Programming with Math and Logic

an invitation to functional programming

Ed Morehouse

Wesleyan University

The Plan

The What and Why of Functional Programming

Imperative vs. Functional

In **imperative programming languages** (e.g. C, Python, Java) there are,

terms: expressions that can be evaluated to a *value*
(e.g. "hello world", 3 + 4, fact (4))

statements: expressions that change the *state* of an executing program
(e.g. x++, for ..., my_list.append("unicorn"))

In **functional programming languages**, we program (primarily) using just terms.

Why Functional Programming?

Q) Why would we want to do that?

A) So we can reason about programs in a mathematical way.

This facilitates:

correctness
: We can prove that our programs behave as intended.

optimization
: A compiler can substitute terms in a program with other terms that compute the same values more efficiently.

parallelism
: A compiler can understand which parts of a program are *independent*, and have those parts run concurrently.

But is it Worth it?

Q) That sounds okay. But I've heard that functional programming is hard to understand. Is that true?

A) Not necessarily, but it is a bit different from imperative programming.

If you're used to the way we reason in math classes, it may feel very familiar.

Today we'll highlight some of the main ideas and benefits of functional programming through examples.

For concreteness, all examples are in the functional language **Haskell**[1].

[1] http://haskell.org/

But is it Worth it?

Q) That sounds okay. But I've heard that functional programming is hard to understand. Is that true?

A) Not necessarily, but it is a bit different from imperative programming.

If you're used to the way we reason in math classes, it may feel very familiar.

Today we'll highlight some of the main ideas and benefits of functional programming through examples.

For concreteness, all examples are in the functional language **Haskell**[1].

[1] http://haskell.org/

Computing with Terms

Defining Terms

Functional programming is all about *terms*:
defining them, *combining* them and *evaluating* them.

Terms are like mathematical expressions.
They don't change identity during the course of program execution.

So we give **definitions** rather than **assignments**:

```
x  =  "foo"  --  a string
y  =  "bar"  --  another string

z  =  x ++ y  --  concatenation from the standard library
```

Now z is "foobar" forever. [2]

[2] in this *scope*, anyway.

Functions are Terms Too

Functional programming is also all about functions.

We can **define a function** by saying how it acts on its arguments using **formal parameters**.

Here's a boring but important function, called the **identity function**:

```
id  x   =   x
```

It does nothing at all to its argument.

Multiple Arguments

Functions can take multiple arguments:

```
add x y  =  x + y  --  addition from the standard library
mul x y  =  x * y  --  likewise, multiplication
```

We juxtapose the arguments after the function name without any "(– , ... , –)" noise.

This is called **currying**.

Applying Functions

To **apply a function** we follow its name by its **arguments**:

```
three  =  add 1 2
twelve  =  mul three (add 2 2)

still_twelve  =  id (id twelve)
```

Parentheses are used only for association.

Partial Application

We can apply functions to fewer arguments than they expect.

This is called **partial application**.

It gives a new function that expects the remaining arguments:

```
double  =  mul 2
```

So the results of functions can themselves be functions.

Function Literals

Functional languages let us write **function literals**.

The function we get by taking the variable x as a formal parameter in the term t is usually written as "$\lambda x\ .\ t$". [3]

But Haskell uses the notation "\ x → t".

```
id' = \ x → x  -- just like id
```

For multiple arguments, we can write "\ x y → t" as shorthand for "\ x → (\ y → t)".

[3] from the *original* model of universal computation, Church's λ-calculus

Higher-Order Functions

In addition to *returning functions as results*, functions may also *take functions as arguments*. These are called **higher-order functions**.

An important higher-order functions is **function composition**:

```
compose f g = \ x → g (f x)
```

it first applies f to its argument and then applies g to the result.

In math, this is usually written “$g \circ f$”, which we can do in Haskell using **infix notation**:

```
g ∘ f = compose f g
```

Recursive Functions

A **recursive function** may call itself in the process of computing its result.

But we must be careful that the recursive calls don't go on forever.

So the arguments to a recursive calls must be "smaller". [4]

Every mathematician's favorite recursive function is the **factorial**:

```
fact 0  =  1
fact n  =  n * fact (n - 1)
```

This is an example of definition by **pattern-matching**.

[4] in some sense

Example: Function Iteration

Using these ideas, let's write a higher-order recursive function, iterate, that takes a function and a number and composes the argument function with itself the specified number of times.

We can pattern match on the number of iterations.

How do we iterate a function zero times?

```
iterate₀ f 0  =  (?)
iterate₀ f n  =  (?)
```

Example: Function Iteration

Using these ideas, let's write a higher-order recursive function, iterate, that takes a function and a number and composes the argument function with itself the specified number of times.

We can pattern match on the number of iterations.

and how do we iterate a function one more time?

```
iterate₁ f 0  =  id
iterate₁ f n  =  (?)  --  using iterate f (n - 1)
```

Example: Function Iteration

Using these ideas, let's write a higher-order recursive function, iterate, that takes a function and a number and composes the argument function with itself the specified number of times.

We can pattern match on the number of iterations.

That's it, we're done!

```
iterate  f 0  =  id
iterate  f n  =  compose f (iterate f (n - 1))
```

Classifying Terms with Types

Types and Terms

Many functional programming languages (including Haskell) are *typed*.

Types *classify* terms.

We write, "`t : A`" to express that "`t` is a term of type `A`". [5]

We've already met a few types, like *numbers*, *strings* and *functions*.

Haskell can figure out the types of many terms by itself.

But it's helpful to write types out explicitly.

So from now on whenever we define a term we'll write its type.

[5] idiosyncratically, Haskell uses "`::`" for "`:`"

Function Types

The type of a function depends on the types of its argument and result.

We write the type of a function with argument type a and result type b as “a → b”.

By *currying* and *partial application*, a function with two arguments of types a and b, and result type c has type a → (b → c).

Conventionally, → associates to the right, so we can drop the parentheses and just write “a → b → c”.

Function Types

The type of a function depends on the types of its argument and result.

We write the type of a function with argument type a and result type b as "$a \to b$".

By *currying* and *partial application*, a function with two arguments of types a and b, and result type c has type $a \to (b \to c)$.

Conventionally, $\to$ associates to the right, so we can drop the parentheses and just write "$a \to b \to c$".

Polymorphism

Some functions don't examine their arguments at all.

They are called (parametrically) **polymorphic**.
(they are functions *parameterized* by a type).

Since id does nothing to its argument, it doesn't care about its type.

So id has type:

$$\text{id} :: \forall\ a\ .\ a \to a$$

Since compose just sticks two functions together, it cares only that the *result* of the first be the same type as the *argument* of the second.

So compose has type:

$$\text{compose} :: \forall\ a\ b\ c\ .\ (a \to b) \to (b \to c) \to (a \to c)$$

What is the type of iterate ?

Polymorphism

Some functions don’t examine their arguments at all.

They are called (parametrically) **polymorphic**.
(they are functions *parameterized* by a type).

Since id does nothing to its argument, it doesn’t care about its type.

So id has type:

$$\text{id} :: \forall\ a\ .\ a \to a$$

Since compose just sticks two functions together, it cares only that
the *result* of the first be the same type as the *argument* of the second.

So compose has type:

$$\text{compose} :: \forall\ a\ b\ c\ .\ (a \to b) \to (b \to c) \to (a \to c)$$

What is the type of iterate ?

Algebraic Types

Most types in functional programming languages are *inductively defined*.

They are called **algebraic types**.

An algebraic type has a fixed set of **constructors**. [6]

The constructors determine all the possible ways to create **values** (i.e. basic terms) of the type.

Each constructor is like a *function* whose result type is the type in question.

But this function doesn't actually *do* anything!

It's just a labeled container that stores its arguments.

Let's look at some examples.

[6] *Caution*: these are not the same "constructors" you may know from object-oriented languages.

Basic Types

The type of **Booleans** has just two constuctors:

```
data  Bool  ::  *   where
   True   ::   Bool
   False   ::   Bool
```

It also has just two values.

The type of **natural numbers** also has two constructors,
but one takes a natural number as argument:

```
data  Nat  ::  *   where
   Zero   ::   Nat
   Succ   ::   Nat  →  Nat
```

This is the *inductive definition of the natural numbers*.
It's the reason why the *principle of mathematical induction* works.

Type Constructors

Types can be parameterized by types too.

We build such types using **type constructors**.

These are like functions from types to types.

The Maybe type constructor represents the possibility of having “no value”:

```
data  Maybe a :: *  where
  Nothing  ::  Maybe a  -- no a for you!
  Just  ::  a → Maybe a  -- here, have an a
```

Maybe a is like a box that may or may not contain an a. (*Surprise!*)

With Maybe types, we don’t need *null pointers*[7].

[7] Tony Hoare’s “billion dollar mistake”

Lists

The List type constructor lets us build linked lists:

```
data  List a :: *  where
  Nil  ::  List a  -- an empty list
  Cons  ::  a → List a → List a  -- a head and tail
```

Haskell has special notation for lists:

- the type constructor List is written as “[]”,
- the type List a is written as “[a]”,
- the (term) constructor Nil is written as “[]” and
- the (term) constructor Cons is written infix as “:”. [8]

We may also write list literals like, “[1 , 2 , 3]” for Cons 1 (Cons 2 (Cons 3 Nil)).

[8] which is why Haskell has to use “::” for the typing relation

Concatenating Lists

We can **concatenate** two lists over the same type into one with a function, cat.

For example, cat [1 , 2] [3 , 4] should compute to [1 , 2 , 3 , 4].

How do we concatenate the empty list to a list?

```
cat₀ :: ∀ a . [a] → [a] → [a]
cat₀ [] ys = (?)
cat₀ (x : xs) ys = (?)
```

Concatenating Lists

We can **concatenate** two lists over the same type into one with a function, cat.

For example, `cat [1 , 2] [3 , 4]` should compute to `[1 , 2 , 3 , 4]`.

and how do we concatenate a longer list to a list?

```
cat₁ :: ∀ a . [a] → [a] → [a]
cat₁ [] ys = ys
cat₁ (x : xs) ys = (?) -- using cat xs ys
```

Concatenating Lists

We can **concatenate** two lists over the same type into one with a function, cat.

For example, cat [1 , 2] [3 , 4] should compute to [1 , 2 , 3 , 4].

That's it, we're done!

```
cat  ::  ∀ a . [a] → [a] → [a]
cat  [] ys  =  ys
cat  (x : xs) ys  =  x : (cat xs ys)
```

A Haskell string is just a list of characters,
and the "++" operator is just cat written infix.

Mapping over Lists

We can **map** a function over the elements of a list with a *higher-order function*, map.

For example, map even [1 , 2 , 3] should compute to [False , True , False].

How do we map a function over the empty list?

```
map₀ :: ∀ a b . (a → b) → [a] → [b]
map₀ f []  = (?)
map₀ f (x : xs) = (?)
```

Mapping over Lists

We can **map** a function over the elements of a list with a *higher-order function*, map.

For example, map even [1 , 2 , 3] should compute to [False , True , False].

and how do we map a function over a longer list?

```
map₁ :: ∀ a b . (a → b) → [a] → [b]
map₁ f []  =  []
map₁ f (x : xs)  =  (?)  -- using map f xs
```

Mapping over Lists

We can **map** a function over the elements of a list with a *higher-order function*, map.

For example, map even [1 , 2 , 3] should compute to [False , True , False].

That's it, we're done!

```
map  ::  ∀ a b . (a → b) → [a] → [b]
map  f  []  =  []
map  f  (x : xs)  =  (f x) : (map f xs)
```

In imperative languages, this is usually done with a for loop.

Note that map is parallelizable: the function can be applied to each list element independently.

Abstracting Types using Interfaces

Type Classes

When we program we often want to take an abstract approach:
we care more about *behavior* than about *structure*.

In object-oriented programming we do this using **interfaces**:
collections of methods that a type must implement.

In functional programming, interfaces are called **type classes**.

For example, the Show class provides string representations for terms:

```
class Show (a :: *) where
  show :: a → String
```

To implement an interface, we declare a type to be an **instance** of the class and define the promised functionality:

```
instance Show Bool where
  show :: Bool → String
  show True = "True"
  show False = "False"
```

Using Type Classes

Type classes let us write functions that work with any type that's in the class.

This is sometimes called **ad-hoc polymorphism**
(in contrast to *parametric polymorphism*).

We express the dependence on class membership using **constraints**:

```
speak_up  ::  Show a ⇒ a → String
speak_up x  =
  "Hi, I'm " ++ (show x) ++ ". Pleased to meet you!"
```

speak_up has type a → String under the *constraint* that type a is an instance of class Show.

That's how we know it's safe to call function show on x.

Algebraic Type Classes

There are type classes for many common programming tasks, like:

- serialization (Show),
- deserialization (Read),
- equality testing (Eq),
- linear order testing (Ord),
- arithmetic operations (Num), etc.

And you're free to define your own.

But some of the most powerful abstractions in functional programming come from a field of math called **algebra**.

Monoids

Intuitively, monoids are structures that let us *combine* multiple things together where the order of combining doesn't matter. [9]

Formally, a **monoid** is a set together with a binary combining operation, $- \cdot -$, and element, ε, that satisfies the **monoid laws**,

(associative law) the order of combining doesn't matter:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

(unit law) ε is "neutral" for combining:

$$\varepsilon \cdot a = a = a \cdot \varepsilon$$

For example, numbers are monoids under both $+$ and $\times$.
What are the respective units?

[9] But the order of the things being combined may well matter!

The Monoid Type Class

There is a Monoid type class that lets us do combining in a very general way.

For example, lists are monoids.

What should be the combining operation for lists?

```
[1 , 2] · [3 , 4] = (?)
```

The Monoid Type Class

There is a Monoid type class that lets us do combining in a very general way.

For example, lists are monoids.

What should be the combining operation for lists?

[1 , 2] · [3 , 4] = [1 , 2 , 3 , 4] = [1 , 2] (?) [3 , 4]

The Monoid Type Class

There is a Monoid type class that lets us do combining in a very general way.

For example, lists are monoids.

What should be the combining operation for lists?

$[1, 2] \cdot [3, 4] = [1, 2, 3, 4] = [1, 2] +\!\!+ [3, 4]$

The Monoid Type Class

There is a Monoid type class that lets us do combining in a very general way.

For example, lists are monoids.

What should be the combining operation for lists?

$[1, 2] \cdot [3, 4] = [1, 2, 3, 4] = [1, 2] +\!\!+ [3, 4]$

The combining operation is concatenation.

What should be the unit?

$(?) \cdot [1, 2, 3] = [1, 2, 3] = [1, 2, 3] \cdot (?)$

The Monoid Type Class

There is a Monoid type class that lets us do combining in a very general way.

For example, lists are monoids.

What should be the combining operation for lists?

$[1, 2] \cdot [3, 4] = [1, 2, 3, 4] = [1, 2] +\!\!+ [3, 4]$

The combining operation is concatenation.

What should be the unit?

$[] \cdot [1, 2, 3] = [1, 2, 3] = [1, 2, 3] \cdot []$

The unit is the empty list.

Since $+\!\!+$ is a monoid operation on lists, the compiler can safely optimize concatenations as long as it obeys the *monoid laws*.

Functors

Another powerful abstraction is the idea of a functor.

Functors let us to apply *type constructors* to *functions* as well as to types.

```
class Functor (f :: * → *) where
  fmap :: ∀ a b . (a → b) → f a → f b
```

The higher-order function fmap, “lifts” a function into a type constructor.

For a type constructor to be a functor, it must obey the **functor laws**,

(identity law) fmap preserves identity:

$$\text{fmap id} = \text{id}$$

(composition law) fmap preserves composition:

$$\text{fmap } (g \circ f) = \text{fmap } g \circ \text{fmap } f$$

Functor Instances

We've already met two functors: Maybe and List.

We've also met the fmap instance for List.

Can you guess what it is based on its type?

```
instance  Functor []  where
  fmap  ::   ∀ a b . (a → b) → [a] → [b]
  fmap  =  (?)
```

Functor Instances

We've already met two functors: Maybe and List.

We've also met the fmap instance for List.

That's right, it's map!

```
instance Functor [] where
  fmap :: ∀ a b . (a → b) → [a] → [b]
  fmap = map
```

But in order to prove it, we must check the functor laws:

$$\text{map id} = \text{id} \qquad \text{and} \qquad \text{map}\ (g \circ f) = \text{map}\ g \circ \text{map}\ f$$

This is not too difficult.

List Functoriality

$$
\begin{aligned}
& \mathrm{map\ id\ []} \\
=\; & [\text{definition map}] \\
& [] \\
=\; & [\text{definition id}] \\
& \mathrm{id\ \ []}
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{map\ id\ (x\ :\ xs)} \\
=\; & [\text{definition map}] \\
& \mathrm{id\ \ x\ :\ map\ id\ xs} \\
=\; & [\text{IH: map id xs} = \text{id xs}] \\
& \mathrm{id\ \ x\ :\ id\ \ xs} \\
=\; & [\text{definition id}] \\
& \mathrm{x\ :\ xs} \\
=\; & [\text{definition id}] \\
& \mathrm{id\ \ (x\ :\ xs)}
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{map\ (g \circ f)\ \ []} \\
=\; & [\text{definition map}] \\
& [] \\
=\; & [\text{definition map}] \\
& \mathrm{map\ g\ []} \\
=\; & [\text{definition map}] \\
& \mathrm{map\ g\ (map\ f\ [])} \\
=\; & [\text{definition } \circ] \\
& \mathrm{(map\ g \circ map\ f)\ \ []}
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{map\ (g \circ f)\ \ (x\ :\ xs)} \\
=\; & [\text{definition map}] \\
& \mathrm{(g \circ f)\ \ x\ :\ map\ (g \circ f)\ \ xs} \\
=\; & [\text{IH: map } (g \circ f) \text{ xs} = (\text{map g} \circ \text{map f}) \text{ xs}] \\
& \mathrm{(g \circ f)\ \ x\ :\ (map\ g \circ map\ f)\ xs} \\
=\; & [\text{definition } \circ] \\
& \mathrm{g\ \ (f\ \ x)\ :\ map\ g\ (map\ f\ xs)} \\
=\; & [\text{definition map}] \\
& \mathrm{map\ g\ (f\ x\ :\ map\ f\ xs)} \\
=\; & [\text{definition map}] \\
& \mathrm{map\ g\ (map\ f\ (x\ :\ xs))} \\
=\; & [\text{definition } \circ] \\
& \mathrm{(map\ g \circ map\ f)\ \ (x\ :\ xs)}
\end{aligned}
$$

Functors and Functions

As mentioned, Maybe is also a functor:

```
instance  Functor Maybe  where
  fmap  ::   ∀ a b . (a → b) → Maybe a → Maybe b
  fmap f Nothing  =  Nothing
  fmap f (Just x)  =  Just (f x)
```

A common scenario is to have a function inside a functor,
e.g. maybe_f :: Maybe (a →b),

and an argument inside the same functor,
e.g. maybe_x ::Maybe a.

We want to somehow “apply” the function to the argument to get a result of type Maybe b.

But normal function application cannot do this.

We need to somehow turn the Maybe (a →b) into Maybe a →Maybe b.

Applicative Functors

This is the role of **applicative functors**: they let us *apply* functions within functors[10]:

```
class  Functor f ⇒ Applicative (f :: * → *)  where
  pure  ::  ∀ a . a → f a
  (⊛)   ::  ∀ a b . f (a → b) → f a → f b
```

Now we can declare the Maybe functor applicative:

```
instance  Applicative Maybe  where
  pure  ::  ∀ a . a → Maybe a
  pure  =  Just

  (⊛)   ::  ∀ a b . Maybe (a → b) → Maybe a → Maybe b
  Nothing ⊛ maybe_x  =  Nothing
  Just f ⊛ maybe_x  =  fmap f maybe_x
```

How could we make List applicative? (can you think of another way?)

[10]There are laws here as well. In particular, compose pure (⊛) =fmap.

Applicative Functors

This is the role of **applicative functors**: they let us *apply* functions within functors[10]:

```
class  Functor f ⇒ Applicative (f :: * → *)  where
  pure  ::  ∀ a . a → f a
  (⊛)   ::  ∀ a b . f (a → b) → f a → f b
```

Now we can declare the Maybe functor applicative:

```
instance  Applicative Maybe  where
  pure  ::  ∀ a . a → Maybe a
  pure  =  Just

  (⊛)   ::  ∀ a b . Maybe (a → b) → Maybe a → Maybe b
  Nothing ⊛ maybe_x  =  Nothing
  Just f ⊛ maybe_x  =  fmap f maybe_x
```

How could we make List applicative? (can you think of another way?)

[10]There are laws here as well. In particular, compose pure (⊛) =fmap.

Programming with Algebra

Programming with algebraic interfaces lets us work at a high level of abstraction, and lets compilers make optimizations based on mathematical properties of programs.

For example, by knowing that List is a functor, a compiler can combine multiple map applications into one (*fusion*).

Monoid, Functor and Applicative are just a few of many algebraic type classes that can be used to write elegant and expressive functional programs.