4.md

class: center, middle

Functional Programming

3. The Functional Paradigm

Frank C Langbein
frank@langbein.org

Version 1.1

Programming Paradigms

- **Imperative**: Use statements, e.g. assignment statements
  - Explicitly change the state / the memory of the computer
- **Logic**: Use terms of mathematical logic (mainly first-order logic)
  - Reasoning with relations
- **Functional**: Evaluate mathematical functions
  - Define functions without side-effects / no state
- **Object-Oriented**: associate behaviour with data-structures (objects), belonging to classes, structured into a hierarchy
  - Change the state of objects, communicate between objects
- Most programming languages use multiple paradigms, but have a preference

The Imperative Paradigm

- Computers have re-usable memory that can change state
  - Imperative languages are explicitly based on von Neumann-Zuse computer architectures
- Statements affect the state of the machine
  - Mathematically this means a sequence of values is associated with a variable/state x
  - `[x(1),...,x(t),...]` where `t` is a discrete time variable
  - Becomes increasingly complicated as the state-changes
- Imperative languages can relatively easily be translated into efficient machine-code
  - They are considered to be highly efficient
- Many people find the imperative paradigm quite natural
- Such languages do have functional features (e.g. arithmetic expression, etc.)

The Logic Paradigm

- Logical paradigm focuses on predicate logic
  - Basic concept is a relation

- Useful for problems where it is no obvious what the functions should be
  - Search and construct valid relations
  - To answer questions
- Mathematical logic plays key role in computation, since Boolean logic is basis of the design of logic circuits
- Of course, we can create a functional definition of a logic evaluation procedure

The Functional Paradigm

- Emphasis on evaluating mathematical functions
  - Combined into expressions
  - Based on lambda calculus
- While common in other languages, pure functional languages have no side-effects
  - Makes proofing correctness simple
  - E.g. `f(x) * f(x) == f(x)^2` (not true if `f` has side-effects)
- Often functional programs are less efficient compared to imperative programs
  - Recursion can be expensive (but does not have to be)
  - Absence of side-effects requires more memory (hard to determine which memory can be reused)

The Object-Oriented Paradigm

- OO associates data with behaviour to create objects
  - Methods change the state of objects (not a procedure operating on data)
  - Objects have a class indicating their behaviour
  - Often classes have a hierarchy to group behaviour and enable inheritance
  - Behaviour is encapsulated to allow only legitimate enquiries
- Based on **closures**
  - An association of behaviour (i.e. some code) with data
  - Data can only be accessed by passing appropriate arguments

The Object-Oriented Paradigm

- Focus is on the data, not functions
  - `f(x,y,z)` can be thought of as `(x,y,z).f ()`
  - But side-effects are allowed in OO
- Message-passing sub-paradigm
  - `send OBJECT MESSAGE` instead of `OBJECT.MESSAGE()`
  - Useful for communication in networks
  - Single dispatch: not so useful to send messages to multiple objects
- Distributed function definition sub-paradigm
  - Functions are defined in multiple places
  - Objects have methods with the same names
  - Overloading

Monoids - Functional Design Patterns

- A **monoid** is a pair of a binary operator `(@@)` and a value `u` where the operator has the value as identity and is associative

  ```
  u @@ x = x
  x @@ u = x
  (x @@ y) @@ z = x @@ (y @@ z)
  ```

- So we can define

  ```
  @@_concat :: m -> m -> m
  @@_concat u ys      = ys
  @@_concat (x:xs) ys = x @@ (xs @@_concat ys)
  ```

  - This should look very familiar, e.g. `( (++), [] )`

    ```
    (++) :: m -> m -> m
    [] ++ ys       = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    ```

- More: `((+),0)` , `((*),1)` , `((||),False)` , `((&&),True)` , `((>>),done)`

Monoid Typeclass

```
class Monoid m where
mempty  :: m
mappend :: m -> m -> m
mconcat :: [m] -> m
mconcat = foldr mappend mempty
```

- This defines the general pattern of Monoids
- Define list append Monoid

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

- Note, only one `Monoid` typeclass per type possible (see `newtype`)

Typeclasses

- **Type** declaration: defines how a particular type is created
- **Typeclass**: defines how a set of types are consumed / used in computations
  - Generalise over a set of types to define and execute a standard set of features for those types
- E.g. `Eq` typeclass for `Bool` (already defined, of course)

  ```
  class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

  data Bool = False | True
  instance Eq Bool
    (==) True  True  = True
    (==) False False = True
    (==) _     _     = False
  ```

- Actually defined via `deriving`, which creates the matching patterns

Functor

- **Functor**: apply a function to elements of a data structure

```
class Functor f where
  fmap :: (a->b) -> f a -> f b
```

```
* Take a function ```a->b```, a structure of ```a``` and create the same
  structure of ```b``` (applying the function)
```

- E.g. `> fmap (\x -> x > 3) [1..6]` (same a `map`)

Applicative

- **Applicative**: monoidal functors
  - Same as functor, but the function is also embedded in a structure

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a->b) -> f a -> f b
```

```
* Take a structure of functions ```a->b```, a structure of ```a``` and
  create the same structure of ```b``` (by applying the
  functions)
* Using ```pure``` that creates a structure of ```a``` from ```a```
```

- E.g. how to apply a list of functions?
  - `> [(*0),(+10),(^2)] <*> [1,2,3]`
- The list type constructor `[]` is an applicative

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <-xs]
```

Monads

- A monad is a specialisation of an applicative (which is a special functor)
  - A Monad allows to run actions depending on the outcomes of earlier actions

```
do text <- getLine
   if null text
      then putStrLn "You did not enter anything"
      else putStrLn "You entered " ++ text
```

````
* Recall `(>>=)` and `return`
```
return v >>= \x -> m          = m[x:=v]
m >>= \x -> return x          = m
(m >>= \x -> n) >>= \y -> o   = m >>=  \x -> (n >>= \y -> o)
-- (m >>= f)     >>= g        = m >>= (\x -> (f x >>= g)     )
```
````

- Anything that is a monad is an applicative, is a functor

- E.g. ````> fmap (+1) [1..3]``` is the same than
- `> [1..3] >>= return . (+1)`

Monad Operations

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- `return` takes a value and returns it inside the structure (like `pure`)
- `>>` (sequencing operator) sequences two actions while discarding any resulting value of the first action
- `>>=` is the bind operator (the special part of monads)

  ```
  >>= :: Monad m
      => m a -> (a -> m b) -> m b
  ```

  - Take a structure of `a` and a function `a -> m b` creating a structure of `b` from `a` and return a structure of `b`
- Monads change the structure involved via a function

Concat, Join and Bind

- Functors:

  ```
  > let andOne x = [x,1]
  > andOne 10             -- [10,1]
  > fmap andOne [4,5,6]   -- [[4,1],[5,1],[6,1]]
  ```

  - What if we only wanted a list, not a list of lists?
  - I.e. `> concat $ fmap andOne [4,5,6]`
  - `concat :: Foldable t => t [a] -> [a]` (or just `concat : [[a]] -> [a]`)
- Monads generalises `concat`

  ```
  > import Control.Monad (join) -- defines: join :: Monad m => m (m a) -> m a
  ```

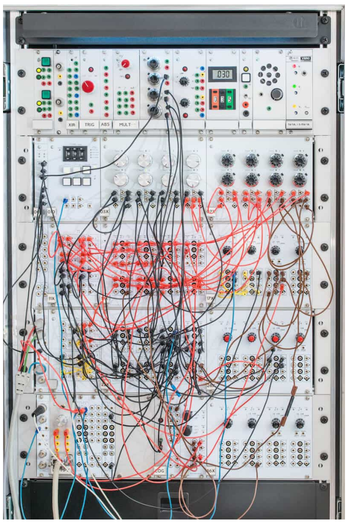
  - Now `bind` or `>>=` is

  ```
  bind :: Monad m => m a -> (a -> m b) -> m b
  bind a b = join (fmap b a)
  ```

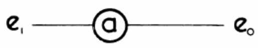
  - E.g. `> bind [1..3] andOne`

Analog Computers

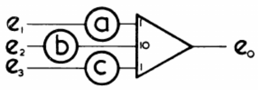
- Network of interconnected components, very similar to functions, and more!
  - Combinatorial FPGA elements!
- Energy efficient? Additional capabilities?



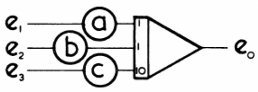
Basic Functions



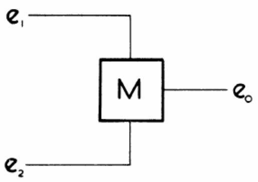
Potentiometer :
$e_0 = ae_1$



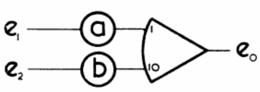
Summer :
$e_0 = -(ae_1 + 10be_2 + ce_3)$



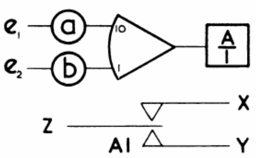
Integrator :
$e_0 = -\int(ae_1 + be_2 + 10ce_3)\,dt$



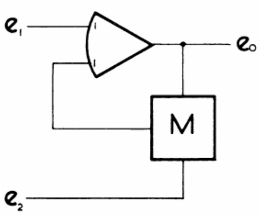
Multiplier :
$e_0 = \frac{e_1 e_2}{100}$



High gain amplifier :
$e_0 = -120\,\mathrm{sgn}(ae_1 + 10be_2)$



Comparator :
$Z \to X$ if $\mathrm{sgn}(10ae_1 + be_2) = -$
$Z \to Y$ if $\mathrm{sgn}(10ae_1 + be_2) = +$



Divider :
$e_0 = -100\frac{e_1}{e_2}$

- Based on feedback loops in operational amplifiers

- Evaluates continuous values!

Solving Differential Equations

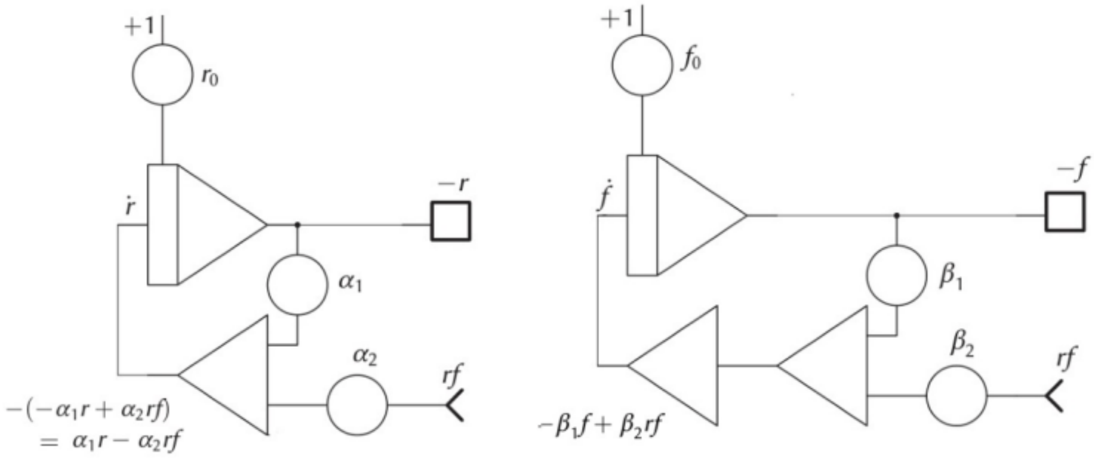
- Predator-prey model (AJ Lotka; V Volterra)
  - Closed eco-system of two species: foxes and rabbits
  - Unlimited food supply for the rabbits
  - Foxes eat rabbits
  - Foxes can die of starvation
- Two coupled differential equations with `r` and `f` denoting the number of rabbits and foxes resp:

```
dr/dt = alpha_1 r - alpha_2 r f
```

```
df/dt = -beta_1 f + beta_2 r f
```

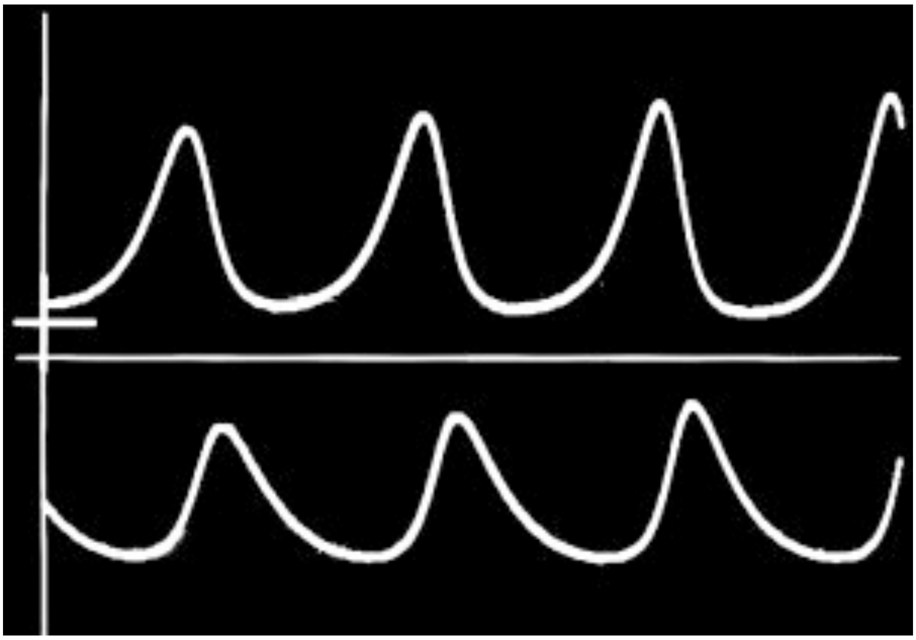
  - `alpha_1` : rabbit fertility rate
  - `alpha_2` : rate of rabbits killed by foxes
  - `beta_1` : fox fertility rate (negative sign, as foxes would die out without rabbits; note, rabbits without foxes would grow)
  - `beta_2` : rate of fox population increase due to rabbits eaten

Analog Predator-Prey Simulator



- Left: Output `-r` with two inputs `r(0)` and `f(t)`
- Right: Output `-f` with input `f(0)`
- Still need a multiplier to get `rf` from `-r` , `-f`

Results



- Easy to change parameters
- Think analog synthesizer