Type Systems,
Chapter 7

Programming Languages

Objective

Today's primary objective:

- You know the purpose of type systems
- You know that type systems can be static/dynamic, weak/strong (not well defined), gradual, duck
- You know pros and cons of static and dynamic typing
- You know what is meant by orthogonal

Type System

- Introduction to type systems
- Forms of type systems
- Languages employing each type and coding examples
- Implementation
- Pros and cons
- Questions

Type System Definition

Type Systems – set of rules that assign a "type" to computer program constructs such as variables, expressions, functions or modules.

Purpose of Type Systems

- Provide an implicit context
- Enable checking to make sure that certain meaningless operations do not occur (It can't catch all, but enough to be useful)
- If declared explicitly, a kind of stylized documentation
- Enable some optimizations

Common Types

Discrete types

integer, float, Boolean, char, enumeration

Composite types

records, arrays (strings), sets, pointers, lists, files

Forms of Type Systems

- Weak versus Strong
- Static versus Dynamic

Typing

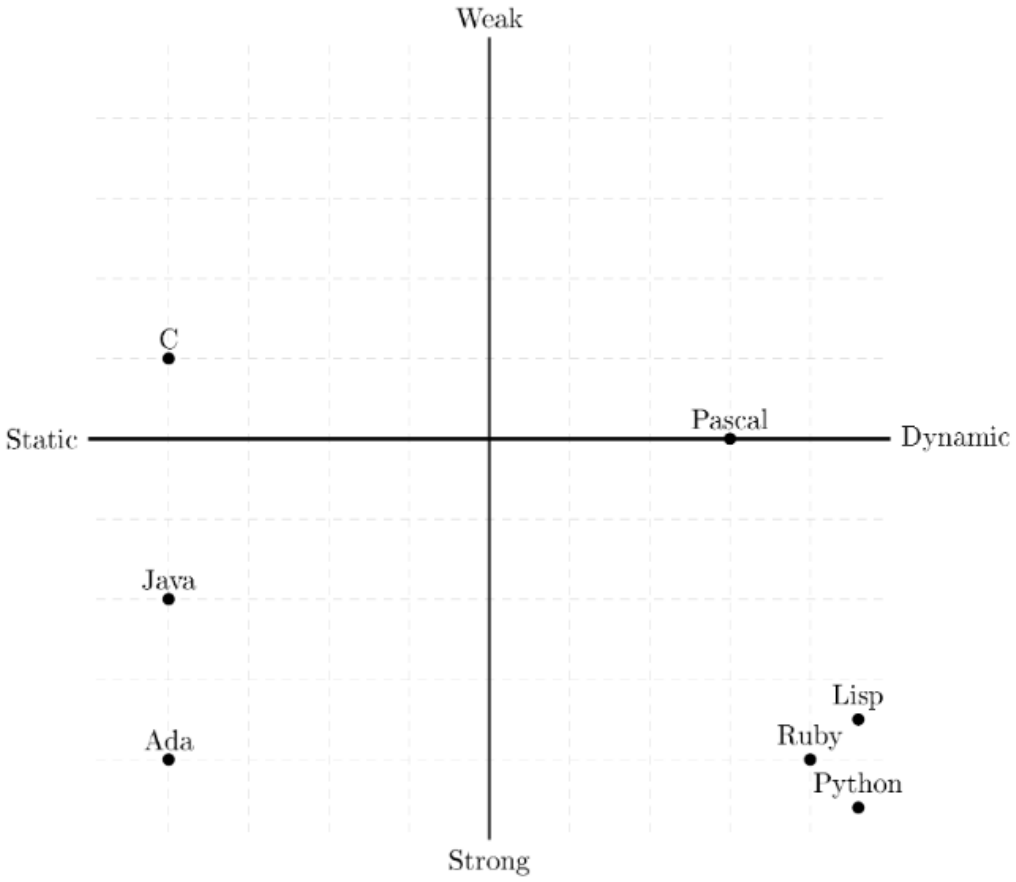

FIGURE 0.0.1. The Static/Dynamic - Strong/Weak Typing Plane

https://stackoverflow.com/questions/2351190/static-dynamic-vs-strong-weak

Forms of Type Systems

	Static	**Dynamic**
Weak	C	Perl JavaScript
Strong	C++ (mostly strong) Objective C (mostly strong) Java C# Rust Scheme	Python Ruby Lisp Borland's Prolog

Other terms

Gradual typing – variables may be typed at compile or run-time. Allow developers to choose static or dynamic within a single language

Duck typing (typically considered style of dynamic typing) – infer the type based on how it is used

Type Inference

Inferring the type (duck typing) is used in some strongly typed languages:

- Rust
- C++ (version 11 and up)
- C# (version 3.0 and up)
- Java (version 10 and up)
- F#
- Go
- functional languages

Strong versus Weak Type Systems

Strong – language defines each type of data. Variables and expressions must be described with one of these data types

Weak – many operations allowed without regard to types of the operands (typically dynamically typed)

Terms not well defined. Wikipedia says "No universally accepted definition for strong and weak typing"

Static versus Dynamic Type Systems

Static – variables are typed at compile time

Dynamic – variables are typed at run-time

Well defined, although a language may use aspects of both (gradual typing)

Type System in C

C is weakly, statically typed

All variables must be declared and typed, however, it doesn't enforce the type.

```
byte foo = 5;
int f = foo;          //this is valid, the variable
                      // "foo" will be implicitly
                      // converted to an int
```

Type System in C continued

C sort of provides dynamic typing via void *type

Example:

```
int i;
float f;
double d;
void *p = &i;
p = &f;
p = &d;
```

Python Implicit Conversion

```
a_int = 1
b_float = 1.0
c_sum = a_int + b_float
print(c_sum)
Print(type(c_sum))
```

```
2.0
<class 'float'>
```

Python Explicit Conversion

```
price_cake = 15
price_cookie = 6
total= price_cake + price_cookie
print (“The total is: “ + total + “$”)
```

Gives error.

```
print (“The total is: “ + str(total) + “$”)
```

JavaScript – dynamic typing

Change object type based on program conditions:

```
var person = {
        getName: function() {
                return 'Zack';
        }
};
if (new Date().getMinutes() > 29 {
        person = 5;     // "weakly" allows changing data type
}
alert('The name is '+ person.getName());
                        // strongly gives error message
```

Strong versus Static

Strong Typing – language prevents programmers from applying an operation to data on which it is not appropriate

Static Typing – compiler can do all the checking at compile time

Common Terms in Type Systems - Orthogonality

A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined

Orthogonality – useful goal in the design of language, particularly in its type system

Orthogonality

Orthogonality makes a language easy to understand, easy to use, and easy to reason about.

Orthogonality maximizes the expressive power of the language, yet increases the number of independent primitive concepts. It avoids deleterious superfluities.

Type Orthogonality Example

Pascal is more orthogonal than Fortran

- In Pascal can make arrays of anything

Orthogonal instruction set - all instruction types can use all addressing modes

Orthogonality – All 1st Class

A value is:

- 1st class if it can be passed as a parameter, returned from a subroutine or assigned into a variable.
- 2nd class can be passed as a parameter but not returned from a subroutine or assigned into a variable
- 3rd class cannot even be passed as a parameter

Strong versus Static Examples

Java – strongly typed with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically

Common Lisp - strongly typed, but not statically typed

Ada – both strongly and statically typed

Pascal - almost statically typed

Pros Static Typing

- Earlier detection of programming mistakes
- Better documentation in the form of type signatures
- More opportunities for compiler optimizations
- Increased runtime efficiency

Pros Dynamic Typing

- Ideally suited for prototyping systems with changing or unknown requirements
- Good when dealing with truly dynamic program behavior
- Code is more reusable
- Code is more concise
- Code is not any less safe (some claim)
- Code is not any less expressive (some claim)