Types

Dynamic types

Types are broken down into many categories

Static types

Duck typing

Dynamic types

Subtypes

Types are broken down into many categories

Classes and subclasses

Static types

“Strong” types

Dependent

Duck typing

Dynamic types

Subtypes

Types are broken down into many categories

Classes and subclasses

Static types

Gradual types

Linear

“Strong” types

A “type” is a classification that says how some data may be used

Essentially all programming languages have the concept of a “dynamic” type

Some languages also have “static” types

In those languages, the types have to be **checked** before running the program

A “dynamic” type is a piece of data’s type
at runtime

If I ask “what is x’s dynamic type” I am asking “what is x’s type **right now**”

In the next few slides, I am **only** going to be talking about **runtime** types

Python has dynamic types

```
>>> x = 12
>>> type(x)
<type 'int'>
>>> x = "Hello"
>>> type(x)
<type 'str'>
```

So does Ruby....

```
2.4.1 :005 > x = 12
 => 12
2.4.1 :006 > x.class
 => Integer
2.4.1 :007 > x = "Hello"
 => "Hello"
2.4.1 :008 > x.class
 => String
```

Everything in C++ also has a dynamic type **at runtime**

At compile time, C++ assigns **static types**

Here’s a **really key thing**

Dynamic types and static types are not necessarily the same!!!

Basically everything in Ruby revolves around classes

Classes are one kind of type

But classes are just one **kind** of type

We’ll learn more about classes in a few lectures…

Even assembly languages have types

They’re just really degenerate types

For example, everything in HERA is a word

This is still a type, but since there’s only one dynamic type in HERA it’s not that useful

Why do we have dynamic types?

To prevent us from doing something we shouldn’t at runtime

The dynamic types throw errors when the language doesn't know how to do something

```
2.4.1 :009 > 1 + "hello"
TypeError: String can't be coerced into Integer
	from (irb):9:in `+'
	from (irb):9
	from /Users/kmicinski/.rvm/rubies/ruby-2.4.1/bin/irb:11:in `<main>'
```

The dynamic types throw errors when the language doesn't know how to do something

```
2.4.1 :009 > 1 + "hello"
TypeError: String can't be coerced into Integer
    from (irb):9:in `+'
    from (irb):9
    from /Users/kmicinski/.rvm/rubies/ruby-2.4.1/bin/irb:11:in `<main>'
```

The dynamic types throw errors when the language doesn’t know how to do something

```
>>> 1 + "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
> (+ 1 "hello")
; +: contract violation
;   expected: number?
;   given: "hello"
;   argument position: 2nd
; [,bt for context]
```

So a **dynamic type system** is a set of rules that apply to program data **at runtime**

A **static type system** is a set of rules that assigns types to data **before running it**

Every language has dynamic types

Some languages **also** have static types

A lot of people act like it’s dynamic types on one end and static types on the other. But that is **false**

Dynamic Types

Static Types

Dynamic Types

Static Types

So when you go out into the world, just remember, a dynamic type is just a **type at runtime**

Now, what are static types?

Question: who here has had a dynamic type error in Racket?

Static types are all about helping you **prevent those errors**

Static types help you ensure **at compile time** that I won’t run into a type error **at runtime**

But type errors aren’t all the bugs in my program

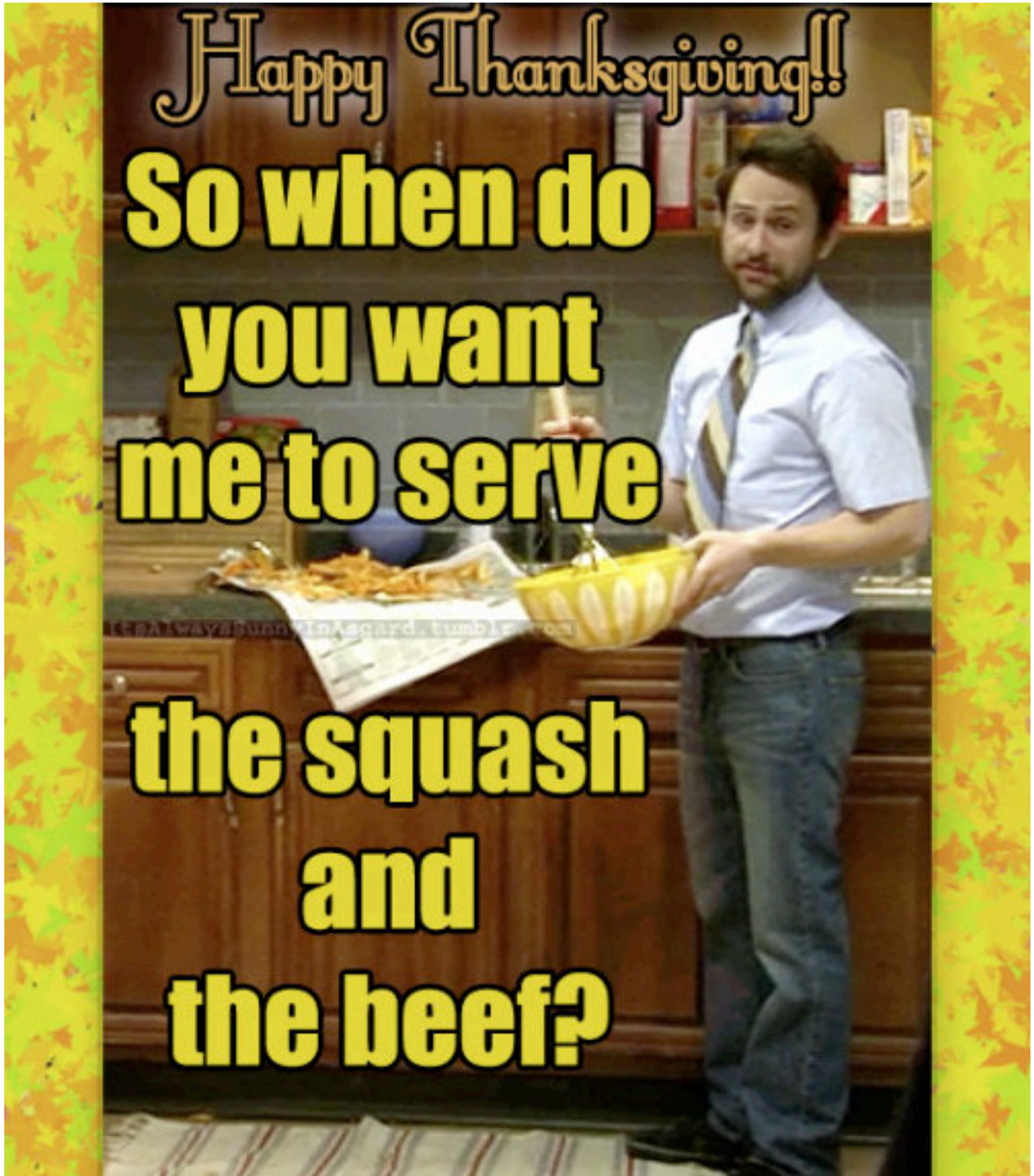
C++ has static types

```
string get_ith(list<string> l, i) {
  string s;
  for(; i > 0; i--) {
    l = rest(l);
  }
}
```

If I call `get_ith(ez_list("1","2"),2))` the program will fail at runtime

```
string get_ith(list<string> l, i) {
  string s;
  for(; i > 0; i--) {
    l = rest(l);
  }
}
```

It turns out that you can actually **beef up** the types


Happy Thanksgiving!!
So when do
you want
me to serve
the squash
and
the beef?

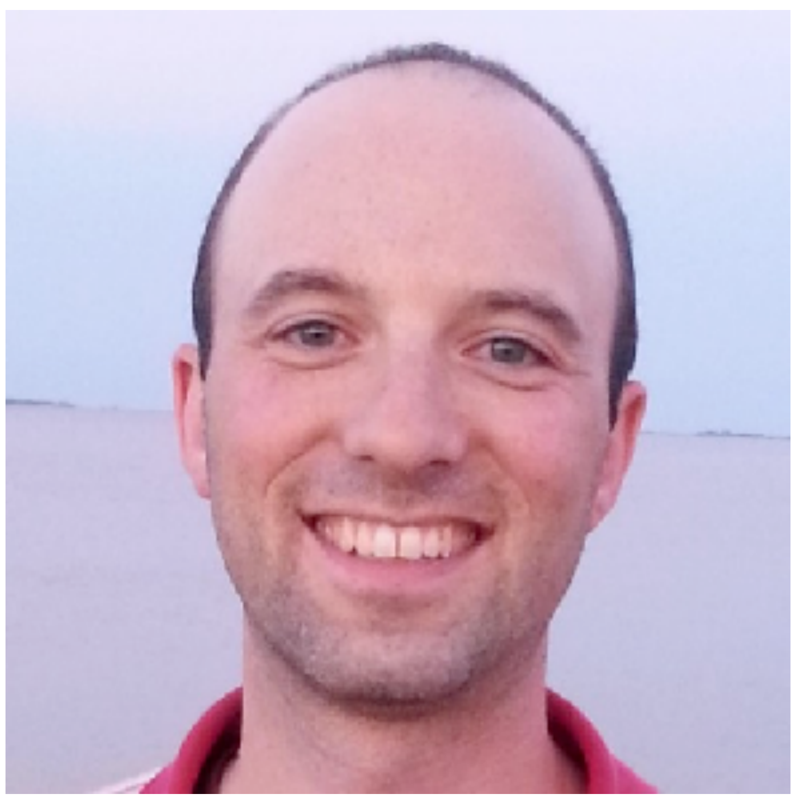
Certain languages allow you to specify constraints on the list size at compile time

```
list(string,n)
```

These are called dependent types because the type "depends" on the integer value n

These types are potentially very useful. Right now they're too hard to use. Few people use dependent types in production

Richard Eisenberg (BMC) and Stephanie Weirich (Penn) both work on efforts toward practical dependent types



In this class we will stick to more conventional types

Which are still very useful for most purposes

Two popular kinds of type systems

Nominal

Two popular kinds of static type systems

(Many real type systems mix the two)

Structural

Nominal Types

Types are assigned based on **name**

```
class C {
  int mX;
  int mY;
}
```

```
class D {
  int mX;
  int mY;
}
```

Nominal Types

In C++ these are different types, because they have different names

```
class C {
  int mX;
  int mY;
}
```

```
class D {
  int mX;
  int mY;
}
```

Structural type systems reason about the **structure** of the types

The Typed Racket Guide

by Sam Tobin-Hochstadt <samth@racket-lang.org>,
Vincent St-Amour <stamourv@racket-lang.org>,
Eric Dobson <endobson@racket-lang.org>,
and Asumu Takikawa <asumu@racket-lang.org>

Typed Racket is Racket's gradually-typed sister language which allows the incremental addition of statically-checked type annotations. This guide is intended for programmers familiar with Racket. For an introduction to Racket, see **The Racket Guide**.

For the precise details, also see The Typed Racket Reference.

We’re going to learn about static types by learning some typed Racket

(Typed Racket won’t be on exam, but concepts from type systems may be, I’ll tell you which)

Racket

```
(struct pt (x y))

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

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))))))
```

Structure type signature

```
(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))))))
```

```
> (pt "Hello" 1)
Type Checker: type mismatch
  expected: Real
  given: String in: "Hello"
>
```

The type checker prevents me from creating data that violates the type invariant

```
(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))))))
```

Function type signature

This is a type signature

```
(: distance (-> pt pt Real))
```

Read this as…

```
pt pt -> Real
```

Function types have the form

```
i1 i2 i3 … in -> output
```

Evocative of math

Sometimes called “arrow types”

```
-> Int Int Int
```

How would we write this in C++

```
(define-type Tree (U leaf node))
(struct leaf ([val : Number]))
(struct node ([left : Tree] [right : Tree]))
```

This is a **union type**

```
(define-type Tree (U leaf node))
(struct leaf ([val : Number]))
(struct node ([left : Tree] [right : Tree]))
```

A union type is a type that includes elements of two **different** types

“Every element of type **leaf** is an element of type Tree”

“Every element of type **node** is an element of type Tree”

```
(define-type Tree (U leaf node))
(struct leaf ([val : Number]))
(struct node ([left : Tree] [right : Tree]))
```

The union type allows you to combine two different types

Exercise: write a union type that allows strings or reals

```
(define-type Tree (U leaf node))
(struct leaf ([val : Number]))
(struct node ([left : Tree] [right : Tree]))
```

Call it string-or-real

I can also force Racket to check the types for me

```
(ann (+ 1 2) Number)
```

“ann” means “annotate”

Exercise: produce a type error with this

```
> (lambda (x) x)
- : (-> Any Any)
#<procedure>
> (lambda ([x : Number]) x)
- : (-> Number Number)
#<procedure>
```

Any means “can be any type”

```
> (lambda (x) x)
- : (-> Any Any)
#<procedure>
> (lambda ([x : Number]) x)
- : (-> Number Number)
#<procedure>
```

Typing rules and judgements

(This won't be on exam)

PL uses a fairly standard notation to write out what are called typing **judgements**

This is a standard mechanism for mathematically defining type systems

The way to read this is "If everything above the line is true, then Conclusion is true"

$$\frac{\text{Assumption 1} \qquad \text{Assumption 2} \qquad \text{Assumption 3}}{\text{Conclusion}}$$

If nothing above the line, it means I don’t have to make any assumptions

Conclusion

I.e., conclusion is vacuously true (don’t need to do any work to *prove* it)

1 : Number

2 : Number

Generally…

$$\frac{}{n : \text{Number}}$$

Typing judgements

A typing judgement

```
x : Number, y : Number |- (+ 1 x) : Number
```

Assumptions that certain variables have certain types

A typing judgement

```
x : Number, y : Number |- (+ 1 x) : Number
```

Assumptions that certain variables have certain types

Conclusion I have drawn about expression
(involving variables on left)

A typing judgement

```
x : Number, y : Number |- (+ 1 x) : Number
```

The thing to the left of the |- is typically called an “environment”

A typing judgement

```
x : Number, y : Number |- (+ 1 x) : Number
```

"If I assume x has type Number, and I assume y has type Number, I can show (+ 1 x) has type Number"

“Under the environment where x has type Number, I have concluded e has type t”

```
x : Number |- e : t
```

$$\frac{x : \mathrm{Number} \vdash e : t}{(\mathrm{lambda}\ (x)\ e) : \to \mathrm{Number}\ t}$$

“If assuming x has type number allows me to conclude e has type t”

$$\frac{\texttt{x : Number |- e : t}}{\texttt{(lambda (x) e) : -> Number t}}$$

"If assuming x has type number allows me to conclude e has type t"

```
x : Number |- e : t
-----------------------------
(lambda (x) e) : -> Number t
```

"Then I am allowed to conclude that…

```
(lambda (x) e)
```

Has type…

```
-> Number t
```

We won’t do this for Typed Racket, and doing so is nontrivial because it involves some elaborate types

Type Inference

Typed Racket will do this

```
> (lambda ([x : Number]) (+ 1 x))
- : (-> Number Number)
```

Why is this the return type?

Why not `(-> Number Any)`?

(Begin fair-game exam stuff again..)

Type Inference

The process by which a programming language ascertains types of expressions by starting with a set of annotations

Most type systems require at least **some** programmer annotation

Does Java have type inference?

Does the C++ we've learned have type inference?

C++17 actually **adds** (some) type inference

```cpp
template<class T, class U>
auto add(T t, U u) { return t + u; }
// the return type is the type of operator+(T, U)
```

```cpp
template<auto n> // C++17 auto parameter declaration
auto f() -> std::pair<decltype(n), decltype(n)>
      // auto can't deduce from brace-init-list
{
    return {n, n};
}
```

```cpp
    auto a = 1 + 2;                // type of a is int
    auto b = add(1, 1.2);          // type of b is double
    static_assert(std::is_same_v<decltype(a), int>);
    static_assert(std::is_same_v<decltype(b), double>);
```

More at…

`http://en.cppreference.com/w/cpp/language/auto`