

Introduction to Functional Programming



Jeremy Gibbons
University of Oxford

Part 0

Aims and objectives

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 become functional programmers)

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

1. value-oriented programming
2. lists and other algebraic datatypes
3. types, polymorphism, type classes
4. monads and applicative functors
5. laziness, infinite data structures, co-programming

Concurrently, John Hughes will be talking about *property-based testing* using *QuickCheck*, illustrating most of these ideas.

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`
 - ▶ execution order is important
$$i := i + 1 ; a := a * i \neq a := a * i ; i := i + 1$$
- 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)
```

- 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 Brevity

- 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

Part 1

Value-oriented programming

1.1 Programs

We'll be using the *GHCi* implementation of *Haskell*.
A *script* is essentially just a collection of definitions:

```
-- 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
```

Bigger *programs* are divided into *modules*.

A standalone program includes a 'main' expression.

1.1 REPL

Value-oriented programming accommodates a *read-eval-print loop* (REPL):

Prelude> :l sample.hs

Ok, one module loaded.

Main> 42

42

Main> 6 × 7

42

Main> square 7 – smaller (3, 4) – square (smaller (2, 3))

42

Main> square 3141592654

9869604403666763716

(I typed the things in green.)

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

smaller::(Integer, Integer) → Integer

smaller (x, y)

= **if**

x ≤ y

then

x

else

y

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

1.2 Evaluation

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

```
square (3 + 4)
⇒      { definition of + }
square 7
⇒      { definition of square }
7 × 7
⇒      { definition of × }
49
```

- 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:

square (3 + 4)

⇒ { definition of *square* }

(3 + 4) × (3 + 4)

⇒ { definition of + }

7 × (3 + 4)

⇒ { definition of + }

7 × 7

⇒ { definition of × }

49

- 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

three :: Integer → Integer

three x = 3 -- NB argument unused

infinity :: Integer

infinity = 1 + infinity

- two different evaluation orders:

three infinity

⇒ { defn of *infinity* }

three (1 + *infinity*)

⇒ { defn of *infinity* }

three (1 + (1 + *infinity*))

⇒ ...

three infinity

⇒ { defn of *three* }

3

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

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 undefined (\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 → Integer*
- in general, $f :: A \rightarrow B$ indicates that function f takes arguments of type A and returns results of type B
- *apply* function to argument: fx
- 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 fx

1.3 Lambda

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

1.3 Declaration vs expression style

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

$$\begin{aligned} \textit{quad} &:: \text{Integer} \rightarrow \text{Integer} \\ \textit{quad } x &= \textit{square } x \times \textit{square } x \end{aligned}$$

- *expression style*: emphasising the use of expressions

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

- 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)
- eg these two functions are equal:

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

- the important thing about a function is its mapping from arguments to results
- *intentional* properties (eg how a mapping is described) are irrelevant

1.3 Currying

- replace single structured argument by several simpler ones

$\text{add} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$

$\text{add } (x, y) = x + y$

$\text{add}' :: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$

$\text{add}' x y = x + y$

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

1.4 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: $17 \text{'mod'} 10$
- make symbolic operator prefix (and curried) with parentheses: $(+)\ 3\ 4$
- thus, *add*' = $(+)$
- extend notion to include one argument too: *sectioning*
- eg $(1/)$ is the reciprocal function, (>0) is the positivity test

1.4 Associativity

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

$x \otimes y \otimes z$

- 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

1.4 Association

- some operators are not associative ($-$, $/$, \uparrow)
- to disambiguate without parentheses, operators may *associate* to the left or 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:
 $\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$ means $\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$

1.4 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, we use *precedence* (binding power)
- eg \times has higher precedence (binds more tightly) than $+$

infixl 7 \times

infixl 6 $+$

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

1.4 Composition

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

$$\begin{aligned} (\circ) &:: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \\ f \circ g &= \lambda x \rightarrow f(g x) \end{aligned}$$

- eg function *square* \circ *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)

1.5 Definitions

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

```
here :: String  
here = "Oxford"
```

- 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

1.5 Conditional definitions

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

$$\begin{aligned} \text{smaller} &:: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\ \text{smaller } (x, y) &= \text{if } x \leq y \text{ then } x \text{ else } y \end{aligned}$$

- could also use *guarded equations*:

$$\begin{aligned} \text{smaller} &:: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\ \text{smaller } (x, y) & \\ | \quad x \leq y &= x \\ | \quad x > y &= y \end{aligned}$$

- 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...

1.5 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

```
day :: Integer → String
day 1 = "Saturday"
day 2 = "Sunday"
day _ = "Weekday"
```

- 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

1.5 Local definitions

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

qroots::(Float, Float, Float) → (Float, Float)

qroots(a, b, c) = $((-b - sd) / (2 \times a), (-b + sd) / (2 \times a))$

where $sd = \text{sqrt}(b \times b - 4 \times a \times c)$

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

demo::Integer → Integer → Integer

demo $x y = (a + 1) \times (b + 2)$

where $a = x - y$

$b = x + y$

(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



1.5 let-expressions

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

```
demo :: Integer → Integer → Integer
demo x y = let a = x - y
            b = x + y
            in (a + 1) × (b + 2)
```

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

1.6 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

1.7 Functions as arguments and results

- functions may be taken as arguments

applyToPair :: (Integer → Integer) → (Integer, Integer) → (Integer, Integer)
applyToPair f (x, y) = (fx, fy)

- functions may also be returned as results

addOrMul :: Bool → (Integer → Integer → Integer)
addOrMul b = if b then (+) else (×)

- those are rather artifical examples...
- partial application
- currying
- function composition (again)
- lots more, especially tomorrow: *parametrizable program schemes*

1.7 Partial application

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

1.7 Currying

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

$\text{add} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$
 $\text{add } (x, y) = x + y$

$\text{add}' :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$
 $\text{add}' x y = x + y$

- 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:: $((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$
curry f = $\lambda a\ b \rightarrow f(a, b)$

uncurry:: $(a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$
uncurry g = $\lambda(a, b) \rightarrow g\ a\ b$

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

flip:: $(a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$
flip f = $\lambda b\ a \rightarrow f\ a\ b$

1.7 Function composition

- recall function composition

$$\begin{aligned} (\circ) &:: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \rightarrow \text{Integer} \rightarrow \text{Integer} \\ (f \circ g) x &= f(g x) \end{aligned}$$

- takes two functions that ‘meet in the middle’ and an argument to one; returns the result from the other
- equivalently, type $(\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer})$
- takes two functions, glues them together to form a third

1.7 Repeated composition

- double application: eg *twice square* 3 = 81

$$\begin{aligned} \textit{twice} &:: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \\ \textit{twice } f &= f \circ f \end{aligned}$$

- generalize: eg *iter* 4 (2 \times) 1 = 2 \times 2 \times 2 \times 1

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

1.8 Reasoning about programs

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

three $x = 3$

can replace 3 anywhere by *three* x for any suitably-typed expression x
(even $x = 1 \text{ `div' } 0$)

- simple proofs by *equational reasoning*

1.9 Equational reasoning

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

$$\text{swap}(x, y) = (y, x)$$

we can show that *swapping twice* is the identity:

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

- program text used as proof rules

1.9 Another simple example

- given

$$\text{curry } f \ a \ b = f(a, b)$$

$$\text{fst } (a, b) = a$$

$$\text{const } a \ b = a$$

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

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