Announcements

- Exam 2 is graded, but I will need some time to go over it.
- I’ll release grades this evening (fingers crossed!)
  - Rainbow grades: HW1-6, Exam 1-2, Quiz 1-5
  - Will post answer key
- Still grading: Quiz 6, HW7 and HW8
  - We will grade HW8 over weekend to use in HW9

Today’s Lecture Outline

- Haskell
  - Covered syntax, lazy evaluation, static typing
  - Algebraic data types and pattern matching
  - Type classes
  - Monads … and more
- Types
- Type systems
  - Type checking
  - Type safety
  - Type equivalence

Types

Read: Scott, Chapter 7

Algebraic Data Types

- Algebraic data types are tagged unions (aka sums) of products (aka records)

```
data Shape = Line Point Point
            | Triangle Point Point Point
            | Quad Point Point Point
```

- Haskell keyword for union
- new constructors (a.k.a. tags, disjuncts, summands)
- Line is a binary constructor, Triangle is a ternary …

Algebraic Data Types in HW9

- Defining a lambda expression

```haskell
  type Name = String
  data Expr = Var Name
             | Lambda Name Expr
             | App Expr Expr
```

> e1 = Var “x” // Lambda term x
> e2 = Lambda “x” e1 // Lambda term λx.x

Pattern Matching

- Examine values of an algebraic data type

```
anchorPnt :: Shape -> Point
anchorPnt s = case s of
  Line   p1 p2 -> p1
  Triangle p3 p4 p5 -> p3
  Quad   p6 p7 p8 p9 -> p6
```

- Two points
  - Test: does the given value match this pattern?
  - Binding: if value matches, bind corresponding values of s and pattern
Pattern Matching

- Pattern matching “deconstructs” a term

```haskell
> let h:t = "ana" in t
  "na"

> let (x,y) = (10,"ana") in x
  10
```

Pattern Matching in HW9

```haskell
isFree :: Name -> Expr -> Bool
isFree v e = case e of
  Var n -> if (n == v) then True else False
  Lambda ...```

Generic Functions in Haskell

- We can generalize a function when a function makes no assumptions about the type:

```haskell
const :: a -> b -> a
const x y = x

apply :: (a->b)->a->b
apply g x = g x```

Generic Functions

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

- Can we write function `sum` over a list of `a`'s?

```haskell
sum :: a -> List a -> a
sum n Nil = n
sum n (Cons x xs) = sum (n+x) xs```

- No. `a` no longer unconstrained. Type and function definition imply that we can apply `+` on `a` but
  - `+` is not defined on all types!
  - Type error: No instance for (Num a) arising from a use of `+`!
```

Haskell Type Classes

- Not to be confused with Java classes/interfaces
- Define a type class containing the arithmetic operators

```haskell
class Num a where
  (==) :: a -> a -> Bool
  (+) :: a -> a -> a
  ...
instance Num Int where
  x == y = ...
instance Num Float where
  ...
```

Generic Functions with Type Class

```haskell
sum :: (Num a) => a -> List a -> a
sum n Nil = n
sum n (Cons x xs) = sum (n+x) xs```

- One view of type classes: predicates
  - `(Num a)` is a predicate in type definitions
  - Constrains the types we can instantiate a generic function to specific types
  - A type class has associated laws
Type Class Hierarchy

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

class (Eq a) => Ord where
  (<), (<=), (>, (>=) :: a -> a -> Bool
  min, max :: a -> a -> a

- Each type class corresponds to one concept
- Class constraints give rise to a hierarchy
- Eq is a superclass of Ord
  - Ord inherits specification of (==) and (/=)
  - Notion of “true subtyping”

Today’s Lecture Outline

- Haskell
  - Algebraic data types and pattern matching
  - Type classes
  - Monads … and more

- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence

Monads

- One source: All About Monads (haskell.org)
- Another source: Scott’s book
- A way to cleanly compose computations
  - E.g., f may return a value of type a or Nothing
  - Composing computations becomes tedious:
    - case (f s) of
      - Nothing -> Nothing
      - Just m -> case (f m) ...
- In Haskell, monads encapsulate IO and other imperative features

An Example: Cloned Sheep

type Sheep = ...
father :: Sheep -> Maybe Sheep
father = ...
mother :: Sheep -> Maybe Sheep
mother = ...
(Note: a cloned sheep may have both parents, or not...
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case mother s of
  Nothing -> Nothing
  Just m -> father m

An Example

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
  Nothing -> Nothing
  Just m -> case (father m) of
    Nothing -> Nothing
    Just gf -> father gf

- Tedious, unreadable, difficult to maintain
- Monads help!

The Monad Type Class

- Haskell’s Monad class requires 2 operations, 
  >>> (bind) and return

  class Monad m where
    // >>> (the bind operation) takes a monad 
    // m a, and a function that takes a and turns 
    // it into a monad m b
    (>>=) :: m a -> (a -> m b) -> m b
    // return encapsulates a value into the monad
    return :: a -> m a
### The Maybe Monad

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just

Cloned Sheep example:
mothersPaternalGrandfather s =
    (return s) >>= mother >>= father >>= father
(Note: if at any point, some function returns
    Nothing, Nothing gets cleanly propagated.)
```

### The List Monad

- The List type is a monad!
- `li >>= f = concat (map f li)`
- `return x = [x]`
- Note: `concat :: [[a]] -> [a]`
e.g., `concat [[1,2],[3,4],[5,6]]` yields `[1,2,3,4,5,6]`
- Use any `f` s.t. `f :: a->[b]`. `f` may yield a list of
  `0,1,2,...` elements of type `b`, e.g.,
  - `f x = [x+1]`
  - `[1,2,3] >>= f` --- yields ?

### The do Notation

- `do` notation is syntactic sugar for monadic `bind`

```haskell
> f x = x+1
> g x = x*5
> [1,2,3] >>= (return . f) >>= (return . g)
```

Or
```
> [1,2,3] >>= (x->[x+1]) >>= (y->[y*5])
```

Or, make encapsulated element explicit with `do`
```
do { x <- [1,2,3]; y <- (x->[x+1]) x; (y->[y*5]) y }
```

### List Comprehensions

- List comprehensions are syntactic sugar on top of the `do` notation!
  - `[ x | x <- [1,2,3,4] ]` is syntactic sugar for
  ```haskell
do { x <- [1,2,3,4]; return x }
```
  - `[ x | x <- [1,2,3,4] ]` is syntactic sugar for
  ```haskell
do { x <- [1,2,3,4]; return [x,y] }
```
- Which in turn, we can translate into monadic `bind`...
So What is the Point of the Monad...

- Conveniently chains (builds) computations
- Encapsulates "mutable" state. E.g., IO:
  - openFile :: FilePath -> IOMode -> IO Handle
  - hClose :: Handle -> IO () – void
  - hIsEOF :: Handle -> IO Bool
  - hGetChar :: Handle -> IO Char

These operations break "referential transparency". For example, hGetChar typically returns different value when called twice in a row!

Today's Lecture Outline

- Haskell
  - Algebraic data types and pattern matching
  - Type classes
  - Monads ... and more

- Types
  - Type systems
    - Type checking
    - Type safety
  - Type equivalence

What is a type?

- A set of values and the valid operations on those values
  - Integers:
    - + - * / < <= = >= > ...
  - Arrays:
    -lookup(&array, &index)
    -assign(&array, &index, &value)
    -initialize(&array), setBounds(&array)
  - User-defined types:
    - Java interfaces

What is the role of types?

- What is the role of types in programming languages?
  - Semantic correctness
  - Data abstraction
    - Abstract Data Types
  - Documentation (static types only)

3 Views of Types

- Denotational (or set) point of view:
  - A type is simply a set of values. A value has a given type if it belongs to the set. E.g.
    - int = {..., -1, 0, 1, 2, ...}
    - char = {'a', 'b', ...}
    - bool = {true, false}

- Abstraction-based point of view:
  - A type is an interface consisting of a set of operations with well-defined meaning

3 Views of Types

- Constructive point of view:
  - Primitive/simple/built-in types: e.g., int, char, bool
  - Composite/constructed types:
    - Constructed by applying type constructors
      - pointer e.g., pointerTo(int)
      - array e.g., arrayOf(char) or arrayOf(char, 20) or ...
      - record/struct e.g., record(age:int, name:string)
      - union e.g., union(int, pointerTo(char))
      - list e.g., list(...)
      - function e.g., float -> int
    - CAN BE NESTED! pointerTo(arrayOf(pointerTo(char)))
  - For most of us, types are a mixture of these 3 views
What is a Type System?
- A mechanism to define types and associate them with programming language constructs
- Additional rules for type equivalence, type compatibility
  - Important from pragmatic point of view

What is Type Checking?
- The process of ensuring that the program obeys the type rules of the language
  - Type checking can be done statically
    - At compile-time, i.e., before execution
    - Statically typed (or statically checked) language
  - Type checking can be done dynamically
    - At runtime, i.e., during execution
    - Dynamically typed (or dynamically checked) language

What is Type Checking?
- Statically typed (better term: statically checked) languages
  - Typically require type annotations (e.g., A a, List<A> list)
  - Typically have a complex type system, and most of type checking is performed statically (at compile-time)
    - Ada, Pascal, Java, C++, Haskell, OCaml
  - A form of early binding
- Dynamically typed (better term: dynamically checked) languages. Also known as Duck typed...
  - Typically require no type annotations!
  - All type checking is performed dynamically (at runtime)
    - Smalltalk, Lisp and Scheme, Python, JavaScript

Language Design Choices
- Design choice: what is the set of forbidden errors?
  - Obviously, we cannot forbid all possible semantic errors...
    - Define a set of forbidden errors
- Design choice: Once we’ve chosen the set of forbidden errors, how does the type system prevent them?
  - Static checks only? Dynamic checks only? A combination of both?
- Furthermore, are we going to absolutely disallow forbidden errors (be type safe), or are we going to allow for programs to circumvent the system and exhibit forbidden errors (i.e., be type unsafe)?

Forbidden Errors
- Example: indexing an array out of bounds
  - a[i], a is of size Bound, i<0 or Bound≤i
  - In C, C++, this is not a forbidden error
    - 0≤i and i<Bound is not checked (bounds are not part of type)
      - What are the tradeoffs here?
    - In Pascal, this is a forbidden error. Prevented with static checks
      - 0≤i and i<Bound must be checked at compile time
      - What are the tradeoffs here?
  - In Java, this is a forbidden error. It is prevented with dynamic checks
    - 0≤i and i<Bound must be checked at runtime
    - What are the tradeoffs here?
Type Safety

- Java vs. C++:
  - Java: Duck q; ... q.quack(); class Duck has quack
  - C++: Duck *q; ... q->quack(); class Duck has quack

  Can we end up with a program that calls quack() on an object that isn’t a Duck?
  * In Java?
  * In C++?

- Java is said to be type safe while C++ is said to be type unsafe

C++ is type unsafe

```c
// #1
void* x = (void *) new A;
B* q = (B*) x;  // a safe downcast?
int case1 = q->foo(); // what happens?

A virtual foo()
B virtual foo()

// #2
void* x = (void *) new A;
B* q = (B*) x;  // a safe downcast?
int case2 = q->foo(66); // what happens?
```

A
B
virtual foo()
virtual foo(int)

static type B* q "promises" the programmer that q will point to a B object. However, language does not "honor" this promise...

What is Type Checking

- Static typing vs. dynamic typing
- What are the advantages of static typing?
- What are the advantages of dynamic typing?

What is Type Checking?

Today’s Lecture Outline

- Haskell
  - Algebraic data types and pattern matching
  - Type classes
  - Monads … and more

- Types
- Type systems
- Type checking
- Type safety
- Type equivalence

Type Equivalence and Type Compatibility

- Discussion centers on non-object-oriented von Neumann languages that use the value model for variables: Fortran, C, Pascal
- Questions:
  - e := expression ∅ or ∅
    Are e and expression of "same type"?
  - a + b ∅ or ∅
    Are a and b of "same type" and type supports +?
  - foo(arg1, arg2, ..., argN) ∅ or ∅
    Do the types of the arguments "match the types" of the formal parameters?
Type Equivalence

Two ways of defining type equivalence

- **Structural equivalence**: based on "shape"
  - Roughly, two types are the same if they consist of the same components, put together in the same way
- **Name equivalence**: based on lexical occurrence of the type definitions
  - Roughly, each type definition introduces a new type
  - **Strict name equivalence**
  - **Loose name equivalence**

Structural Equivalence

- A type name is structurally equivalent to itself
- Two types are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent arguments
- After type declaration `type n = T` the type name `n` is structurally equivalent to `T`
  - E.g., in Haskell, we saw `type Name = String`
  - Declaration makes `n` an alias of `T`. `n` and `T` are said to be aliased types

Name Equivalence

Name equivalence

- Two types are name equivalent if they correspond to the same type definition
- `T: type array [1..20] of int;
  x,y: array [1..20] of int;
  w,z,T;
  v: T;
  x and y are of same type, w, z, v are of same type, but x and w are of different types!
- A application of a type constructor is a type definition. Red array... is ONE TYPE DEFINITION. Blue array... is ANOTHER TYPE DEFINITION.
Name Equivalence

type cell = ... // record/struct type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink