Overview

- What is a Type?
- Static vs. Dynamic Typing
- Kinds of Typing
- Polymorphic types
- Overloading
- User Data Types

References

What Is a Type?

- Type errors:
  - ? 5 + []
    - ERROR: Illegal Haskell 98 class constraint in inferred type
    - *** Expression : 5 + []
    - *** Type : Num [a] => [a]

- A type is a set of values:
  - Int = {..., -2, -1, 0, 1, 2, ...}
  - Bool = {True, False}
  - Point = { [x=0, y=0], [x=1, y=0], [x=0, y=1], ...}

- A type is a partial specification of behavior:
  - n, m : Int ➔ n+m is valid, but not(n) is an error
  - n : Int ➔ n = 1 is valid, but n = “hello world” is an error
Static and Dynamic Typing

- Values have static types defined by the programming language. Variables and expressions have dynamic types determined by the values they assume at runtime.

- A language is *statically typed* if it is always possible to determine the (static) type of an expression based on the program text alone.

- A language is *strongly typed* if it is possible to ensure that every expression is type consistent based on the program text alone.

- A language is dynamically typed if only values have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

- Type consistency may be assured by (i) compile-time type-checking, (ii) type inference, or (iii) dynamic type-checking.
Kinds of Types

All programming languages provide some set of built-in types.

- Most strongly-typed modern languages provide for additional user-defined types:
  - **Primitive types**: booleans, integers, floats, chars ...
  - **Composite types**: functions, lists, tuples ...
  - **User-defined types**: enumerations, recursive types, generic types ...

The Type Completeness Principle (Watt):

No operation should be arbitrarily restricted in the types of values involved.

- First-class values can be evaluated, passed as arguments and used as components of composite values. Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.
Types in Haskell

- Luca Cardelli: Haskell is a *typeful* language.

- All Haskell values are “first-class”. They may be passed as arguments to functions, returned as results, placed in data structures, etc.

- Haskell types, on the other hand, are *not* first-class. Types describe values, and the association of a value with its type is called *typing*.

- Haskell’s type system is different and somewhat richer than the most. An adjustment to the power and complexity of Haskell’s type system may be very difficult for those having only experience with relatively “untypeful” languages like Perl, Tcl, or Scheme. It will be easier for those, which are familiar with Java, C, Modula, or ML.
Function Types

- Functions types allow one to deduce the types of expressions without the need to evaluate them:
  \[ \text{fac :: Int -> Int} \]
  \[ \text{42 :: Int} \rightarrow \text{fac 42 :: Int} \]

- Curried types:
  \[ \text{a}_1 \rightarrow \text{a}_2 \rightarrow \ldots \rightarrow \text{a}_n \equiv \text{a}_1 \rightarrow (\text{a}_2 \rightarrow (\ldots \rightarrow \text{a}_n \ldots)) \]
  and
  \[ f \text{x}_1 \text{x}_2 \ldots \text{x}_n \equiv ((f \text{x}_1) \text{x}_2) \ldots \text{x}_n \]
  so
  \[ (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

This example also demonstrates the first-class nature of functions (partial application of a curried function), which when used in this way are usually called higher-order functions. See also “Higher-order Functions”.

Com S 541
List and Tuple Types

- List types:
  A list of values of type a has the type [a]:
  
  \[
  [1] :: [\text{Int}]
  \]

  Note: All elements in a list must have the same type!
  
  ['a', 2, False] - this is illegal! It cannot be typed!

- Tuple types:
  If the expressions \(x_1, x_2, \ldots, x_n\) have types \(a_1, a_2, \ldots, a_n\) respectively, then the tuple \(x_1, x_2, \ldots, x_n\) has type \((a_1, a_2, \ldots, a_n)\):
  
  \[(1, [2], 3) :: (\text{Int}, [\text{Int}], \text{Int})\]
  
  \[('a', \text{False}) :: (\text{Char}, \text{Bool})\]
  
  \[((1,2), (3,4)) :: ((\text{Int,Int}), (\text{Int,Int}))\]

  The unit type is written () and has a single element, which is also written as ().
Polymorphism

Languages like Pascal have monomorphic type systems: every constant, variable, parameter and function result has a unique type. Such languages hinders, however, the definition of generic abstraction, if possible at all.

Haskell incorporates *polymorphic* types. In fact, these types are universally quantified. Polymorphic type expressions describe families of types. For example, $(\forall a) [a]$ is the family of types consisting of, for every type $a$, the type of lists of $a$.

Haskell has only universally quantified types. Therefore, one does not need to write out the symbol for universal quantification. All type variables are implicitly universally quantified.

See also: “Polymorphic Types”
We can deduce the types of expressions using polymorphic functions by simply binding type variables to concrete types:

Consider:

- `length :: [a] -> Int`
- `map :: (a -> b) -> [a] -> [b]`

Then:

- `map length :: [[a]] -> Int`
- `[ "Hello", "World" ] :: [[Char]] -- String = [Char]`
- `map length [ "Hello", "World" ] :: [Int]`
The polymorphic type inference yields the function’s *principle type*, which is the least general type that, intuitively, “contains all instances of the expression”. The existence of unique principle types is a feature of all Hindley-Milner type systems.

The Hindley-Milner type inference provides an effective algorithm for automatically determining the types of polymorphic functions. The corresponding type system is used in many modern functional languages, including ML and Haskell.
User-defined Types

- New data types can be introduced by specifying (i) optionally a context, (ii) a datatype name, (iii) a set of parameter types, and (iv) a set of constructors for elements of the type:

  \[
  \text{data} \ [\text{Context} \Rightarrow \ ] \ \text{DatatypeName} \ a_1 \ldots \ a_n = \text{Constructor}_1 \ | \ \ldots \ | \ \text{Constructor}_m
  \]

- An important predefined type in Haskell is that of truth values:

  \[
  \text{data} \ \text{Bool} = \text{False} \ | \ \text{True}
  \]

The type Bool has exactly two values: True and False. Type Bool is an example of a nullary type constructor, and True and False are also nullary data constructors.
Examples of User-defined Data Types

- Enumeration types:

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

whatShallIDo Sun = "relax"
whatShallIDo Sat = "go shopping"
whatShallIDo _ = "looks like I’ll have to go to work"
```

- Union types:

```haskell
data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```
Polymorphic User-defined Data Types

- An example of a polymorphic user-defined data type is Point:

  \[
  \text{data Point } a = \text{Pt } a \ a
  \]

  Because of the single constructor, the type Point is also called a tuple type, since it is essentially just a cartesian product of other types (tuples are like records in other languages). In contrast, multi-constructor types, such as Bool and Day, are called (disjoint) union or sum type.

- More importantly, Point is a polymorphic type: for any type \( t \), it defines the type of the cartesian points that use \( t \) as the coordinate type.

- Furthermore, the type of the binary data constructor Pt is \( a \rightarrow a \rightarrow \text{Point } a \), and thus the following typings are valid:

  - \( \text{Pt 2.0 3.0 :: Point Float} \)
  - \( \text{Pt ‘a’ ‘b’ :: Point Char} \)
  - \( \text{Pt True False :: Point Bool} \)
Recursive Data Types

- Types can also be recursive, as in the type of binary trees:
  
  ```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Here we have defined a polymorphic binary tree whose elements are either leaf nodes containing a value of type `a`, or internal ("branches") containing (recursively) two sub-trees.

- When reading data declarations such as this, remember again that `Tree` is a type constructor, whereas `Leaf` and `Branch` are data constructors, which have the following types:
  
  ```haskell
Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a
```

- Examples:
  
  ```haskell
  countLeaves :: Tree a -> Int
  leaves :: Tree a -> [a]
  countLeaves (Leaf _) = 1
  leaves (Leaf x) = [x]
  countLeaves (Branch l r) =
  (countLeaves l) + (countLeaves r)
  (leaves l) ++ (leaves r)
  ```
Kinds of Polymorphism

- **Polymorphism**:  
  - **Universal**:  
    - Parametric: polymorphic map function in Haskell; nilpointer type in Pascal  
    - Inclusion: subtyping – graphic objects

- **Ad Hoc**:  
  - Overloading: + applies to both integers and floating point numbers  
  - Coercion: integer values can be used where floating point numbers are expected and vice versa

- Coercion or overloading – how does one distinguish?
  - \( 3 + 4 \)
  - \( 3.0 + 4 \)
  - \( 3 + 4.0 \)
  - \( 3.0 + 4.0 \)
Overloading

- The overloaded behavior of operators is different for each type. In fact, sometimes the behavior is undefined, or error.

- On the other hand, parametric polymorphism the type truly does not matter. For example, the function countLeaves really does not care what kind of elements are found in the leaves of the tree.

- In Haskell, *type classes* and *instance declaration* provide a structured way to control *ad hoc* polymorphism, or overloading.
Overloaded operators are introduced by means of type classes:

```haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

The class Eq defines two operations, one for equality, the other for inequality. It also demonstrates the use of a default method, in this case for the operation inequality. If a method for a particular operation is omitted in an instance declaration, then the default method of the class, if it exists, is used instead.
Instances of Eq

For each overloaded instance a separate definition must be given:

```
instance Eq Int where
    x == y = x `primEqInt` y

instance Eq Bool where
    True == True = True
    False == False = True
    _ == _ = False

instance (Eq a) => Eq (Tree a) where
    Leaf a == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) && (r1 == r2)
    _ == _ = False
```
We may wish to define a class Ord, which inherits all of the operations of Eq, but in addition has a set of comparison operations and minimum and maximum functions:

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

This class declaration uses a context, (Eq a), which states that Eq is a superclass of Ord (conversely, Ord is a subclass of Eq), and any type which is an instance of Ord must also be an instance of Eq.

As an example of the use of Ord, consider the principle type of the function quicksort:

```haskell
quicksort :: (Ord a) => [a] -> [a]
```

In other words, quicksort only operates on lists of values of ordered types.
Higher-order Types

- The type constructor `Tree` has always been paired with an argument, as in `Tree Int` (the tree containing integer values) or `Tree a` (representing the family of trees containing `a` values).

- But `Tree` by itself is a type constructor, and as such takes a type as an argument and returns a type as a result. There are no values in Haskell that have this type, but such "higher-order" types can be used in class declarations:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```
The Function \( \text{fmap} \)

The function \( \text{fmap} \) is a generalization of \( \text{map} \) and the type variable \( f \) is applied to other types in \( f \, a \) and \( f \, b \). An instance of \( \text{Functor} \) for type \( \text{Tree} \) would be:

\[
\text{instance} \quad \text{Functor} \quad \text{Tree} \quad \text{where} \\
\quad \text{fmap} \, f \, (\text{Leaf} \, x) = \text{Leaf} \, (f \, x) \\
\quad \text{fmap} \, f \, (\text{Branch} \, l \, r) = \text{Branch} \, (\text{fmap} \, f \, l) \, (\text{fmap} \, f \, r)
\]

This instance declaration declares that \( \text{Tree} \), rather than \( \text{Tree} \, a \), is an instance of \( \text{Functor} \). The function \( \text{fmap} \) can work uniformly over arbitrary data types.

Category theory: A functor is a structure-preserving map between categories (both objects and arrows are mapped).
The type system detects typing errors in expressions. But what about errors due to malformed type expressions? The expression

\[(+) \ 1 \ 2 \ 3\]

results in a type error, since \((+)\) takes only two arguments. Similarly, the type Tree Int Int should produce the same kind of error.

The solution is a second type system, which ensures the correctness of types! Each type has an associated \textit{kind}, which ensures that the type is used correctly.

The expressions are classified into different \textit{kinds}, which can take two possible forms:

- The symbol \(*\) represents the kind of type associated with concrete data objects. That is, if the value \(v\) has type \(t\), the kind of \(v\) must be \(*\).
- If \(k_1\) and \(k_2\) are kinds, then \(k_1 \to k_2\) is the kind of types that take a type of kind \(k_1\) and return a type of kind \(k_2\).
A general statement about OOP, simply substituting type class for class, and type for object, yields a valid summary of Haskell’s type class mechanism:

“Classes capture common sets of operations. A particular object may be an instance of a class, and will have a method corresponding to each operation. Classes may be arranged hierarchically, forming notions of super classes and sub classes, and permitting inheritance of operations/methods. A default method may also be associated with an operation.”

In contrast to OOP, types are not objects, and in particular there is no notion of an object’s or type’s internal state. Methods in Haskell are completely type-safe. Furthermore, methods are not “looked up” at runtime but are simply passed as higher-order functions.
The classes used by Haskell are similar to those used in other object-oriented languages like C++ and Java. There are, however, some significant differences:

- Haskell separates the definition of a type from the definition of the methods associated with that type.
- The class methods defined by a Haskell class correspond to virtual functions in a C++ class. Each instance of a class provides its own definition for each method; class defaults correspond to default definitions for a virtual function in the base class.
- Haskell classes are roughly similar to Java interfaces. Like an interface declaration, a Haskell class declaration defines a protocol for using an object rather than defining an object itself.
- Haskell does not support the C++ overloading style, in which functions with different types share a common name.
- The type of a Haskell object cannot be implicitly coerced; there is no universal base class such as Object, which values can be projected into or out of.
- There is no access control (e.g. public or private). Instead, the module system must be used to hide or reveal components of a class.