1 Tipos (Clase teórica 9-04)

Building blocks of programming languages. They divide values by sets; can think of them as a set division of values in a language. Formally, it is a collection of values sharing structural properties. The set of values in a type is usually finite because they have a binary representation in the computar. For example, 32-bit integers range in $[-2147483648, 2147483647]$. This is important because it affects behavior; e.g. if a sum of two integer surpasses this range, the result will not be the expected one.

Primitive types are those included built-in in a language (generally booleans, ints, reals, characters, but not strings).

Types are useful for documenting the language and avoid errors. Furthermore, they are useful when dealing, for example, with memory allocation. Memory used to be highly expensive. Thus, it was important to know how much memory different types of values needed; this was an original distinction (values that take so and so much memory, etc).

Then came more semantic distinctions—this is, based on logics. The larger the set of characteristics that define a type, the less elements this type contains. Types allow for a more structured organization of languages, which on its turn makes it less likely to commit mistakes on runtime (assuming the language is strongly typed).

Static type systems are those where variable types are fixed in compile time; dynamic type systems are those where types are set in run-time.

A strongly typed system is one that imposes strong type restrictions so that the program behaves in a predictible manner. In general, strongly typed languages are more robust and easier to understand. The downside is that strongly typed programs are rigid and unflexible, and thus break more easily.

Weak type systems are harder to define and more of a grey zone. Languages that are rigid but can make some exceptions use *casting*. To *cast* a type is to convert it from one type into another. Castings imposed by the compiler (for example, the ones emerging from printing an array or a dictionary) are usually arbitrary. In strongly typed languages, castings have to be explicit.

Types can be atomic or composite. Composite types have multiple parts. There are also user-defined types (vs. bult-in types). User-defined types that are defined on rum-time (not in compile time) cannot be syntatically decomposed into a tree.

Most operations in a language are defined with relation to types $(+, -, /, *, \ldots)$.

Sobrecaraga, polimorfismo. Overhead (sobrecarga) is the variation in the meaning of a function or operator as a function of the type of its arguments or terms. For example, we can sum integers and floats; in this case, the result may be a float. For example, multiple dispatch is an example of overhead.

> El significado de un eprador o function cambia dependiendo de los tipos de sus operandos o argumentos o resultado.

A type or function is polymorphic if it can be applied to any associated type.

The difference between polymorphhism and overhead is that, in overhead, a single symbol refers to multiple algorithms (depending on the type of the parameters or operands)—each algorithm has different types, ant the algorithm is chosen based on the types used. In polymorphhism, a single algorithm may take multiple types—there is a unique implementation—the type variable is replaced by the type in question.

Type inference. Types can be *checked* (comprobados) or infered. In type checking, the body of the function is analyzed and the declared types are used to check that everything matches.

In type inference, the code of the function is looked at without declaring types and the types are infered based on the operations used.