CS2104

Untyped Lambda Calculus

Concrete syntax

- Usually written in infix/distfix format
- Has ambiguities, disambiguated using either brackets, or by defining a convention

$t ::=$	terms
x	variable
$\mid \lambda x\,.\,t$	abstraction/function
$\mid t\ t$	application

Note that the space between the two ts in application is important.

Conventions for ambiguities

- Application is left associative
  $x\ y\ z = (x\ y)\ z$
- λ function extends as far to right as possible
  $\lambda x\,.\,x\ y = \lambda x\,.\,(x\ y)$

Abstract syntax

Written in prefix format | No ambiguities | Can be visualized as a tree

$t ::= x$	terms
$\mid \mathrm{lam}(x, t)$	abstraction/function
$\mid \mathrm{app}(t, t)$	application

Properties

- Turing-complete, i.e. can solve any computation problem given enough time and memory
- Can only represent fns with one param; use currying to represent fns with more params

Reduction rules

Beta-reduction

- Essentially function calling/unrolling
- Rule: $(\lambda x\,.\,t1)\ t2 \quad \rightarrow \quad t1[x \rightarrow t2]$
- Example
  $$(\lambda x\,.\,x\ y)\ (z\ z) \rightarrow x\ y\ [x \rightarrow (z\ z)]$$
  $$\rightarrow (z\ z)\ y$$

Alpha-renaming

- Renames a **bound** parameter, to avoid name clashes that may occur in beta-reduction
- Ensure that new name chosen is not in use
- Rule: $(\lambda x\,.\,t1) \quad \rightarrow \quad (\lambda z\,.\,t1\ [x \rightarrow z])$
- Example
  $$(\lambda x\,.\,x\ y) \rightarrow (\lambda z\,.\,x\ y\ [x \rightarrow z])$$
  $$\rightarrow (\lambda z\,.\,z\ y)$$

Reduction strategies

- A beta redex is an application, where the first term is a function.

Call-by-Name (CBN)

- Leftmost outermost, but not inside lambda
- Lazy evaluation

$$\underline{(\lambda x\,.\,(\lambda z\,.\,x\ z)\ x)\ ((\lambda x\,.\,x)\ y)}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow \underline{((\lambda z\,.\,(\lambda x\,.\,x)\ y)\ z)\ (\lambda x \cdot x)\ y))}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow \underline{((\lambda x \cdot x)\ y)}\ ((\lambda x \cdot x)\ y)\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow (y)\ \underline{((\lambda x \cdot x)\ y)}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow (y)\ (y)\ \underline{((\lambda x\,.\,x)\ z)}$$
$$\rightarrow (y)\ (y)\ (z)$$

Call-by-Value (CBV)

- Leftmost innermost, but not inside lambda
- Eager evaluation, may fail to terminate
- Using same example as above:

$$(\lambda x\,.\,(\lambda z\,.\,x\ z)\ x)\ \underline{((\lambda x\,.\,x)\ y)}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow \underline{(\lambda x\,.\,(\lambda z\,.\,x\ z)\ x)\ (y)}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow \underline{((\lambda z\,.\,y\ z)\ y)}\ ((\lambda x\,.\,x)\ z)$$
$$\rightarrow (y\ y)\ \underline{((\lambda x\,.\,x)\ z)}$$
$$\rightarrow (y\ y)(z)$$

Example where CBV fails to terminate
$$(\lambda x\,.\,z)\ ((\lambda x\,.\,x\ x)\ (\lambda x\,.\,x\ x))$$

Church-Rosser Theorem

Regardless of strategies used, if two evaluations by two different evaluation strategies both terminate, then it always leads to the same normal form.

Expressiveness

Booleans

$\mathrm{true} = \lambda t\,.\,\lambda f\,.\,t$	$\mathrm{if} = \lambda l\,.\,\lambda m\,.\,\lambda n\,.\,l\ m\ n$
$\mathrm{false} = \lambda t\,.\,\lambda f\,.\,f$	$\mathrm{and} = \lambda a\,.\,\lambda b\,.\,\mathrm{if}\ a\ b\ \mathrm{false}$

Non-terminating computation

Use $(\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$

Recursion

- Let binding allows recursion
- Abstract recursive call with fact as a fix-point
  $$\text{let } \underline{\text{fact}} = (\lambda f\,.\,\lambda n\,.\,\text{if } n == 0$$
  $$\text{then } 1 \text{ else } n * f(n-1))\ \underline{\text{fact}} \text{ in } \cdots$$
- i.e. we have the fix-point form
  $$\text{let } x = h\ x \text{ in } \cdots$$

Fix-point operator

- A fix-point can be expanded infinitely
  $$\text{let } x = h\ x \text{ in } \cdots$$
  $$\text{let } x = h\ (h\ x) \text{ in } \cdots$$
- Define **fix** as a fix-point operator, and it can also be expanded infinitely
  $$\text{let } \mathbf{fix}\ h = h\ (\mathbf{fix}\ h) \text{ in } \cdots$$
  $$\text{let } \mathbf{fix}\ h = h\ (h\ (\mathbf{fix}\ h)) \text{ in } \cdots$$
- With **fix**, we can define fact in another way:
  $$\text{let } \underline{\text{fact}} = \mathbf{fix}\ (\lambda f\,.\,\lambda n\,.\,\text{if } n == 0$$
  $$\text{then } 1 \text{ else } n * f(n-1)) \text{ in } \cdots$$
- We can also defined **fix** using untyped lambda calculus:
  $$\text{let } \mathbf{fix}\ h = (\lambda x\,.\,h(x\ x))(\lambda x\,.\,h(x\ x))$$

Extended Lambda Calculus

Types

$\tau ::=$	type
Int	integer
$\mid$ Bool	boolean
$\mid \alpha$	type variable
$\mid \tau \rightarrow \tau$	function type

- Type annotation: $t : \tau$
- Help support data structures and methods
- Useful for documentation and important for static checks

Syntax with Types

The same conventions for ambiguities applies.

$t ::=$	terms
x	variable
$\mid \lambda x : \tau\,.\,t$	abstraction/function
$\mid t\ t$	application
$\mid c$	constants, e.g. int/bool
$\mid \mathrm{op}(t, \cdots, t)$	primitive functions
$\mid \text{let } x : \tau = t \text{ in } t$	let binding
$\mid \text{if } t \text{ then } t \text{ else } t$	conditional

Let binding

- Supports local variables with fixed scope
- Supports recursion (impt for expressivity)

Usage let $x = t$ in t

- x is the local variable;
- t is where the scope of the let variable is limited to; e.g.
  - let $x = 2$ in $x * x$
  - let fact $= \lambda n\,.$ if $n == 0$ then 1 else $n * (\mathrm{fact}(n-1))$ in fact 10
  - let fact $n =$ if $n == 0$ then 1 else $n * (\mathrm{fact}(n-1))$ in fact 10

Type system

- Is an instance of static semantics
  - Static/Dynamic semantics refer to formal analysis that can be done at compile-time/run-time.

Syntax

$$\Gamma \vdash t : \tau$$

Γ is the type env | t is a term/expression | τ is the type, e.g.

- $x : \mathrm{Int} \vdash 1 + x : \mathrm{Int}$
- $y : \beta \vdash \lambda x : \alpha\,.\,y : \alpha \rightarrow \beta$

Rules

1. $\Gamma \vdash c_{\mathrm{Int}} : \mathrm{Int}$
2. Get variable type from typing context
   $$\Gamma \vdash x : \Gamma(x)$$
3. Lambda abstraction
   $$\frac{\Gamma, x : \tau \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau\,.\,t : \tau \rightarrow \tau_2}$$
4. Function application
   $$\frac{\Gamma \vdash t_2 : \tau_2 \quad \Gamma \vdash t_1 : \tau_2 \rightarrow \tau}{\Gamma \vdash t_1 t_2 : \tau}$$
5. Conditional
   $$\frac{\Gamma \vdash t : \mathrm{Bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau}$$
6. Let binding (we define x to have type τ)
   $$\frac{\Gamma + [x : \tau] \vdash t_1 : \tau \qquad \Gamma + [x : \tau] \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau = t_1 \text{ in } t_2 : \tau_2}$$
7. Support polymorphic types
   $$\frac{\Gamma \vdash t : \forall \alpha\,.\,\tau \quad \text{fresh } \alpha_1}{\Gamma \vdash t : [\alpha \rightarrow \alpha_1]\tau}$$

Safety properties of type system

Two key properties that can guarantee type system is sound and safe:

1. **Type preservation**
   (type is a static property)

   Given a type environment Γ and a well-typed expression e such that $\Gamma \vdash e : \tau$. Assume there is a reduction sequence $e \rightarrow^* e2$, then the resulting expression $e2$ always preserves its original type via $\Gamma \vdash e2 : \tau$

2. **Progress**
   (well-typed program cannot have run-time type errors)

   Given a type environment Γ and a well-typed expression e such that $\Gamma \vdash e : \tau$. For every well-typed expression e, its reduction cannot fail due to type error. That is, either e is irreducible, or we must have $e \rightarrow e2$.

Uses

- Can be used to implement type-checking system, given t and τ
- Can be used to implement type-inference system, given t, returning a τ

Dynamic semantics

- Refers to run-time semantics
- Usually involves a machine model suitable for program reduction/execution
- For lambda calculus, a simple execution model could be
$$\langle t \rangle \rightarrow \langle t_2 \rangle$$
- Then we can introduce an evaluation order, either CTX_{name} or CTX_{value} that should not change. Hence
$$\frac{\langle t \rangle \rightarrow \langle t_2 \rangle}{\langle CTX[t] \rangle \rightarrow \langle CTX[t_2] \rangle}$$

Repeated evaluation

- Call by name can cause sub-expressions to be duplicated and thus repeatedly evaluated
- To avoid duplication, use an environment binding, that allows unevaluated arguments to be shared at several locations

Usage $\langle E, t \rangle$
E is the variable to expression enviornment binding | t is a term

$$\langle [\,], (\lambda x \,.\, (\lambda z \,.\, x\ z)\ x)((\lambda x \,.\, x)y)\ ((\lambda x \,.\, x)\ z) \rangle$$
$$\rightarrow \langle [x \rightarrow ((\lambda x \,.\, x)y)], ((\lambda z \,.\, \underline{x}\ z)\ \underline{x})\ ((\lambda x \,.\, x)\ z) \rangle$$
$$\rightarrow \langle [x \rightarrow y], ((\lambda z \,.\, \underline{y}\ z)\ \underline{y})\ ((\lambda x \,.\, x)\ z) \rangle$$
$$\rightarrow \langle [x \rightarrow y], (y\ y)\ ((\lambda x \,.\, x)\ z) \rangle$$

Haskell types

Primitive types

- Types are usually boxed
- Unboxed types are built-in but seldom used, except when implementing the compiler using `-fglasgow-exts`
- Boxed types support polymorphism and lazy evaluation

```
data Bool = False | True
type String = [Char]
data Int = GHC.Types.I# GHC.Prim.Int#
data Float = GHC.Types.F#
    GHC.Prim.Float#
data Double = GHC.Types.D#
    GHC.Prim.Double#
data Char = GHC.Types.C# GHC.Prim.Char#
```

User-defined types

Product types

Includes tuples and records; Similar to conjunction $a \wedge b$

```
  type Pair a b = (a, b)
  ("Hello", 42) :: Pair String Int
```

Sum types

Includes ordinals and general algebraic data types; Similar to disjunction $a \vee b$

```
  type Either a b = Left a | Right b
  (Left "Hello") :: Either String Int
  Right 42 :: Either String Int
```

Ordinal types

- Use GHC built-in `Enum` class
- Note: `succ` and `pred` not cyclic

```
-- Built-in
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
{- succ Tue = Wed; pred Sat = Fri
   toEnum 1 = Tue; fromEnum Sun = 6 -}
data DaysObj = Mon | Tue | Wed | Thu |
    Fri | Sat | Sun
```

Algebraic data type

- `I, F, S` are data constructor tags
- Obtains type-safe values through pattern matching

```
data Data = I Int | F Float | S String
    deriving Show
v1 = I 3; v2 = F 4.5; v3 = S "CS2104";
print_data v = case v of
  I a -> show a
  F v -> show v
  S v -> v
```

Tuple types

- Special case of algebraic data type
- The following is polymorphic
- For a 2-tuple, can use `fst` or `snd` to access first/second element
- Otherwise, use pattern matching

```
-- Haskell definition
(,,) :: a -> b -> c -> (a,b,c)
data (,,) a b c = (,,) a b c
-- Usage
stud1 = ("John", "A1234567J", 2013);
```

Type synonym

- Can give a name to a type
- Usually used for tuple types
- Cannot be recursive

```
type Student = (String, String, Int)
type Pair a b = (a, b)
type String = [Char]
```

Constructors

- `P` is a data constructor, use `P 3 4` to instantiate a `Pair`
- `Pair` is a type constructor

```
P :: a -> b -> Pair a b
data Pair a b = P a b
```

Record type

- Extension of algebraic data type
- Automatically derives access methods for each attribute

```
-- :t Student is String -> String ->
     Int -> Student
data Student = Student {
  name :: String,
  matrix :: String,
  year :: Int
};
-- automatically derive function
-- name :: Student -> String
```

- Can be used with algebraic data types

```
data X = A | B {name::String}
myfn :: X -> Int
myfn A = 50
myfn B{} = 200
-- Instantiating
bb = B "some_string"
```

Pointer types

- Algebraic data type is already implemented as a pointer to a boxed value, hence no need for pointer types
- Allows for recursive type
- e.g. `data Rnode = Rnode(Int, Rnode)`

Haskell constructs

Statement Executed for its effects

Expression Computes a result, may have effects

Pattern matching

- Use `;` to mark the end of a pattern body
- Use underscore for catch-all pattern
- Each pattern can only contain one case

```
-- Non-exhaustive
let weekend x = case x of {
Mon -> False; Sat -> True; Sun -> True;}
-- Exhaustive
let weekend x = case x of {
Sat -> True; Sun -> True; _ -> False; }
```

where vs let

- `let` is an expression. With nested `let` with same varaibles used, refer to the nearest variable. For example, the following expression results in an in infinite loop:

```
let x = c1 in
  let x = x+1 in x+x
```

- `where` is tied to the scope of a syntactic construct, so can share bindings

```
f x
  | cond1 x  = a
  | cond2 x  = g a
  | otherwise = f (h x a)
  where
    a = w x
```

Indentation

- Grouped statements should have the same indentation
- To "close" a declaration (`in`), it should be on the same indentation level as opening (`let`)

```
foo :: Double -> Double
foo x =
    let s = sin x
        c = cos x
    in 2 * s * c
```

- Can use `;` and `{}` to avoid indentation
- Useful when writing one-liners in GHCi

```
-- ghci oneliner
let foo :: Double -> Double; foo x =
    let { s = sin x; c = cos x } in 2
    * s * c
```

Functions

Types

- `(x-> x * x)` has type `Num a => a -> a`, where `Num a` means that `a` is a member of `Num` type class
- `fst` has type `(a,b) -> a`

Tupled vs curried functions

- Haskell supports functions with multiple args via tupled and curried functions
- Curried function takes one arg at a time, returning a function that takes the next arg
- Tupled function takes all args as a tuple
- Tupled and curried functions are isomorphic

Partially applied functions

- Curried fns: just apply the function partially
- Tupled fns: need lambda abstraction

Lists

- Prepend: Use cons - `(:) :: a->[a]->[a]`
- Append: `(++) :: [a] -> [a] -> [a]`, O(size of left list)

Reverse

- Tail-recursive, O(size of list)
  - A fn is tail-recursive if the recursive call is always the last op in recursive invocations
  - Equivalent to a loop
- `acc` accumulates a list of reversed elements

```
revit xs =
  let aux xs acc =
    case xs of
      [] -> acc
      x:xs -> aux xs (x:acc)
    in aux xs []
```

Take

Can take first k elements of an infinite list

```
sum(take 2 [2..]) -- evaluates to 5
```

Type classes

- Support overloading by allowing types to be parameterized
- `(+) :: Num a => a -> a -> a` ≡ `(+) ::` $\forall a \in$ `Num . a -> a -> a`
- `map :: (a -> b)-> [a] -> [b]` ≡ `map ::` $\forall a \forall b$ `. (a -> b)-> [a] -> [b]`

Equality class

- Equality can be tested for any data structure except functions
- `(==) :: Eq a => a -> a -> Bool`

```
data Y = Y{a::Int, b::Int} deriving Show
instance Eq Y where
  (==) Y{a=xa, b=xb} Y{a=ya, b=yb} = xa
      == ya
aa = Y 3 2; bb = Y 3 4
-- aa == bb is True
```

- Recursive types can be handled, but may need type qualifiers

```
instance (Eq t) => Eq (Tree t) where
  Leaf a == Leaf b = a == b
  (Branch l1 r1) == (Branch l2 r2)
                   = (l1 == l2) && (r1
                      == r2)
  _ == _           = False
```

Class extension

- Can extend a subclass from some superclass
- Multiple inheritance possible

```
class (Eq a) => Ord a where ...
class (Eq a, Show a) => C a where ...
```

- Allows using definitions from base class (e.g. /= below)
- The following are the default definitions for `Ord` class

```
x >= y    = y <= x
x > y     = y < x
min x y  = if x<=y then x else y
max x y  = if x>=y then x else y
```

- Notice everything is defined in terms of `/=` and `<=`, so you only need to implement `<=`

Enum class

```
class (Enum a)
  enumFromThen :: a -> a -> [a]
  fromEnum :: a -> Int
  toEnum :: Int -> a
-- Arithmetic sequences
[Mon..Wed] -> [Mon, Tue, Wed]
```

Show class

- Conversion for character string, for printing

```
class Show where
  show :: a -> String
  -- with accumulating parameter
  shows :: a -> String -> String
  show x = shows x ""
```

Read class

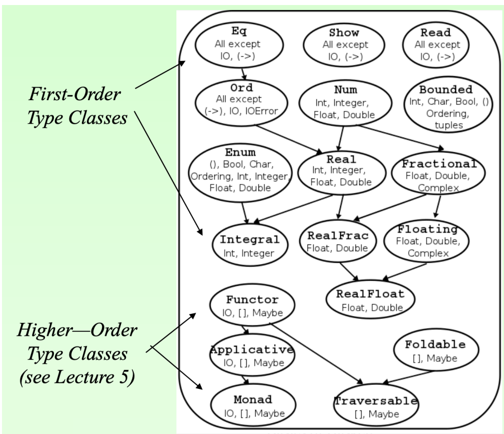
- Empty list denotes failure of parsing, multiple answers denote non-determinism

```
class Read a
  reads :: String -> [(a,String)]
```

Derived instances

- Haskell supports automatic derivation, where it implements the necessary functions for the classes where possible
- `data Tree a = ... deriving (Eq, Ord)`

Predefined type classes



Higher-order functions

- Like data structures, fns should be first class:
  - Has value and type
  - Can be passed as arg, returned as result, constructed at run-time, stored in data structures
- HO functions are useful in supporting: Code reuse | Laziness | Data abstraction | Design patterns

Lazy evaluation

- Default for Haskell
- Any expression `e` can be abstracted into a function: `\() -> e`
- Called a closure when it is coupled with value environment of free vars of `e`: `(E, \() -> e)`
- Allows handling of non-terminating code, as long as it is not evaluated by context, like infinite data structures such as `ones = 1:ones`

Strict evaluation

Helps avoid using memory for building closures, if result is definitely needed, e.g. `inc !y = y+1`

Folding

- Implement generalized functions
- Haskell fold types expect a `Foldable`

```
foldl :: Foldable t => (b -> a -> b) ->
    b -> t a -> b
foldr :: Foldable t => (a -> b -> b) ->
    b -> t a -> b
```

- Example impl below only support lists
- `foldr` process from R to L, not tail-recursive

```
foldr :: (a -> z -> z) -> z -> [a] -> z
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
-- Usage
sum xs = foldr (+) 0 xs
prod xs = foldr (*) 1 xs
```

- `foldl` process from L to R, is tail-recursive

```
foldl :: (z -> a -> z) -> z -> [a] -> z
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Which to use

- Can be interchanged if reduction operator f is associative
- `foldl` typically more efficient due to tail recursion

List mapping

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

List filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs) =
  if (f x) then x:(filter f xs)
  else filter f xs
```

Mutual recursion

Mutual-recursive functions supported by multiple declarations within a single let construct

```
f n =
  let foo n =
        if n <= 1 then 1
        else foo(n-1) + goo(n-2)
      goo n =
        if n <= 1 then 1
        else goo(n-1) + foo(n-2)
  in foo n
```

Function composition

Summary

```
f (g x)     -- use space
(f . g) x   -- composition (R-to-L)
x |> g |> f -- pipeline (left-to-right)
f $ g x     -- weak precedent apply
```

Types

```
-- Regular function composition
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
-- Left assoc
(|>) :: a -> (a -> b) -> b
a |> f = f a
-- Right assoc, low precedence
($) :: (a -> b) -> a -> b
f $ x = f x
-- Usage of weak precedent apply
inc $ x*2 <-> inc (x*2)
inc x*2 <-> (inc x)*2
```

List comprehension

- Equivalence with map:

`[f x | x<-xs]` ≡ `map (\x -> f x) xs`

- Supports filtering:

`[f x | x<-xs, x>5]` ≡ `map (\x -> f x) (filter (\x -> x>5) xs)`

- Can have multiple generators:

`[(x,y) | x<-xs, y<-ys]` ≡ `concatMap (\x -> map (\y -> (x,y)) ys) xs`

General translation scheme

- `[e | x<-xs]` ≡ `map (x-> e)xs`
- `[e | x<-xs, y<-ys, rest]` ≡ `concatMap (x -> [e | y<-ys, rest]) xs`
- `[e | x<-xs, test, rest]` ≡ `[e | x <- filter (x -> test) xs, rest]`

concatMap

```
concatMap :: (a->b) -> [[a]] -> [b]
concatMap f [] = []
concatMap f (x:xs) = (f x) ++
    (concatMap f xs)
```

Summary on Types/Expr

- Each type belongs to a kind - `type :: kind`
- Each type has `*` as its kind, e.g. `Int :: *`
- Type constructors are HO-functions over types, e.g.

```
Pair :: * -> * -> *
Pair Int :: * -> *
```

- Three levels of typings - `expr::type::kind`
  - Allows more type-safe expressions and kind-safe types to be safely constructed

Arrays

- Use `import Data.Array` to access
- `Ix a => Array a b`
- Can be regarded as fns from indices to values, where indices are contiguous and bounded
- Array index up to a 5-tuple
- Cannot be infinite

```
type Ix :: * -> Constraint
class (Ord a) => Ix a where
  range   :: (a,a) -> [a]
  inRange :: (a,a) -> a -> Bool
  index   :: (a,a) -> a -> Int
```

- `range` enumerates a list of idx in index order
- `inRange` checks if an index is between a pair of bounds
- `index` gets zero-origin offset of an index from its bounds

Array creation

- Build using indices and a list of elements
- Can be defined recursively

```
array :: (Ix a) => (a,a) -> [(a,b)] ->
    Array a b
-- Usage
squares = array (1,100) [(i, i*i) | i
    <- [1..100]]
squares ! 7 <=> 49
bounds squares <=> (1,100)
```

Accumulation

- Accumulate values into each index
- `accumArray :: (Ix a) => (b->c->b) -> b -> (a,a) -> [Assoc a c] -> Array a b`
- Parameters are: accumulating function | initial value | bounds | elements to accumulate

Incremental updates

- `(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b`
- e.g. `a // [(i,v), (j,w)]`
- If an index appears multiple times, the last value takes precedence

Semi-group and monoids

```
class SemiGroup a where
  op :: a -> a -> a
class SemiGroup a => Monoid a where
  unit :: a
```

- Associative | Op with `unit` returns itself
- A monad is a higher-order monoid

Monads

Referential transparency

- An expression is **referentially transparent** if it can be replaced with equivalent value (and vice versa) without changing the program's behavior/meaning
- Requires the expression to be pure

Summary

- Functors apply a function to a wrapped value
- Applicatives apply a wrapped function to a wrapped value
- Monads apply a function that returns a wrapped value, to a wrapped value
- Maybe implements all three

Contexts

- `data Maybe a = Nothing | Just a`
- List `[a]`: Empty list means no soln | `[r1,r2,r3]` means 3 possible solns
- `IO a` for input-output interaction
- `DB a = state -> (state, a)` for imperative state that can be updated
- `Parser a = String -> [(a,String)]` for non-deterministic parsing

Functor

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  -- replacing value in context with a
  (<$) :: a -> f b -> f a
  -- infix variant of fmap
  (<$>) :: (a -> b) -> f a -> f b
instance Functor Maybe where
  fmap f (Just val) = Just (f val)
  fmap f Nothing = Nothing
```

Applicative

- Can work with functions of any no. of args

```
class Functor f => Applicative (f :: *
    -> *) where
  (<*>) :: f (a->b) -> f a -> f b
  -- takes a pure value and wraps it
  pure :: a -> f a
```

Monad

```
class Monad m where
  -- bind operator
  >>= :: m a -> (a -> m b) -> m b
  return :: a -> m a
  >> :: (m a) -> (m b) -> m b
instance Monad Maybe where
  Nothing >>= f = Nothing
  Just val >>= f = f val
  m1 >> m2 = m1 >>= (\_ -> m2)
```

Monad laws

```
-- Left identity
(return a) >>= k  = k a
-- Right identity
m >>= return      = m
-- Associativity
(m >>= f) >>= g   =
  m >>= (\x -> f x >>= g)
```

IO Monads

```
getChar :: IO Char; getLine :: IO String
putChar :: Char -> IO ()
putStrLn :: String -> IO ()
readFile :: FilePath -> IO String
```

Do comprehension

- List is an instance of Monad; List comprehension is an instance of Do comprehension
- Every line is a monadic value, and can refer to computations in previous lines
- Using `<-` lets us get the pure value, but cannot use for last line
- Each line of do comprehension is just a `(>>=)`

```
foo :: Maybe String
foo = Just 3 >>= (\x -> Just "!" >>=
    (\y -> Just (show x ++ y)))
-- equivalent to
foo = do
  x <- Just 3
  y <- Just "!"
  Just (show x ++ y)
```

Error handling

```
-- using try-catch
class Monad m => MonadError e m | m ->
    e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
-- using Either
data Either a b = Left a | Right b
Left e  -- an error value of type a
Right x -- a successful val x of type b
-- using Maybe
data Maybe a b = Nothing | Just b
Nothing  -- error value
Just x   -- success
```