Types for Quack

Why types?

Seriously. What's the point?

Why types?

My provisional answer:

The meaning of an operation like '+' depends on the types of its operands.

At the machine level, the (assumed) types of operands are encoded in the operations themselves. For example, integer plus and floating point plus are different instructions.

At the language level, we'd rather have one operation '+' do the right thing, or report an error if the operation is inappropriate.

Types as damage control

What should happen when I add a string to an array of floating point numbers?

“Throws an exception” is probably ok

“Doesn’t compile” is probably ok

”Subtracts a random amount of money from my bank account” or “swings robot arm in some random direction” are probably not ok.

Programs with types still fail, but perhaps less catastrophically than programs that don’t check types (dynamically or statically)

Choices in type system design

- What are the things that need types?
  - Variables or values?
    (Dynamic vs. static typing)
  - Functions? Collections? Other things?
    (What is "first class" in the language)
- Named or structural?
  - Can types be anonymous? Can types with different names be equivalent?
- Safe? What can happen if we break the rules?
  - If my program has a bug, is it ok if it sets the server room on fire?
- When can I use type T in place of type Q?
  - What is the subtyping rule?
- What do I have to describe explicitly?
  - Declaration vs. inference

Typed values or typed variables?

Consider Python:

```
def plus(a,b):
    return a + b

int_val = plus(7,9)
str_val = plus("Ostrich", "Giraffe")
```

Each value has a type (represented by a tag in memory)

a and b don't have fixed types. plus(8, "Ostrich") compiles, although 8+"Ostrich" will fail at run-time.

This is a "dynamic" type system.

Typed values or typed variables (2)

C has a static type system

```
int plus(int a, int b) {
   return a + b;
}
```

The values in a and b don't need tags. We know *at compile time* that a+b can be translated to integer addition, with no further checking during execution. (Fast!)

The compiler will complain if I write "Ostrich" + 7

Does this prevent errors? Maybe. It's complicated.

Typed variables AND typed values

Java and C++ and Quack have static *and* dynamic types

```
class Foo {
  Foo zog(Foo x) { … }
 }
class Bar extends Foo {
  Bar zog(Foo x) { … }
}
```

```
Foo x = new Bar(…);  Foo y = new Foo(…);
Foo z = x.zog(y);
```

Static type: Is this call of x.zog(y) ok?

Dynamic type: What method does it call?

What things need types?

Typically: What is "first class" in the language?

A "first class" value is something I can use like other values, passing to functions, returning from functions, storing in variables, etc.

- First class functions? Lambdas? Closures? Other things?
  - In Quack: Nope nope nope
  - Useful, but they make type checking and inference harder (a lot)

Named or structural?

If I have two objects that are identical except for the name of their class, are they compatible?

```
class C(x: Int) { this.x = x;  def val(): Int { return this.x } }
class D(x: Int) { this.x = x;  def val(): Int { return this.x } }
y = C(5);
z = D(5);
```

Do y and z have the same type? Can I use one where the other is expected? (Most useful if types can be anonymous, e.g., the type of a lambda)

Quack types are named. They are never anonymous, and every named type is distinct. (Way, way easier than structural.)

Is it safe? What if we break the rules?

Java is safe, with a few checks at run-time.

"Undefined behavior" is limited (e.g., no "out of thin air" results).
Example: Run-time checks on array accesses, null references.

Python is safe, with all the checks at run-time.

Typical of a dynamically typed language.

C and C++ are unsafe: You can cheat the type system

- Casts, unions, address arithmetic: Here is a chainsaw, have fun
- "Catch fire" semantics: Whatever happens, it's your own fault
- Not a mistake. Enables highly tuned code by experts.

Quack is statically safe. No run-time checks or type errors.

Example: There are no null pointers in Quack.
There are no array bounds errors, because there are no arrays.

When can I use type T in place of type Q?

Most languages have a "subtype of" relation, <:

T <: Q means "I can use a value of type T where a value of type Q is expected"

Quack has the usual rule for object-oriented languages like Java, C#, etc: Inheritance ("extends") implies subtyping

No primitives: Everything is part of the type hierarchy

No untyped values (like "null" in Java)

Note subtyping is not the same as coercion (e.g., changing int values to float values in C) or overloading.

What do I have to describe explicitly?

Do I have to declare a variable's type before I use it?

Are there constraints on order of declaration?

We can have strong, static typing without declarations by inferring types. Example: "auto" in C++, inferred types in many contemporary languages (Kotlin, Swift, etc).

Can be simple (as in Quack) or very complex (as in ML, Haskell, etc)

Quack: Declarations required at method boundaries (so that each method can be analyzed separately.) Inference for variables within a method (local and instance variables).

Static semantics to enforce

- Class hierarchy must be tree rooted at Obj
- No shadowing or redefinition of classes or methods within a class
- Overriding methods is permitted, but overriding definition must be compatible
  - Argument types must be supertypes (usually the same)
  - Result type must be subtype (often the same)
- Variables must be assigned a value before they are used
- Each method call must be compatible with (static) types of the arguments (checked during type inference)
- Method return types must conform to method signature

Am I forgetting anything?

Well-formed class hierarchy

Classes may be defined in any order (not like C++)

This is perfectly ok:

```
class Foo extends Bar {
     def answer_me(): Int { return this.answer; }
 }
class Bar {  this.answer = 42; }
```

Malformed class hierarchy

```
class Foo extends Bar { }
class Bar extends Foo { }            // Circular!
class OhNo extends NoSuch { }   // No such parent!
```

The first thing to check: The class hierarchy is a tree rooted at Obj.

Error? Abandon ship!

Easier to code the other checks if the class hierarchy is well-formed.

Initialize before use

Legal:

```
class Foo(color: Int) {
    if color < 5 {
        this.weight = 17;
    } else {
        this.weight = 14;
    }

    def is_heavy(): Boolean {
        excess = this.weight – 12;
        return excess > 4;
    }
```

Initialization errors

On any ***syntactic*** path

```
class BadInit() {

  def bad_init() {
    rep = 0;
    n_reps = 3;
    while rep < n_reps {
      if rep > 0 {
        x = y;
      }
      rep = rep + 1;
      y = 42;
    }
  }
}
```

Skips first iteration

In execution, y will always be initialized before it is copied to x. It's an error anyway!

Y is initialized here

Why do you think we require initialization before use on every *syntactic* path? (Java does too.)

Type checking and inference

(In more depth Thursday)

Basic issues:

Inference: What type is this expression?

Checking: Can I use this type here?

Especially in method calls:

```
x : T = o.foo(y);
```

type of o must have a method foo with one formal argument

type of y must be subtype of that formal argument

foo must return a subtype of T