

# Programming Language Concepts

## Type Systems

Onur Tolga Şehitoğlu

Computer Engineering



# Outline

- 1 Type Systems
- 2 Polymorphism
  - Inclusion Polymorphism
  - Haskell Class Hierarchy
- Parametric Polymorphism
- 3 Overloading
- 4 Coercion
- 5 Type Inference

# Type Systems

Design choices for types:

- **monomorphic** vs **polymorphic** type system.
- **overloading** allowed?
- **coercion**(auto type conversion) applied, how?
- **type relations and subtypes** exist?

# Polymorphism

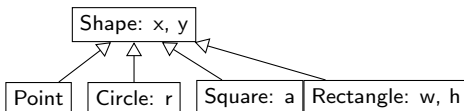
- **Monomorphic** types: Each value has a single specific type. Functions operate on a single type. C and most languages are monomorphic.
- **Polymorphism**: A type system allowing different data types handled in a uniform interface:
  - 1 **Ad-hoc polymorphism**: Also called overloading. Functions that can be applied to different types and behave differently.
  - 2 **Inclusion polymorphism**: Polymorphism based on subtyping relation. Function applies to a type and all subtypes of the type (class and all subclasses).
  - 3 **Parametric polymorphism**: Functions that are general and can operate identically on different types

# Subtyping

- C types:  
 $\text{char} \subseteq \text{short} \subseteq \text{int} \subseteq \text{long}$
- Need to define arithmetic operators on them separately?
- Consider all strings, alphanumeric strings, all strings from small letters, all strings from decimal digits.  
Need to define special concatenation on those types?
- $f : T \rightarrow V$  ,  $U \subseteq T \Rightarrow f : U \rightarrow V$
- Most languages have arithmetic operators operating on different precisions of numerical values.

# Inheritance

- `struct Point { int x, y; };  
    struct Circle { int x, y, r; };  
    struct Square { int x, y, a; };  
    struct Rectangle { int x, y, w, h; };`
- `void move (Point p, int nx, int ny) {  
       p.x=nx; p.y=ny;}`
- Moving a circle or any other shape is too different?



## Haskell extensible records:

```
import Hugs.Trex;  -- Only in -98 mode!!!

type Shape = Rec (x::Int, y::Int)
type Circle = Rec (x::Int, y::Int, r::Int)
type Square = Rec (x::Int, y::Int, w::Int)
type Rectangle = Rec (x::Int, y::Int, w::Int, h::Int)

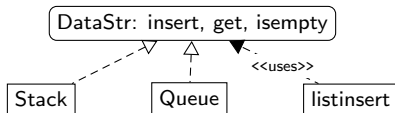
move (x=_,y=_|rest) b c = (x=b,y=c|rest)

(a::Shape)=(x=12,y=24)
(b::Circle)=(x=12,y=24,r=10)
(c::Square)=(x=12,y=24,w=4)
(d::Rectangle)=(x=12,y=24,w=10,h=5)

Main> move b 4 5
(r = 10, x = 4, y = 5)
Main> move c 4 5
(w = 4, x = 4, y = 5)
Main> move d 4 5
(h = 5, w = 10, x = 4, y = 5)
```

# Haskell Classes

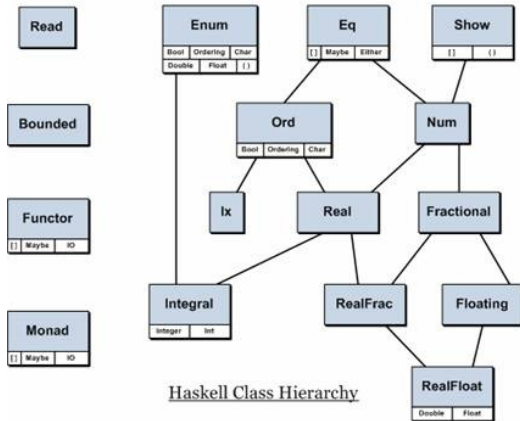
- Subtyping hierarchy based on classes
- An instance implements interface functions of the class
- Functions operating on classes (using interface functions) can be defined



- Called **interface** in OO programming



# Haskell Class Hierarchy



```

class DataStr a where
  insert :: (a v) -> v -> (a v)
  get :: (a v) -> Maybe (v, (a v))
  isempty :: (a v) -> Bool

instance DataStr Stack where
  insert x v = push v x
  get x = pop x
  isempty Empty = True
  isempty _ = False

instance DataStr Queue where
  insert x v = enqueue v x
  get x = dequeue x
  isempty EmptyQ = True
  isempty _ = False

insertlist :: DataStr a => (a v) -> [v] -> (a v)
insertlist x [] = x
insertlist x (el:rest) = insertlist (insert x el) rest

data Stack a = Empty | St [a] deriving Show
data Queue a = EmptyQ | Qu [a] deriving Show

```

# Parametric Polymorphism

- **Polymorphic** types: A value can have multiple types.  
Functions operate on multiple types **uniformly**
- **identity** `x = x` function. type:  $\alpha \rightarrow \alpha$   
`identity 4 : 4`, `identity "ali" : "ali"` , `identity (5,"abc") : (5,"abc")`  
 $int \rightarrow int$ ,  $String \rightarrow String$ ,  $int \times String \rightarrow int \times String$
- **compose** `f g x = f (g x)` function  
 type:  $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$   
`compose square double 3 : 36`,  
 $(int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \rightarrow int$ .  
`compose listsum reverse [1,2,3,4] : 10`  
 $([int] \rightarrow int) \rightarrow ([int] \rightarrow [int]) \rightarrow [int] \rightarrow int$

- `filter f [] = []`  
`filter f (x:r) = if (f x) then x:(filter f r) else (filter f r)`  
 $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$   
`filter ((<) 3) [1,2,3,4,5,6] : [4,5,6]`  
 $(\text{int} \rightarrow \text{Bool}) \rightarrow [\text{int}] \rightarrow [\text{int}]$   
`filter identity [True, False, True, False] :`  
`[True, True]`  
 $(\text{Bool} \rightarrow \text{Bool}) \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]$
- Operations are same, types are different.
- Types with type variables: [polytypes](#)
- Most functional languages are polymorphic
- Object oriented languages provide polymorphism through inheritance

# Overloading

- **Overloading**: Using same identifier for multiple places in same scope
- Example: Two different functions, two distinct types, same name.
- Polymorphic function: one function that can process multiple types.
- C++ allows overloading of functions and operators.

```
typedef struct Comp { double x, y; } Complex;
double mult(double a, double b) { return a*b; }
Complex mult(Complex s, Complex u) {
    Complex t;
    t.x = s.x*u.x - s.y*u.y;
    t.y = s.x*u.y + s.y*u.x;
    return t;
}
Complex a,b; double x,y; ... ; a=mult(a,b) ; x=mult(y,2.1);
```

- Binding is more complicated. not only according to name but **according to name and type**

- Function type:

`name : parameters → result`

- **Context dependent overloading:**   
Overloading based on function name, parameter type and return type.
- **Context independent overloading :**   
Overloading based on function name and parameter type. No return type!

# Context dependent overloading

## ■ Which

type does the expression calling the function expects (context) ?

```
int f(double a) { .... ① }
int f(int a) { .... ② }
double f(int a) { .... ③ }
double x,y;
int a,b;
```

- `a=f(x);` ① (x double)
- `a=f(a);` ② (a int, assign int)
- `x=f(a);` ③ (a int, assign double)
- `x=2.4+f(a);` ③ (a int, mult double)
- `a=f(f(x));` ②(①) ( x double, f(x):int, assign int)
- `a=f(f(a));` ②(②) or ①(③) ???
- Problem gets more complicated. (even forget about coercion)

# Context independent overloading

- Context dependent overloading is more expensive.
- Complex and confusing. Useful as much?
- Most overloading languages are context independent.
- Context independent overloading forbids ② and ③ functions defined together.
- “name: parameters” part should be unique in “name: parameters → result”, in the same scope
- Overloading is not much useful. So languages avoid it.

## Use carefully:

Overloading is useful only for functions doing same operations. Two functions with different purposes should not be given same names. Confuses programmer and causes errors

- Is variable overloading possible? What about same name for two types?



# Coercion

- Making implicit type conversion for ease of programming.

```
double x;      int k;
x = k+4.2;     /* x = (double) k + 4.2 */
k = x+3.45;    /* k=(int) (x+3.45); */
k = x+2;       /* k=x+(double)2; */
k = x+k-2;     /* k=(int)(x+ (double)k - (double)2) ; */
```

- C makes *int*  $\leftrightarrow$  *double* coercions and pointer coercions (with warning)
- Are other type of coercions are possible? (like  $A * \rightarrow A$ ,  $A \rightarrow A *$ ). Useful?
- May cause programming errors:  $x=k=3.25$  :  $x$  becomes 3.0
- Coercion + Overloading: too complex.
- Most newer languages quit coercion completely (Strict type checking)

# Type Inference

- Type system may force user to declare all types (C and most compiled imperative languages), or
- Language processor infers types. How?
- Each expression position provide information (put a constraint) on type inference:
  - Equality  $e = x, x :: \alpha, y :: \beta \Rightarrow \alpha \equiv \beta$
  - Expressions  $e = a + f\ x, + :: Num \rightarrow Num \rightarrow Num \Rightarrow a :: Num, f :: \alpha \rightarrow Num, e :: Num$
  - Function application  $e = f\ x \Rightarrow e :: \beta, x :: \alpha, f :: (\alpha \rightarrow \beta)$
  - Type constructors  $f\ (x : r) = t \Rightarrow x :: \alpha, t :: \beta, f :: ([\alpha] \rightarrow \beta)$
- Inference of all values start from the most general type (i.e: any type  $\alpha$ )
- Type inference finds the **most general type** satisfying the constraints.