Type Systems

Luca Cardelli

Digital Equipment Corporation, Systems Research Center

The fundamental purpose of a type system is to prevent the occurrence of type errors during the execution of a program. This informal statement motivates the study of type systems, but requires clarification. Its accuracy depends, first of all, on the rather subtle issue of what constitutes a type error. Even when that is settled, the *type soundness* of a programming language (the absence of type errors in all program runs) is a non-trivial property. A fair amount of careful analysis is required to avoid false and embarrassing claims of type soundness; as a consequence, the classification, description, and study of type systems has emerged as a formal discipline.

The formal framework of type systems has practical applicability; it provides conceptual tools with which to judge the adequacy of important aspects of language definitions. Informal language descriptions often fail to specify a type system in sufficient detail to allow unambiguous implementation: different compilers for the same language may implement slightly different type systems. Moreover, many language definitions have been found to be type-unsound, allowing a program to *crash* even though it is judged acceptable by a typechecker. Formal techniques need not be applied in full to be useful; a knowledge of the main principles of type systems can help avoiding obvious and not-so-obvious pitfalls, and can inspire regularity and orthogonality in language design.

For the great majority of programming languages, type systems have not been formally specified and type soundness has not been formally established. Adequate technology, however, is now available. It involves defining the notion of type error for a given language, formalizing the type system by a set of type rules, and verifying that program execution of well-typed programs cannot produce type errors. This process, if successful, guarantees the type-soundness of a language as a whole. Type checking algorithms can then be developed as a separate concern, and their correctness can be verified with respect to a given type system; this process guarantees that typecheckers satisfy the language definition.

Program types

A type system describes the types that can be used to annotate programs, and the relationship between programs and types; common program types include Booleans, Integers, Reals, Records, Unions, Arrays, Objects, and Procedures. Annotations about the behavior of programs can, in general, range from informal comments to formal specifications subject to theorem proving. Type systems sit in the middle of this spectrum: they are more precise than comments, and more easily mechanizable than full specifications.

Type systems are designed to be *decidably verifiable*: there should be an algorithm (called a *typechecking algorithm*) that can tell whether or not a program is type-correct. (*cf.*: general specifications do not have correctness algorithms.) Type systems should be *transparent*: a programmer should be able to predict easily whether a program will typecheck. If it fails to typecheck, the reason for the failure should be self-evident. (*cf.*: automatic theorem proving may fail in mysterious ways.) Type systems should be *enforceable*: type declarations should be

statically checked as much as possible, and otherwise dynamically checked. The consistency between type declarations and their associated programs should be routinely verified. (*cf.*: program comments and conventions are not checked.)

These considerations dictate the kind of constructions that are appropriate for typing programs. For example, they explain why many ordinary set-theoretical constructions are inappropriate, since they are not easily manipulated by a typechecker. Type systems for programming languages take inspiration, instead, from *type theory*, which is a branch of constructive logic separate from set theory.

Formal type systems

Type systems are normally described via a particular formalism based on assertions called *judgments*. The most important judgment is the *typing judgment*, which asserts that a program fragment M has a type A with respect to a typing environment for the free variables of M. This judgment is written in the form:

$\Gamma \vdash M : A$	program M has type A in environment Γ

For example:

$\emptyset \vdash \mathit{true} : \mathit{Bool}$	*true* has type *Bool* in the empty environment
$\emptyset, x{:}\mathit{Int} \vdash x{+}1 : \mathit{Int}$	x+1 has type *Int*, provided that x has type *Int*

Any given judgment can be regarded as *valid* (e.g. $\Gamma \vdash \mathit{true} : \mathit{Bool}$) or *invalid* (e.g. $\Gamma \vdash \mathit{true} : \mathit{Int}$). The collection of valid judgments is described via *type rules*, which assert the validity of certain judgments on the basis of other valid judgments. In contrast to monolithic typechecking algorithms, type rules are highly modular: rules for different program constructs can be written separately. Therefore, type rules are comparatively easy to read, understand, analyze, and reuse.

A collection of type rules is called a *(formal) type system*. A program fragment M is *well-typed* if there is a type A such that $\Gamma \vdash M : A$ is a valid judgment for some Γ; that is, if the program fragment M can be given some type. If there is no possible type for a given program, the program is *not typeable* (it has a *type error*). Thus, type errors are formalized independently of any particular typechecking algorithm.

Type systems are not explicitly algorithmic: the discovery of a type for a term in a given type system is called the *type inference problem*. Its solution requires the construction of appropriate typechecking algorithms, which is in itself an important and sometimes complex endeavor.

Status and outlook

A large variety of formal type systems have been developed, with corresponding typechecking algorithms. They cover imperative, functional, and concurrent languages. They can handle simple and structured types, data abstraction, polymorphism, object-orientation, and modularization.

At the moment, some advanced constructions used in programming escape proper type-theoretical foundations. This could be either because the programming constructions are ill-conceived, or because our type theories are not yet sufficiently expressive: only the future will tell. Examples of active research areas are the typing of advanced object-orientation and modularization constructs, and the typing of concurrency and distribution.