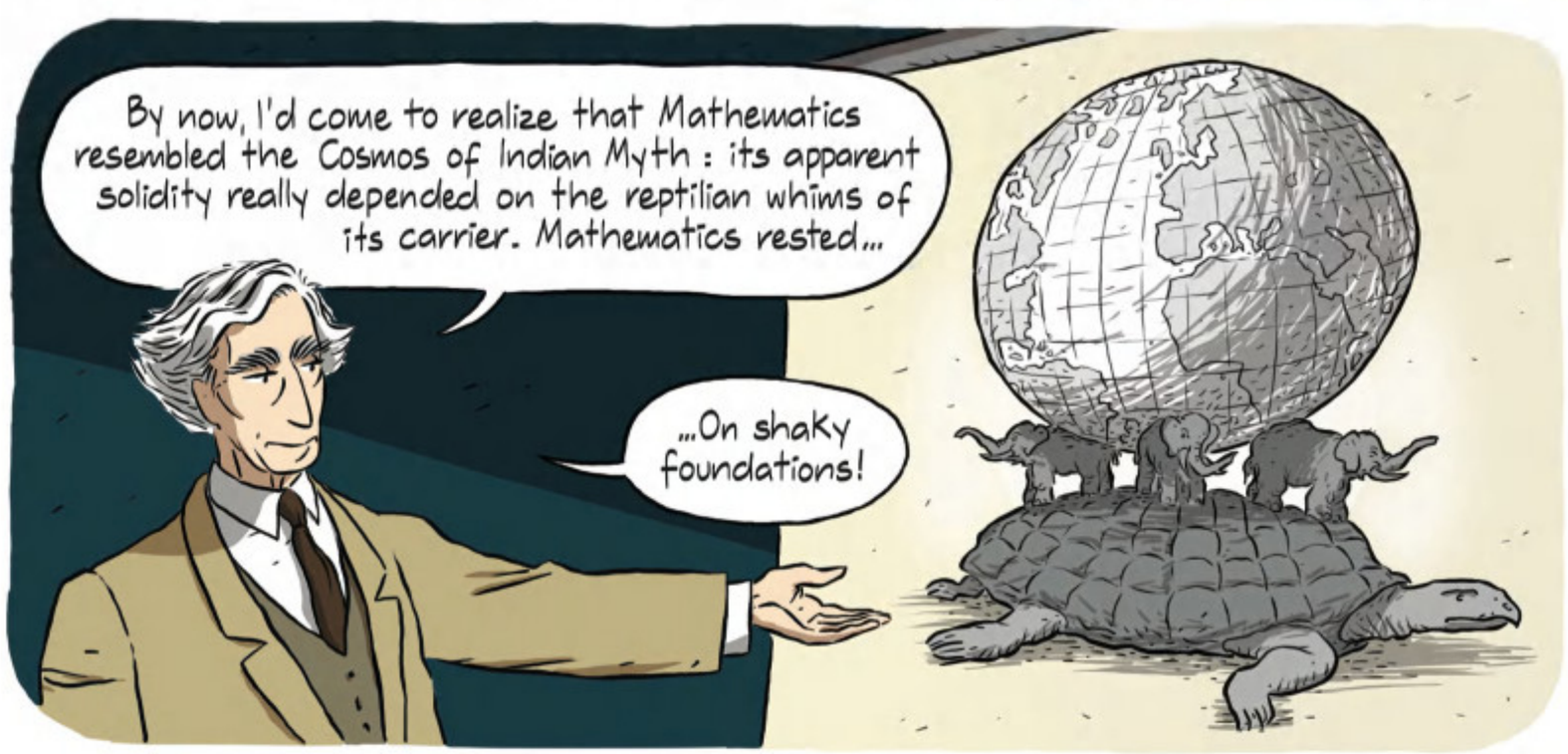
Introduction to types

A bit of history

The term **type systems** (or type theory) refers to a broad field of study in logic, mathematics, and philosophy. Researchers first formalised type systems in this sense in the early 1900s as ways of avoiding the logical paradoxes, such as Russell's paradox, that threatened the foundations of mathematics.



Doxiadis, Apostolos, and Christos Papadimitriou. ***Logicomix: An epic search for truth***. Bloomsbury Publishing USA, 2015.

A bit of history

The term **type systems** (or type theory) refers to a broad field of study in logic, mathematics, and philosophy. Researchers first formalised type systems in this sense in the early 1900s as ways of avoiding the **logical paradoxes**, such as Russell's paradox, that threatened the foundations of mathematics.

Let $R = \{x \mid x \notin x\}$

then $R \in R \iff R \notin R$

Doxiadis, Apostolos, and Christos Papadimitriou. ***Logicomix: An epic search for truth***. Bloomsbury Publishing USA, 2015.

A bit of history

The term **type systems** (or type theory) refers to a broad field of study in logic, mathematics, and philosophy. Researchers first formalised type systems in this sense in the early 1900s as ways of avoiding the logical paradoxes, such as Russell's paradox, that threatened the foundations of mathematics.

Let $R = \{x \mid x \notin x\}$
then $R \in R \iff R \notin R$

Russel's intuition on the introduction of types: set inclusion is conditional to types.

Let $Set \vdash x, R = \{x \mid x \notin x\} \implies Set(Set) \vdash R \wedge R \notin R$

Let $Set(Set) \vdash x, R = \{x \mid x \notin x\} \implies Set(Set(Set)) \vdash R \wedge R \notin R$

. . .

type of

A bit of history

During the twentieth century, types have become standard tools in logic, especially in proof theory, and have permeated the language of philosophy and science.

In computer science, there are two major research branches on type systems: the more **practical** one concerns applications to programming languages; the more **abstract** one focuses on connections with varieties of logic. Both branches use similar concepts, notations, and techniques, but with some important differences in orientation, e.g., abstract research usually concerns systems in which every well-typed computation is guaranteed to terminate, whereas most practical applications sacrifice this property for the sake of features like recursive definitions.

A bit of history

A plausible definition of type system in computer science is:

> *A tractable syntactic method for proving the absence of certain program behaviours by classifying clauses according to the types of values they compute.*

An important element in the above definition is its emphasis on classification of terms—syntactic clauses—according to the properties of the values that they will compute when executed.

A type system can be regarded as calculating a kind of static approximation to the runtime behaviours of the terms in a program.

Types of data

Looking at the more practical side of types, we can interpret them as

collections of homogenous and actually-present values

Hence, a type is a collection of values, e.g., integers. The adjective **homogeneous**, somewhat informal, suggests that such values must share some structural properties, which makes them all similar to each other.

The **actually-present** part of the interpretation above speaks to the practical side of calling and manipulating values that can be presented (written, named) in a finite way. For example, the real numbers are not actually presentable, because there are real numbers with infinite decimal expansion, which cannot be obtained by means of any algorithm. Hence, their approximations in programming languages (e.g., `real` and `float`) are their representable subset.

Types of data • Support to conceptual organisation

Types can help discipline the conceptual organisation of programs.

Indeed, types let programmers express the difference among the entities that form the solution to a given problem. For example, a hotel reservation program likely contains concepts such as *customers*, *dates*, *prices*, *rooms*, etc. The programmer can define a type for each of these concepts. Using types helps to logically separate conceptually-different elements, such as a room and a price. These might share similar implementations, e.g., they could both be integers, but at the design level, they are considered separate types.

The use of distinct types is both a **documentation** and **design** tool: knowing the type of a variable help us understanding what role that variable has in the program. In this sense, types play a role similar to that of comments. However, unlike comments, we can use types to reason on the programs they annotate, e.g., by signalling the wrongful assignment of a variable declared as “Room” to a value annotated as “Price” (see the “correctness” part, in the continuation).

Types of data • Support to abstraction

A particular declination of support for conceptual organisation offered by types is that of giving concrete backing to module systems in languages—where modules package and tie together different software units.

The typical example of this kind of usage of types are **interfaces**, which **associate a type** (e.g., `Integer`) **to operations** one can apply on it (e.g., +, -, %, conversions). Indeed, we can view an interface as a kind of “summary" of the facilities provided by the module or as part of a contract between implementors and users.

Having software units in terms of types (modules with interfaces) leads to a more abstract style of design. Since programmers can abstract from each unit’s implementation, they can focus on designing software top-down: from the contracts modules offer to each other to their independent implementation (this also supports code restructuring and reuse).

Types of data • Support for correctness

The most famous benefit of types is **type-checking**, i.e., the possibility to use types to detect programming errors. For example, if we can discover an error before running our program, we can fix it immediately, instead of discovering it while we use the program (or worse, when our users do). Moreover, the error reports from checking types are often more accurate and easier to act upon than analysing some runtime stack-trace of the error (if any).

Simple type systems can prevent us from assigning the wrong value to a variable, but more advanced ones can pinpoint, e.g., the mismatching between return types of if-then-else clauses, which manifest as inconsistencies at the level of types.

Types of data • Support for correctness

Of course, the strength of the effect of types on program correctness depends on the expressiveness of the type system and on the programming task in question. For example, if we encode all our data structures as lists we will not get as much help from the compiler as if we defined a different type for each of them.

Types also greatly help **refactoring** (the act of restructuring existing code). For example, without the support of a (static) type system, if we change the definition of a data structure we need to search and update all the code that involves said data structure. With a (static) type system, once we changed the declaration of the type of the structure, a passage of the type checker will point out all of those sites where the type is used inconsistently and need fixing.

Types of data • Support for correctness (safety)

The support for correctness mentioned before speaks to the more general concept of “language safety” — as in “well-typed programs do not go wrong” [1] or “safe languages make it impossible to shoot yourself in the foot while programming” [2].

Broadly, safety refers to a language’s ability to **guarantee the integrity of its abstractions** (and of higher-level abstractions introduced by the programmer using the definitional facilities of the language). Safe languages are also called **strongly-typed** ones (and unsafe ones weakly-typed).

For example, a language may provide arrays, with access and update operations, as an abstraction of the underlying memory. Using this language, we might expect to change an array only by using the update operation on it explicitly—and not, for example, by writing past the end of some other data structure. Similarly, we can expect to access lexically-scoped variables only from within their scopes.

[1] Milner, Robin. "**A theory of type polymorphism in programming.**" *Journal of computer and system sciences* 17.3 (1978): 348-375.
[2] Pierce, Benjamin C. **Types and programming languages**. MIT press, 2002.

Types of data • Support for correctness (safety)

Notably, we can achieve language safety through type safety, but types are not the only arrow in our quiver. For example, we can have runtime checks that trap nonsensical operations at the moment the program attempts them and stop it or raise an exception.

Conversely, unsafe languages provide “best effort” safety guarantees that help programmers eliminate the most obvious slips but do not guarantee the preservation of their abstractions.

In this perspective, we can consider “safe” a language like Java—where its compiler (using its type system) can detect a plethora of problems but the language deals with other classes of issues via runtime checks, e.g., by raising exceptions on out-of-bound array access and null-pointer references—but not a language like C since, e.g., it can let programs access arrays beyond their end.

Types of data • Support for implementation

Types can help improve the **efficiency** of programs. Indeed, designers introduced the first type systems in the 1950s in languages such as Fortran to improve the efficiency of numerical calculations by distinguishing between integer-valued arithmetic expressions and real-valued ones; this allowed the compiler to use different representations and generate optimised machine instructions.

In safe languages, types help improve efficiency by eliminating many of the dynamic checks to guarantee safety. Modern high-performance compilers heavily rely on information gathered by the type-checker to optimise code-generation.

Types of data • Other applications

Types have been used in computer and network security, e.g., typing lies at the core of the security model of Java and of the JINI “plug and play” architecture for network devices. Similarly, researchers have used type-checking algorithms in program analysis tools other than compilers, e.g., alias and exception analysis.

Automated theorem provers use type systems—usually powerful ones, based on dependent types—to represent logical propositions and proofs.

Also databases and data-management systems use types, e.g., in the form of Document Type Definitions and other kinds of schemas (XML, JSON schemas) and query languages that integrate these schema languages to type-check the correctness of queries.

A bit of history, timeline

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^{ω}	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)

1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

Dynamic vs Static Typing

A language is “**statically** typed” if we can check types at compile time, on the program text. Unless the runtime needs type-level information (we will see some examples where we might want to preserve some of this information), the compiler can remove it (and its related checks) from the generated code.

We talk about “**dynamically** typed” languages when type checking takes place while the program is running. Specifically, dynamic type checking requires that each value has a runtime descriptor that specifies its type and, at each operation, the runtime checks that the program preforms operations only on operands of the correct type.

Manifest vs Inferred typing

As seen, static vs dynamic typing relates to **when** a language (interpreter or compiler) performs type-checking. Manifest vs inferred typing determines **how much** information about types the programmer must put in their programs.

A manifest-typed language needs the programmer to type-annotate all variables and operations. An inferred-typed language needs no annotations, since it equips algorithms that deduce types from the context (declarations, operations).

While static vs dynamic typing leave little (useful) room to hybridisation, manifest vs inferred typing is a **spectrum** determined by both ergonomics and algorithmic factors. For example, reading the program `x = 5` we (as programmers) can deduce that the type of x might be `integer`. However, with more complex programs, we could struggle to keep in mind all details and would rather benefit from the additional documental information provided by types. At the other end, asking to annotate everything can sensibly slow down programming. For these reasons, **balancing manifest vs inferred typing is not a clean-cut decision** and it concerns what support/effort a language offer to/requires from the programmers.

Dynamic vs Static Typing

When talking about the trade-offs of using dynamically- vs statically-typed languages, frequently programmers make some confusion.

One example is the assumption that statically-typed languages are manifest-typed and dynamically-typed are inferred. This is somewhat true for many languages, but at different degrees. For example, Java is a language famous for its verbose manifest, static typing, but new versions refined the inference capabilities of its type checker to let users omit many type annotations.

Another assumption is that dynamically-typed languages are interpreted. This comes from some folklore belief that languages determine how one executes their programs. We can have compilers for dynamically-typed languages that equip the operations performed by the source program with the necessary control code to check types at runtime. The other direction is true too, e.g., Futamura projections [1] use partial evaluation to obtain compilers from interpreters.

[1] Futamura, Yoshihiko. "**Partial evaluation of computation process—an approach to a compiler-compiler**." *Higher-Order and Symbolic Computation* 12.4 (1999): 381-391.

Dynamic vs Static Typing

Hence, asking to choose between a dynamically- vs statically-typed language does not concern how much information programmers (must) put regarding types (manifest vs inferred) or whether the programs is interpreted or compiled (implementation). Instead, the main factors that can orient the choice between either approach are correctness, performance, and program expressiveness.

As mentioned, static typing means finding errors before we run our program. This eliminates the need for annotating terms and performing checks at runtime and unlocks some optimisations, increasing performance. However, statically-typed languages have a common “defect”: depending on the expressiveness of types and the accuracy of the type-checking algorithm, they can reject programs that would execute correctly (e.g., the code on the right). This is due to the undecidability of programs (termination), which makes it impossible to know what branches will or will not execute and forces type-checking to consider all possible states of computation.

```
int x
if(e) x=“A”
else x = 5
```