Types in the “Real World”

April 24, 2018

So far in 98-317...

- What types are
- What lambda calculus is
- What category theory is

How does this help me?

- Many programming languages have types
- But some do not look like they do...
- Surely types are too boring for the average programmer to care about?

Why types?

- It's `\the\year`. Why can my booleans still take on the value `-98317`?
- Why do I need a default `switch` case when I know there are no more cases?
- How can a bug like Heartbleed still happen? (`memcpy` bounds? Come on, man.)

Why types?

- Why did I train my CNN for 24 hours, only then to be greeted with a `TypeError`?
- How can I stop typing `if err != nil`?
- Why can my enums only carry constants? Why wouldn't I just use a constant...

Why types?

- How do I write less code...
- and have it be more likely to be correct, more of the time?
- and learn things about programs that I would otherwise have to figure out myself?

Why types?

Necessity

Because without types, the world is bug-ridden hell.

Convenience

Because without types, you waste so much of your time.

Innovation

Because without types, there are so many things you cannot do.

But first, all the obvious reasons...

- Better, earlier error reporting
- Fewer “wat” moments[1]
- Self-documentation
- Faster execution
- Better IDE support
- Encourages better software design
- Real patterns (monads, not `AbstractSingletonFactoryProxyBeans`)

[1] `https://www.destroyallsoftware.com/talks/wat`

Our goal

- “Just use a strongly, statically typed programming language.”
- That war is probably lost. (Was it ever fought?)
- Types are still everywhere!
- Novel applications of them are gaining prominence.

Our goal

- “Just use a strongly, statically typed programming language.”
- That war is probably lost. (Was it ever fought?)
- Types are still everywhere!
- Novel applications of them are gaining prominence.

Today we will sample ~15 ways that types can make life easier in the **real world**!

No inference rules.

Whether you use Python or Idris, types are central to software.

Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck.

Duck typing

```
def print_it(it):
    try:
        while True:
            print(it.next())
    except StopIteration:
        pass

class It(object):
    def next(self):
        return "Hype for Types"

print_it(It())
```

Duck typing

```
template <typename T>
void print_me(T x) {
    x.print();
}

struct Me {
    void print() {
        // print
    }
};

int main() {
    print_me(Me());
}
```

Duck typing

Pros:

- Easy to work with
- Compatible with static type checking
- Easy to extend, lightweight interfaces

Cons:

- No concrete interface
- Dispatch

Inference in practice

```
jshell> var x = "Hello world!";
x ==> "Hello world!"
jshell> var y = (x) -> x;
|  Error:
|  cannot infer type for local variable y
|  (lambda expression needs an explicit target-type)
|  var y = (x) -> x;
|  ^---------------^

scala> (x : Any) => x;
res1: Any => Any = $$Lambda$1102/1796415927
scala> null;
res2: Null = null
```

Hmm... Why doesn't everyone just do the right thing?

Gradual typing

```
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

mypy[2]

[2] http://mypy-lang.org

Gradual typing

```
function print<T: {x: number,
                   y: number}> (point: T): T {
    var x: number = point.x;
    var y: number = point.y;
    console.log('x: ' + Math.abs(x) +
                ', y: ' + Math.abs(y));
    return point;
}
```

Flow[3]

[3] https://flow.org

Gradual typing

```
#lang typed/racket
(struct pt ([x : Real] [y : Real]))

(: distance (-> pt pt Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

Typed Racket[4]

[4] https://docs.racket-lang.org/ts-guide/

Gradual typing

Pros:

- Encourages typing discipline
- Easier than converting to static typing
- Best of both worlds

Cons:

- Very hard to implement on top of language
- Interoperability? Compatibility?
- Worst of both worlds

Mutation and effects

```haskell
sumST :: Num a => [a] -> a
sumST xs = runST $ do
    n <- newSTRef 0
    forM_ xs $ \x -> do
        modifySTRef n (+x)
    readSTRef n
```

ST monad[5]

[5] https://wiki.haskell.org/Monad/ST

Mutation and effects

```
public static void main(String[] args)
  throws IOException, NoSuchMethodException,
      SocketException
  { ... }
```

Mutation and effects

Java checked exceptions[6]

6 https://blog.takipi.com/ignore-checked-exceptions-all-the-cool-devs-are-doing-it-based-on-600000-java-projects/

Mutation and effects

```
double maximum(const double d1, const double d2) {
    double dResult = d1;
    if (d2 > dResult) {
        dResult = d2;
    }
    d1 = 0.0;  // Illegal
    d2 = 0.0;  // Illegal
    return dResult;
}
```

Or, declare methods as `const` to indicate they don't change instance state.

Mutation and effects

Now we've seen gradual typing. What about checked exceptions and `const` correctness?

Mutation and effects

Now we've seen gradual typing. What about checked exceptions and `const` correctness?

- Poisoning
- Adding to codebase much more painful
- Hacks to work around
- Rest of type system ignorant (method resolution)
- Lack of familiarity with monads!
- Useful only if *everyone does it*

Protocols and interfaces

```
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }
  repeated PhoneNumber phones = 4;
}
```

Protocol Buffers[7]

[7] https://github.com/google/protobuf

Protocols and interfaces

- Cross-platform RPC and message interfaces
- Somewhat specialized to message protocols (repeated messages, etc).
- How to serialize and deserialize? GADTs?[8]

[8] `https://blog.janestreet.com/why-gadts-matter-for-performance/`

Generic programming

- Not the same as polymorphism!
- Polymorphism describes how to compute regardless of a type
- Generic programming describes how to compute *across many types*

- Typeclasses (`Eq`, `Read`, `Foldable`)
- Deriving instances (scrap your boilerplate)

Generic programming

```haskell
data Bit = O | I
class Serialize a where
  put :: a -> [Bit]

instance Serialize Bool where
  put True = [I]
  put False = [O]

instance Serialize a => Serialize [a] where
  put []    = []
  put (h:t) = put h ++ put t
```

Generic programming

```haskell
data Bit = O | I
class Serialize a where
  put :: a -> [Bit]

instance Serialize Bool where
  put True = [I]
  put False = [O]

instance Serialize a => Serialize [a] where
  put []     = []
  put (h:t) = put h ++ put t

data Tree a = Leaf | Node a (Tree a) (Tree a)
```

Generic programming

```haskell
data Tree a = Leaf | Node a (Tree a) (Tree a)

data U1 p = U1
data (:+:) f g p = L1 (f p) | R1 (g p)
data (:*:) f g p = f p :*: g p

type RepTree a =
       U1
  :+: a :*: Tree a :*: Tree a
```

Generic programming

```
class Generic a where
  type Rep a :: * -> *
  from  :: a -> (Rep a) x
  to    :: (Rep a) x -> a

instance Generic (Tree a) where
  type Rep (Tree a) = RepTree a
  from Leaf           = L1 U1
  from (Node a l r) = R1 (a :*: l :*: r)
  to (L1 U1)                = Leaf
  to (R1 (a :*: l :*: r)) = Node a l r
```

Generic programming

```
class GSerialize f where
  gput :: f a -> [Bit]

instance GSerialize U1 where
  gput U1 = []
instance (GSerialize a, GSerialize b) =>
  GSerialize (a :*: b) where
  gput (a :*: b) = gput a ++ gput b
instance (GSerialize a, GSerialize b) =>
  GSerialize (a :+: b) where
  gput (L1 x) = O : gput x
  gput (R1 x) = I : gput x
```

Generic programming

```haskell
{-# LANGUAGE DeriveGeneric #-}
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving Generic

-- Magically connect Serialize with GSerialize
-- See reference!
instance (Serialize a) => Serialize (Tree a)
```

GHC.Generics[9]

Not just in Haskell![10]

[9] https://wiki.haskell.org/GHC.Generics

[10] http://willcrichton.net/notes/type-directed-metaprogramming-in-rust/

Nominal typing

```
type p1 = int * string
type p2 = int * string

datatype s1 = L of int | R of string
datatype s2 = L of int | R of string
```

Structural vs. nominal typing

Nominal typing

```
type stack = < pop: int option; push: int -> unit >
class stack2 = object
  val mutable v : int list = []
  method pop = match v with x::xs -> v <- xs; Some x
                          | _ -> None
  method push x = v <- x::v
end
let s : stack = object
  val mutable v = []
  method pop = match v with x::xs -> v <- xs; Some x
                          | _ -> None
  method push x = v <- x::v
end
let s2 : stack2 = new stack2
let _ = (s : stack2)
let _ = (s2 : stack)
```

Nominal typing

```
class Stack<T> {
    LinkedList<T> v = new LinkedList<>();
    public T pop() { return v.poll(); }
    public void push(T x) { v.push(x); }
}
class Stack2<T> {
    LinkedList<T> v = new LinkedList<>();
    public T pop() { return v.poll(); }
    public void push(T x) { v.push(x); }
}
```

Nominal typing

```
class Stack<T> {
    LinkedList<T> v = new LinkedList<>();
    public T pop() { return v.poll(); }
    public void push(T x) { v.push(x); }
}
class Stack2<T> {
    LinkedList<T> v = new LinkedList<>();
    public T pop() { return v.poll(); }
    public void push(T x) { v.push(x); }
}
```

Try assigning instances of `Stack` to `Stack2`.

Nominal typing

- See `typealias` in Swift
- See `newtype` in Haskell
- When is structural typing useful?
- When is nominal typing useful?
- Are classes types?
- Should they be?

Union and intersection types

```
function padLeft(value: string,
                 padding: string | number) {
    // ...
}
```

Union and intersection types

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}
```

TypeScript[11]

[11] http://www.typescriptlang.org/docs/handbook/advanced-types.html

Union and intersection types

"Programming with Intersection Types and Bounded Polymorphism"

Benjamin C. Pierce's PhD thesis, 1991[12]

[12] `https://www.cs.cmu.edu/~rwh/theses/pierce.pdf`

Resource analysis

```
let rec append l1 l2 =
  match l1 with
    | [] -> l2
    | x::xs ->
      let _ = Raml.tick(1.2) in
      x::(append xs l2)

== append :
  ['a list; 'a list] -> 'a list
  Simplified bound:
     3.00 + 11.00*M
```

Resource Aware ML[13]

[13] http://www.raml.co/

Modularity

Buzzwords:

- What's a module?
- What's an abstraction?
- What's an *existential type*?
- What's a *higher kind*?

Modularity

Deep questions:

- How do I express a module having a type and operations on that type?
- How do I seal a module with some signature?
- Why do functors generate new types when I apply them over again?
- How do I allow users to build robust software?

Modularity

Deep questions:

- How do I express a module having a type and operations on that type?
- How do I seal a module with some signature?
- Why do functors generate new types when I apply them over again?
- How do I allow users to build robust software?
- What on earth is the meaning of this code?

```
- let datatype foo = Foo in Foo end;
val it = Foo : ?.foo
```

Formal systems

- How do I verify that my compiler does not have a hidden backdoor?
- How do I make sure my Pentiums divide floating point numbers correctly?
- How do I anticipate and automatically repair program bugs?
- How does the Coq theorem prover help find a mathematical proof?
- How do I solve generalized satisfiability problems?
- How do I ensure certain security properties about my code?

Formal systems

```
int{Alice→;Alice←*} b;
int{Alice→Bob;Alice←*} y = 0;
if (b) {
    // pc is at level {Alice→;Alice←*} at this point.
    declassify ({Alice→;Alice←*} to {y}) {
        // at this point, pc has been declassified
        // to the label of the local variable y
        // (that is,  {Alice→Bob;Alice←*} ) permitting
        // the assignment to y
        y = 1;
    }
}
```

Jif[14]

[14] https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html

Communication and concurrency

- How do I securely define a communication protocol?
- How do I reason about high-level stateful programs?
- How do I model concurrency in a systems programming language?

Communication and concurrency

```
choice natstream {
  int /\ choice natstream next;
  void stop;
};
typedef choice natstream nats;
nats $c from(int n) {
    switch ($c) {
    case next:
        send($c, n);
        $c = from(n+1);
    case stop:
        close($c);
    }
}
```

Session types[15]

[15] http://cs.cmu.edu/~janh/courses/411/16/lec/23-concur.pdf

Communication and concurrency

```
state File {
    public final String filename;
}
state OpenFile extends File {
    private CFilePtr filePtr;
    public int read() { ... }
    public void close() [OpenFile>>ClosedFile]
      { ... }
}
state ClosedFile extends File {
    public void open() [ClosedFile>>OpenFile]
      { ... }
}
```

Plaid[16]

[16] http://www.cs.cmu.edu/~aldrich/plaid/

Substructural type systems

- What if a variable must be used *at most once*?
- What if a variable must be used *at least once*?
- What if a variable must be used *exactly once*?
- What if a variable must be used exactly once, *but in some fixed order*?

Substructural type systems

- What if a variable must be used *at most once*?
- What if a variable must be used *at least once*?
- What if a variable must be used *exactly once*?
- What if a variable must be used exactly once, *but in some fixed order*?

- Linear, affine, and other type systems achieve these goals.
- **Clean** language[17]
- `unique_ptr` in C++, with move semantics!
- Rust ownership tracking[18]

[17] `https://clean.cs.ru.nl/Clean`

[18] `https://doc.rust-lang.org/1.12.1/book/ownership.html`

Type-directed compilation

- Why are generics sometimes erased, sometimes reified?
- How do we determine whether values need to be boxed?
- How do we tell whether two pointers are aliased?
- Can we optimize a program given additional type information?
- Can we even put types on *assembly language*? (Yes!)
- Can we make it compile fast? (Sadly, not yet...)

Registration!

This Fall:

- **15-312 Foundations of Programming Languages** with Bob Harper**
- **15-317 Constructive Logic** with Karl Crary
- **15-354 Computational Discrete Mathematics** with Klaus Sutner*
- **15-411 Compiler Design** with Jan Hoffmann and Jean Yang*
- **15-414 Bug Catching** with Matt Fredrickson

Next Spring:

- **15-316 Software Foundations of Security and Privacy**
- **15-417 HOT Compilation**
- **15-819 Advanced Topics** (?)

Next Week:

- **98-317 Hype for Types** with Final Exam!****

Tell your friends!

Thanks for a great semester!