

What is a *formal method*?

What are the two kinds?

A mathematical technique for specifying, developing and verifying software.

- Maximally powerful methods like Hoare logic and modal logic.
- *Lightweight* formal method like type systems and model checkers.

Define *type system*.

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

Name a major difference between the type systems studied in the context of programming languages and those studied in the context of pure typed lambda-calculi.

The type systems in pure typed lambda calculi require computations to halt. Hence they are not Turing complete. Type systems in the context of programming languages lack this requirement in order to allow general recursion.

What is the relationship between *terms*, *types*,
and *values*.

Expressions are syntactic phrases. Terms are expressions representing a computation. Types apply to term expressions and describe the possible values that the term will compute when run. Values are the outputs of evaluated term expressions. They are themselves terms, ones that cannot be further reduced.

What is problematic with the term *dynamically typed language*.

Those "types" are tags on the heap for identifying different kinds of values. They are not static *approximations* on terms of run-time values.

In what sense are type systems *conservative*?

They can only prove the *absence* of some behaviors, never their presence. They therefore must reject some programs that will also lack these behaviors at run-time.

What is the main research goal in the study of type systems?

To allow more programs to be typed by improving the accuracy of static type approximations.

The particular bad behaviors a type system is trying to prevent are called ...

A type system that successfully prevents all such behaviors is called ...

... *run-time type errors*.

... *safe or sound*.

What are the benefits of type systems?

- Early detection of some programming errors.
- Maintenance/refactoring.
- Abstraction.
- Documentation.
- Language safety.
- Efficiency.

What makes a language *safe*?

Don't confuse with ...

Pierce: A safe language guarantees the integrity of its abstractions.

Cardelli: A safe language traps its errors, meaning they halt computation immediately or raise an exception that can be handled. Unsafe languages have untrapped errors that allow computation to proceed.

Also: A safe language lacks undefined behavior. Or in other words, it's portable between implementations.

Don't confuse language safety/soundness with the safety/soundness of a type system.

Why are there virtually no unsafe dynamically checked languages?

There is little marginal cost to checking the safety of all operations at run-time once most are.

Why can't safe languages rely solely upon the type system to guarantee safety?

Certain violations of abstractions (e.g., array indexes that are out of bounds) are beyond any currently widely used type system.

Contrast *run-time type error* with the more general notion of unsafe behaviors.

Run-time type errors are only those the type system is aiming to prevent. For example, a language can be type-safe despite allowing out-of-bounds array accesses because array bounds checking is not one of the properties the type system is aiming to guarantee (*i.e.*, not a run-time type error). In that case it is simply an unsafe behavior an unsafe language would allow and a safe language would prevent through dynamic checks.

What was the goal of the first type systems for programming languages?

To distinguish between integers and floating-point numbers,
for the sake of efficiency.