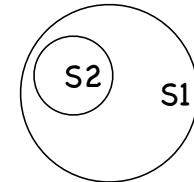


Subtyping and Polymorphism

©Andrey Kruglyak, 2010

Subtyping

- ◀ Subtyping is a reflexive, transitive relation on types
- ◀ $S2 < S1$
 - = S2 is a subtype of S1
 - = S1 is supertype of S2
 - = S2 can be used in every context where S1 is expected
- ◀ Timber allows multiple subtyping ($A < B, C$), but many languages with subtyping (e.g. Java) only allow single subtyping to simplify implementation



Subtyping example 1

```
struct Product where  
  getPrice :: Request Int
```

```
struct Car<Product where  
  getMilage ::Request Int
```

- ◀ Car has two methods: getPrice and getMilage

- ◀ Note that we cannot have two struct types with identical selector name in the same module in Timber

```
f ::Product->Request Int  
f p = p.getPrice
```

```
g :: Car->Request Int  
g c = c.getMilage
```

```
test1 (p::Product) = do  
  f p  
  g p -- type error
```

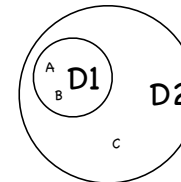
```
test2 (c::Car) = do  
  f c -- okay  
  g c
```

Subtyping example 2

```
data D1 = A | B
```

```
data D2 > D1 = C
```

- ◀ D2 has three constructors: A, B, and C



```
f :: D1->Int  
f x = if x == A then 1  
      else 0
```

```
g :: D2->Int  
g x = if X == C then 1  
      else 0
```

```
test1 (d1::D1) = do  
  f d1  
  g d1 -- okay
```

```
test1 (d2::D2) = do  
  f d2 -- type error  
  g d2
```

Subtyping & type constructors

- One could reasonably expect that most type constructors are covariant in their arguments, i.e. if $A < B$, then $[A] < [B]$,
(A,A) < (B,B), Array A < Array B, Class A < Class B,
Request A < Request B, Maybe A < Maybe B, ...
- However, this has not yet been implemented in Timber (but this has been studied and implemented in O'Haskell, a precursor to Timber)
- But the type constructor " \rightarrow " is contra-variant in its argument and co-variant in its result

\rightarrow : covariance in result

- If $A < B$, then for any type T holds:
 $(T \rightarrow A) < (T \rightarrow B)$
- This is easy to understand:
 - the only thing one can do with a function is to apply it
 - if I expect a function that returns B and I'm given a function that returns A, then I can apply this function on some value $t :: T$, get an A, but - any A is a B, so get what I need

\rightarrow : contravariance in argument

- On the other hand, if $A < B$, then for any type T holds:
 $(B \rightarrow T) < (A \rightarrow T)$
- Absence of covariance:
If I expect a function that takes B and I'm given a function that takes A, then I can apply this function on some value $b :: B$, and that may lead to an error since my value $b :: B$ may not be an A

\rightarrow : contravariance in argument

- On the other hand, if $A < B$, then for any type T holds:
 $(B \rightarrow T) < (A \rightarrow T)$
- Proof of contravariance:
If I expect a function that takes A and I'm given a function that takes B, then I can apply this function on some value $a :: A$ which is also of type B (since $A < B$), and that will work

```

1 module Test where
2
3 import POSIX
4
5 struct Product where
6   getPrice :: Request Int
7
8 mkProd p = class
9   getPrice = request
10  result p
11  result Product{..}
12
13 struct Car < Product where
14   getMilage :: Request Int
15
16 mkCar p = class
17   getPrice = request
18   result p
19   getMilage = request
20   result m
21   result Car{..}
22
23 f :: Product -> Request Int
24 f p = p.getPrice
25
26 g :: Car -> Request Int
27 g c = c.getMilage
28
29 test1 :: (Product -> Request Int) -> Product -> Cmd () Int
30 test1 fcn x = do
31   v <- fcn x
32   result v
33
34 test2 :: (Car -> Request Int) -> Car -> Cmd () Int
35 test2 fcn x = do
36   v <- fcn x
37   result v
38
39 root :: RootType
40 root w = do
41   c = new mkCar 1 3
42   p = new mkProd 7
43   test1 g p -- type error:
44   -- cannot send "g" of type (Car -> Request Int) to a context that expects
45   -- (Product -> Request Int),
46   -- so (Car -> Request Int) is not a subtype of (Product -> Request Int)
47   test2 f c -- okay:
48   -- can send "f" of type (Product -> Request Int) to a context that expects
49   -- (Car -> Request Int),
50   -- so (Product -> Request Int) < (Car -> Request Int)

```

Wednesday, April 28, 2010

9

Polymorphism

- ◀ Sometimes the same function can work for all types


```
head x:xs = x
applyTwice f x = f (f x)
```
- ◀ Such functions are called polymorphic, and we use type variables in their types (read: "for any type a")


```
head :: [a] -> a
applyTwice :: (a->a) -> a -> a
```
- ◀ Note that the type is correct even though head [] is not defined; this application would result in an error value, which is a member of all types

Wednesday, April 28, 2010

10

Restricted polymorphism

- ◀ However, sometimes there are additional requirements on the type of a polymorphic function
- ◀ Let us consider the equality operator (==). We can try to define its type as:


```
(==) :: a -> a -> Bool
```

 Here lies the problem – it may not work on some type T since (==) may not be defined on that type! This will work with Int, Char, Bool, but can we make it work with some of the types that we define ourselves?

Wednesday, April 28, 2010

11

Typeclasses

- ◀ As we still want the type to be as general as possible, we define `typeclass Eq a where`

```
(==), (/=) :: a -> a -> Bool
```
- ◀ So for any type that belongs to this typeclass, we can compare terms of the type using (==)
- ◀ Then we can use this typeclass to restrict the type of a polymorphic function (it will have a qualified type):


```
compare :: [a] -> [a] -> Bool \\Eq a
compare [] [] = True
compare (x:xs) (y:ys) = (x == y) && (compare xs ys)
compare _ _ = False
```

Wednesday, April 28, 2010

12

Instances of typeclasses

- ◀ But how do we make a type into a member of this typeclass?

- ◀ We need to provide implementation of the operators (==) and (/=). Here we make the type Bool into an instance of the typeclass Eq:

```
instance eqBool :: Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
  x     /= y     = not (x==y)
```

- ◀ And we can do the same to a pair of any types that belong to Eq:

```
instance eqPair :: Eq (t1,t2) \\  
Eq t1, Eq t2 where  
  (a,b) == (c,d) = a==c && b==d  
  x /= y       = not (x==y)
```

Important typeclasses from Prelude

- ◀ `typeclass Eq a where`
`(==), (/=) :: a -> a -> Bool`
- ◀ `typeclass Ord a < Eq a where`
`(<), (<=), (>), (>=) :: a -> a -> Bool`
- ◀ `typeclass Show a where`
`show :: a -> String`
- ◀ `typeclass Parse a where`
`parse :: String -> Either String a`
- ◀ `typeclass Enum a where`
`fromEnum :: a -> Int`
`toEnum :: Int -> a`

More on Timber's typeclasses

- ◀ Instances are defined separately from typeclasses
=> we can easily extend typeclasses to new types, even those that did not exist when we defined the typeclass, and the code written for elements of the typeclass will work on elements of the new type as well
- ◀ Instance definition for a type is separate from a class definition of that type
=> we can make a type into an instance of a typeclass after defining classes without updating all class definitions

Default instances

- ◀ For some type classes, the compiler can add default implementations of class instances if we declare them:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun  
default eqDay :: Eq Day  
default showDay :: Show Day
```

- ◀ This will, for example, work for Eq and Show instances of data types where constructors do not take any arguments
- ◀ Note that we do not discuss defaulting here, these are just examples!