FPR

Functional Programming

Introduction to Functional Programming

Jeremy Gibbons

University of Oxford

February 2015

Part 0

Course aims and objectives

0.0 Outline

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

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

$$i{:=}i+1\ ;\ a{:=}a*i \quad \neq \quad a{:=}a*i\ ;\ i{:=}i+1$$

- 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:
  - branch merging:

```
    if b then p else p end
=   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

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

```
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, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.
- Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, No Starch Press, 2011.
- 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.
- 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

1.0 Outline

1.1 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 ...

1.1 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)

1.1 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
```

1.1 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
```

1.1 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

$$
\begin{array}{l}
\mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\
\mathit{smaller}\ (x, y) \\
\quad = \textbf{if} \\
\qquad\quad x \leqslant y \\
\qquad \textbf{then} \\
\qquad\quad x \\
\qquad \textbf{else} \\
\qquad\quad y
\end{array}
$$

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

1.1 A session

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

1.2 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)

1.2 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)

1.2 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:

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

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

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

1.2 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)

1.2 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

1.3 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

1.3 Function types

- *type declaration* in script specifies type of function
- eg $square :: Integer \to Integer$
- in general, $f :: A \to B$ 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 $f\ x$

1.3 Lambda

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

1.3 Declaration vs expression style

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

$$
\begin{aligned}
&quad :: Integer \rightarrow Integer\\
&quad\ x = square\ x * square\ x
\end{aligned}
$$

- *expression style*: emphasising the use of expressions

$$
\begin{aligned}
&quad :: Integer \rightarrow Integer\\
&quad = \lambda x \rightarrow square\ x * square\ x
\end{aligned}
$$

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

1.3 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:

$$
\begin{array}{l}
\mathit{double}, \mathit{double'} :: \mathit{Integer} \to \mathit{Integer} \\
\mathit{double}\ x\ \ = x + x \\
\mathit{double'}\ x = 2 * x
\end{array}
$$

1.3 Currying

- replace single structured argument by several simpler ones

$$add :: (Integer, Integer) \to Integer$$
$$add\ (x, y) = x + y$$

$$add' :: Integer \to (Integer \to Integer)$$
$$add'\ x\ y = x + y$$

- 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)

1.3 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

1.3 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

- the operator $+$ has also a neutral element

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

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

1.3 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 $Integer \to (Integer \to Integer)$

1.3 Precedence

- association does not help when operators are mixed

  $$x \oplus y \otimes z$$

  what does this mean: $(x \oplus y) \otimes z$ or $x \oplus (y \otimes z)$?
- 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 $\mathit{square}\ 3 + 4 = 13$

1.3 Composition

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

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

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

1.4 Definitions

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

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

- 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)

1.4 Conditional definitions

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

$$
\begin{aligned}
&\mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\
&\mathit{smaller}\ (x, y) = \mathbf{if}\ x \leqslant y\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{aligned}
$$

- could also use *guarded equations*:

$$
\begin{aligned}
&\mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\
&\mathit{smaller}\ (x, y) \\
&\quad \mid x \leqslant y = x \\
&\quad \mid x > y = y
\end{aligned}
$$

- 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**...

1.4 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

$$
\begin{array}{l}
\mathit{day} :: \mathit{Integer} \rightarrow \mathit{String} \\
\mathit{day}\ 1 = \texttt{"Saturday"} \\
\mathit{day}\ 2 = \texttt{"Sunday"} \\
\mathit{day}\ \_ = \texttt{"Weekday"}
\end{array}
$$

- 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

1.4 Local definitions

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

$$
\begin{array}{l}
qroots :: (Float, Float, Float) \rightarrow (Float, Float) \\
qroots\ (a, b, c) = ((-b - sd)\ /\ (2 * a), (-b + sd)\ /\ (2 * a)) \\
\quad \mathbf{where}\ sd = sqrt\ (b * b - 4 * a * c)
\end{array}
$$

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

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

  (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

1.4 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

1.5 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

2.0 Outline

2.1 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

2.2 Simple types

- booleans
- characters
- strings
- numbers

2.2 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

2.2 Boole pattern

- every type comes with a pattern of definition
- *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)

2.2 Characters

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

2.2 Strings

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

2.2 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)

2.3 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)

2.4 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

2.5 Polymorphism

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

$$fst :: (a, b) \rightarrow a$$

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

2.5 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

2.6 Type synonyms

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

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

- cannot be recursive
- just a 'macro': no new type

2.7 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)

2.7 Some standard type classes

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

2.7 Derived type classes

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

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Day}\ \textbf{where} \\
\quad \mathit{Mon} == \mathit{Mon} = \mathit{True} \\
\quad \mathit{Tue} == \mathit{Tue} = \mathit{True} \\
\quad \mathit{Wed} == \mathit{Wed} = \mathit{True} \\
\quad \mathit{Thu} == \mathit{Thu} = \mathit{True} \\
\quad \mathit{Fri} == \mathit{Fri} = \mathit{True} \\
\quad \mathit{Sat} == \mathit{Sat} = \mathit{True} \\
\quad \mathit{Sun} == \mathit{Sun} = \mathit{True} \\
\quad \_ == \_ = \mathit{False}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Day} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Sun} \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Enum}, \mathit{Bounded}, \mathit{Show}, \mathit{Read})
\end{array}
$$

2.8 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: suggests case analysis
- f produces a U value: suggests 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 $reverse$ `"jeremy"` $=$ `"ymerej"`
- $map :: (a \to b) \to ([a] \to [b])$ 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 $tails$ `"jeremy"` $=$
  `["jeremy","eremy","remy","emy","my","y",""]`

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

$$\begin{aligned} &\mathit{unwrap} :: \mathit{String} \to \mathit{String} \\ &\mathit{unwrap} = \mathit{concat} \circ \mathit{lines} \end{aligned}$$

- count the number of lines

$$\begin{aligned} &\mathit{countLines} :: \mathit{String} \to \mathit{Int} \\ &\mathit{countLines} = \mathit{length} \circ \mathit{lines} \end{aligned}$$

- flip text upside down

$$\begin{aligned} &\mathit{upsideDown} :: \mathit{String} \to \mathit{String} \\ &\mathit{upsideDown} = \mathit{unlines} \circ \mathit{reverse} \circ \mathit{lines} \end{aligned}$$

- flip text from left to right

$$\begin{aligned} &\mathit{leftRight} :: \mathit{String} \to \mathit{String} \\ &\mathit{leftRight} = \mathit{unlines} \circ \mathit{map}\ \mathit{reverse} \circ \mathit{lines} \end{aligned}$$

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

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

  (why is this different from $(==\,[\,])$?)

- eg to return first element of non-empty list

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

3.4 Case analysis

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

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

- *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

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

- eg length of a list of elements

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

3.4 List definition pattern

- remember: every type comes with a definition 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: $\mathit{concat}\,[[1,2],[\,],[3]] = [1,2,3]$

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

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

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

(exercise: complexity? improve!)

- is a list ordered?

$$\begin{aligned}
&\mathit{ordered} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow \mathit{Bool} \\
&\mathit{ordered}\ [\,] && = \mathit{True} \\
&\mathit{ordered}\ [\,x\,] && = \mathit{True} \\
&\mathit{ordered}\ (x : y : \mathit{xs}) && = x \leqslant y \;\&\&\; \mathit{ordered}\ (y : \mathit{xs})
\end{aligned}$$

  we distinguish three cases

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

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

  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

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

- 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 $\mathit{filter}\ \mathit{isDigit}$ `"more4u2say"` $=$ `"42"`
- eg $(\mathit{sum} \circ \mathit{map}\ \mathit{square} \circ \mathit{filter}\ \mathit{odd})\ [1\,..\,5] = 35$
- definition

$$
\begin{array}{l}
\mathit{filter} :: (a \rightarrow \mathit{Bool}) \rightarrow ([a] \rightarrow [a]) \\
\mathit{filter}\ p\ [\,] = [\,] \\
\mathit{filter}\ p\ (x : \mathit{xs}) \\
\quad \mid p\ x \qquad\quad\ \ = x : \mathit{filter}\ p\ \mathit{xs} \\
\quad \mid \mathit{otherwise} = \qquad \mathit{filter}\ p\ \mathit{xs}
\end{array}
$$

3.6 Comprehensions

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

$$
\begin{aligned}
&\mathit{overdue} = \\
&\quad [\,(\mathit{name}, \mathit{addr}) \mid (\mathit{key}, \mathit{name}, \mathit{addr}) \leftarrow \mathit{names}, \\
&\qquad\qquad (\mathit{key}', \mathit{date}, \mathit{amount}) \leftarrow \mathit{invoices}, \mathit{key} == \mathit{key}', \\
&\qquad\qquad \mathit{date} < \mathit{today}\,]
\end{aligned}
$$

- 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 (int a in x)
  foreach (int b in g a)
    if (p b)
      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

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

3.7 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 definition 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
- parametric 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

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

- 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

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

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

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

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

```
data Temp = Cels Float | Fahr Float
  deriving (Show)
```

- define our own equality function

```
instance Eq Temp where
  Cels x == Cels y = x == y
  Fahr x == Fahr y = x == y
  Cels x == Fahr y = x * 1.8 == y - 32.0
  Fahr x == Cels y = Cels y == Fahr x
```

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

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

- 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* definition pattern

- recursive definitions by pattern-matching

$$
\begin{array}{l}
\mathit{evaluate} :: \mathit{Expr} \rightarrow \mathit{Integer} \\
\mathit{evaluate}\ (\mathit{Lit}\ i) \qquad\ \ = i \\
\mathit{evaluate}\ (e1 :\!+\!: e2) = \mathit{evaluate}\ e1 + \mathit{evaluate}\ e2 \\
\mathit{evaluate}\ (e1 :\!*\!: e2) = \mathit{evaluate}\ e1 * \mathit{evaluate}\ e2
\end{array}
$$

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

4.4 *Expr* definition pattern

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

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

- *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 3:* 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)

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

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

$$\begin{array}{l}
nat2int :: Nat \to Integer \\
nat2int\ Zero \qquad\quad = 0 \\
nat2int\ (Succ\ n) = 1 + nat2int\ n
\end{array}$$

- addition

$$\begin{array}{l}
plus :: Nat \to Nat \to Nat \\
plus\ Zero \qquad\quad n = n \\
plus\ (Succ\ m)\ n = Succ\ (plus\ m\ n)
\end{array}$$

(does this look familiar?)

4.4 Peano definition pattern

- remember: every datatype comes with a definition 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)

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

- 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

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

4.4 List definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: List\ P \rightarrow S$
- *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)

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

4.4 General trees

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

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

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

$$\begin{array}{l} gametree :: (Pos \to [\,Pos\,]) \to (Pos \to Gtree\ Pos) \\ gametree\ m\ p = Branch\ p\ (map\ (gametree\ m)\ (m\ p)) \end{array}$$

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

$$\begin{aligned}&[\,]\\ &4 : [\,]\\ &7 : 4 : [\,]\\ &11 : 7 : 4 : [\,]\\ &18 : 4 : [\,]\\ &72 : [\,]\end{aligned}$$

- we also show the correctness of compiler and VM

4.5 Warm-up: showing expressions

- *showExpr* maps an expression to its string representation

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

- 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

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

- *idea:* pass in the precedence level of the enclosing operator

$$
\begin{aligned}
& \mathit{showPrec} :: \mathit{Int} \to \mathit{Expr} \to \mathit{String} \\
& \mathit{showPrec}\ \_\ (\mathit{Lit}\ i) \\
& \quad = \mathit{show}\ i \\
& \mathit{showPrec}\ p\ (e1\ {:}{+}{:}\ e2) \\
& \quad = \mathit{parenthesis}\ (p > 6)\ (\mathit{showPrec}\ 6\ e1 +\!\!+ \texttt{"\ +\ "} +\!\!+ \mathit{showPrec}\ 6\ e2) \\
& \mathit{showPrec}\ p\ (e1\ {:}{*}{:}\ e2) \\
& \quad = \mathit{parenthesis}\ (p > 7)\ (\mathit{showPrec}\ 7\ e1 +\!\!+ \texttt{"\ *\ "} +\!\!+ \mathit{showPrec}\ 7\ e2)
\end{aligned}
$$

4.5 Respecting precedence—continued

- optional parenthesis

$$\begin{aligned}&\mathit{parenthesis} :: \mathit{Bool} \to \mathit{String} \to \mathit{String}\\&\mathit{parenthesis}\ \mathit{True}\ s = \texttt{"("} +\!\!+\ s +\!\!+ \texttt{")"}\\&\mathit{parenthesis}\ \mathit{False}\ s = s\end{aligned}$$

- we start off with the lowest precedence

$$\begin{aligned}&\mathit{showExpr} :: \mathit{Expr} \to \mathit{String}\\&\mathit{showExpr} = \mathit{showPrec}\ 0\end{aligned}$$

- eg $\mathit{showExpr}\ \mathit{expr1}$ = `"4 * 7 + 11"` and $\mathit{showExpr}\ \mathit{expr2}$ = `"(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

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

- eg *showCode code1* = `"push 47 ; push 11 ; add"`

4.5 Compilation

- the compiler follows the *Expr* definition pattern

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

- 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 VM follows the *Code* definition 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*

$$
\begin{array}{l}
push :: Integer \rightarrow (Stack \rightarrow Stack) \\
push\ i\ xs = i : xs
\end{array}
$$

$$
\begin{array}{ll}
add\ :: Stack \rightarrow Stack & \\
add\ [\,] & = error\ msg \\
add\ [\_] & = error\ msg \\
add\ (x1 : x2 : xs) & = x2 + x1 : xs
\end{array}
$$

$$
\begin{array}{ll}
mul\ :: Stack \rightarrow Stack & \\
mul\ [\,] & = error\ msg \\
mul\ [\_] & = error\ msg \\
mul\ (x1 : x2 : xs) & = x2 * x1 : xs
\end{array}
$$

$$
\begin{array}{l}
msg :: String \\
msg = \texttt{"VM: empty stack"}
\end{array}
$$

4.5 Advanced: proof of correctness ⋆

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

$$\mathit{push}\ (\mathit{evaluate}\ e) = \mathit{execute}\ (\mathit{compile}\ e)$$

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

4.5 Proof of correctness: base case ⋆

Case $e = \mathit{Lit}\ i$:

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

4.5 Proof of correctness: inductive step ⋆

Case $e = e1 :+: e2$:

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

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

$$\begin{array}{l} addOrMul :: Bool \to (Integer \to Integer \to Integer) \\ addOrMul\ b = \mathbf{if}\ b\ \mathbf{then}\ (+)\ \mathbf{else}\ (*) \end{array}$$

- partial application
- currying
- function composition (again)

5.3 Partial application

- consider $add'\ x\ y = x + y$
- type $Integer \rightarrow Integer \rightarrow Integer$; takes two $Integer$s and returns an $Integer$ (eg $add'\ 3\ 4 = 7$)
- another view: type $Integer \rightarrow (Integer \rightarrow Integer)$ (remember, $\rightarrow$ associates to the right); takes a single $Integer$ and returns an $Integer \rightarrow Integer$ function (eg $add'\ 3$ is the $Integer$-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 application 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

$$curry :: ((a, b) \to c) \to (a \to b \to c)$$
$$curry\ f\ a\ b = f\ (a, b)$$

$$uncurry :: (a \to b \to c) \to ((a, b) \to c)$$
$$uncurry\ f\ (a, b) = f\ a\ b$$

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

$$flip :: (a \to b \to c) \to (b \to a \to c)$$
$$flip\ f\ b\ a = f\ a\ b$$

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 $\mathit{twice}\ \mathit{square}\ 3 = 81$

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

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

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

- more on this in a minute ...

5.4 Functions as datastructures

consider a dictionary (associative array)

$$
\begin{array}{l}
\textbf{type}\ \mathit{Dict}\ k\ v \\
\mathit{empty} :: \mathit{Dict}\ k\ v \\
\mathit{insert} :: (\mathit{Eq}\ k) \Rightarrow (k, v) \rightarrow \mathit{Dict}\ k\ v \rightarrow \mathit{Dict}\ k\ v \\
\mathit{lookup} :: (\mathit{Eq}\ k) \Rightarrow \mathit{Dict}\ k\ v \rightarrow k \rightarrow v
\end{array}
$$

5.4 Implementation as list

```
type Dict k v = [(k, v)]

empty :: Dict k v
empty = []

insert :: (Eq k) => (k, v) -> Dict k v -> Dict k v
insert kv kvs = kv : kvs

lookup :: (Eq k) => Dict k v -> k -> v
lookup [] _ = error "item not present"
lookup ((k, v) : kvs) k'
  | k == k'   = v
  | otherwise = lookup kvs k'
```

5.4 Implementation as function

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

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

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

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

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.

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

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

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

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 definition 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 $foldr$

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

- difference to $reduce$?

5.5 Examples of fold right

- many examples:

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

- right-to-left computation
- operator may $(+, +\!\!+)$ but need not $(:, \mathit{snoc})$ be associative

5.5 Sorting

- given

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

- 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*

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

- definition

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

5.5 Accumulating parameter

- recall *reverse* program

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

- another definition

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

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

5.5 Fold over non-empty lists

- consider computing maximum element in a list of numbers
- $maximum\ (x : xs) = x \text{ `}max\text{` } maximum\ xs$ suggests $foldr$
- but what is the starting value e?
- e should be $maximum\ [\,]$; also require $x \text{ `}max\text{` } e = x$
- want to choose e to be smallest possible value
- could restrict to context $Bounded$, with $e = minBound$
- alternatively, don't use on empty list!

- capture pattern via two folds on non-empty lists

$$
\begin{aligned}
&\mathit{foldr1} :: (a \to a \to a) \to [\,a\,] \to a \\
&\mathit{foldr1}\ \mathit{op}\ (x : \mathit{xs}) \\
&\quad \mid \mathit{null}\ \mathit{xs} \qquad = x \\
&\quad \mid \mathit{otherwise} = x\ \text{`}\mathit{op}\text{`}\ \mathit{foldr1}\ \mathit{op}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl1} :: (a \to a \to a) \to [\,a\,] \to a \\
&\mathit{foldl1}\ \mathit{op}\ (x : \mathit{xs}) = \mathit{foldl}\ \mathit{op}\ x\ \mathit{xs}
\end{aligned}
$$

- eg

$$
\begin{aligned}
&\mathit{foldr1}\ \mathit{op}\ [\,x, y, z\,] = x\ \text{`}\mathit{op}\text{`}\ (y\ \text{`}\mathit{op}\text{`}\ z) \\
&\mathit{foldl1}\ \mathit{op}\ [\,x, y, z\,]\ = (x\ \text{`}\mathit{op}\text{`}\ y)\ \text{`}\mathit{op}\text{`}\ x
\end{aligned}
$$

- now $\mathit{maximum} = \mathit{foldr1}\ \mathit{max} = \mathit{foldl1}\ \mathit{max}$

5.5 Scan

- sometimes convenient to apply *foldl* to every initial segment of a list

$$\begin{aligned} scanl\ op\ e\ [x, y, z] = [\,&e, e\ \text{`}op\text{`}\ x, (e\ \text{`}op\text{`}\ x)\ \text{`}op\text{`}\ y, \\ &((e\ \text{`}op\text{`}\ x)\ \text{`}op\text{`}\ y)\ \text{`}op\text{`}\ z\,] \end{aligned}$$

- eg $scanl\ (+)\ 0$ computes *prefix sums*
- eg $scanl\ (*)\ 1\ [1\,..\,n]$ computes first $n+1$ factorials
- start with specification

$$\begin{aligned} &scanl :: (b \to a \to b) \to b \to [a] \to [b] \\ &scanl\ op\ e = map\ (foldl\ op\ e) \circ inits \end{aligned}$$

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

(exercise: *inits* matches pattern for *foldr*)

5.5 Efficient scan

- inefficient, as quadratically many applications of *op*
- straightforward to synthesize more efficient implementation

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

- dually,

$$\begin{aligned} &\mathit{scanr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [\,a\,] \rightarrow [\,b\,] \\ &\mathit{scanr}\ \mathit{op}\ e = \mathit{map}\ (\mathit{foldr}\ \mathit{op}\ e) \circ \mathit{tails} \end{aligned}$$

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

has more efficient implementation

$$\mathit{scanr}\ \mathit{op}\ e = \mathit{foldr}\ (\lambda x\ \mathit{ys} \rightarrow \mathit{op}\ x\ (\mathit{head}\ \mathit{ys}) : \mathit{ys})\ [\,e\,]$$

5.5 Duality: unfold

- so far we have focused on *consumers*
- *producers* are important too
- producers (unfolds) are *dual* to consumers (folds)
- common pattern

$$
\begin{array}{l}
\mathit{unfoldr} :: (b \to \mathit{Maybe}\,(a, b)) \to (b \to [\,a\,]) \\
\mathit{unfoldr}\ \mathit{coalg}\ x \\
\quad = \mathbf{case}\ \mathit{coalg}\ x\ \mathbf{of} \\
\qquad \mathit{Nothing} \quad\ \ \to [\,] \\
\qquad \mathit{Just}\,(a, x') \to a : \mathit{unfoldr}\ \mathit{coalg}\ x'
\end{array}
$$

- *unfoldr* is *dual* to *fold* (in what way...?)
- relation to OO iterators?

5.5 Examples of unfold

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

$$
\begin{array}{l}
enumFromTo :: (Num\ a, Ord\ a) \Rightarrow a \rightarrow a \rightarrow [\,a\,] \\
enumFromTo\ m\ n \\
\quad = unfoldr\ (\lambda i \rightarrow \mathbf{if}\ i > n\ \mathbf{then}\ Nothing \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ Just\ (i, i+1))\ m
\end{array}
$$

- map can also be expressed as an unfold

$$
\begin{array}{l}
map :: (a \rightarrow b) \rightarrow ([\,a\,] \rightarrow [\,b\,]) \\
map\ f = unfoldr\ (\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \quad\ \rightarrow Nothing \\
\qquad\qquad\qquad\qquad\qquad a : x' \rightarrow Just\ (f\ a, x'))
\end{array}
$$

5.5 Sorting

- given

$$
\begin{aligned}
&\mathit{insertList} :: (\mathit{Ord}\ a) \Rightarrow \mathit{Maybe}\ (a, [\,a\,]) \to [\,a\,] \\
&\mathit{insertList}\ \mathit{Nothing} \qquad\quad = [\,] \\
&\mathit{insertList}\ (\mathit{Just}\ (x, [\,])) = [\,x\,] \\
&\mathit{insertList}\ (\mathit{Just}\ (x, y : ys)) \\
&\quad \mid x \leqslant y \qquad\ \ = x : y : ys \\
&\quad \mid \mathit{otherwise} = y : \mathit{insertList}\ (\mathit{Just}\ (x, ys))
\end{aligned}
$$

  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

$$
\begin{array}{l}
\mathit{deleteMin} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow \mathit{Maybe}\ (a, [\,a\,]) \\
\mathit{deleteMin}\ [\,]\ = \mathit{Nothing} \\
\mathit{deleteMin}\ (x : \mathit{xs}) \\
\quad = \mathbf{case}\ \mathit{deleteMin}\ \mathit{xs}\ \mathbf{of} \\
\qquad \mathit{Nothing} \qquad\quad\ \ \rightarrow \mathit{Just}\ (x, [\,]) \\
\qquad \mathit{Just}\ (y, \mathit{ys}) \\
\qquad\quad \mid x \leqslant y \qquad\quad \rightarrow \mathit{Just}\ (x, y : \mathit{ys}) \\
\qquad\quad \mid \mathit{otherwise} \rightarrow \mathit{Just}\ (y, x : \mathit{ys})
\end{array}
$$

we have

$$
\begin{array}{l}
\mathit{selectSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
\mathit{selectSort} = \mathit{unfoldr}\ \mathit{deleteMin}
\end{array}
$$

- (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)
```

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

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

$$\begin{aligned}
&tequilaBody = \\
&\quad cut\ 16\ (repeatM\ (\\
&\quad\quad twice\ (drums :=: bass :=: guitar) :=: brass))
\end{aligned}$$

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

```
drumIntro = Instr "Drums" (cut 4 (repeatM (
                p0 qn :+: p0 en1 :+: p0 en2)))

drums = Instr "Drums" (drumIntro :=: cut 4 (repeatM (
            (qnr :+: p2 en1 :+: p2 en2) :=: p3 hn)))

drumCoda = Instr "Drums" (cut 2 drums :+:
  line [
    chord [p1 qn, p2 qn, p3 qn],
    chord [p1 qn, p2 qn, p3 qn],
    chord [p1 qn, p2 qn, p3 qn],
    chord [p1 qn, p2 qn, p3 qn, p4 (tie qn wn)]])

p1 d = perc RideCymbal2   d [Volume 50]
p2 d = perc AcousticSnare d [Volume 30]
p3 d = perc LowTom        d [Volume 50]
p4 d = perc SplashCymbal  d [Volume 100]
p0 d = perc PedalHiHat    d [Volume 50]
```

```
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)]]
```

```haskell
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 [ ]
```

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

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

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

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

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

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

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

Laziness

6.0 Outline

6.1 Evaluation orders

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

6.1 Different evaluation orders

- recall different evaluation orders from before:

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

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

- not two different answers
- but sometimes no answer at all!
- which order to choose?

6.1 Applicative-order evaluation

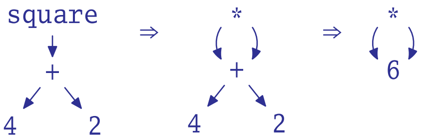
- to reduce the application $f\ e$:
  - first reduce e to normal form
  - then expand definition of f and continue reducing
- (recall, an expression is in *normal form* when it cannot be reduced any further)
- simple and obvious
- easy to implement
- may not terminate when other evaluation orders would

6.1 Normal-order evaluation

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

6.1 A third way: Lazy evaluation

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



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

6.1 Weak head normal form

- an expression is in *weak head normal form* (WHNF) if it is a lambda, or a constructor applied to zero or more arguments
- eg $\lambda n \rightarrow 2 * 3 + n$, $f\,x : map\,f\,xs$, $(1 + 2, 1 - 2)$
- partially normalised
- an expression in normal form is in weak head normal form, but converse does not hold

6.1 Lazy evaluation via let

- equivalently, expand application to **let** expression, reduce to WHNF, substitute normalised fragments
- eg consider $\mathit{sqfst}\ (\mathit{mkpr}\ 3\ 0)$, where

$$\begin{aligned} \mathit{sqfst}\ z &= \mathit{fst}\ z * \mathit{fst}\ z \\ \mathit{mkpr}\ x\ y &= (x + y, x \text{ `}\mathit{div}\text{` } y) \end{aligned}$$

- we have

$$\begin{aligned} & \mathit{sqfst}\ (\mathit{mkpr}\ 3\ 0) \\ \Rightarrow\ & \mathbf{let}\ z = \mathit{mkpr}\ 3\ 0\ \mathbf{in}\ \mathit{sqfst}\ z \\ \Rightarrow\ & \mathbf{let}\ z = \mathit{mkpr}\ 3\ 0\ \mathbf{in}\ \mathit{fst}\ z * \mathit{fst}\ z \\ \Rightarrow\ & \mathbf{let}\ x = 3; y = 0; z = (x + y, x \text{ `}\mathit{div}\text{` } y)\ \mathbf{in}\ \mathit{fst}\ z * \mathit{fst}\ z \\ \Rightarrow\ & \ldots \end{aligned}$$

6.1 Lazy evaluation in Haskell

- leftmost outermost *redex* (reducible expression) reduced at each stage, eg in $(2 + 3) * (4 + 5)$, leftmost $+$ first
- pattern-matching may trigger reduction of arguments to WHNF

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

- patterns matched top to bottom, left to right

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

- guards may also trigger reduction

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

- local definitions not reduced until needed

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

6.2 Demand-driven programming

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

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

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

6.2 A pipeline

$$
\begin{array}{ll}
& \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ [1\,..\,100]) \\
\Rightarrow & \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ (1 : [2\,..\,100])) \\
\Rightarrow & \mathit{foldl}\ (+)\ 0\ (1 : \mathit{map}\ \mathit{square}\ [2\,..\,100]) \\
\Rightarrow & \mathit{foldl}\ (+)\ 1\ (\mathit{map}\ \mathit{square}\ [2\,..\,100]) \\
\Rightarrow & \mathit{foldl}\ (+)\ 1\ (\mathit{map}\ \mathit{square}\ (2 : [3\,..\,100])) \\
\Rightarrow & \mathit{foldl}\ (+)\ 1\ (4 : \mathit{map}\ \mathit{square}\ [3\,..\,100]) \\
\Rightarrow & \mathit{foldl}\ (+)\ 5\ (\mathit{map}\ \mathit{square}\ [3\,..\,100]) \\
\Rightarrow & \ldots \\
\Rightarrow & \mathit{foldl}\ (+)\ 14\ (\mathit{map}\ \mathit{square}\ [4\,..\,100]) \\
\Rightarrow & \ldots \\
\Rightarrow & 338350
\end{array}
$$

6.2 Another pipeline

- insertion sort

$$
\begin{array}{l}
\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,] \\
\mathit{insertSort}\ [\,]\qquad\quad = [\,] \\
\mathit{insertSort}\ (x : \mathit{xs}) = \mathit{insert}\ x\ (\mathit{insertSort}\ \mathit{xs})
\end{array}
$$

$$
\begin{array}{l}
\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [\,a\,] \to [\,a\,] \\
\mathit{insert}\ a\ [\,]\qquad = [\,a\,] \\
\mathit{insert}\ a\ (b : \mathit{xs}) \\
\quad \mid a \leqslant b \qquad\ \ = a : b : \mathit{xs} \\
\quad \mid \mathit{otherwise} = b : \mathit{insert}\ a\ \mathit{xs}
\end{array}
$$

- minimum

$$
\begin{array}{l}
\mathit{minimum} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to a \\
\mathit{minimum} = \mathit{head} \circ \mathit{insertSort}
\end{array}
$$

- complexity?

6.3 Efficiency

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

6.3 Simplifications

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

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

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

6.3 Accumulating parameters

- improve efficiency by adding an extra parameter
- common use: to remove expensive $++$
- eg naive reverse

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

- specify $\mathit{revCat}\ \mathit{xs}\ \mathit{ys} = \mathit{reverse}\ \mathit{xs} ++ \mathit{ys}$
- then $\mathit{reverse}\ \mathit{xs} = \mathit{revCat}\ \mathit{xs}\ [\,]$, and

$$
\begin{aligned}
&\mathit{revCat} :: [a] \to [a] \to [a] \\
&\mathit{revCat}\ [\,] \qquad\ \ \mathit{ys} = \mathit{ys} \\
&\mathit{revCat}\ (x : \mathit{xs})\ \mathit{ys} = \mathit{revCat}\ \mathit{xs}\ (x : \mathit{ys})
\end{aligned}
$$

(we've seen this program before!)

- eg flattening a tree

6.3 Tupling

- dually, improve efficiency by adding an extra result
- eg naive Fibonacci

$$
\begin{aligned}
\mathit{fib}\ 0 &= 0 \\
\mathit{fib}\ 1 &= 1 \\
\mathit{fib}\ n &= \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)
\end{aligned}
$$

- introduce $\mathit{fibtwo}\ n = (\mathit{fib}\ n, \mathit{fib}\ (n+1))$
- then $\mathit{fib} = \mathit{fst} \circ \mathit{fibtwo}$, and

$$
\begin{aligned}
\mathit{fibtwo}\ 0 &= (0,1) \\
\mathit{fibtwo}\ n &= (b, a+b)\ \mathbf{where}\ (a,b) = \mathit{fibtwo}\ (n-1)
\end{aligned}
$$

- eg building balanced binary tree from list

6.3 Fusion and deforestation

- *deforestation* is the removal of intermediate data structures
- *fusion* combines a recursive computation with another computation
- often combined, eg to fuse producer and consumer

- eg deforest treesort to get quicksort

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\begin{array}{l}
\mathit{grow} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow \mathit{Tree}\ a \\
\mathit{grow}\ [\,] \qquad\quad = \mathit{Empty} \\
\mathit{grow}\ (x : xs) = \mathit{Fork}\ (\mathit{grow}\ \mathit{littles})\ x\ (\mathit{grow}\ \mathit{bigs}) \\
\quad \mathbf{where}\ \mathit{littles} = [\,a \mid a \leftarrow xs, a < x\,] \\
\qquad\qquad\quad \mathit{bigs} \quad = [\,a \mid a \leftarrow xs, a \geqslant x\,]
\end{array}$$

$$\begin{array}{l}
\mathit{inorder} :: \mathit{Tree}\ a \rightarrow [\,a\,] \\
\mathit{inorder}\ \mathit{Empty} \qquad\quad = [\,] \\
\mathit{inorder}\ (\mathit{Fork}\ t\ a\ u) = \mathit{inorder}\ t \mathbin{+\!\!+} [\,a\,] \mathbin{+\!\!+} \mathit{inorder}\ u
\end{array}$$

$$\begin{array}{l}
\mathit{treeSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
\mathit{treeSort} = \mathit{inorder} \circ \mathit{grow}
\end{array}$$

- unfold, fold?
- accumulating parameter?

6.3 Strictness

- recall summing a list:

$$\begin{array}{ll} & \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ [1\,..\,100]) \\ \Rightarrow & \mathit{foldl}\ (+)\ 1\ (\mathit{map}\ \mathit{square}\ [2\,..\,100]) \\ \Rightarrow & \mathit{foldl}\ (+)\ 5\ (\mathit{map}\ \mathit{square}\ [3\,..\,100]) \\ \Rightarrow & \ldots \end{array}$$

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

$$\begin{array}{ll} & \mathit{foldl}\ (+)\ 0\ (\mathit{map}\ \mathit{square}\ [1\,..\,100]) \\ \Rightarrow & \mathit{foldl}\ (+)\ (0 + \mathit{square}\ 1)\ (\mathit{map}\ \mathit{square}\ [2\,..\,100]) \\ \Rightarrow & \mathit{foldl}\ (+)\ ((0 + \mathit{square}\ 1) + \mathit{square}\ 2)\ (\mathit{map}\ \mathit{square}\ [3\,..\,100]) \\ \Rightarrow & \ldots \end{array}$$

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

- judicious mix of outermost and innermost evaluation to force additions (safe, because $+$ is strict in both arguments)
- *seq a b* reduces *a* to WHNF, then returns *b*

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

- same reductions (on strict functions), but in different order

$$
\begin{array}{llcll}
& \mathit{succ}\ (\mathit{succ}\ (8 * 5)) & & & \mathit{strict}\ \mathit{succ}\ (\mathit{strict}\ \mathit{succ}\ (8 * 5))\\
\Rightarrow & \mathit{succ}\ (8 * 5) + 1 & & \Rightarrow & \mathit{strict}\ \mathit{succ}\ (\mathit{strict}\ \mathit{succ}\ 40)\\
\Rightarrow & ((8 * 5) + 1) + 1 & & \Rightarrow & \mathit{strict}\ \mathit{succ}\ (40 + 1)\\
\Rightarrow & (40 + 1) + 1 & & \Rightarrow & \mathit{strict}\ \mathit{succ}\ 41\\
\Rightarrow & 41 + 1 & & \Rightarrow & 41 + 1\\
\Rightarrow & 42 & & \Rightarrow & 42
\end{array}
$$

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

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

6.4 Infinite data structures

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

6.4 Infinite lists

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

6.4 No magic

- can apply functions to infinite data structures

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

- can return finite results

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

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

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

- to interrupt, ctrl-C

6.4 What does it mean?

- essential idea is that infinite data structure is *limit* of series of *approximations*
- eg infinite list

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

is limit of series of approximations

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

6.4 Primes

- recall bounded sequences of primes

$$\begin{array}{l} primes\ m = [\, n \mid n \leftarrow [1\,..\,m], divisors\ n == [1, n]\,] \\ divisors\ n = [\, d \mid d \leftarrow [1\,..\,n], n\ \text{`}mod\text{`}\ d == 0\,] \end{array}$$

- infinite sequence of primes

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

- much more efficient version: *sieve of Eratosthenes*

$$\begin{array}{l} primes = sieve\ [2\,..]\ \mathbf{where} \\ \quad sieve\ (x : xs) = x : sieve\ [\, y \mid y \leftarrow xs, y\ \text{`}mod\text{`}\ x \neq 0\,] \end{array}$$

(write *sieve* as an *unfold*)

6.4 Pythagorean triples

- obvious definition

$$\begin{aligned} \mathit{pyth} = {} & [\,(a,b,c) \\ & \mid a \leftarrow [1\,..\,], b \leftarrow [1\,..\,], c \leftarrow [1\,..\,], \\ & \quad a * a + b * b == c * c\,] \end{aligned}$$

  doesn't work — why?
- instead

$$\begin{aligned} \mathit{pyth} = {} & [\,(a,b,c) \\ & \mid c \leftarrow [1\,..\,], b \leftarrow [1\,..\,c-1], a \leftarrow [1\,..\,b-1], \\ & \quad a * a + b * b == c * c\,] \end{aligned}$$

- then

$$\mathit{pyth} = [\,(3,4,5), (6,8,10), (5,12,13), \ldots$$

6.4 What's the point?

- better abstraction: some real-world entities are infinite
- better modularity: separation of concerns, reuse of components
- provides a model of interaction (but now superseded by monads)
- fun!

Part 7

Reasoning and calculating

7.0 Outline

7.1 Reasoning about programs

- functional programs are just equations
- lazy semantics means that rules of ordinary algebra (substitution of equals for equals) apply
- given

  $$\mathit{three}\ x = 3$$

  can replace 3 anywhere by $\mathit{three}\ x$ for any suitably-typed expression x (even $x = 1\ /\ 0$)
- simple proofs by *equational reasoning*

7.2 Equational reasoning

- equations as definitions intended for evaluation
- …but also useful for reasoning: proofs
- better than testing, because exhaustive
- eg given

$$swap\ (x, y) = (y, x)$$

we can show that *swap*ping twice is the identity:

$$\begin{array}{ll} & swap\ (swap\ (a, b)) \\ = & \quad \{ \text{definition of } swap \} \\ & swap\ (b, a) \\ = & \quad \{ \text{definition of } swap \} \\ & (a, b) \end{array}$$

- program text used as proof rules

7.2 Another simple example

- given

$$
\begin{aligned}
&\mathit{curry}\ f\ a\ b = f\ (a, b)\\
&\mathit{fst}\ (a, b) = a\\
&\mathit{const}\ a\ b = a
\end{aligned}
$$

prove that $\mathit{const} = \mathit{curry}\ \mathit{fst}$:

$$
\begin{aligned}
& \quad \mathit{curry}\ \mathit{fst}\ a\ b\\
= & \qquad \{\ \text{definition of } \mathit{curry}\ \}\\
& \quad \mathit{fst}\ (a, b)\\
= & \qquad \{\ \text{definition of } \mathit{fst}\ \}\\
& \quad a\\
= & \qquad \{\ \text{definition of } \mathit{const}\ \}\\
& \quad \mathit{const}\ a\ b
\end{aligned}
$$

7.3 Proof by induction

- proofs about recursive definitions typically require *induction*
- in order to show that some property $P(xs)$ holds for every finite list xs, it suffices to show
  - base case: $P([\,])$
  - inductive step: if $P(xs)$ holds, then so does $P(x : xs)$ for any x
- this is called *structural induction* (cf mathematical and commonplace induction)
- think of climbing a ladder
- induction is valid for the same reason that recursive definitions are valid: every list is either $[\,]$ or of the form $x : xs$

7.3 Example: map distributes over $\circ$

- to prove $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$ for every finite list xs we use induction on lists
- recall definition of map

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

- base case

$$\begin{aligned} & map\ f\ (map\ g\ [\,]) \\ = \quad & \quad \{ \text{ definition of } map \} \\ & map\ f\ [\,] \\ = \quad & \quad \{ \text{ definition of } map \} \\ & [\,] \\ = \quad & \quad \{ \text{ definition of } map \} \\ & map\ (f \circ g)\ [\,] \end{aligned}$$

- inductive step: assume $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$; then

$$
\begin{array}{ll}
 & map\ f\ (map\ g\ (x : xs)) \\
= & \quad \{ \text{ definition of } map \ \} \\
 & map\ f\ (g\ x : map\ g\ xs) \\
= & \quad \{ \text{ definition of } map \ \} \\
 & f\ (g\ x) : map\ f\ (map\ g\ xs) \\
= & \quad \{ \text{ inductive hypothesis } \} \\
 & f\ (g\ x) : map\ (f \circ g)\ xs \\
= & \quad \{ \text{ definition of } \circ \ \} \\
 & (f \circ g)\ x : map\ (f \circ g)\ xs \\
= & \quad \{ \text{ definition of } map \ \} \\
 & map\ (f \circ g)\ (x : xs)
\end{array}
$$

7.3 Example: associativity of append

- to prove

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

  for all finite lists xs, ys, zs

- recall definition

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

- use induction over xs (why?)
- base case

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

- inductive step: assume $xs \mathbin{+\!\!\!+} (ys \mathbin{+\!\!\!+} zs) = (xs \mathbin{+\!\!\!+} ys) \mathbin{+\!\!\!+} zs$; then

$$
\begin{array}{cl}
 & (x : xs) \mathbin{+\!\!\!+} (ys \mathbin{+\!\!\!+} zs) \\
= & \quad \{ \text{ definition of } \mathbin{+\!\!\!+} \} \\
 & x : (xs \mathbin{+\!\!\!+} (ys \mathbin{+\!\!\!+} zs)) \\
= & \quad \{ \text{ inductive hypothesis } \} \\
 & x : ((xs \mathbin{+\!\!\!+} ys) \mathbin{+\!\!\!+} zs) \\
= & \quad \{ \text{ definition of } \mathbin{+\!\!\!+} \} \\
 & (x : (xs \mathbin{+\!\!\!+} ys)) \mathbin{+\!\!\!+} zs \\
= & \quad \{ \text{ definition of } \mathbin{+\!\!\!+} \} \\
 & ((x : xs) \mathbin{+\!\!\!+} ys) \mathbin{+\!\!\!+} zs
\end{array}
$$

7.3 Structural induction on other datatypes

- exactly the same principle applies for any recursive datatype
- eg for naturals

  $$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

  one has to show
  - base case: $P(Zero)$
  - inductive step: if $P(n)$ holds, then so does $P(Succ\ n)$
- eg for binary trees

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

  one has to show
  - base case: $P(Tip\ a)$ holds for any a
  - inductive step: if $P(t)$ and $P(u)$ hold, then so does $P(Bin\ t\ u)$

7.3 Inductive proofs about infinite structures

- basic structural induction as above works only for finite data structures
- to prove for infinite (and partially defined) data structures too, must also verify second base case $P(\bot)$
- eg $+\!\!+$ is associative on infinite and partial lists too, because

$$
\begin{array}{ll}
 & \bot +\!\!+ (ys +\!\!+ zs) \\
= & \quad \{ +\!\!+ \text{ is strict in its first argument} \} \\
 & \bot \\
= & \quad \{ +\!\!+ \text{ is strict in its first argument} \} \\
 & \bot +\!\!+ zs \\
= & \quad \{ +\!\!+ \text{ is strict in its first argument} \} \\
 & (\bot +\!\!+ ys) +\!\!+ zs
\end{array}
$$

- but $\mathit{filter}\ (\mathit{const}\ \mathit{False})\ xs = [\,]$ holds only for finite xs, because $\mathit{filter}\ (\mathit{const}\ \mathit{False})\ \bot = \bot$

7.3 When induction doesn't apply

- recall

$$\begin{aligned}&\mathit{iterate} :: (a \to a) \to a \to [\,a\,]\\&\mathit{iterate}\; f\; x = x : \mathit{iterate}\; f\; (f\; x)\end{aligned}$$

- expect $\mathit{map}\; f\; (\mathit{iterate}\; f\; x) = \mathit{iterate}\; f\; (f\; x)$
- how to prove? no argument over which to induct
- *approximation lemma*: to prove $xs = ys$, suffices to show

$$\mathit{approx}\; n\; xs = \mathit{approx}\; n\; ys$$

  for every n (by induction), where

$$\begin{aligned}&\mathit{approx}\; (n+1)\; [\,]\; &&= \bot\\&\mathit{approx}\; (n+1)\; (x:xs) &&= x : \mathit{approx}\; n\; xs\end{aligned}$$

7.4 Program synthesis

- derive program from specification
- eg from specification $revCat\ xs\ ys = reverse\ xs \mathbin{+\!\!+} ys$, synthesize

$$
\begin{array}{ll}
 & revCat\ [\,]\ ys \\
= & reverse\ [\,] \mathbin{+\!\!+} ys \\
= & [\,] \mathbin{+\!\!+} ys \\
= & ys
\end{array}
\qquad
\begin{array}{ll}
 & revCat\ (x : xs)\ ys \\
= & reverse\ (x : xs) \mathbin{+\!\!+} ys \\
= & (reverse\ xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys \\
= & reverse\ xs \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} ys) \\
= & reverse\ xs \mathbin{+\!\!+} (x : ys) \\
= & revCat\ xs\ (x : ys)
\end{array}
$$

- eg given *add* on *Nat*, synthesize *sub* such that

$$(m \text{ `}add\text{` } n) \text{ `}sub\text{` } n = m$$

7.5 Calculating programs

- sometimes the specification is a (not efficient enough) program
- given enough laws, synthesis may then be a linear calculation
- eg maximum segment sum problem

$$\begin{aligned}&\mathit{mss} :: [\mathit{Integer}] \to \mathit{Integer}\\&\mathit{mss} = \mathit{maxlist} \circ \mathit{map}\ \mathit{sum} \circ \mathit{segs}\end{aligned}$$

$$\begin{aligned}&\mathit{segs} :: [a] \to [[a]]\\&\mathit{segs} = \mathit{concat} \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails}\end{aligned}$$

$$\begin{aligned}&\mathit{sum} :: (\mathit{Num}\ a) \Rightarrow [a] \to a\\&\mathit{sum} = \mathit{foldl}\ (+)\ 0\end{aligned}$$

$$\begin{aligned}&\mathit{maxlist} :: (\mathit{Ord}\ a) \Rightarrow [a] \to a\\&\mathit{maxlist} = \mathit{foldr1}\ \mathit{max}\end{aligned}$$

7.5 Laws

- polymorphism of *concat*:

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

- 'bookkeeping law': for associative *op*,

$$foldr1\ op \circ concat = foldr1\ op \circ map\ (foldr1\ op)$$

- *map* distributes over composition:

$$map\ (f \circ g) = map\ f \circ map\ g$$

- Horner's rule: if g is associative with unit e, and g distributes over f, then

$$foldr1\ f \circ scanl\ g\ e = foldr\ h\ e$$

where $h\ x\ y = f\ e\ (g\ x\ y)$

7.5 Calculation

$$
\begin{array}{cl}
 & \mathit{mss} \\
= & \quad \{ \text{ definition of } \mathit{mss} \} \\
 & \mathit{maxlist} \circ \mathit{map}\ \mathit{sum} \circ \mathit{segs} \\
= & \quad \{ \text{ definition of } \mathit{segs} \} \\
 & \mathit{maxlist} \circ \mathit{map}\ \mathit{sum} \circ \mathit{concat} \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & \quad \{ \text{ polymorphism of } \mathit{concat} \} \\
 & \mathit{maxlist} \circ \mathit{concat} \circ \mathit{map}\ (\mathit{map}\ \mathit{sum}) \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & \quad \{ \text{ bookkeeping law } \} \\
 & \mathit{maxlist} \circ \mathit{map}\ \mathit{maxlist} \circ \mathit{map}\ (\mathit{map}\ \mathit{sum}) \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & \quad \{ \mathit{map} \text{ distributes over composition } \} \\
 & \mathit{maxlist} \circ \mathit{map}\ (\mathit{maxlist} \circ \mathit{map}\ \mathit{sum} \circ \mathit{inits}) \circ \mathit{tails}
\end{array}
$$

$$
\begin{aligned}
& \mathit{maxlist} \circ \mathit{map}\ (\mathit{maxlist} \circ \mathit{map}\ \mathit{sum} \circ \mathit{inits}) \circ \mathit{tails} \\
= \quad & \quad \{\ \text{definition of } \mathit{scanl}\ \} \\
& \mathit{maxlist} \circ \mathit{map}\ (\mathit{maxlist} \circ \mathit{scanl}\ (+)\ 0) \circ \mathit{tails} \\
= \quad & \quad \{\ \text{Horner's rule: let } h\ x\ y = 0\ \text{`}\mathit{max}\text{`}\ (x + y)\ \} \\
& \mathit{maxlist} \circ \mathit{map}\ (\mathit{foldr}\ h\ 0) \circ \mathit{tails} \\
= \quad & \quad \{\ \text{definition of } \mathit{scanr}\ \} \\
& \mathit{maxlist} \circ \mathit{scanr}\ h\ 0
\end{aligned}
$$

Part 8

Monads

8.0 Outline

8.1 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?

8.1 Gedankenexperiment

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

$$\mathit{putStr} :: \mathit{String} \rightarrow ()$$

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

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

- and of this one?

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

- if we noticed different effects, then we would no longer be able to replace equals by equals!

8.1 Monadic IO

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

$$\mathit{putStr} :: \mathit{String} \rightarrow \mathit{IO}\ ()$$

- *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

8.1 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
```

8.2 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

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

8.2 Example

- a simple interactive program

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

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

8.2 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 pattern

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

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

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

8.2 More IO operations

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

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

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

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

and many more . . .

8.3 Do notation

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

$$
\begin{array}{ll}
\mathbf{do}\ \{m\} & = m \\
\mathbf{do}\ \{x \leftarrow m; ms\} & = m \ggg\!= \lambda x \rightarrow \mathbf{do}\ \{ms\} \\
\mathbf{do}\ \{m; ms\} & = m \ggg\!= \lambda\_ \rightarrow \mathbf{do}\ \{ms\} \\
\mathbf{do}\ \{\mathbf{let}\ ds; ms\} & = \mathbf{let}\ ds\ \mathbf{in}\ \mathbf{do}\ \{ms\}
\end{array}
$$

where x 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.

8.3 Examples: character I/O

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

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

8.3 File I/O

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

8.3 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)
```

8.4 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

8.4 An evaluator

Here's a simple datatype of terms:

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

$$\begin{aligned} &good, bad :: Expr \\ &good = Div\ (Lit\ 7)\ (Div\ (Lit\ 4)\ (Lit\ 2)) \\ &bad\ \ \ = Div\ (Lit\ 7)\ (Div\ (Lit\ 2)\ (Lit\ 4)) \end{aligned}$$

...and an evaluation function:

$$\begin{aligned} &eval :: Expr \rightarrow Integer \\ &eval\ (Lit\ n) \qquad = n \\ &eval\ (Div\ d\ e) = eval\ d\ \text{`}div\text{`}\ eval\ e \end{aligned}$$

8.4 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 d e) =
  case evalE d of
    Raise x    -> Raise x
    Result m -> case evalE e of
                  Raise x   -> Raise x
                  Result n ->
                    if n == 0 then Raise "division by zero"
                              else  Result (m `div` n)
```

8.4 Counting

We could instrument the evaluator to count evaluation steps:

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

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

8.4 Tracing

…or to trace the evaluation steps:

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

$$
\begin{aligned}
&\mathit{evalT} :: \mathit{Expr} \to \mathit{Trace}\ \mathit{Integer}\\
&\mathit{evalT}\ (\mathit{Lit}\ n) \quad = T\ (\mathit{line}\ (\mathit{Lit}\ n)\ n, n)\\
&\mathit{evalT}\ (\mathit{Div}\ d\ e) = \textbf{let}\\
&\qquad\qquad\qquad\qquad T\ (s,\ m) = \mathit{evalT}\ d\\
&\qquad\qquad\qquad\qquad T\ (s', n) = \mathit{evalT}\ e\\
&\qquad\qquad\qquad\qquad p = m\ \text{‘}\mathit{div}\text{‘}\ n\\
&\qquad\qquad\qquad\ \ \textbf{in}\ T\ (s + s' + \mathit{line}\ (\mathit{Div}\ d\ e)\ p, p)
\end{aligned}
$$

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

8.4 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?

8.5 The monad type class

These are the methods of a type class:

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

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

8.5 Original evaluator, monadically

$$
\begin{aligned}
&\mathit{evalM} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Expr} \rightarrow m\ \mathit{Integer} \\
&\mathit{evalM}\ (\mathit{Lit}\ n) \quad\ \ = \mathit{return}\ n \\
&\mathit{evalM}\ (\mathit{Div}\ d\ e) = \mathit{evalM}\ d \gg\!= \lambda m \rightarrow \\
&\qquad\qquad\qquad\qquad\ \ \mathit{evalM}\ e \gg\!= \lambda n \rightarrow \\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\ (m\ `\mathit{div}`\ n)
\end{aligned}
$$

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

8.5 Original evaluator, using do notation

$$
\begin{array}{l}
\mathit{evalM} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Expr} \rightarrow m\ \mathit{Integer} \\
\mathit{evalM}\ (\mathit{Lit}\ n) \quad\;\; = \mathbf{do}\ \mathit{return}\ n \\
\mathit{evalM}\ (\mathit{Div}\ d\ e) = \mathbf{do}\ m \leftarrow \mathit{evalM}\ d \\
\qquad\qquad\qquad\qquad\quad\;\; n \;\leftarrow \mathit{evalM}\ e \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{return}\ (m\ \text{`}\mathit{div}\text{`}\ n)
\end{array}
$$

8.5 The exception instance

Exceptions instantiate the class:

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

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

The effect-specific behaviour is to throw an exception:

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

8.5 Exceptional evaluator, monadically

$$
\begin{array}{l}
\mathit{evalE} :: \mathit{Expr} \rightarrow \mathit{Exc}\ \mathit{Integer} \\
\mathit{evalE}\ (\mathit{Lit}\ n) \quad\ \ = \mathbf{do}\ \mathit{return}\ n \\
\mathit{evalE}\ (\mathit{Div}\ d\ e) = \mathbf{do}\ m \leftarrow \mathit{evalE}\ d \\
\qquad\qquad\qquad\qquad\quad\ n \leftarrow \mathit{evalE}\ e \\
\qquad\qquad\qquad\qquad\quad\ \mathbf{if}\ n == 0\ \mathbf{then}\ \mathit{throw}\ \texttt{"division by zero"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{else}\ \ \mathit{return}\ (m\ \text{`}\mathit{div}\text{`}\ n)
\end{array}
$$

8.5 The counter instance

Counters instantiate the class:

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

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

The effect-specific behaviour is to increment the count:

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

8.5 Counting evaluator, monadically

$$
\begin{array}{lll}
\mathit{evalC} :: \mathit{Expr} \to \mathit{Counter}\ \mathit{Integer} \\
\mathit{evalC}\ (\mathit{Lit}\ n) & = \mathbf{do} & \mathit{tick} \\
 & & \mathit{return}\ n \\
\mathit{evalC}\ (\mathit{Div}\ d\ e) & = \mathbf{do} & \mathit{tick} \\
 & & m \leftarrow \mathit{evalC}\ d \\
 & & n\ \ \leftarrow \mathit{evalC}\ e \\
 & & \mathit{return}\ (m\ \text{`}\mathit{div}\text{`}\ n)
\end{array}
$$

8.5 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:

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

8.5 Tracing evaluator, monadically

$$
\begin{aligned}
&\mathit{evalT} :: \mathit{Expr} \rightarrow \mathit{Trace}\ \mathit{Integer} \\
&\mathit{evalT}\ (\mathit{Lit}\ n) \quad\ \ = \textbf{do}\ \mathit{trace}\ (\mathit{line}\ (\mathit{Lit}\ n)\ n) \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ n \\
&\mathit{evalT}\ (\mathit{Div}\ d\ e) = \textbf{do}\ m \leftarrow \mathit{evalT}\ d \\
&\qquad\qquad\qquad\qquad\quad\ \ n\ \leftarrow \mathit{evalT}\ e \\
&\qquad\qquad\qquad\qquad\quad\ \ \textbf{let}\ p = m\ \text{`}\mathit{div}\text{`}\ n \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{trace}\ (\mathit{line}\ (\mathit{Div}\ d\ e)\ p) \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ p
\end{aligned}
$$

8.5 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.

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

8.6 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

END

I hope you've enjoyed the journey!