A Tour of Type

1

Prelude

Starting from C

```c
#include <stdio.h>

int main() {
    // an integer
    int i = 42;
    // a double precision floating point
    double d = 0.1;

    int wth = i + d;
    // ???
    printf("%d\n", wth);
}
```

So … WHAT IS TYPE, for real ?

A type is a **classification of data** which tells the compiler or interpreter how the programmer intends to use the data. By Wikipedia

~~Not exactly.~~

Type as interpretation of bytes

```
#include <stdio.h>
#include <stdint.h>

int main() {
    char a = 'a';
    // ?
    printf("%d\n", a);

    int64_t i = 0x000A214F4C4C4548;
    // ???
    printf("%s", (char*)&i);
}
```

2

Basic Terminologies

Type Checking: dynamic vs. static

python

```python
def main():
    print("Program start\n")

    im_not_array = 1
    print(im_not_array[0])
```

C

```c
#include <stdio.h>

int main() {
    printf("program start\n");

    int im_not_array = 0;
    printf("%d\n", im_not_array[0]);
}
```

Type Checking: weak vs. strong

C

```
void Test() {
    double d = 123.321;
    int i = d; // ok
}
```

java

```
public class NotSoWeak {
    static public void Test() {
        double d = 123.321;
        int i2 = d; // error
        int i = (int)d; // ok
    }
}
```

haskell

```
d :: Double
d = 123.321

i :: Int
i = floor d -- or i = ceiling d
```

Type Safety

Type safety is the extent to which a programming language discourages or prevents type errors.

```
void Unsafe() {
    double d = 2.5;
    // ???
    printf("%d\n", d);

    int array[5] = {1, 0, 0, 8, 6};
    int n = 10086;
    // out-of-range indexing is not type error
    printf("%d\n", array[n]);
}
```

3

Basic Types

Primitives (maybe..?)

- **Boolean**: true, false
- **Character**: ‘a’, ‘b’, ‘我’
- **Integer**: -1, 2, 3, …
- **Floating Point**: -1.5, 0.111
- **String**: “hello world”, “大吉大利”
- …

Array

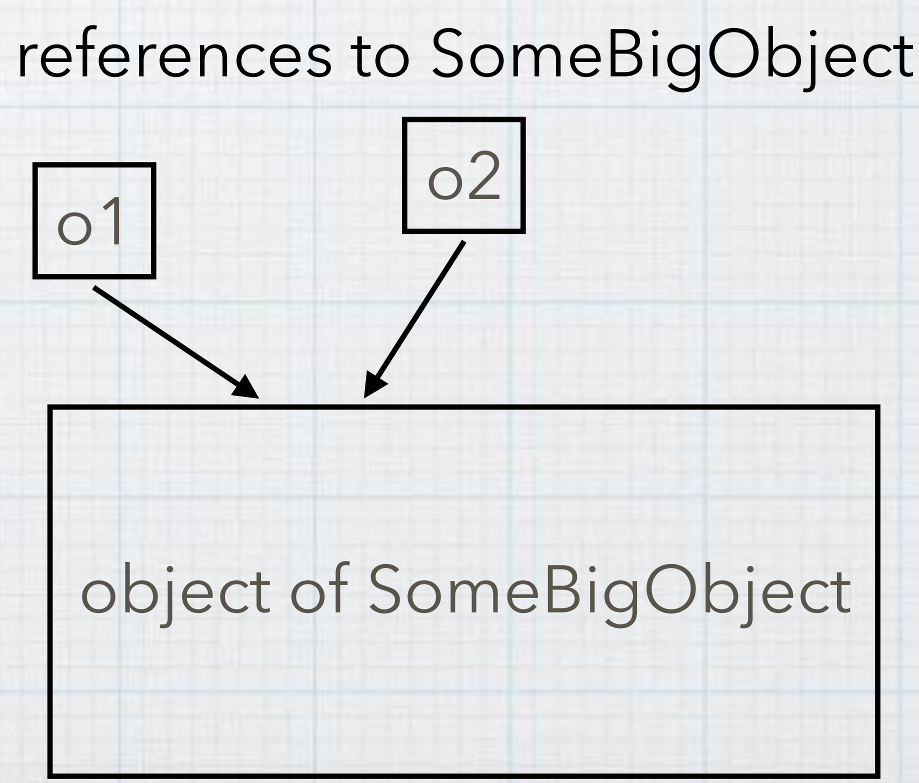
- represent the sequence of some type
- usually stored in continuous memory
- fast integer indexing
- static or dynamic growing

```
func array() {
    dynamic_array := make([]int, 0)
    for i := 0; i < 100; i++ {
        // the array can grow at runtime
        append(dynamic_array, i)
    }
    for i := 0; i < len(dynamic_array); i++ {
        fmt.Println(dynamic_array[i])
    }
}
```

Pointer & Reference

- point to the actual object of some type
- reduce cost of parameter passing
- allow different variable to share same data
- implicit or explicit dereferencing

```
static void Reference() {
    SomeBigObject o1 = new SomeBigObject();
    SomeBigObject o2 = o1;
    Foo(o1);
}
static void Foo(SomeBigObject o) {
    // do something
}
```



Tuple & Aggregation

- combine multiple types to another type
- promote type abstraction

```
struct Point {
    double x;
    double y;
};

double Mod(const Point *p) {
    return sqrt(pow(p->x, 2) + pow(p->y, 2));
}

Point Normalize(const Point *p) {
    double m = Mod(p);
    Point result = {p->x / m, p->y / m};
    return result;
}
```

Function Type

- the mapping between different types
- specify actions, behaviours or predicates

```
def Sort(a: Array[Int], compare: (Int, Int) => Boolean) = {
  // Sort the array with special compare function
}

def LessThan(i: Int, j: Int) = i < j
def GreaterThan(i: Int, j: Int) = i > j

def SortAscent(a: Array[Int]) = Sort(a, LessThan)
def SortDescent(a: Array[Int]) = Sort(a, GreaterThan)
```

Generic Type

- allow single interface being used with different type parameters
- promote type abstraction
- parametric polymorphism

```
def Sort[T](a: Array[T], compare: (T, T) => Boolean) = {
  // ..
}
def IntAscent(i: Int, j: Int) = i > j
def StringAscentByLength(i: String, j: String) = i.length > j.length

def Test(args: Array[String]): Unit = {
  val a1 = Array(2, 3, 1, 4, 5)
  val a2 = Array("hello", "generic", "sort")

  Sort(a1, IntAscent) // 5 4 3 2 1
  Sort(a2, StringAscentByLength) // generic hello sort
}
```

4

Object Oriented

Type as simulation of real world

```
class Person {
  public:
    // methods are behaviours
    void Kill(Person p) {
        if (p.weight < weight and p.height < height) {
            p.alive = false;
        }
    }
    void Suicide() {
        alive = false;
    }
    // .. other "methods"
  private:
    // fields are properties
    double weight;
    double height;
    bool alive;
    string name;
    // ... other "fields"
};
```

Well-known features of OO

- Encapsulation
- Inheritance
- Polymorphism

Abstract Data Type

Abstract data type is a class of objects whose logical behaviour is defined by a set of values and a set of operations.

```
interface IntArray {
    int get(int index);
    int set(int index, int n);
    int length();
    int[] toArray();
    // ..
}

boolean Has(IntArray arr, int n) {
    for (int i = 0; i < arr.length(); ++i) {
        if (arr.get(i) == n) return true;
    }
    return false;
}
```

Subtyping

- a form of type polymorphism
- supertypes can be “substituted” by its subtypes where appropriate
- `S` is a subtype of `T` is written as `S <: T`
- Nominal & Structural subtyping

Nominal Subtyping

```
class Animal {
    public String sound() { return "Awww!"; }
}
class Dog extends Animal {
    public String sound() { return "bark!!"; }
}
class Cat extends Animal {
    public String sound() { return "Meow"; }
}
class Cow {
    public String sound() { return "Mooooooooooo"; }
}
boolean Louder(Animal a, Animal b) {
    return a.sound().length() > b.sound().length();
}
void Animals() {
    Dog dog = new Dog();
    Cat cat = new Cat();
    Cow cow = new Cow();
    Louder(dog, cat); // true
    // error: Cow is not a subtype of Animal
    Louder(cow, dog);
}
```

Structural Subtyping

```
class Point {
    x: number;
    y: number;
}
class Line {
    a: Point;
    b: Point;
}
function Length(l: Line) { return 10086; }

let p1 = {x: 1, y: 1};
let p2 = {x: 1, y: 1, z: 1};

let l1 = {a: p1, b: p1};
let l2 = {a: p1, b: p1, c: "hello"};
let l3 = {a: p1, b: p2};

Length(l1); // exact
Length(l2); // subtype by width
Length(l3); // subtype by depth
```

Bounded type parameter

- able to declare type relationship of type parameters
- more expressive generic types

```
def Concat[A](as1: List[A], as2: List[A]): List[A] = {
  // ..
}
def Concat2[A, B <: A](as1: List[A], as2: List[B]): List[A] = {
  // ..
}
def Join[S <: String](ss: List[S]): String = {
  // ..
}
```

Interesting issues of subtyping

Subtyping of function types

- S <: T, U is a type ==> U -> S ?? U -> T
- S <: T, U is a type ==> S -> U ?? T -> U
- S1 <: T1, S2 <: T2 ==> S1 -> S2 ?? T1 -> T2
- S1 <: T1, S2 <: T2 ==> S1 -> T2 ?? T1 -> S2

Higher Order Subtyping

- S <: T ==> Array<S> ?? Array<T>
- S <: T, U is a generic type ==> U<S> ?? U<T>

5

Algebraic Data Type

Type as set of values

How many values do these types have:

```
bool   char

(bool, char)

union {bool b; char c;}

()

void

bool -> char
```

Algebraic Data Type (ADT)

Notations (informal):

- `S and T are ADTs ==> S + T is ADT (sum/variant)`
- `S and T are ADTs ==> S * T is ADT (product/tuple)`
- `S and T are ADTs ==> S -> T is ADT (function)`

Informally, define `N(T)` be the number of values in some ADT `T`, then

- `N(S + T) = N(S) + N(T)`
- `N(S * T) = N(S) * N(T)`
- `N(S -> T) = N(T) ^ N(S)`

Simple Examples

Function Totality

```
indexOf :: (Array<T>, T) -> int
type Optional<T> = T * ()
indexOf :: (Array<T>, T) -> Optional<T>
```

Currying

```
type Uncurried = (A, B) -> C
type Curried = A -> (B -> C) // or simply A -> B -> C
```

List

```
type List<A> = () + A * List<A>
```

Pattern Matching

- Matching the structure of values of ADTs recursively
- Destructure values into values with “smaller” type

```
data List a = Null | Cons a (List a)

tail :: List a -> List a
tail Null = Null
tail (Cons a t) = t

tail2 :: List a -> List a
tail2 Null = Null
tail2 (Cons a Null) = Null
tail2 (Cons a (Cons a' t)) = t
```

6

Type & Proof

Type as proposition: the intuition

```
void PrintInteger(int i) {
    printf("I got an integer: %d!\n", i);
}
```

```
validateForm :: Form -> Maybe ValidForm

submitForm :: ValidForm -> Result

processForm :: Form -> Either Error Result
processForm form =
    case validateForm of
        Just validForm -> Right (submitForm validForm)
        Nothing        -> Left invalidFormError
```

Curry-Howard Isomorphism

- the direct correspondence between mathematical proofs and computer programs
- values as proofs, types as propositions

Propositional Logic

Proposition	Type
`A`	`A`
`A ∧ B`	`A * B`
`A ∨ B`	`A + B`
`A ⇒ B`	`A -> B`
`¬A`	`A -> void`

Currying: the propositional view:

```
A ∧ B ⇒ C ⇔ A ⇒ B ⇒ C
(A, B) -> C ⇔ A -> B -> C
```

Quantifiers & Dependent type

- a dependent type is a type whose definition depends on a value.
- e.g. `std::array<int, 5>`

Proposition	Type
`(∀x:A).P(x)`	`(x: A) -> P(x)`
`(∃x:A).P(x)`	`(x: A) * P(x)`

```
concat : Vect m a -> Vect n a -> Vect (m + n) a
```

$$\forall m,n,a.\ \mathrm{Vect}(m, a) \wedge \mathrm{Vect}(n, a) \Rightarrow \mathrm{Vect}(m + n, a)$$

Induction & Recursion

```
data Natural = Zero
             | Succ Natural


data List a = Null
            | Cons a (List a)


-- illustration purpose, need extra refinement
-- Nil : Vect 0 a
concat : Vect m a -> Vect n a -> Vect (m + n) a

-- base case: Vect 0 a -> Vect n a -> Vect (0 + n) a
concat Nil v = v

-- induction case:
-- given Vect k a -> Vect n a -> Vect (k + n) a
-- prove Vect (1 + k) a -> Vect n a -> Vect (1 + k + n) a
concat (a :: as) v = a :: concat as v
```

Predicate & Type: motivation

Expressing equality as result of “function” ?

```
zip : Vect n a -> Vect n b -> Vect n (a, b)

tryZip : Vect m a -> Vect n b -> Maybe (Vect m (a, b))

-- did m == n say anything about equality?
wrongTryZip : Vect m a -> Vect n b -> Maybe (Vect m (a, b))
wrongTryZip {m} {n} v1 v2 =
  if m == n
    then Just (zip v1 v2)
    else Nothing
```

Equality as a type

Equality of two things is a proposition!

```
data (=) : A -> B -> Type where
  Refl : x = x

zip : Vect n a -> Vect n b -> Vect n (a, b)

equal : (m: Nat) -> (n: Nat) -> Maybe (m = n)

tryZip : Vect m a -> Vect n b -> Maybe (Vect m (a, b))
tryZip {m} {n} v1 v2 =
  case equal m n of
      Nothing => Nothing
      -- the proof here tell compiler m and n are the same
      (Just Refl) => Just (zip v1 v2)
```

Summary & Questions