Pure vs. Impure

- Pure: No side effects
  - Example: Haskell
- Impure: Has side effects
  - Example: ML
    - Print
    - Assignment
- Litmus test for side effect:
  - $f \implies f; f$

Mutability vs. Immutability of Data

- ML:
  - val $x = ref 5; x := 10; val y = l x + 1;
  - Val $x = ref 5 : int ref$
  - Val $l t = () : unit$
  - Val $y = 11 : int$
- Prolog:
  - put_assoc(FnName, EnvIn, FnBody, EnvOut).
- Arrays are often mutable data structures

Lazy Evaluation

- quickSort $[] = []$
- quickSort $(x:xs) = quickSort (filter (< x) xs) ++ [x] ++ quickSort (filter (>= x) xs)$
- minimum $ls = head (quickSort ls)$
- In practice purity is almost a prerequisite for lazy evaluation
- Many imperative languages will lazily evaluate conditions:
  - if $(a \&\& b) \ldots$
  - if $(a || b) \ldots$
Type System

- Some languages like part of the Lisp Family are based on an untyped $\lambda$-Calculus.
  - Also Scheme

- Some languages like the ML Family use a typed $\lambda$-Calculus and a Hindley-Milner Type System
  - Also Haskell, with some modifications

- Many languages provide algebraic data types (tuples, records, ...) as a first class citizen
  - Except for the Lisp Family (only Lists)

Explicit vs. Implicit Typing

- Explicit vs. Implicit
  - Explicit: Type annotations
    - Some languages only support explicit type declaration
      - Java, C, Ada, Pascal, C++, ...
  - Implicit: Type inference
    - ML, OCaml, Haskell, F#, Scala, Go, C++11 (auto), ...
    - Most languages with type inference also support explicit type annotations

Static vs. Dynamic Typing

- Static vs. Dynamic
  - Static: Compile time
    - Java, Scala, Haskell, ML, C, C++
  - Dynamic: Runtime
    - Ruby, Lisp Family, Objective C, Python, Perl, PHP, ...

- Some languages do both...
  - Java: for downcasting
  - Static type vs. dynamic type

Abstract Data Types (ADT)

- Combining data with associated operations

Haskell:

```haskell
module Stack (Stack, empty, isEmpty, push, top, pop) where

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a, Stack a)
```
Abstract Data Types

- ML:
  ```ml
type ‘a stack = a' list
val empty = []
val push = op ::;
fun pop [] = NONE
    | pop (tos::rest) = SOME tos
end
```

- Stack.push(100, Stack.empty);

Recursive Types

- ML:
  ```ml
datatype mylist = Nil | Cons of int * mylist
val l = Cons(10,Nil)
```

- Haskell:
  ```haskell
data List a = Nil | Cons a (List a)
```

Ad-Hoc Polymorphism

- Polymorphism: Functions (or operators) can operate on multiple different types.
- Parametric Polymorphism
  - `a' → (a' → b') → b''`
- Subtype Polymorphism
  - `S <: T`

- Ad-Hoc: not rooted in the type system.
  - E.g., operator overloading
  - Arguments to an operator do not need to be related in any form in the type system.
  - E.g., "+" for Int addition and String concatenation.

Type Classes

- Expressing operator overloading in the type system

  - Haskell:
    ```haskell
class Eq a where
      (==) :: a -> a -> Bool
      (/=) :: a -> a -> Bool
    instance (Eq a) => Eq (Tree a) where
      Leaf a == Leaf b = a == b
      (Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
      _      == _      = False
    ```
Existential Types

- Idea: Fixed but not further specified type
  - Think of J.P. Sartre: Existence precedes Essence.
- Method of abstraction

  Haskell:
  - data Litem = forall a. (Show a) => Litem a
  - xs :: [Litem]
  - doShow :: [Litem] -> String
  - doShow [] = ""
  - doShow ((Litem x):xs) = show x ++ doShow xs

Warmup: Monoids

- Monoid
  - Set $S$
  - Binary operation $\cdot$
    - Closure: $\forall a, b \in S : a \cdot b \in S$
    - Associativity: $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
    - Identity element: $\exists e \in S : \forall a \in S : e \cdot a = a \cdot e = a$

Category Theory

- Categories = Algebraic Structures
- Attempt to axiomatically capture commonalities between similar structures
- Category (informal):
  - Collection of Objects
  - Morphisms: Structure-preserving mapping from one structure to another
- Example: Category of Sets:
  - Objects = Sets
  - Morphisms = Functions

Category Theory

- Objects $(S, T, U, ...)$
- Morphisms $(f : S \to T, ...)$
- Binary operation $\circ$ (composition of morphisms)
  - Associativity: $f : S \to T, g : T \to U, h : U \to V$ then $h \circ (g \circ f) = (h \circ g) \circ f$
  - Identity: For every object $x$ there is a morphism $id_x : X \to X$ such that $id_x X = X = X \cdot id_x$
CCC
- Axioms are relatively weak
- We often want more restrictive categories with more interesting properties
- Example: Cartesian Closed Categories
  - It has a terminal object
    - \( T \) is a terminal if \( \forall X \in C : \exists \text{ exactly one } f : X \to T \)
  - Any two objects \( X \) and \( Y \) have a product \( X \times Y \) in \( C \)
  - Any two objects \( Y \) and \( Z \) have an exponential \( Z^Y \) in \( C \)

\[ \lambda - Calculus! \]
- Currying is a consequence of the category-theoretical background:
  - \( X \times Y \to Z \rightarrow X \to Z^Y \)

Functors
- In Category Theory: A Functor maps between Categories.
  - Objects to Objects
  - Morphism to Morphism
  
  \[ \text{In Haskell:} \]
  - The \textbf{Functor} class is defined like this:
    
    ```haskell```
    ```
    class Functor f where
    fmap :: (a -> b) -> f a -> f b
    ```
    
    All instances of Functor should obey:
    ```haskell```
    ```
    fmap id = id
    fmap (p . q) = (fmap p) . (fmap q)
    ```

Monads
- In Category Theory: Monad = Endofunctor with two natural transformations.
  - Natural transformation: transforming one Functor into another while preserving structure

\[ C \text{ a Category, Monad over } C \text{ consists of functor } T: C \to C \text{ with natural transformations} \]
  - \( 1_C \to T \)
  - \( T \circ T \to T \)

\[ \text{In Haskell: A Monad is a} \]
  - Type constructor that defines for every underlying type a monadic type
  - A unit function that maps a value of the underlying type to a value of the monadic type \((\text{return})\)
  - A binding operation \((M \ t) \to (t \to M \ u) \to (M \ u) (\text{represented by } >>=)\)
Monads

- **Identity Monad**
  
  \[ \text{Id } t = t \]
  
  \[ \text{return } x = x \]
  
  \[ x >>= f = f x \]

- **State Monad**
  
  \[ \text{type } \text{State } s t = s \to (t, s) \]
  
  \[ \text{return } x = \backslash s \to (x, s) \]
  
  \[ m >>= f = \\backslash r \to \text{let } (x, s) = m r \text{ in } (f x) s \]

- **IO Monad**

Uniqueness Types

- Pure functional languages have their problems
  
  - Side effects or “destructive state”
  
  - Function \text{read}(\text{File } f) \text{ returns String}

- Clean: Uniqueness Types as an alternative to Monads
  
  - Unique type means at most one reference to it
  
  - Helps with concurrency
    
    - (by preventing it...)
  
  - Function \text{readUnique}(\text{unique } \text{File } f) \text{ returns}
    
    - (\text{File}, \text{String})

Recursion

- Remember recursion...
  
  - Function calls itself with a subpart of the task
  
  - Fixpoint combinator

- **Scala:**
  
  def factorial(number: Int): Int = {
    if (number == 1)
      return 1
    number * factorial(number - 1)
  }

Tail Recursion

- Tail recursion
  
  - Result of the recursive call is immediately returned
  
  - No need to remember the return address

  def factorial(number: Int): Int = {
    def factorialWithAcc (number: Int, acc: Int) : Int = {
      if (number == 1)
        return acc
      else
        factorialWithAcc (number - 1, acc * number)
    }
    factorialWithAcc (number, 1)
Closures

```javascript
function foo(x : int) {
    return x + a;
}
```

- Where does the “a” come from?
  - Context.

- A (lexical) Closure is a combines the function with its referencing environment (free variables).

Closures

- “Capture by reference”
  - JavaScript
```javascript
var y = 1;
function foo(x) {
    return x + y;
}
y = 2;
alert(foo(1));
```

Closures

- “Capture by value.”
  - Java: Annonymous inner classes
```java
void foo() {
    final int x;
    new Thread() {
        public void run() {
            System.out.println(x);
        }
    }.start();
}
```

Continuations

- Traditional functions:
```python
def sq(x)
    return x*x
```

- Continuation
  - Abstract representation of control state
  ```python
def sq(x,c)
    return c(x*x)
```

- Scheme: call/cc
  - Captures current environment as a continuation
  - Related to the idea of setjmp/longjmp coroutines
Overview