Types

A Type is a Set of Values

Consider the C statement:

```
int n = 3;
```

Here we constrain n to take on any value from the set of all integer values.

Reading: MPL Chap 6

Types

Def: A type is a set of values.

Def: A primitive type is a type that is built into the language, e.g., integer, string.

Def: A constructed type is a user defined type, e.g., any type introduced by the user. In Asteroid this is done through the 'structure' statement.

Example: C, 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

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

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

Rectangle r = new Rectangle();
```

Now the variable r only accepts values that are members of type Rectangle;
☞ <u>object instantiations</u> of class Rectangle.

Types

Example: Asteroid, constructed type

```
structure Rectangle with
  data xdim.
  data ydim.
end

let r:%Rectangle = Rectangle(4,2).
```

an element of
type Rectangle.

Types

In statically typed languages arrays are also considered ‘constructed 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}

That is, ‘int a[3]’ defines the set of all integer arrays of size three.

Subtypes

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

Example: C

The notation A < B means A is a subtype of B.

Short is a subtype of int: short < 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)

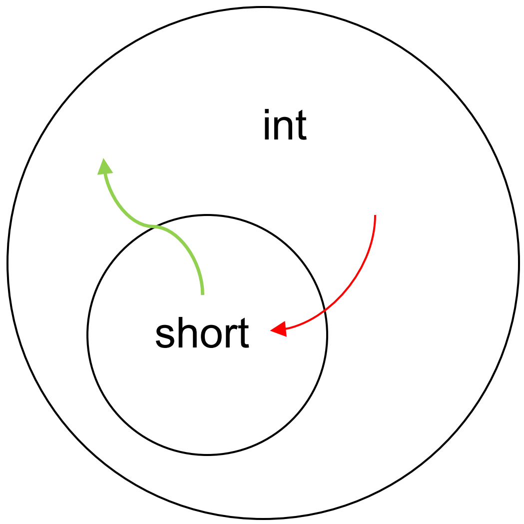
Example: C, partial type hierarchy

char < short < int < float < double

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

Subtypes

- A convenient way to visualize subtypes is using Venn diagrams
- Consider,
  short < int
- It is easy to see that the shorts are a subset of the integer values
- The green arrow represents a widening type conversion is always safe
- The red arrow represents a narrowing type conversion and is never safe



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

- Fundamental to type checking is the notion of type equivalence:
  - Figuring out whether two type description are equivalent or not
  - This is especially important for constructed types like class/struct objects.

Type Equivalence

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

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

```
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 <u>name equivalence</u>

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 and to make sure that all expressions and statements are properly typed.
  - We often refer to this as “type checking” a program
- 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 C:

```
int x = 3;
int y = (2 * x);
```

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

Type Inference

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

Type Inference

```
int x = 3;
int y = (2 * x);
```

Type Inference

```
int x = 3.integer;
int y = (2 * x);
```

Start with the primitives on the right-hand side for the first statement

Type Inference

```
int x = 3.integer;  ✓
int y = (2 * x);
```

If we have evaluated a top-level entity, then check against left-hand side.
If it type checks accept it, if not reject it. If you not at top-level keep inferencing.

Type Inference

```
int x = 3.integer;
int y = (2.integer * x.integer);
```

Process the next statement

Type Inference

```
int x = 3.integer;
int y = (2.integer * x.integer).integer;
```

If you not at top-level keep inferencing.

Type Inference

```
int x = 3.integer;
int y = (2.integer * x.integer).integer;   ✓
```

Accept: we can assign an integer value to an integer variable.

If we have evaluated a top-level entity, then check against left-hand side.
If it type checks accept it, if not reject it.

Type Inference

- Let's try a program with a bug in it. In C we have the hierarchy, short < int

```
int x = 3;
short y = (2 * x);
```

Type Inference

```
int x = 3.integer;
short y = (2* x);
```

Type Inference

```
int x = 3.integer;   ✓
short y = (2 * x);
```

If we have evaluated a top-level entity, then check against left-hand side.
If it type checks accept it, if not reject it. If you not at top-level keep inferencing.

Type Inference

```
int x = 3.integer;
short y = (2.integer * x.integer);
```

Type Inference

```
int x = 3.integer;
short y = (2.integer * x.integer).integer;
```

Type Inference

int x = 3.integer;
short y = (2.integer * x.integer).integer; ✗

Reject: cannot assign a member of a supertype to a subtype.

Example: To see this check out the repl at
https://replit.com/@lutzhamel/C-types#int.c

Type Inferencing in Asteroid

- Type inferencing for assignment statements works a little bit different in Asteroid:
  - The types must match exactly, no type conversion is supported during assignments
  - This is because Asteroid does not support a type hierarchy.

```
ast> let r:%real = 1.
error: pattern match failed: conditional pattern match failed
ast> let r:%real = toreal(1).
ast> let r:%real = 1 + 2.5.
error: found 'integer + real' expected 'real + real'
ast> let r:%real = toreal(1) + 2.5.
ast>
```

Exercises

- Let Q be the set of all negative integer values less than zero,
  Q = {-1,-2,-3,-4,-5,-6,-7,-8,-9,…}
- Let P be the set of all negative integer values evenly divisible by two,
  P = {-2,-4,-6,-8,…}
- Then, is the following statement type safe assuming that x is declared as type Q and y is declared as type P?

  x := (y+(-1))

  where -1 is a member of type Q.

  Hint: A type is a set of values!

Exercise

- Answer:
  - First, we have to determine if there is a subtype-supertype relationship between Q and P. There is, because P is a subset of Q,

    $P < Q$

  - Second, now we can do our type inferencing on the statement

    x := (y+(-1))

Exercise

$$x := (y.P + (-1).Q)$$

Start with primitives on rhs.

Exercise

x := (y.Q+(-1).Q)

Both operands of + have to have the same type. We know that P < Q, therefore we can replace P with Q on the left operand to +.

Exercise

$$x := (y.Q+(-1).Q).Q$$

If the input type to + is Q then the output type is also Q

Exercise

$$x.Q := (y.Q+(-1).Q).Q$$ ✓

The variable x was declared as type Q. Therefore, we have an assignment of a Q value to a Q variable which is always safe.

Exercise

- What about the assignment,

  y := (-1)

  is it type safe?

Exercise

$$y := (-1).Q$$

Start with primitives on rhs

Exercise

y.P := (-1).Q ✘

Look at type on lhs. NOT type safe because P < Q. You cannot store a value from a supertype into a variable of a subtype.

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

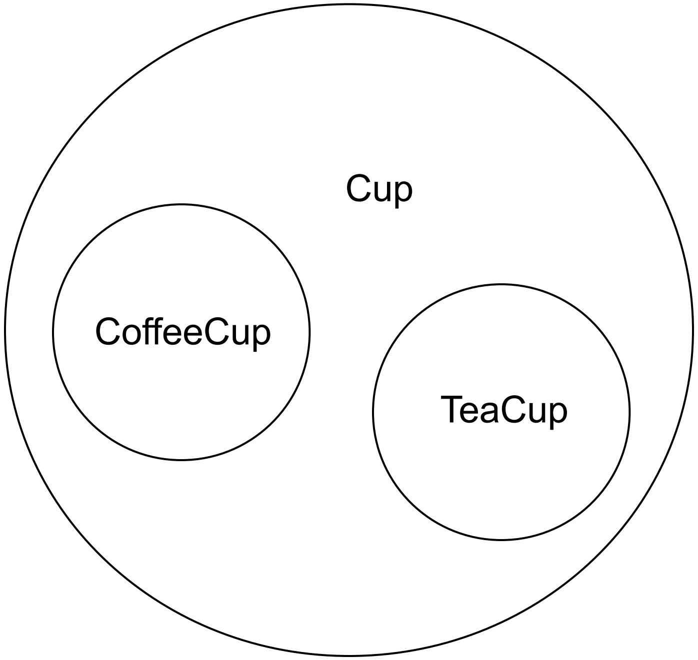
Types & Objects

Example: 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;



CoffeeCup < Cup
TeaCup < Cup

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

Object-Oriented Programming

- Classic OO languages are based around inheritance hierarchies.
- The main distinguishing feature between them is whether they support single or multiple inheritance.
  - C++ and Python support multiple inheritance
  - Java supports single inheritance
- There are three main problems with inheritance-based OO languages.

Problem #1

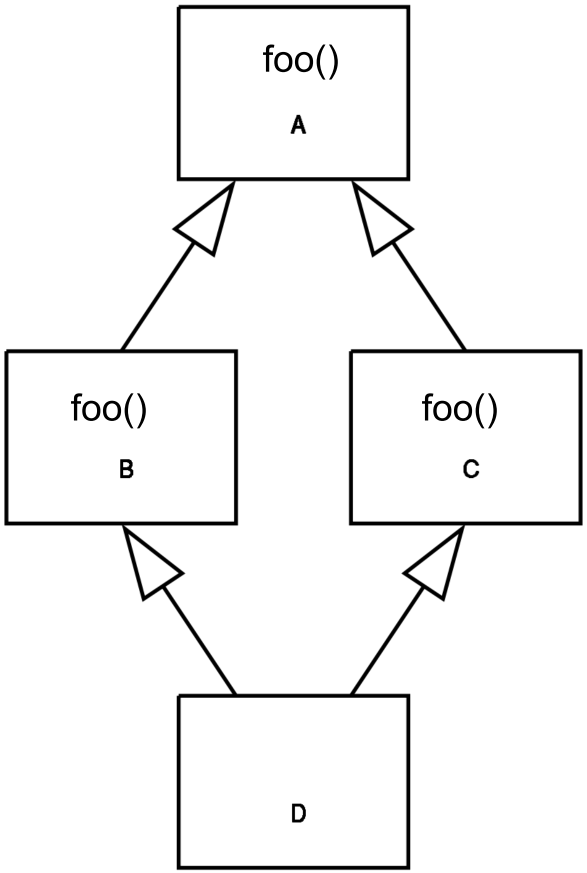
- *Bloated method inheritance* – that is, each child in an inheritance hierarchy will inherit ALL of the methods of its ancestors.
- This is true for both single and multiple inheritance.

Problem #2

- The *diamond problem* – sometimes referred to as the ‘deadly diamond of death’
- This occurs in languages with multiple inheritance

The Diamond Problem

- Briefly:
  - An ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
  - If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?
  - That is: D.foo() – which foo() should be called?
- This gets really problematic in deep inheritance structures.



The Diamond Problem

- Different languages deal with the diamond problem in different ways
  - C++ uses a fully qualified syntax
  - Python uses a class hierarchy linearization algorithm (C3 linearization or MRO) to resolve ambiguities

CSC301-git — python3 — 117×21

```
(base) MacBook:CSC301-git lutz$ python3
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(B,C):
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>> 
```

MRO: Method Resolution Order

Problem #3

- A third problem that frequently arises in inheritance-based OO languages are *rigid class structures*
  - This usually manifests itself in class hierarchies that are difficult to evolve in face of changing software requirements

Object-Based Programming

- A response to these problems is that recent languages no longer support inheritance and are *object-based*
- Of the three new big languages, Rust, Go, and Swift, only Swift supports a full OO model.
- Asteroid is object-based, that is, it supports objects but not inheritance.

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/bugs 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.

Exercise

- Let P={1,2,3,4,5} and Q={2,4}
- Let x:P and y:Q, determine if the following are legal:
  1. x = (1+1) with 1.P
  2. x = (4+2) with 4.P and 2.P
  3. x = (1+y) with 1.P

Assignments

- Reading: MPL Chap 6
- Assignment #2 – See BrightSpace