Types

A Type is a Set of Values

Consider the statement:

```
let mut n : i32;
```

Here we declare n to be a variable of <u>type</u> i32; what we mean, n can take on any value from the <u>set of all 32 bit integer values</u>.

Types

Def: A type is a set of values.

Def: A primitive type is a type a programmer can use but not define.

Def: A constructed type is a user defined type.

Example: Java, primitive type

float q;

type float ⇒ set of all
possible floating point values

q is of type float, only
a value that is a member
of the set of all floating point
values can be assigned to q.

Types

Example: Java, constructed type

```
class Rectangle { int xdim; int ydim; };

Rectangle r = new Rectangle();
```

Now the variable c only accepts values that are members of type Rectangle;
☞ object instantiations of class Rectangle.

Types

Example: Rust, constructed type

```rust
#[allow(dead_code)]

struct Rectangle {
  xdim : i32,
  ydim : i32
}

fn main() {
  let _r: Rectangle = Rectangle {xdim:3, ydim:4};
}
```

an element of
type Rectangle.

Types

<u>Example</u>: C, constructed type

int a[3];

the variable a will accept values
which are arrays of 3 integers.

e.g.: int a[3] = {1,2,3};
int a[3] = {7,24,9}

<u>Example</u>: Rust, constructed type

let a : [i64;3] = ...;

the variable a will accept values
which are arrays of 3 i64 integers.

Subtypes

Def: a subtype is a subset of the elements of a type.

Example: Java

Short is a subtype of int: short $\subset$ int

Observations:

(1) converting a value of a subtype to a values of the super-type is called widening type conversion. (safe)

(2) converting a value of a supertype to a value of a subtype is called narrowing type conversion. (not safe)

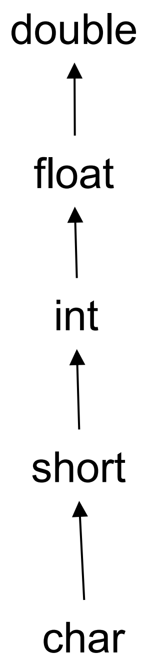
Example: Java

float $\subset$ double

Subtypes give rise to type hierarchies and type hierarchies allow for automatic type coercion – widening conversions!

Subtypes & Type Hierarchies

- Here the arrow means "subtype of", e.g., int is a subtype of float.
- In type hierarchies it is always safe to move from subtype to supertype – widening conversion (coercion)
  - E.g. short -> int ✓
- Never the other way around
  - E.g. int -> short ✗



Subtypes

- Rust does NOT have subtypes
  - Therefore, it does NOT support automatic type coercion
  - It is an error in Rust to do:

    ```
    let x : i64 = 3;
    let i : f64 = x;
    ```

  - You would have to explicitly cast the integer variable as a floating point:

    ```
    let x : i64 = 3;
    let i : f64 = x as f64;
    ```

Function Types

Rust and Haskell (as we will see) treat functions as just another data type that can be manipulated in a very elegant way.

☞Functions can be passed as values; just as values that belong to other data types
☞Functions belong to **function types**

Example: in Rust consider the function type

x: fn(f64) $\rightarrow$ f64

This type represents the set of all functions from f64 to f64.

Looking at the Rust standard library std::f64 we can find some examples from this set of functions

floor: fn(f64) $\rightarrow$ f64
ceil: fn(f64) $\rightarrow$ f64
round: fn(f64) $\rightarrow$ f64

Function Types

<u>Example:</u> Functions as values

```
fn myround (x:f64) -> i64 {
  x.round() as i64
}

fn main () {
  let foo: fn(f64)->i64 = myround;
  let v : i64 = foo(3.4);
  println!("{}", v);
}
```

☞ A function is just an element of a particular function set.

Function Types

Example: Functions as function arguments

```
fn myfloor (x:f64) -> i64 {
  x.floor() as i64
}

fn myceil (x:f64) -> i64 {
  x.ceil() as i64
}

fn myfun (x: fn(f64)->i64, v:f64) -> i64 {
    x(v)
}

fn main () {
    println!("{}", myfun(myfloor, 3.4));
    println!("{}", myfun(myceil, 3.4));
}
```

☞ A function is just an element of a particular function set.

Why do we use types?

- Types allow the computer/language system to assist the developer write better programs. Type mismatches in a program usually indicate some sort of programming error.
  - Static type checking – check the types of all statements and expressions at compile time.
  - Dynamic type checking – check the types at runtime.

Type Equivalence

I. Name (nominal) Equivalence – two objects are of the same type if and only if they share the same type name.

Example: Rust

```
struct Type1 {x:i64, y:i64}
struct Type2 {x:i64, y:i64}

fn main () {
    let x: Type1 = Type1{x:1,y:2};
    let y: Type2 = x;
    println!("{:?}",y);
}
```

Error; even though the types look the same, their names are different, therefore, Rust will not compile.

☞Rust uses name equivalence

Type Equivalence

II. Structural Equivalence – two objects are of the same type if and only if they share the same type structure.

Example: Haskell

```haskell
type Type1 = (Integer, Integer)
type Type2 = (Integer, Integer)

x :: Type1
y :: Type2

x = (1,2)
y = x
```

Even though the type names are different, Haskell correctly recognizes this statement.

☞ Haskell uses structural equivalence.

Type Inference

- Type inference refers to the automatic detection of the data type of an expression in a programming language.
- To see how this might work let's work through an example.

Type Inference

- Assume we have the following statements in a programming language like Rust:

```
let x : i64 = 3;
let y : i64 = 2 * x;
```

- We want to make sure that all the assignments are legal.
- We will use the type notation '3.{i64}' indicating that this syntactic unit has the type i64.

Type Inference

- We start at the primitives on the right side of the assignments starting with the first statement and then stepping through all the remaining statements

Type Inference

let x : i64 = 3.{i64};
let y : i64 = 2 * x;

let x : i64 = 3.{i64};
let y : i64 = 2 * x;

let x : i64 = 3;
let y : i64 = 2.{i64} * x;

let x : i64 = 3;
let y : i64 = 2.{i64} * x.{i64};

let x : i64 = 3;
let y : i64 = 2 *.{fn(i64,i64)->i64} x;

let x : i64 = 3;
let y : i64 = 2 *.{fn(i64,i64)->i64} x;

let x : i64 = 3;
let y : i64 = 2 * x;

Type Inference

- Let’s try a program with a bug in it:

```
let x : i64 = 3;
let y : f64 = 2 * x;
```

Type Inference

```
let x : i64 = 3.{i64};
let y : f64 = 2 * x;
```

```
let x : i64 = 3.{i64}; ✓
let y : f64 = 2 * x;
```

```
let x : i64 = 3; ✓
let y : f64 = 2.{i64} * x;
```

```
let x : i64 = 3; ✓
let y : f64 = 2.{i64} * x.{i64};
```

```
let x : i64 = 3; ✓
let y : f64 = 2 *.{fn(i64,i64)->i64} x;
```

```
let x : i64 = 3; ✓
let y : f64 = 2 *.{fn(i64,i64)->i64} x; ✗
```

```
let x : i64 = 3; ✓
let y : f64 = 2 * x; ✗
```

Types & Objects

- In any OO language class definitions create new types
- Objects are the values in those types
- In OO languages that support inheritance, inheritance creates a subtype-supertype relationship in the class hierarchy

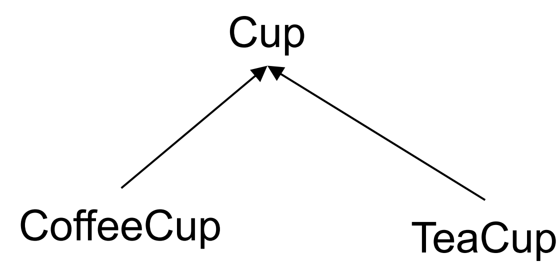
Types & Objects

<u>Example</u>: Java

```
class Cup { ... };
class CoffeeCup extends Cup { ... };
class TeaCup extends Cup { ... };
```

Which ones of the following statements are safe and which ones are not?

1. Cup x = new Cup();
2. Cup y = new CoffeeCup();
3. TeaCup z = new Cup();
4. TeaCup t = new TeaCup();
   Cup c = t;



Notation:
A → B means A is subtype of B

Note: Type coercion in type hierarchies gives rise to polymorphic programming in OO - objects can appear in different type contexts.

Exercises

- Describe the type associated with the set of values {-1,-2,-3,-4,…}, call this type Q.
- Describe the type associate with the set of values {-2,-4,-6,-8,…}, call this type P.
- Is there a subtype-supertype relationship between this types? If so, what is it?
- Let x be a variable of type Q and y be a variable of type P, then is the assignment

  x := y

  a safe assignment? Why? Why not?
- Describe the type associated with set $Q \rightarrow P$.

Take Away

- Types are sets of values, typically with a common representation and common set of operations.
- Types in programming languages allows compilers and interpreters to check for consistency in your programs.
- Inconsistencies usually show up a type mismatches.
- Type equivalence between constructed types can be established in one of two ways, name equivalence or structural equivalence.
- Class hierarchies in OO languages give rise to subtype-supertype relationships due to inheritance.

Assignments

- Assignment #2 – See BrightSpace