpr}\ (\mathit{e1} :\!*\!: \mathit{e2}) \\
&\quad = \texttt{"("} +\!\!+ \mathit{showExpr}\ \mathit{e1} +\!\!+ \texttt{" * "} +\!\!+ \mathit{showExpr}\ \mathit{e2} +\!\!+ \texttt{")"}
\end{aligned}
$$

- parentheses is necessary for products of sums eg
  showExpr expr2 = `"((4 + 7) * 11)"`
- some parentheses is redundant, however, eg
  showExpr expr1 = `"((4 * 7) + 11)"`

4.5 Respecting precedence

- string representation should respect precedence
- *idea:* pass in the precedence level of the enclosing operator

```
showPrec :: Int → Expr → String
showPrec _ (Lit i)
  = show i
showPrec p (e1 :+: e2)
  = parenthesis (p > 6) (showPrec 6 e1 ++ " + " ++ showPrec 6 e2)
showPrec p (e1 :*: e2)
  = parenthesis (p > 7) (showPrec 7 e1 ++ " * " ++ showPrec 7 e2)
parenthesis :: Bool → String → String
parenthesis True s = "(" ++ s ++ ")"
parenthesis False s = s
```

- eg *showPrec* 0 *expr1* = `"4 * 7 + 11"` and
  showPrec 0 *expr2* = `"(4 + 7) * 11"`

4.5 Instructions of a stack machine

- the operations of the VM operate on a stack

```
infixr 2 :^:
data Code
   = Push Integer       -- push integer onto stack
   | Add                -- add topmost two elements and push result
   | Mul                -- multiply
   | Code :^: Code      -- sequencing
  deriving (Show)
```

- eg

```
code1 :: Code
code1 = Push 47 :^: Push 11 :^: Add
```

4.5 Warm-up: showing code

- *showCode* maps a piece of code to its string representation

$$
\begin{aligned}
&\mathit{showCode} :: \mathit{Code} \to \mathit{String}\\
&\mathit{showCode}\ (\mathit{Push}\ i) &&= \texttt{"push "} +\!\!+\ \mathit{show}\ i\\
&\mathit{showCode}\ (\mathit{Add}) &&= \texttt{"add"}\\
&\mathit{showCode}\ (\mathit{Mul}) &&= \texttt{"mul"}\\
&\mathit{showCode}\ (c1 :\hat{}: c2) &&= \mathit{showCode}\ c1 +\!\!+\ \texttt{" ; "} +\!\!+\ \mathit{showCode}\ c2
\end{aligned}
$$

- eg $\mathit{showCode}\ \mathit{code1} = \texttt{"push 47 ; push 11 ; add"}$

4.5 Compilation

- the definition of the compiler follows the *Expr* design pattern

$$
\begin{aligned}
&\mathit{compile} :: \mathit{Expr} \to \mathit{Code}\\
&\mathit{compile}\ (\mathit{Lit}\ i) &&= \mathit{Push}\ i\\
&\mathit{compile}\ (e1 :+: e2) &&= \mathit{compile}\ e1 :\hat{}: \mathit{compile}\ e2 :\hat{}: \mathit{Add}\\
&\mathit{compile}\ (e1 :*: e2) &&= \mathit{compile}\ e1 :\hat{}: \mathit{compile}\ e2 :\hat{}: \mathit{Mul}
\end{aligned}
$$

- for addition we first generate code for the two subexpressions and then emit an *Add* instruction
- eg $\mathit{compile}\ \mathit{expr1} = \mathit{Push}\ 4 :\hat{}: \mathit{Push}\ 7 :\hat{}: \mathit{Mul} :\hat{}: \mathit{Push}\ 11 :\hat{}: \mathit{Add}$

4.5 Execution

- we implement a stack using a list of integers

$$\textbf{type}\ \mathit{Stack} = [\,\mathit{Integer}\,]$$

- the definition of the VM follows the *Code* design pattern

$$
\begin{aligned}
&\mathit{execute} :: \mathit{Code} \to (\mathit{Stack} \to \mathit{Stack})\\
&\mathit{execute}\ (\mathit{Push}\ i) &&= \mathit{push}\ i\\
&\mathit{execute}\ (\mathit{Add}) &&= \mathit{add}\\
&\mathit{execute}\ (\mathit{Mul}) &&= \mathit{mul}\\
&\mathit{execute}\ (c1 :\hat{}: c2) &&= \mathit{execute}\ c2 \circ \mathit{execute}\ c1
\end{aligned}
$$

- syntax (*Push*) is replaced by semantics (*push*)

4.5 Helper functions

- *push* etc are *stack transformers*

$$
\begin{aligned}
&push :: Integer \to (Stack \to Stack)\\
&push\ i\ xs = i : xs\\
&add\ :: Stack \to Stack\\
&add\ [\,] \qquad\qquad\ = error\ msg\\
&add\ [\_] \qquad\qquad = error\ msg\\
&add\ (x1 : x2 : xs) = x2 + x1 : xs\\
&mul\ :: Stack \to Stack\\
&mul\ [\,] \qquad\qquad\ = error\ msg\\
&mul\ [\_] \qquad\qquad = error\ msg\\
&mul\ (x1 : x2 : xs) = x2 * x1 : xs\\
&msg :: String\\
&msg = \texttt{"VM: empty stack"}
\end{aligned}
$$

4.5 Advanced: Proof of correctness

Evaluating a compiled expression has the same effect as evaluating the expression and then pushing the result:

$$push\ (evaluate\ e) = execute\ (compile\ e)$$

The proof proceeds by induction over the structure of the expression e.

4.5 Proof of correctness: base case

Case $e = Lit\ i$:

$$
\begin{aligned}
& push\ (evaluate\ (Lit\ i)) \\
= \quad & \{\ \text{definition of } evaluate\ \} \\
& push\ i \\
= \quad & \{\ \text{definition of } execute\ \} \\
& execute\ (Push\ i) \\
= \quad & \{\ \text{definition of } compile\ \} \\
& execute\ (compile\ (Lit\ i))
\end{aligned}
$$

4.5 Proof of correctness: inductive step

Case $e = e1 :\!+\!: e2$:

$$
\begin{aligned}
& \mathit{push}\ (\mathit{evaluate}\ (e1 :\!+\!: e2)) \\
= \quad & \{ \text{ definition of } \mathit{evaluate} \ \} \\
& \mathit{push}\ (\mathit{evaluate}\ e1 + \mathit{evaluate}\ e2) \\
= \quad & \{ \text{ property of add: } \mathit{add} \circ \mathit{push}\ n \circ \mathit{push}\ m = \mathit{push}\ (m + n) \ \} \\
& \mathit{add} \circ \mathit{push}\ (\mathit{evaluate}\ e2) \circ \mathit{push}\ (\mathit{evaluate}\ e1) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& \mathit{add} \circ \mathit{execute}\ (\mathit{compile}\ e2) \circ \mathit{execute}\ (\mathit{compile}\ e1) \\
= \quad & \{ \text{ definition of execute } \} \\
& \mathit{execute}\ (\mathit{compile}\ e1 :\hat{}: \mathit{compile}\ e2 :\hat{}: \mathit{Add}) \\
= \quad & \{ \text{ definition of compile } \} \\
& \mathit{execute}\ (\mathit{compile}\ (e1 :\!+\!: e2))
\end{aligned}
$$

Likewise for $e1 :\!*\!: e2$.

4.6 The art of functional programming

- model static aspects of the real world using datatypes
- model dynamic aspects using functions
- don’t shy away from introducing new types

Part 5

Higher-order programming

5.0 Outline

5.1 Functions as first-class citizens

- *functional programming* concerns functions (of course!)
- functions are first-class citizens of the language
- functions have all the rights of other types:
  - may be passed as arguments
  - may be returned as results
  - may be stored in data structures
  - etc
- functions that manipulate functions are *higher order*

Slogan: higher-order functions allow new and better means of modularizing programs

5.2 Functions as arguments

- we have already seen many examples of higher-order operators encapsulating patterns of computation: *map*, *filter*, *reduce*
- each is a parameterizable program scheme
- parameterization improves modularity, and hence understanding, modification and reuse

5.3 Functions as results

- functions may also be returned as results

$$\mathit{addOrMul} :: \mathit{Bool} \to (\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer})$$
$$\mathit{addOrMul}\ b = \mathbf{if}\ b\ \mathbf{then}\ (+)\ \mathbf{else}\ (*)$$

- partial application
- currying
- function composition (again)

5.3 Partial application

- consider $\mathit{add'}\ x\ y = x + y$
- type $\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}$; takes two $\mathit{Integer}$s and returns an $\mathit{Integer}$ (eg $\mathit{add'}\ 3\ 4 = 7$)
- another view: type $\mathit{Integer} \to (\mathit{Integer} \to \mathit{Integer})$ (remember, $\to$ associates to the right); takes a single $\mathit{Integer}$ and returns an $\mathit{Integer} \to \mathit{Integer}$ function (eg $\mathit{add'}\ 3$ is the $\mathit{Integer}$-transformer that adds three)
- need not apply function to all its arguments at once: *partial application*; result will then be a function, awaiting remaining arguments
- in fact, partial evaluation is the norm; every function takes exactly one argument
- sectioning ($(3+)$, $(+)$) is partial application of binary ops

5.3 Currying

- a function taking pair of arguments can be transformed into a function taking two successive arguments, and vice versa

$$
\begin{aligned}
&\mathit{add} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\
&\mathit{add}\ (x, y) = x + y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{add'} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{add'}\ x\ y = x + y
\end{aligned}
$$

- *add'* is called the *curried* version of *add*
- named after logician Haskell B. Curry (like the language), though actually due to Schönfinkel
- thus, pair-consuming functions are unnecessary

- transformations are implementable as higher-order operations

$$
\begin{aligned}
&\mathit{curry} :: ((a, b) \to c) \to (a \to b \to c)\\
&\mathit{curry}\ f\ a\ b = f\ (a, b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{uncurry} :: (a \to b \to c) \to ((a, b) \to c)\\
&\mathit{uncurry}\ f\ (a, b) = f\ a\ b
\end{aligned}
$$

- eg $\mathit{add'} = \mathit{curry}\ \mathit{add}$
- a related higher-order operation: flip arguments of binary function (later: $\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]$)

$$
\begin{aligned}
&\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)\\
&\mathit{flip}\ f\ b\ a = f\ a\ b
\end{aligned}
$$

5.3 Function composition

- recall function composition (now with polymorphic type)

$$(\circ) :: (b \to c) \to (a \to b) \to a \to c$$
$$(f \circ g)\ x = f\ (g\ x)$$

- takes two functions that 'meet in the middle' and an argument to one; returns the result from the other
- equivalently, type $(b \to c) \to (a \to b) \to (a \to c)$
- takes two functions, glues them together to form a third
- *exercise:* show that $\circ$ is associative

5.3 Repeated composition

- double application: eg $\mathit{twice}\ \mathit{square}\ 3 = 81$

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

- generalize: eg $\mathit{iter}\ 4\ (2*)\ 1 = 2 * 2 * 2 * 2 * 1$

$$\begin{aligned}&\mathit{iter} :: \mathit{Integer} \to (a \to a) \to (a \to a)\\&\mathit{iter}\ 0\ \_ = \mathit{id}\\&\mathit{iter}\ n\ f = f \circ \mathit{iter}\ (n - 1)\ f\end{aligned}$$

- more on this in a minute ...

5.4 Functions as datastructures

consider a dictionary (associative array)

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Dict}\ k\ v\\
&\mathit{empty}\ :: \mathit{Dict}\ k\ v\\
&\mathit{insert}\ :: (\mathit{Eq}\ k) \Rightarrow (k, v) \to \mathit{Dict}\ k\ v \to \mathit{Dict}\ k\ v\\
&\mathit{lookup} :: (\mathit{Eq}\ k) \Rightarrow \mathit{Dict}\ k\ v \to k \to v
\end{aligned}
$$

5.4 Implementation as list

$$\textbf{type}\ \mathit{Dict}\ k\ v = [\,(k, v)\,]$$

$$\begin{aligned}
&\mathit{empty} :: \mathit{Dict}\ k\ v\\
&\mathit{empty} = [\,]
\end{aligned}$$

$$\begin{aligned}
&\mathit{insert} :: (\mathit{Eq}\ k) \Rightarrow (k, v) \to \mathit{Dict}\ k\ v \to \mathit{Dict}\ k\ v\\
&\mathit{insert}\ \mathit{kv}\ \mathit{kvs} = \mathit{kv} : \mathit{kvs}
\end{aligned}$$

$$\begin{aligned}
&\mathit{lookup} :: (\mathit{Eq}\ k) \Rightarrow \mathit{Dict}\ k\ v \to k \to v\\
&\mathit{lookup}\ [\,]\ _ = \mathit{error}\ \texttt{"item not present"}\\
&\mathit{lookup}\ ((k, v) : \mathit{kvs})\ k'\\
&\quad \mid k == k' \qquad\quad = v\\
&\quad \mid \mathit{otherwise} = \mathit{lookup}\ \mathit{kvs}\ k'
\end{aligned}$$

5.4 Implementation as function

$$\textbf{type}\ \mathit{Dict}\ k\ v = k \to v$$

$$\begin{aligned}
&\mathit{empty} :: \mathit{Dict}\ k\ v\\
&\mathit{empty}\ _ = \mathit{error}\ \texttt{"item not present"}
\end{aligned}$$

$$\begin{aligned}
&\mathit{insert} :: (\mathit{Eq}\ k) \Rightarrow (k, v) \to \mathit{Dict}\ k\ v \to \mathit{Dict}\ k\ v\\
&\mathit{insert}\ (k, v)\ f\ k'\\
&\quad \mid k == k' \qquad = v\\
&\quad \mid \mathit{otherwise} = f\ k'
\end{aligned}$$

$$\begin{aligned}
&\mathit{lookup} :: (\mathit{Eq}\ k) \Rightarrow \mathit{Dict}\ k\ v \to k \to v\\
&\mathit{lookup}\ f = f
\end{aligned}$$

The dictionary *is* the look-up function.

5.4 Natural numbers as functions

Functions can be used to represent other data structures.
In fact, we've already seen how to represent the natural numbers as functions, via repeated composition.

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

$$\mathit{zero} :: \mathit{Natural}$$
$$\mathit{zero}\ \_ = \mathit{id}$$

$$\mathit{succ} :: \mathit{Natural} \to \mathit{Natural}$$
$$\mathit{succ}\ n\ f = f \circ n\ f$$

The $\forall$ makes explicit that these functions are polymorphic.
These are called *Church numerals*. We could define:

$$\mathit{one}, \mathit{two} :: \mathit{Natural}$$
$$\mathit{one} = \mathit{succ}\ \mathit{zero}$$
$$\mathit{two} = \mathit{succ}\ \mathit{one}$$

Conversion from *Integer* using *iter*; how about back again?

5.5 Fold and unfold

- many recursive definitions on lists share a *pattern* of computation
- capture that pattern as a function (abstraction, conciseness, general properties, familiarity, . . .)
- *map* and *filter* are two common patterns
- folds and unfolds capture many more

5.5 Fold right

- consider following pattern of definition

$$\begin{aligned} h\,[\,] &= e \\ h\,(x : xs) &= x \,`op`\, h\,xs \end{aligned}$$

  (simple variant of list design pattern: xs is only used in the recursive call)
- then

$$h\,(x : (y : (z : [\,]))) = x \,`op`\, (y \,`op`\, (z \,`op`\, e))$$

- h replaces constructors by functions
- capture pattern as *foldr*

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

- difference to *reduce*?

5.5 Examples of fold right

- many examples:

$$
\begin{aligned}
&\mathit{sum} &&= \mathit{foldr}\ (+)\ 0\\
&\mathit{copy} &&= \mathit{foldr}\ (:)\ [\,]\\
&\mathit{length} &&= \mathit{foldr}\ (\lambda x\ n \to 1 + n)\ 0\\
&\mathit{map}\ f &&= \mathit{foldr}\ ((:) \circ f)\ [\,]\\
&\mathit{concat} &&= \mathit{foldr}\ (+\!\!+)\ [\,]\\
&\mathit{reverse} &&= \mathit{foldr}\ \mathit{snoc}\ [\,]\ \mathbf{where}\ \mathit{snoc}\ x\ \mathit{xs} = \mathit{xs} +\!\!+ [x]\\
&\mathit{xs} +\!\!+ \mathit{ys} &&= \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs}
\end{aligned}
$$

- right-to-left computation
- operator may ($+$, $+\!\!+$) or may not ($:$, snoc) be associative

5.5 Sorting

- given

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

- we have

$$
\begin{aligned}
&\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,] \\
&\mathit{insertSort} = \mathit{foldr}\ \mathit{insertList}\ [\,]
\end{aligned}
$$

5.5 Fold left

- not every list function is a *foldr* (eg *drop*)
- even those that are may have better definitions
- eg $decimal\ [1, 2, 3] = 123$
- efficient algorithm using *Horner's Rule*:

$$decimal\ [x, y, z] = 10 * (10 * (10 * 0 + x) + y) + z$$

- left-to-right computation — hence *foldl*

$$foldl\ op\ e\ [x, y, z] = ((e\ \text{`}op\text{`}\ x)\ \text{`}op\text{`}\ y)\ \text{`}op\text{`}\ z$$

- definition

$$\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b\\
&foldl\ _\ \ e\ [\,] \qquad\ = e\\
&foldl\ op\ e\ (x : xs) = foldl\ op\ (e\ \text{`}op\text{`}\ x)\ xs
\end{aligned}$$

5.5 Accumulating parameter

- recall *reverse* program

$$reverse :: [a] \to [a]$$
$$reverse = foldr\ (\lambda x\ xs \to xs \mathbin{+\!\!\!+} [x])\ [\,]$$

- another definition

$$reverse' :: [a] \to [a]$$
$$reverse' = foldl\ (flip\ (:))\ [\,]$$

- (now what is complexity?)
- second argument of *foldl* is an *accumulating parameter*

5.5 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 are *dual* to consumers
- to exhibit the duality we first re-define *foldr*
- a *non-recursive* variant of the list data type

$$\mathbf{data}\ List\ a\ b = Nil \mid Cons\ a\ b$$

- *foldr* reformulated

$$\begin{aligned}
&fold :: (List\ a\ b \to b) \to ([a] \to b)\\
&fold\ inn\ [\,] \quad\quad = inn\ Nil\\
&fold\ inn\ (a : x) = inn\ (Cons\ a\ (fold\ inn\ x))
\end{aligned}$$

5.5 Examples of fold

- summing a list of numbers

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

- *map* can be expressed as an fold

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

5.5 Duality: unfold

- folds consume lists
- *dually*, unfolds generate lists
- common pattern

$$
\begin{aligned}
&\mathit{unfold} :: (b \to \mathit{List}\ a\ b) \to (b \to [\,a\,]) \\
&\mathit{unfold}\ \mathit{out}\ x \\
&\quad = \mathbf{case}\ \mathit{out}\ x\ \mathbf{of} \\
&\qquad \mathit{Nil} \qquad\quad\ \to [\,] \\
&\qquad \mathit{Cons}\ a\ x' \to a : \mathit{unfold}\ \mathit{out}\ x'
\end{aligned}
$$

- *unfold* is *dual* to *fold*
- relation to OO iterators?

5.5 Examples of unfold

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

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

- map can also be expressed as an unfold

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

5.5 Sorting

- given

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

  we have

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

- (exercise: write *insertList* itself as an unfold)

- dually, given

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

we have

$$
\begin{aligned}
&\mathit{selectSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,] \\
&\mathit{selectSort} = \mathit{unfold}\ \mathit{deleteMin}
\end{aligned}
$$

- (exercise: write *deleteMin* itself as a fold)

5.6 Component-oriented and combinator-style programming

- higher-order functions make a good framework for gluing programs together
- *component-oriented programming*: pluggable units of code, software assembly instead of programming
- manifests itself in a functional language as *combinator style* programming, as higher-order functions sometimes called combinators
- eg functional parsers, see Hutton's *Programming in Haskell*
- eg functional graphics, see Hudak's *The Haskell School of Expression*
- eg functional music composition, ditto

5.6 Music

- Hudak's *Haskore* combinators for expressing musical structure
- primitive entities: notes, rests, durations
- transformations (transposition, tempo-scaling)
- combinations (sequential and parallel, looping)
- translation to MIDI
- algorithmic composition

5.6 A datatype for music

```
data Music
  = Note Pitch Dur [NoteAttribute]      -- a note (atomic object)
  | Rest Dur                            -- a rest (atomic object)
  | Music :+: Music                     -- sequential composition
  | Music :=: Music                     -- parallel composition
  | Tempo (Ratio Int) Music             -- scale the tempo
  | Trans  Int Music                    -- transposition
  | Instr  IName Music                  -- instrument label
  | Player PName Music                  -- player label
  | Phrase [PhraseAttribute] Music      -- phrase attributes
  deriving (Show, Eq)
```

$$tequila = tequilaIntro :+: tequilaBody :+: tequilaCoda$$

$$\begin{aligned}
&tequilaIntro = \\
&\quad drumIntro :+: \\
&\quad (drums :=: bass) :+: \\
&\quad (drums :=: bass :=: guitar) :+: \\
&\quad (drums :=: bass :=: guitar :=: brassIntro)
\end{aligned}$$

$$\begin{aligned}
&tequilaBody = \\
&\quad cut\ 32\ (repeatM\ (\\
&\quad\quad twice\ (drums :=: bass :=: guitar) :=: brass))
\end{aligned}$$

$$\begin{aligned}
&tequilaCoda = \\
&\quad drumCoda :=: bassCoda :=: guitarCoda :=: brassCoda
\end{aligned}$$

$$
\begin{aligned}
&\mathit{drumIntro} = \mathit{Instr}\ \texttt{"Drums"}\ (\mathit{cut}\ 4\ (\mathit{repeatM}\ (\\
&\qquad\qquad \mathit{p0}\ \mathit{qn} :+: \mathit{p0}\ \mathit{en1} :+: \mathit{p0}\ \mathit{en2})))\\[1ex]
&\mathit{drums} = \mathit{Instr}\ \texttt{"Drums"}\ (\mathit{drumIntro} :=: \mathit{cut}\ 4\ (\mathit{repeatM}\ (\\
&\qquad\qquad (\mathit{qnr} :+: \mathit{p2}\ \mathit{en1} :+: \mathit{p2}\ \mathit{en2}) :=: \mathit{p3}\ \mathit{hn})))\\[1ex]
&\mathit{drumCoda} = \mathit{Instr}\ \texttt{"Drums"}\ (\mathit{cut}\ 2\ \mathit{drums} :+:\\
&\quad \mathit{line}\ [\\
&\qquad \mathit{chord}\ [\mathit{p1}\ \mathit{qn}, \mathit{p2}\ \mathit{qn}, \mathit{p3}\ \mathit{qn}],\\
&\qquad \mathit{chord}\ [\mathit{p1}\ \mathit{qn}, \mathit{p2}\ \mathit{qn}, \mathit{p3}\ \mathit{qn}],\\
&\qquad \mathit{chord}\ [\mathit{p1}\ \mathit{qn}, \mathit{p2}\ \mathit{qn}, \mathit{p3}\ \mathit{qn}],\\
&\qquad \mathit{chord}\ [\mathit{p1}\ \mathit{qn}, \mathit{p2}\ \mathit{qn}, \mathit{p3}\ \mathit{qn}, \mathit{p4}\ (\mathit{tie}\ \mathit{qn}\ \mathit{wn})]])\\[1ex]
&\mathit{p1}\ d = \mathit{perc}\ \mathit{RideCymbal2}\ \ d\ [\mathit{Volume}\ 50]\\
&\mathit{p2}\ d = \mathit{perc}\ \mathit{AcousticSnare}\ d\ [\mathit{Volume}\ 30]\\
&\mathit{p3}\ d = \mathit{perc}\ \mathit{LowTom}\ \ d\ [\mathit{Volume}\ 50]\\
&\mathit{p4}\ d = \mathit{perc}\ \mathit{SplashCymbal}\ d\ [\mathit{Volume}\ 100]\\
&\mathit{p0}\ d = \mathit{perc}\ \mathit{PedalHiHat}\ \ d\ [\mathit{Volume}\ 50]
\end{aligned}
$$

```
bass = Instr "Fretless Bass" bassline

bassline = cut 4 (repeatM (
              line [g 2 (tie qn en1) [ ],
                    f 3 (tie en2 en1) [ ],
                    c 3 en2 [ ],
                    a 2 qn [ ]]))

bassCoda = Instr "Fretless Bass" (
  cut 2 bassline :+:
  line [g 2 qn [ ], g 2 qn [ ], f 2 qn [ ], g 2 en1 [ ],
        en2r, wnr])
```

```
guitar = Instr "Electric Guitar (jazz)" chordSeq
chordSeq = line [
  g qn, g qn, f (tie qn en1), g (tie en2
  en1), g (tie en2 en1), g en2, f en1, f en2, f en1, f en2,
  g qn, g qn, f (tie qn en1), g (tie en2
  en1), f (tie en2 (tie qn en1)), f en2, f en1, f en2]
  where g = eChord G; f = eChord F

eChord :: PitchClass → Dur → Music
eChord key d
  | pc < pcE   =  Trans (12 + pc − pcE) (chord (eShape d))
  | otherwise  =  Trans (pc − pcE) (chord (eShape d))
  where
    pc = pitchClass key
    pcE = pitchClass E
    eShape dur = [ n o dur [Volume 30]
                 | (n, o) ← [(e, 3), (b, 3), (e, 4)]]
```

```
brass = Instr "Brass Section" brassRiff
brassRiff = line [
  g qn, g en1, f en2, a en1, f (tie en2 en1), g (tie en2
  en1), d (tie en2 (tie hn en1)), d en2,
  g qn, g en1, f en2, a en1, f (tie en2 en1), g (tie en2
  (tie dhn en1)), d en2,
  g qn, g en1, f en2, a en1, f (tie en2 en1), g (tie en2
  en1), d (tie en2 (tie hn en1)), d en2,
  g qn, g en1, f en2, a en1, f (tie en2 en1), d (tie en2
  (tie hn qn)), en1r, d en2]
  where
    g d = Note (G, 4) d [ ]
    f d = Note (F, 4) d [ ]
    a d = Note (A, 4) d [ ]
    d d = Note (D, 4) d [ ]
```

$$
\begin{aligned}
&\mathit{rep} :: (\mathit{Music} \to \mathit{Music}) \to (\mathit{Music} \to \mathit{Music}) \to \mathit{Int} \to \\
&\qquad \mathit{Music} \to \mathit{Music} \\
&\mathit{rep}\ f\ g\ 0\ m = \mathit{Rest}\ 0 \\
&\mathit{rep}\ f\ g\ n\ m = m :=: g\ (\mathit{rep}\ f\ g\ (n - 1)\ (f\ m))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{run} &= \mathit{rep}\ (\mathit{Trans}\ 5)\ (\mathit{delay}\ \mathit{tn})\ 8\ (c\ 4\ \mathit{tn}\ [\,]) \\
\mathit{cascade} &= \mathit{rep}\ (\mathit{Trans}\ 4)\ (\mathit{delay}\ \mathit{en})\ 8\ \mathit{run} \\
\mathit{cascades} &= \mathit{rep}\ \mathit{id}\ (\mathit{delay}\ \mathit{sn})\ 2\ \mathit{cascade} \\
\mathit{t4} &= \mathit{test}\ (\mathit{Instr}\ \texttt{"piano"} \\
&\qquad\qquad (\mathit{cascades} :+: \mathit{revM}\ \mathit{cascades}))
\end{aligned}
$$

$$
\begin{array}{l}
\mathbf{type}\ \mathit{SNote} = [\,(\mathit{AbsPitch}, \mathit{Dur})\,] \\
\mathit{pat4'} :: [\,\mathit{SNote}\,] \\
\mathit{pat4'} = [\,[\,(3, 0.5)\,], [\,(4, 0.25)\,], [\,(0, 0.25)\,], [\,(6, 1.0)\,]\,]
\end{array}
$$

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Cluster} = \mathit{Cl}\ \mathit{SNote}\ [\,\mathit{Cluster}\,] \\
\mathit{sim} :: [\,\mathit{SNote}\,] \to [\,\mathit{Cluster}\,] \\
\mathit{sim}\ \mathit{pat} = \mathit{map}\ \mathit{mkCl}\ \mathit{pat} \\
\quad \mathbf{where}\ \mathit{mkCl}\ \mathit{ns} = \mathit{Cl}\ \mathit{ns}\ (\mathit{map}\ (\mathit{mkCl} \circ \mathit{addmult}\ \mathit{ns})\ \mathit{pat}) \\
\mathit{addmult} = \mathit{zipWith}\ (\lambda(p, d)\ (i, s) \to (p + i, d * s))
\end{array}
$$

$$
\begin{array}{l}
\mathit{simFringe}\ n\ \mathit{pat} = \mathit{fringe}\ n\ (\mathit{Cl}\ [\,(0, 0)\,]\ (\mathit{sim}\ \mathit{pat})) \\
\mathit{fringe}\ 0\ (\mathit{Cl}\ \mathit{note}\ \mathit{cls}) = [\,\mathit{note}\,] \\
\mathit{fringe}\ n\ (\mathit{Cl}\ \mathit{note}\ \mathit{cls}) = \mathit{concat}\ (\mathit{map}\ (\mathit{fringe}\ (n - 1))\ \mathit{cls})
\end{array}
$$

$$
\begin{array}{l}
\mathit{sim4s}\ n = \mathit{l1} :=: \mathit{l2}\ \mathbf{where} \\
\quad \mathit{l1} = \mathit{Instr}\ \texttt{"flute"}\ s \\
\quad \mathit{l2} = \mathit{Instr}\ \texttt{"bass"}\ (\mathit{Trans}\ (-36)\ (\mathit{revM}\ s)) \\
\quad s\ \ = \mathit{Trans}\ 60\ (\mathit{Tempo}\ 2\ (\mathit{simToHask}\ (\mathit{simFringe}\ n\ \mathit{pat4'})))
\end{array}
$$

5.7 Abstraction, abstraction, abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- higher-order functions (HOFs) allow you to capture control structures, in particular, common patterns of recursion

Part 6

Type classes

6.7 Outline

6.8 Haskell's approach to overloading

- inventing names is hard!
- sometimes we wish to use the same name for semantically different, but related functions
  - $+$, $*$ etc: arithmetic operations (*Int*, *Integer*, *Float*, *Double* ...)
  - $==$, $\neq$: equality and inequality (almost any type)
  - *show*, *read*: converting to and fro strings (almost any type)
- we want to *overload* the identifiers
- (put differently, we are too lazy to think of different names)
- Haskell's major innovation: a systematic approach to overloading
- (ad-hoc polymorphism *vs* universal polymorphism)

6.8 The equality type class

- overloaded functions typically come in groups
- a type class declares a group of identifiers as overloaded

$$\begin{array}{l}\textbf{class}\ \mathit{Eq}\ a\ \textbf{where}\\ \quad (==) :: a \to a \to \mathit{Bool}\\ \quad (\neq) :: a \to a \to \mathit{Bool}\end{array}$$

- $==$ and $\neq$ are member functions of the type class Eq (also called methods)
- types of the member functions:

$$\begin{array}{l}(==) :: (\mathit{Eq}\ a) \Rightarrow a \to a \to \mathit{Bool}\\ (\neq) :: (\mathit{Eq}\ a) \Rightarrow a \to a \to \mathit{Bool}\end{array}$$

- *read:* for all types a that are instances of the type class Eq, the method $==$ has type $a \to a \to \mathit{Bool}$
- $(\mathit{Eq}\ a) \Rightarrow$ is a *class context*; it constrains the type variable a

6.8 Overloaded functions

- since $==$ is overloaded, $x == y$ can be ambiguous
- what happens if the compiler can't resolve the ambiguity?
- eg list membership uses equality:

$$
\begin{aligned}
&elem :: (Eq\ a) \Rightarrow a \to [a] \to Bool \\
&elem\ \_\ [\,] \qquad\quad = False \\
&elem\ x\ (y : ys) = x == y \mathrel{||} elem\ x\ ys
\end{aligned}
$$

- *elem* becomes overloaded!
- (most programming languages insist that the problem of ambiguity is resolvable at compile-time)
- the class constraint $(Eq\ a) \Rightarrow$ is like an infectious disease: using $==$ or *elem* means that "the disease spreads"

6.8 Class instances

- instances of type classes have to be declared explicitly

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Gender} = \mathit{Female} \mid \mathit{Male} \\
\mathbf{instance}\ \mathit{Eq}\ \mathit{Gender}\ \mathbf{where} \\
\quad \mathit{Female} == \mathit{Female} = \mathit{True} \\
\quad \mathit{Female} == \mathit{Male} \quad\; = \mathit{False} \\
\quad \mathit{Male} \quad\; == \mathit{Female} = \mathit{False} \\
\quad \mathit{Male} \quad\; == \mathit{Male} \quad\; = \mathit{True} \\
\quad x \neq y \qquad = \mathit{not}\ (x == y)
\end{array}
$$

- the body of the instance declaration specifies how (in-) equality is implemented for elements of type *Gender*

6.8 Default definitions

- equality is typically defined in terms of inequality (or vice versa)

$$
\begin{aligned}
&\textbf{class } \mathit{Eq}\ a \textbf{ where}\\
&\quad (==), (\neq) :: a \to a \to \mathit{Bool}\\
&\quad x \neq y = \ \mathit{not}\ (x == y)\\
&\quad x == y = \ \mathit{not}\ (x \neq y)
\end{aligned}
$$

- *default declarations* allow us to define the boilerplate code once and for all
- in an instance declaration it suffices now to provide either the code for $==$ or the code for $\neq$
- (one has to implement at least one method to break the vicious circle)

6.8 Instances of parametric types

- to define equality on a parametric type, say, *Tree a* we require equality on the element type *a*
- an instance declaration can have a context too

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Leaf}\ x1 & == \mathit{Leaf}\ x2 & = x1 == x2 \\
\mathit{Leaf}\ \_ & == \mathit{Fork}\ \_\ \_ & = \mathit{False} \\
\mathit{Fork}\ \_\ \_ & == \mathit{Leaf}\ \_ & = \mathit{False} \\
\mathit{Fork}\ l1\ r1 & == \mathit{Fork}\ l2\ r2 & = l1 == l2\ \&\&\ r1 == r2
\end{array}
\end{array}
$$

- *read:* if *a* supports equality, then *Tree a* supports equality too
- *exercise:* seven occurrences of ==; which is which?

6.8 Subclasses

- classes can be extended

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\
\quad \begin{array}{ll}
compare & :: a \to a \to Ordering \\
(<), (\leqslant), (\geqslant), (>) & :: a \to a \to Bool \\
max, min & :: a \to a \to a
\end{array}
\end{array}
$$

- *Ord* is a *subclass* of *Eq*
- conversely, *Eq* is a *superclass* of *Ord*
- subclasses keep class contexts manageable
- necessary if method of superclass is used in one of the default methods (see next slide)

6.8 Ordering

$$\mathbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}$$

$$\begin{array}{l}
\mathbf{class}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Ord}\ a\ \mathbf{where} \\
\quad \begin{array}{ll}
\mathit{compare} & :: a \to a \to \mathit{Ordering} \\
(<), (\leqslant), (\geqslant), (>) & :: a \to a \to \mathit{Bool} \\
\mathit{max}, \mathit{min} & :: a \to a \to a
\end{array} \\
\quad \begin{array}{lll}
\mathit{compare}\ x\ y & \mid x == y & = \mathit{EQ} \\
 & \mid x \leqslant y & = \mathit{LT} \\
 & \mid \mathit{otherwise} & = \mathit{GT}
\end{array} \\
\quad \begin{array}{l}
x \leqslant y = \mathit{compare}\ x\ y \not= \mathit{GT} \\
x < y = \mathit{compare}\ x\ y == \mathit{LT} \\
x \geqslant y = \mathit{compare}\ x\ y \not= \mathit{LT} \\
x > y = \mathit{compare}\ x\ y == \mathit{GT}
\end{array} \\
\quad \begin{array}{lll}
\mathit{max}\ x\ y & \mid x \leqslant y & = y \\
 & \mid \mathit{otherwise} & = x \\
\mathit{min}\ x\ y & \mid x \leqslant y & = x \\
 & \mid \mathit{otherwise} & = y
\end{array}
\end{array}$$

6.8 Bounded

- instances of *Ord* have to implement a *total* order
- occasionally, a type has a least and a greatest element with respect to that ordering

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

- the type *Int* of machine integers is bounded, the type *Integer* of mathematical integers isn't

```
? maxBound :: Int
9223372036854775807
? maxBound :: Integer
No instance for Bounded Integer
```

6.8 Enum

- the dot-dot notation is overloaded too

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{Enum}\ a\ \textbf{where}} \\
\quad \mathit{succ}, \mathit{pred} & :: a \to a & \\
\quad \mathit{toEnum} & :: \mathit{Int} \to a & \\
\quad \mathit{fromEnum} & :: a \to \mathit{Int} & \\
\quad \mathit{enumFrom} & :: a \to [a] & \text{-- } [n\,..] \\
\quad \mathit{enumFromThen} & :: a \to a \to [a] & \text{-- } [n, n'\,..] \\
\quad \mathit{enumFromTo} & :: a \to a \to [a] & \text{-- } [n\,..\,m] \\
\quad \mathit{enumFromThenTo} & :: a \to a \to a \to [a] & \text{-- } [n, n'\,..\,m]
\end{array}
$$

- jolly useful for generating test data

$$
\begin{array}{l}
?\ [\mathit{Mon}\,..\,\mathit{Sun}] \\
[\mathit{Mon}, \mathit{Tue}, \mathit{Wed}, \mathit{Thu}, \mathit{Fri}, \mathit{Sat}, \mathit{Sun}]
\end{array}
$$

6.8 Pretty printing

- converting data into textual representation: *pretty printing*

$$
\begin{array}{l}
\mathbf{type}\ \mathit{ShowS} = \mathit{String} \to \mathit{String} \\
\mathbf{class}\ \mathit{Show}\ a\ \mathbf{where} \\
\quad \begin{array}{ll}
\mathit{show} & :: a \to \mathit{String} \\
\mathit{showsPrec} & :: \mathit{Int} \to a \to \mathit{ShowS} \\
\mathit{showList} & :: [\,a\,] \to \mathit{ShowS}
\end{array}
\end{array}
$$

- for reasons of efficiency, *Show* uses the monoid $(\mathit{ShowS}, \mathit{id}, \circ)$ instead of $(\mathit{String}, [\,], +\!\!+)$
- Hughes' efficient representation of lists (more later)
- operator precedences can be taken into account
- for each type we can also decide how to format lists of elements of that type
- you almost always want to say **deriving** (*Show*)

6.8 Parsing

- converting textual representation into data: *parsing*

$$\textbf{type}\ \mathit{ReadS}\ a = \mathit{String} \to [\,(a, \mathit{String})\,]$$

$$\begin{aligned}&\textbf{class}\ \mathit{Read}\ a\ \textbf{where}\\ &\quad \mathit{readsPrec} :: \mathit{Int} \to \mathit{ReadS}\ a\\ &\quad \mathit{readList}\ \ :: \mathit{ReadS}\ [\,a\,]\end{aligned}$$

- *Read* uses “list of successes” technique
- $\mathit{read} \circ \mathit{show}$ should be the identity

6.8 Deriving instances

- defining equality is tedious, can be derived automatically:

  data *Gender* = *Female* | *Male*
  deriving (*Eq*, *Ord*, *Enum*, *Bounded*, *Show*, *Read*)

- the compiler generates the 'obvious' code:
  - identity for *Eq*,
  - lexicographic ordering for *Ord* etc
- *Bounded* and *Enum* only work for enumerations (*Bounded* also works for records of bounded types)
- **deriving** works for parametric types too

  data *Tree a* = *Leaf a* | *Fork* (*Tree a*) (*Tree a*)
  deriving (*Eq*, *Ord*, *Show*, *Read*)

6.8 The mother of all numeric type classes

- Haskell offers an abundance of numeric types and type classes
- *Num* is the mother of these type classes

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a, Show\ a) \Rightarrow Num\ a\ \textbf{where} \\
\quad (+), (-), (*) :: a \to a \to a \\
\quad negate :: a \to a \\
\quad abs, signum :: a \to a \\
\quad fromInteger :: Integer \to a \\
\quad x - y = x + negate\ y \\
\quad negate\ x = 0 - x
\end{array}
$$

- numerals are overloaded too!
- 4711 is shorthand for $fromInteger\ (4711 :: Integer)$

6.9 Case study: monoids

- map-reduce builds on monoids
- why not define a class for monoids?

$$\begin{array}{l} \textbf{class}\ \mathit{Monoid}\ a\ \textbf{where} \\ \quad \epsilon \quad :: a \\ \quad (\bullet) :: a \to a \to a \end{array}$$

- we require $\bullet$ to be associative with ϵ as its neutral element
- the implementation of *mapReduce* simplifies to

$$\begin{array}{l} \mathit{reduce} :: (\mathit{Monoid}\ m) \Rightarrow [\,m\,] \to m \\ \mathit{reduce}\ [\,] \qquad = \epsilon \\ \mathit{reduce}\ (x : xs) = x \bullet \mathit{reduce}\ xs \end{array}$$

$$\begin{array}{l} \mathit{mapReduce} :: (\mathit{Monoid}\ m) \Rightarrow (a \to m) \to ([\,a\,] \to m) \\ \mathit{mapReduce}\ f = \mathit{reduce} \circ \mathit{map}\ f \end{array}$$

- the monoid operations are now passed implicitly

6.9 Examples of monoids

- lists form a monoid

$$\begin{aligned}&\textbf{instance } \mathit{Monoid}\,[\,a\,] \textbf{ where}\\&\quad \epsilon \quad = [\,]\\&\quad (\bullet) = (+\!\!+)\end{aligned}$$

- for lists, *reduce* amounts to *concat*

$$\begin{aligned}&?\ \mathit{reduce}\,[\,[\,4,7\,],[\,],[\,1\,],[\,1\,]\,]\\&[\,4,7,1,1\,]\end{aligned}$$

- *problem: Int* gives rise to a monoid in at least four different ways—which one to pick?

$$\begin{aligned}&\textbf{instance } \mathit{Monoid\ Integer} \textbf{ where}\\&\quad \epsilon \quad = 0\\&\quad (\bullet) = (+)\end{aligned}$$

6.9 Examples of monoids—continued

- for the remaining instances we have to introduce new types

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Mul} = M\ \mathit{Integer} \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monoid}\ \mathit{Mul}\ \textbf{where} \\
\quad \epsilon \qquad\qquad = M\ 1 \\
\quad M\ x \bullet M\ y = M\ (x * y)
\end{array}
$$

- **newtype** is like **type** in that a new type is defined in terms of an old one; **newtype** is like **data** in that the type defined is unequal to all other types

$$
\begin{array}{l}
?\ \mathit{reduce}\ [1\,..\,100] \\
5050 \\
?\ \mathit{reduce}\ [M\ i \mid i \leftarrow [1\,..\,100]] \\
M\ 3628800
\end{array}
$$

- note that we *can't* say $4711 + M\ 0815$

6.9 Cayley representation

- the list monoid is slow when $+\!\!+$ is nested to the left (cf first implementation of *reverse*)
- this is why the *Show* class uses the monoid $(\mathit{ShowS}, \mathit{id}, \circ)$ instead of $(\mathit{String}, [\,], +\!\!+)$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Show}\ \mathit{Expr}\ \textbf{where} \\
\quad \mathit{showsPrec}\ \_\ (\mathit{Lit}\ i) \qquad\quad = \mathit{shows}\ i \\
\quad \mathit{showsPrec}\ d\ (e1 :\!+\!: e2) = \mathit{showParen}\ (d > 6) \\
\qquad (\mathit{showsPrec}\ 6\ e1 \circ \mathit{showsString}\ \texttt{"\ \ +\ \ "} \circ \mathit{showsPrec}\ 6\ e2) \\
\quad \mathit{showsPrec}\ d\ (e1 :\!*\!: e2) = \mathit{showParen}\ (d > 7) \\
\qquad (\mathit{showsPrec}\ 7\ e1 \circ \mathit{showsString}\ \texttt{"\ \ *\ \ "} \circ \mathit{showsPrec}\ 7\ e2)
\end{array}
$$

- *showsString* embeds a string into *ShowS*:

$$
\begin{array}{l}
\mathit{showsString} :: \mathit{String} \to \mathit{ShowS} \\
\mathit{showsString}\ s = (s +\!\!+)
\end{array}
$$

6.9 Cayley representation—continued

- the list monoid can be made more efficient by turning it into a monoid of functions
- this trick works for an arbitrary monoid

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Cayley}\ m = C\ (m \to m) \\[4pt]
\textbf{instance}\ \mathit{Monoid}\ (\mathit{Cayley}\ m)\ \textbf{where} \\
\quad \epsilon \qquad\qquad\ \ = C\ \mathit{id} \\
\quad C\ f \bullet C\ g = C\ (f \circ g) \\[4pt]
\mathit{toCayley} :: (\mathit{Monoid}\ m) \Rightarrow m \to \mathit{Cayley}\ m \\
\mathit{toCayley}\ a = C\ (a\bullet) \\[4pt]
\mathit{fromCayley} :: (\mathit{Monoid}\ m) \Rightarrow \mathit{Cayley}\ m \to m \\
\mathit{fromCayley}\ (C\ f) = f\ \epsilon
\end{array}
$$

- for some monoids $(\bullet a)$ may be a better choice
- the idea is usually attributed to Hughes, 1986 (but actually, it first appeared in work by Cayley, 1854)
- (Cayley: every monoid is equivalent to a monoid of functions)

6.9 New monoids from old

- reversing a monoid

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Reverse}\ m = R\ m \\
&\textbf{instance}\ (\mathit{Monoid}\ m) \Rightarrow \mathit{Monoid}\ (\mathit{Reverse}\ m)\ \textbf{where} \\
&\quad \epsilon \qquad\qquad = R\ \epsilon \\
&\quad R\ x \bullet R\ y = R\ (y \bullet x) \\
&\mathit{toReverse} :: m \to \mathit{Reverse}\ m \\
&\mathit{toReverse}\ x = R\ x \\
&\mathit{fromReverse} :: \mathit{Reverse}\ m \to m \\
&\mathit{fromReverse}\ (R\ x) = x
\end{aligned}
$$

- efficient reverse

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse} = \mathit{fromCayley} \circ \mathit{fromReverse} \circ \\
&\qquad\qquad \mathit{mapReduce}\ (\mathit{toReverse} \circ \mathit{toCayley} \circ \lambda a \to [a])
\end{aligned}
$$

6.9 New monoids from old—continued

- product of monoids (computing in parallel)

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Monoid}\ m, \mathit{Monoid}\ n) \Rightarrow \mathit{Monoid}\ (m, n)\ \textbf{where}\\
&\quad \epsilon = (\epsilon, \epsilon)\\
&\quad (a, x) \bullet (b, y) = (a \bullet b, x \bullet y)
\end{aligned}$$

- replacing a double traversal by a single traversal

$$\begin{aligned}
&\mathit{sumProduct} :: [\mathit{Integer}] \to (\mathit{Integer}, \mathit{Mul})\\
&\mathit{sumProduct} = \mathit{mapReduce}\ (\lambda n \to (n, M\ n))
\end{aligned}$$

- eg $\mathit{sumProduct}\ [1\,..\,10]$ yields $(55, M\ 3628800)$

6.9 New monoids from old—continued

- semi-directed product of monoids

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Semi}\ m\ n = S\ m\ n \\
&\mathbf{class}\ \mathit{Homo}\ m\ n\ \mathbf{where} \\
&\quad \mathit{homo} :: m \to n \to n \\
&\mathbf{instance}\ (\mathit{Monoid}\ m, \mathit{Monoid}\ n, \mathit{Homo}\ m\ n) \Rightarrow \\
&\qquad\qquad \mathit{Monoid}\ (\mathit{Semi}\ m\ n)\ \mathbf{where} \\
&\quad \epsilon = S\ \epsilon\ \epsilon \\
&\quad S\ a\ x \bullet S\ b\ y = S\ (a \bullet b)\ (x \bullet \mathit{homo}\ a\ y)
\end{aligned}
$$

- we require $\mathit{homo}\ a$ to be a curried *homomorphism*: $\mathit{homo}\ \epsilon = \mathit{id}$ and $\mathit{homo}\ (a \bullet b) = \mathit{homo}\ a \circ \mathit{homo}\ b$, and $\mathit{homo}\ a\ \epsilon = \epsilon$ and $\mathit{homo}\ a\ (x \bullet y) = \mathit{homo}\ a\ x \bullet \mathit{homo}\ a\ y$
- *exercise:* show that $\mathit{Semi}\ m\ n$ is indeed a monoid
- Homo is a *multiple-parameter type class* (Haskell extension)

6.9 Application: evaluation of polynomials

- a polynomial can be represented by a list of coefficients eg

$$p :: \mathit{Integer} \to \mathit{Integer}$$
$$p\ (x) = 4 + 7 * x + x \uparrow 2 + x \uparrow 3$$

  is represented by $[4, 7, 1, 1]$

- parallel evaluation of polynomials:

$$\textbf{instance}\ \mathit{Homo}\ \mathit{Mul}\ \mathit{Integer}\ \textbf{where}$$
$$\quad \mathit{homo}\ (M\ a)\ x = a * x$$
$$\mathit{evaluate} :: \mathit{Integer} \to [\mathit{Integer}] \to \mathit{Semi}\ \mathit{Mul}\ \mathit{Integer}$$
$$\mathit{evaluate}\ x = \mathit{mapReduce}\ (\lambda a \to S\ (M\ x)\ a)$$

- eg $\mathit{evaluate}\ 2\ [4, 7, 1, 1]$ yields $S\ (M\ 16)\ 30$, ie $S\ (M\ (2 \uparrow 4), p\ (2))$

6.10 Mapping functions

- the type of lists is the prime example of a *container type*
- *recall:* *map* applies a given function to each element of a list

$$
\begin{aligned}
&map :: (a \to b) \to ([\,a\,] \to [\,b\,]) \\
&map\ \_\ [\,] \qquad\;\; = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

- map changes the elements but keeps the structure intact
- map is also known as an *internal iterator*
- (external iterators correspond to lazy lists)

6.10 Examples of container types

- *Maybe* is also an example of a container type

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

- either an empty or a singleton container
- *Maybe* also supports a mapping function

$$\begin{aligned}
&\mathit{mapMaybe} :: (a \to b) \to (\mathit{Maybe}\ a \to \mathit{Maybe}\ b)\\
&\mathit{mapMaybe}\ _\ \mathit{Nothing} = \mathit{Nothing}\\
&\mathit{mapMaybe}\ f\ (\mathit{Just}\ a) \ \ = \mathit{Just}\ (f\ a)
\end{aligned}$$

6.10 Examples of container types—continued

- map on binary trees

$$\mathbf{data}\ Btree\ a = Tip\ a \mid Bin\ (Btree\ a)\ (Btree\ a)$$

$$\begin{aligned}
&mapBtree :: (a \to b) \to (Btree\ a \to Btree\ b)\\
&mapBtree\ f\ (Tip\ a) \quad = Tip\ (f\ a)\\
&mapBtree\ f\ (Bin\ t\ u) = Bin\ (mapBtree\ f\ t)\ (mapBtree\ f\ u)
\end{aligned}$$

- map on general trees

$$\mathbf{data}\ Gtree\ a = Branch\ a\ [\,Gtree\ a\,]$$

$$\begin{aligned}
&mapGtree :: (a \to b) \to (Gtree\ a \to Gtree\ b)\\
&mapGtree\ f\ (Branch\ x\ ts) = Branch\ (f\ x)\ (map\ (mapGtree\ f)\ ts)
\end{aligned}$$

6.10 The functor class

- the types of the mapping functions are very similar

$$
\begin{array}{llll}
\mathit{map} & :: (a \to b) \to ([\,a\,] & \to [\,b\,] &) \\
\mathit{mapMaybe} & :: (a \to b) \to (\mathit{Maybe}\ a & \to \mathit{Maybe}\ b &) \\
\mathit{mapBtree} & :: (a \to b) \to (\mathit{Btree}\ \ a & \to \mathit{Btree}\ \ b &) \\
\mathit{mapGtree} & :: (a \to b) \to (\mathit{Gtree}\ \ a & \to \mathit{Gtree}\ \ b &)
\end{array}
$$

- the functor class abstracts away from the container type

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \\
\quad \mathit{fmap} :: (a \to b) \to (f\ a \to f\ b)
\end{array}
$$

- note that f is a type constructor!
- *Functor* is a so-called *constructor class*
- (*functor* is a term from category theory, purloined from Carnap's "Logische Sprache der Syntax")

6.10 Instances of the functor class

- every container type should be made an instance of the functor class

instance *Functor Maybe* **where**
  $fmap = mapMaybe$

instance *Functor Btree* **where**
  $fmap = mapBtree$

instance *Functor Gtree* **where**
  $fmap = mapGtree$

- *exercise:* three occurrences of *fmap*; which is which?

instance *Functor Gtree* **where**
  $fmap\ f\ (Branch\ x\ ts) = Branch\ (f\ x)\ (fmap\ (fmap\ f)\ ts)$

6.11 Abstraction, abstraction, abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- type classes allow you to capture commonalities across datatypes
- type classes make ad-hoc polymorphism less ad-hoc
- overloaded functions implement a family of algorithms
- classes are most useful if the type uniquely determines the instance (example: functor, counterexample: monoid)

Part 7

Monads

7.0 Outline

7.1 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 (eg printing to the console or writing to the file system?)
- how to incorporate these into Haskell?

7.1 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!

7.1 Monadic IO

- *idea: putStr* "ha" has *no* effect at all
- introduce a new type of IO computations

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

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

7.1 Interpreting strings

- if evaluator evaluates non-monadic 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
```

7.2 The monad interface

- $IO\ a$ is an abstract datatype of IO computations
- $return$ turns a value into an IO 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

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

- every monad supports these three operations
- every monad also supports additional effect-specific operations eg

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

7.2 Example

- a simple interactive program

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

- remember: $\lambda s \to \ldots$ is an anonymous function

7.2 IO computations as first-class citizens

- we can freely mix IO 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

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

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

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

7.2 More IO operations

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

$$
\begin{aligned}
&\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()\\
&\mathit{getChar} :: \mathit{IO}\ \mathit{Char}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathbf{data}\ \mathit{StdGen} = \ldots && \text{-- standard random generator}\\
&\mathbf{class}\ \mathit{Random}\ \mathbf{where}\ \ldots && \text{-- randomly generatable}\\
&\mathit{randomR} :: (\mathit{Random}\ a) \Rightarrow (a, a) \to \mathit{StdGen} \to (a, \mathit{StdGen})\\
&\mathit{getStdRandom} :: (\mathit{StdGen} \to (a, \mathit{StdGen})) \to \mathit{IO}\ a
\end{aligned}
$$

and many more ...

7.3 Do notation

Special syntactic sugar for monadic expressions.
Inspired by (in fact, a generalization of) list comprehensions.

$$
\begin{aligned}
\mathbf{do}\ \{m\} &= m \\
\mathbf{do}\ \{x \leftarrow m; ms\} &= m \gg\!= \lambda x \rightarrow \mathbf{do}\ \{ms\} \\
\mathbf{do}\ \{m; ms\} &= m \gg\!= \lambda\_ \rightarrow \mathbf{do}\ \{ms\}
\end{aligned}
$$

where a can appear free in ms.

$$x \leftarrow m$$

Pronounce "x is drawn from m". Note that m has type $IO\ a$, whereas x has type a.

7.3 Character I/O

$$
\begin{aligned}
&\mathit{putStr}, \mathit{putStrLn} :: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{putStr}\ \texttt{""} \qquad\quad = \mathbf{do}\ \{\mathit{return}\ ()\}\\
&\mathit{putStr}\ (c : s) = \mathbf{do}\ \{\mathit{putChar}\ c; \mathit{putStr}\ s\}\\
&\mathit{putStrLn}\ s \qquad = \mathbf{do}\ \{\mathit{putStr}\ s; \mathit{putChar}\ \texttt{'\textbackslash n'}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getLine'} :: \mathit{IO}\ \mathit{String}\\
&\mathit{getLine'} = \mathbf{do}\ c \leftarrow \mathit{getChar}\\
&\qquad\qquad\qquad \mathbf{if}\ c == \texttt{'\textbackslash n'}\ \mathbf{then}\ \mathit{return}\ \texttt{""}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{do}\ s \leftarrow \mathit{getLine'}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (c : s)
\end{aligned}
$$

7.3 File I/O

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

7.3 Random numbers

```
import System.Random

rollDice :: IO Int
rollDice = getStdRandom (randomR (1, 6))

rollThrice :: IO Int
rollThrice = do x ← rollDice
                y ← rollDice
                z ← rollDice
                return (x + y + z)
```

```
                                    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,?$$$$P"l$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$,`$$$$$cccd$$$$$?$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$ ?$$$$$$$$$$$$",$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$)$$$$$$$$$,"$$$$$$P",c$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$FJ$$$$$$$$$$b,,...,,d$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

7.4 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.

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

7.4 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 (ie tips) and in the type of labels of internal nodes.

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

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

7.4 The game logic

```
guess :: Tree String String → IO ()
guess (Tip s)
  = putStrLn s
guess (Node q l r)
  = do b ← yesOrNo q
       if b then
         guess l
       else
         guess r

yesOrNo :: String → IO Bool
yesOrNo question
  = do putStrLn question
       answer ← getLine
       return (elem (map toLower answer) ["y","yes"])
```

7.4 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"))
```

7.4 I guess the number

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

7.5 Composition of effectful functions

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

$$(\odot) :: (b \to IO\ c) \to (a \to IO\ b) \to (a \to IO\ c)$$
$$(f \odot g)\ x = g\ x \gg\!= f$$

- turning a pure into an effectful function

$$lift :: (a \to b) \to (a \to IO\ b)$$
$$lift\ f\ x = return\ (f\ x)$$

- example

$$processFile :: FilePath \to (String \to String) \to FilePath \to IO\ ()$$
$$processFile\ outFile\ f$$
$$= writeFile\ outFile \odot lift\ f \odot readFile$$

7.5 Monad laws

IO is a monad because it satisfies the monad laws (expressed in terms of *return* and $\odot$):

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

(so monads are intimately related to monoids)

7.6 Define your own monad

- *IO* is a *monad*
- monads form *an abstract datatype of computations*.
- computations in general may have *effects*: I/O, exceptions, mutable state, etc.
- monads are a mechanism for cleanly incorporating such impure features in a pure setting
- other monads encapsulate exceptions, state, non-determinism, etc
- the following slides motivate the need for a general notion of computation

7.6 An evaluator

Here's a simple datatype of terms:

$$\textbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Integer} \mid \mathit{Div}\ \mathit{Expr}\ \mathit{Expr}$$
$$\quad \textbf{deriving}\ (\mathit{Show})$$

$$\mathit{good}, \mathit{bad} :: \mathit{Expr}$$
$$\mathit{good} = \mathit{Div}\ (\mathit{Lit}\ 7)\ (\mathit{Div}\ (\mathit{Lit}\ 4)\ (\mathit{Lit}\ 2))$$
$$\mathit{bad} \ \ = \mathit{Div}\ (\mathit{Lit}\ 7)\ (\mathit{Div}\ (\mathit{Lit}\ 2)\ (\mathit{Lit}\ 4))$$

... and an evaluation function:

$$\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}$$
$$\mathit{eval}\ (\mathit{Lit}\ n) \quad\ \ = n$$
$$\mathit{eval}\ (\mathit{Div}\ x\ y) = \mathit{eval}\ x\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ y$$

7.6 Exceptions

Evaluation may fail, because of division by zero.
Let's handle the exceptional behaviour:

```
data Exc a = Raise Exception | Result a
type Exception = String

evalE :: Expr → Exc Integer
evalE (Lit n)     = Result n
evalE (Div x y) =
  case evalE x of
  Raise e  → Raise e
  Result u → case evalE y of
             Raise e  → Raise e
             Result v →
               if v == 0 then Raise "division by zero"
                         else Result (u `div` v)
```

7.6 Counting

We could instrument the evaluator to count evaluation steps:

$$
\begin{array}{l}
\mathbf{newtype}\ \mathit{Counter}\ a = C\ (\mathit{State} \to (a, \mathit{State})) \\
\mathbf{type}\ \mathit{State} = \mathit{Int} \\
\mathit{run} :: \mathit{Counter}\ a \to \mathit{State} \to (a, \mathit{State}) \\
\mathit{run}\ (C\ f) = f
\end{array}
$$

$$
\begin{array}{l}
\mathit{evalC} :: \mathit{Expr} \to \mathit{Counter}\ \mathit{Integer} \\
\mathit{evalC}\ (\mathit{Lit}\ n) \quad\ \ = C\ (\lambda i \to (n, i + 1)) \\
\mathit{evalC}\ (\mathit{Div}\ x\ y) = C\ (\lambda i \to \\
\qquad\qquad\qquad\qquad \mathbf{let}\ (u, i') \ = \mathit{run}\ (\mathit{evalC}\ x)\ (i + 1) \\
\qquad\qquad\qquad\qquad \phantom{\mathbf{let}}\ (v, i'') = \mathit{run}\ (\mathit{evalC}\ y)\ i' \\
\qquad\qquad\qquad\qquad \mathbf{in}\ (u\ \text{`}\mathit{div}\text{`}\ v, i''))
\end{array}
$$

7.6 Tracing

... or to trace the evaluation steps:

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Trace}\ a = T\ (\mathit{Output}, a)\\
\textbf{type}\ \mathit{Output} = \mathit{String}
\end{array}
$$

$$
\begin{array}{l}
\mathit{evalT} :: \mathit{Expr} \to \mathit{Trace}\ \mathit{Integer}\\
\mathit{evalT}\ (\mathit{Lit}\ n) \quad = T\ (\mathit{line}\ (\mathit{Lit}\ n)\ n, n)\\
\mathit{evalT}\ (\mathit{Div}\ x\ y) = \textbf{let}\\
\qquad\qquad\qquad\quad T\ (s,\ u) = \mathit{evalT}\ x\\
\qquad\qquad\qquad\quad T\ (s', v) = \mathit{evalT}\ y\\
\qquad\qquad\qquad\quad p = u\ \text{`}\mathit{div}\text{`}\ v\\
\qquad\qquad\qquad \textbf{in}\ T\ (s +\!\!+\ s' +\!\!+\ \mathit{line}\ (\mathit{Div}\ x\ y)\ p, p)
\end{array}
$$

$$
\begin{array}{l}
\mathit{line} :: \mathit{Expr} \to \mathit{Integer} \to \mathit{Output}\\
\mathit{line}\ t\ n = \texttt{"\ \ \ "} +\!\!+\ \mathit{show}\ t +\!\!+\ \texttt{" yields "} +\!\!+\ \mathit{show}\ n +\!\!+\ \texttt{"\textbackslash n"}
\end{array}
$$

7.6 Ugly!

- none of these 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?

7.7 The monad type class

These are the methods of a type class:

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

We can also use **do**-notation for *Monad* instances.

7.7 Original evaluator, monadically

$$
\begin{aligned}
&\mathit{evalM} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Expr} \to m\ \mathit{Integer}\\
&\mathit{evalM}\ (\mathit{Lit}\ n) \quad\ \ = \mathit{return}\ n\\
&\mathit{evalM}\ (\mathit{Div}\ x\ y) = \mathit{evalM}\ x \gg\!= \lambda u \to\\
&\qquad\qquad\qquad\quad\ \ \mathit{evalM}\ y \gg\!= \lambda v \to\\
&\qquad\qquad\qquad\quad\ \ \mathit{return}\ (u\ \text{`}\mathit{div}\text{`}\ v)
\end{aligned}
$$

Still pure, but written in the monadic style; much easier to extend.

7.7 Original evaluator, using do notation

$$
\begin{aligned}
&\mathit{evalM} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Expr} \to m\ \mathit{Integer}\\
&\mathit{evalM}\ (\mathit{Lit}\ n) \quad\ \ = \mathbf{do}\ \mathit{return}\ n\\
&\mathit{evalM}\ (\mathit{Div}\ x\ y) = \mathbf{do}\ u \leftarrow \mathit{evalM}\ x\\
&\qquad\qquad\qquad\qquad\quad\ \ v \leftarrow \mathit{evalM}\ y\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (u\ \text{`}\mathit{div}\text{`}\ v)
\end{aligned}
$$

7.7 The exception instance

Exceptions instantiate the class:

$$\textbf{data}\ \mathit{Exc}\ a = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Result}\ a$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ \mathit{Exc}\ \textbf{where} \\ \quad \mathit{return}\ a \qquad\quad = \mathit{Result}\ a \\ \quad \mathit{Raise}\ e \gg\!= \_ \ \ = \mathit{Raise}\ e \\ \quad \mathit{Result}\ a \gg\!= f = f\ a\end{array}$$

The effect-specific behaviour is to throw an exception:

$$\begin{array}{l}\mathit{throw} :: \mathit{Exception} \to \mathit{Exc}\ e \\ \mathit{throw}\ e = \mathit{Raise}\ e\end{array}$$

7.7 Exceptional evaluator, monadically

$$
\begin{aligned}
&\mathit{evalE} :: \mathit{Expr} \to \mathit{Exc}\ \mathit{Integer}\\
&\mathit{evalE}\ (\mathit{Lit}\ n) \quad\ \ = \mathbf{do}\ \mathit{return}\ n\\
&\mathit{evalE}\ (\mathit{Div}\ x\ y) = \mathbf{do}\ u \leftarrow \mathit{evalE}\ x\\
&\qquad\qquad\qquad\qquad\quad\ v \leftarrow \mathit{evalE}\ y\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{if}\ v == 0\ \mathbf{then}\ \mathit{throw}\ \texttt{"division by zero"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{return}\ (u\ \text{`}\mathit{div}\text{`}\ v)
\end{aligned}
$$

7.7 The counter instance

Counters instantiate the class:

$$\mathbf{newtype}\ \mathit{Counter}\ a = C\ (\mathit{State} \to (a, \mathit{State}))$$

$$\begin{aligned}&\mathbf{instance}\ \mathit{Monad}\ \mathit{Counter}\ \mathbf{where}\\&\quad \mathit{return}\ a = C\ (\lambda n \to (a, n))\\&\quad \mathit{ma} \gg\!= f = C\ (\lambda n \to \mathbf{let}\ (a, n') = \mathit{run}\ \mathit{ma}\ n\ \mathbf{in}\ \mathit{run}\ (f\ a)\ n')\end{aligned}$$

The effect-specific behaviour is to increment the count:

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

7.7 Counting evaluator, monadically

$$
\begin{aligned}
&\mathit{evalC} :: \mathit{Expr} \to \mathit{Counter\ Integer} \\
&\mathit{evalC}\ (\mathit{Lit}\ n) = \mathbf{do}\ \mathit{tick} \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ n \\
&\mathit{evalC}\ (\mathit{Div}\ x\ y) = \mathbf{do}\ \mathit{tick} \\
&\qquad\qquad\qquad\qquad\quad u \leftarrow \mathit{evalC}\ x \\
&\qquad\qquad\qquad\qquad\quad v \leftarrow \mathit{evalC}\ y \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (u\ \text{`}\mathit{div}\text{`}\ v)
\end{aligned}
$$

7.7 The tracing instance

Tracers instantiate the class:

$$\textbf{newtype}\ \mathit{Trace}\ a = T\ (\mathit{Output}, a)$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Trace}\ \textbf{where} \\
&\quad \mathit{return}\ a \quad = T\ (\texttt{""}, a) \\
&\quad T\ (s, a) \gg\!\!= f = \textbf{let}\ T\ (s', b) = f\ a\ \textbf{in}\ T\ (s +\!\!+ s', b)
\end{aligned}$$

The effect-specific behaviour is to log some output:

$$\begin{aligned}
&\mathit{trace} :: \mathit{String} \to \mathit{Trace}\ () \\
&\mathit{trace}\ s = T\ (s, ())
\end{aligned}$$

7.7 Tracing evaluator, monadically

$$
\begin{array}{lll}
\mathit{evalT} :: \mathit{Expr} \to \mathit{Trace}\ \mathit{Integer} & & \\
\mathit{evalT}\ (\mathit{Lit}\ n) & = \mathbf{do} & \mathit{trace}\ (\mathit{line}\ (\mathit{Lit}\ n)\ n) \\
 & & \mathit{return}\ n \\
\mathit{evalT}\ (\mathit{Div}\ x\ y) & = \mathbf{do} & u \leftarrow \mathit{evalT}\ x \\
 & & v \leftarrow \mathit{evalT}\ y \\
 & & \mathbf{let}\ p = u\ `\mathit{div}`\ v \\
 & & \mathit{trace}\ (\mathit{line}\ (\mathit{Div}\ x\ y)\ p) \\
 & & \mathit{return}\ p
\end{array}
$$

7.7 The IO monad

- There's no magic to monads in general: all the monads above are just plain (perhaps higher-order) data, implementing a particular interface.
- But there is one magic monad: the *IO* monad. Its implementation is abstract, hard-wired in the language implementation.

$$\textbf{data}\ \mathit{IO}\ a = \ldots$$
$$\textbf{instance}\ \mathit{Monad}\ \mathit{IO}\ \textbf{where}\ \ldots$$

7.8 Abstraction, abstraction, abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- monads allow you to abstract over patterns of computations (effects)
- Haskell allows you to implement your own computational effect or combination of effects (how cool is this?)
- IO computations are first-class values!
- in general, try to minimize the IO part of your program