Type Systems

Luca Cardelli

Microsoft Research

1 Introduction

The fundamental purpose of a **type system** is to prevent the occurrence of *execution errors* during the running 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 an execution error, which we will discuss in detail. Even when that is settled, the absence of execution errors is a nontrivial property. When such a property holds for all of the program runs that can be expressed within a programming language, we say that the language is **type sound**. It turns out that a fair amount of careful analysis is required to avoid false and embarrassing claims of type soundness for programming languages. As a consequence, the classification, description, and study of type systems has emerged as a formal discipline.

The formalization of type systems requires the development of precise notations and definitions, and the detailed proof of formal properties that give confidence in the appropriateness of the definitions. Sometimes the discipline becomes rather abstract. One should always remember, though, that the basic motivation is pragmatic: the abstractions have arisen out of necessity and can usually be related directly to concrete intuitions. Moreover, formal techniques need not be applied in full in order to be useful and influential. A knowledge of the main principles of type systems can help in avoiding obvious and not so obvious pitfalls, and can inspire regularity and orthogonality in language design.

When properly developed, type systems provide conceptual tools with which to judge the adequacy of important aspects of language definitions. Informal language descriptions often fail to specify the type structure of a language in sufficient detail to allow unambiguous implementation. It often happens that different compilers for the same language 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**. Ideally, formal type systems should be part of the definition of all typed programming languages. This way, typechecking algorithms could be measured unambiguously against precise specifications and, if at all possible and feasible, whole languages could be shown to be type sound.

In this introductory section we present an informal nomenclature for typing, execution errors, and related concepts. We discuss the expected properties and benefits of type systems, and we review how type systems can be formalized. The terminology used in the introduction is not completely standard; this is due to the inherent inconsistency of standard terminology arising from various sources. In general, we avoid the words *type* and *typing* when referring to run time concepts; for example we replace dynamic typing with dynamic checking and avoid common but ambiguous terms such as strong typing. The terminology is summarized in the Defining Terms section.

In section 2, we explain the notation commonly used for describing type systems. We review **judgments**, which are formal assertions about the typing of programs, **type rules**, which are implications between judgments, and **derivations**, which are deductions based on type rules. In section 3, we review a broad spectrum of simple types, the analog of which can be found in common languages, and we detail their type rules. In section 4, we present the type rules for a simple but complete imperative language. In section 5, we discuss the type rules for some advanced type constructions: *polymorphism* and *data abstraction*. In section 6, we explain how type systems can be extended with a notion of *subtyping*. Section 7 is a brief commentary on some important topics that we have glossed over. In section 8, we discuss the *type inference* problem, and we present type inference algorithms for the main type systems that we have considered. Finally, section 9 is a summary of achievements and future directions.

Execution errors

The most obvious symptom of an execution error is the occurrence of an unexpected software fault, such as an illegal instruction fault or an illegal memory reference fault.

There are, however, more subtle kinds of execution errors that result in data corruption without any immediate symptoms. Moreover, there are software faults, such as divide by zero and dereferencing *nil*, that are not normally prevented by type systems. Finally, there are languages lacking type systems where, nonetheless, software faults do not occur. Therefore we need to define our terminology carefully, beginning with what is a type.

Typed and untyped languages

A program variable can assume a range of values during the execution of a program. An upper bound of such a range is called a **type** of the variable. For example, a variable x of type *Boolean* is supposed to assume only boolean values during every run of a program. If x has type *Boolean*, then the boolean expression $not(x)$ has a sensible meaning in every run of the program. Languages where variables can be given (nontrivial) types are called **typed languages**.

Languages that do not restrict the range of variables are called **untyped languages**: they do not have types or, equivalently, have a single universal type that contains all values. In these languages, operations may be applied to inappropriate arguments: the result may be a fixed arbitrary value, a fault, an exception, or an unspecified effect. The pure λ-calculus is an extreme case of an untyped language where no fault ever occurs: the only operation is function application and, since all values are functions, that operation never fails.

A type system is that component of a typed language that keeps track of the types of variables and, in general, of the types of all expressions in a program. Type systems are used to determine whether programs are **well behaved** (as discussed subsequently). Only program sources that comply with a type system should be considered real programs of a typed language; the other sources should be discarded before they are run.

A language is typed by virtue of the existence of a type system for it, whether or not types actually appear in the syntax of programs. Typed languages are **explicitly typed** if types are part of the syntax, and **implicitly typed** otherwise. No mainstream language is purely implicitly typed, but languages such as ML and Haskell support writing large program fragments where

type information is omitted; the type systems of those languages automatically assign types to such program fragments.

Execution errors and safety

It is useful to distinguish between two kinds of execution errors: the ones that cause the computation to stop immediately, and the ones that go unnoticed (for a while) and later cause arbitrary behavior. The former are called **trapped errors**, whereas the latter are **untrapped errors**.

An example of an untrapped error is improperly accessing a legal address, for example, accessing data past the end of an array in absence of run time bounds checks. Another untrapped error that may go unnoticed for an arbitrary length of time is jumping to the wrong address: memory there may or may not represent an instruction stream. Examples of trapped errors are division by zero and accessing an illegal address: the computation stops immediately (on many computer architectures).

A program fragment is *safe* if it does not cause untrapped errors to occur. Languages where all program fragments are safe are called **safe languages**. Therefore, safe languages rule out the most insidious form of execution errors: the ones that may go unnoticed. Untyped languages may enforce **safety** by performing run time checks. Typed languages may enforce safety by statically rejecting all programs that are potentially unsafe. Typed languages may also use a mixture of run time and **static checks**.

Although safety is a crucial property of programs, it is rare for a typed language to be concerned exclusively with the elimination of untrapped errors. Typed languages usually aim to rule out also large classes of trapped errors, along with the untrapped ones. We discuss these issues next.

Execution errors and well-behaved programs

For any given language, we may designate a subset of the possible execution errors as **forbidden errors**. The forbidden errors should include all of the untrapped errors, plus a subset of the trapped errors. A program fragment is said to have **good behavior**, or equivalently to be well behaved, if it does not cause any forbidden error to occur. (The contrary is to have *bad behavior*, or equivalently to be *ill behaved*.) In particular, a well behaved fragment is safe. A language where all of the (legal) programs have good behavior is called **strongly checked**.

Thus, with respect to a given type system, the following holds for a strongly checked language:

- No untrapped errors occur (safety guarantee).
- None of the trapped errors designated as forbidden errors occur.
- Other trapped errors may occur; it is the programmer's responsibility to avoid them.

Typed languages can enforce good behavior (including safety) by performing static (i.e., compile time) checks to prevent unsafe and ill behaved programs from ever running. These languages are **statically checked**; the checking process is called **typechecking**, and the algorithm that performs this checking is called the typechecker. A program that passes the typechecker is said to be **well typed**; otherwise, it is **ill typed**, which may mean that it is actually ill-behaved,

or simply that it could not be guaranteed to be well behaved. Examples of statically checked languages are ML, Java, and Pascal (with the caveat that Pascal has some unsafe features).

Untyped languages can enforce good behavior (including safety) in a different way, by performing sufficiently detailed run time checks to rule out all forbidden errors. (For example, they may check all array bounds, and all division operations, generating recoverable exceptions when forbidden errors would happen.) The checking process in these languages is called **dynamic checking**; LISP is an example of such a language. These languages are strongly checked even though they have neither static checking, nor a type system.

Even statically checked languages usually need to perform tests at run time to achieve safety. For example, array bounds must in general be tested dynamically. The fact that a language is statically checked does not necessarily mean that execution can proceed entirely blindly.

Several languages take advantage of their static type structures to perform sophisticated dynamic tests. For example Simula67's INSPECT, Modula-3's TYPECASE, and Java's instanceof constructs discriminate on the run time type of an object. These languages are still (slightly improperly) considered statically checked, partially because the dynamic type tests are defined on the basis of the static type system. That is, the dynamic tests for type equality are compatible with the algorithm that the typechecker uses to determine type equality at compile time.

Lack of safety

By our definitions, a well behaved program is safe. Safety is a more primitive and perhaps more important property than good behavior. The primary goal of a type system is to ensure language safety by ruling out *all* untrapped errors in all program runs. However, most type systems are designed to ensure the more general good behavior property, and implicitly safety. Thus, the declared goal of a type system is usually to ensure good behavior of all programs, by distinguishing between well typed and ill typed programs.

In reality, certain statically checked languages do not ensure safety. That is, their set of forbidden errors does not include all untrapped errors. These languages can be euphemistically called **weakly checked** (or *weakly typed*, in the literature) meaning that some unsafe operations are detected statically and some are not detected. Languages in this class vary widely in the extent of their weakness. For example, Pascal is unsafe only when untagged variant types and function parameters are used, whereas C has many unsafe and widely used features, such as pointer arithmetic and casting. It is interesting to notice that the first five of the ten commandments for C programmers [30] are directed at compensating for the weak-checking aspects of C. Some of the problems caused by weak checking in C have been alleviated in C++, and even more have been addressed in Java, confirming a trend away from weak checking. Modula-3 supports unsafe features, but only in modules that are explicitly marked as unsafe, and prevents safe modules from importing unsafe interfaces.

Most untyped languages are, by necessity, completely safe (e.g., LISP). Otherwise, programming would be too frustrating in the absence of both compile time and run time checks to protect against corruption. Assembly languages belong to the unpleasant category of untyped unsafe languages.

Table 1. Safety

	Typed	Untyped
Safe	ML, Java	LISP
Unsafe	C	Assembler

Should languages be safe?

Some languages, like C, are deliberately unsafe because of performance considerations: the run time checks needed to achieve safety are sometimes considered too expensive. Safety has a cost even in languages that do extensive static analysis: tests such as array bounds checks cannot be, in general, completely eliminated at compile time.

Still, there have been many efforts to design safe subsets of C, and to produce development tools that try to execute C programs safely by introducing a variety of (relatively expensive) run-time checks. These efforts are due to two main reasons: the widespread use of C in applications that are not largely performance critical, and the security problems introduced by unsafe C programs. The security problems includes buffer overflows and underflows caused by pointer arithmetic or by lack of array bounds checks, that can lead to overwriting arbitrary areas of memory and that can be exploited for attacks.

Safety is cost effective according to different measures that just pure performance. Safety produces fail-stop behavior in case of execution errors, reducing debugging time. Safety guarantees the integrity of run time structures, and therefore enables garbage collection. In turn, garbage collection considerably reduces code size and code development time, at the price of some performance. Finally, safety has emerged as a necessary foundation for systems security, particularly for systems (such as operating system kernels and web browsers) that load and run foreign code. Systems security is becoming one of the most expensive aspects of program development and maintenance, and safety can reduce such costs.

Thus, the choice between a safe and unsafe language may be ultimately related to a trade-off between development and maintenance time, and execution time. Although safe languages have been around for many decades, it is only recently that they are becoming mainstream, uniquely because of security concerns.

Should languages be typed?

The issue of whether programming languages should have types is still subject to some debate. There is little doubt that production code written in untyped languages can be maintained only with great difficulty. From the point of view of maintainability, even weakly checked unsafe languages are superior to safe but untyped languages (e.g., C vs. LISP). Here are the arguments that have been put forward in favor of typed languages, from an engineering point of view:

- *Economy of execution*. Type information was first introduced in programming to improve code generation and run time efficiency for numerical computations, for example, in FORTRAN. In ML, accurate type information eliminates the need for *nil*-checking on pointer dereferencing. In general, accurate type information at compile time leads to the application of the appropriate operations at run time without the need of expensive tests.

- *Economy of small-scale development*. When a type system is well designed, typechecking can capture a large fraction of routine programming errors, eliminating lengthy debugging sessions. The errors that do occur are easier to debug, simply because large classes of other errors have been ruled out. Moreover, experienced programmers adopt a coding style that causes some logical errors to show up as typechecking errors: they use the typechecker as a development tool. (For example, by changing the name of a field when its invariants change even though its type remains the same, so as to get error reports on all its old uses.)
- *Economy of compilation*. Type information can be organized into *interfaces* for program modules, for example as in Modula-2 and Ada. Modules can then be compiled independently of each other, with each module depending only on the interfaces of the others. Compilation of large systems is made more efficient because, at least when interfaces are stable, changes to a module do not cause other modules to be recompiled.
- *Economy of large-scale development*. Interfaces and modules have methodological advantages for code development. Large teams of programmers can negotiate the interfaces to be implemented, and then proceed separately to implement the corresponding pieces of code. Dependencies between pieces of code are minimized, and code can be locally rearranged without fear of global effects. (These benefits can be achieved also by informal interface specifications, but in practice typechecking helps enormously in verifying adherence to the specifications.)
- *Economy of development and maintenance in security areas*. Although safety is necessary to eliminate security breaches such as buffer overflows, typing is necessary to eliminate other catastrophic security breaches. Here is a typical one: if there is any way at all, no matter how convoluted, to cast an integer into a value of pointer type (or object type), then the whole system is compromised. If that is possible, then an attacker can access any data anywhere in the system, even within the confines of an otherwise typed language, according to any type the attached chooses to view the data with. Another helpful (but not necessary) technique is to convert a given typed pointer into an integer, and then into a pointer of different type as above. The most cost effective way to eliminate these security problems, in terms of maintenance and probably also of overall execution efficiency, is to employ typed languages. Still, security is a problem at all levels of a system: typed languages are an excellent foundation, but not a complete solution.
- *Economy of language features*. Type constructions are naturally composed in orthogonal ways. For example, in Pascal an array of arrays models two-dimensional arrays; in ML, a procedure with a single argument that is a tuple of n parameters models a procedure of n arguments. Thus, type systems promote orthogonality of language features, question the utility of artificial restrictions, and thus tend to reduce the complexity of programming languages.

Expected properties of type systems

In the rest of this chapter we proceed under the assumption that languages should be both safe and typed, and therefore that type systems should be employed. In the study of type systems, we do not distinguish between trapped and untrapped errors, nor between safety and good behavior: we concentrate on good behavior, and we take safety as an implied property.

Types, as normally intended in programming languages, have pragmatic characteristics that distinguish them from other kinds of program annotations. In general, annotations about the behavior of programs can range from informal comments to formal specifications subject to theorem proving. Types sit in the middle of this spectrum: they are more precise than program comments, and more easily mechanizable than formal specifications. Here are the basic properties expected of any type system:

- Type systems should be *decidably verifiable*: there should be an algorithm (called a typechecking algorithm) that can ensure that a program is well behaved. The purpose of a type system is not simply to state programmer intentions, but to actively capture execution errors before they happen. (Arbitrary formal specifications do not have these properties.)
- 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. (Automatic theorem proving does not have these properties.)
- 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. (Program comments and conventions do not have these properties.)

How type systems are formalized

As we have discussed, type systems are used to define the notion of well typing, which is itself a static approximation of good behavior (including safety). Safety facilitates debugging because of fail-stop behavior, and enables garbage collection by protecting run time structures. Well typing further facilitates program development by trapping execution errors before run time.

But how can we guarantee that well typed programs are really well behaved? That is, how can we be sure that the type rules of a language do not accidentally allow ill behaved programs to slip through?

Formal type systems are the mathematical characterizations of the informal type systems that are described in programming language manuals. Once a type system is formalized, we can attempt to prove a *type soundness* theorem stating that *well-typed programs are well behaved*. If such a soundness theorem holds, we say that the type system is sound. (Good behavior of all programs of a typed language and soundness of its type system mean the same thing.)

In order to formalize a type system and prove a soundness theorem we must in essence formalize the whole language in question, as we now sketch.

The first step in formalizing a programming language is to describe its syntax. For most languages of interest this reduces to describing the syntax of types and *terms*. Types express static

knowledge about programs, whereas terms (statements, expressions, and other program fragments) express the algorithmic behavior.

The next step is to define the *scoping* rules of the language, which unambiguously associate occurrences of identifiers to their binding locations (the locations where the identifiers are declared). The scoping needed for typed languages is invariably *static*, in the sense that the binding locations of identifiers must be determined before run time. Binding locations can often be determined purely from the syntax of a language, without any further analysis; static scoping is then called *lexical scoping*. The lack of static scoping is called *dynamic scoping*.

Scoping can be formally specified by defining the set of *free variables* of a program fragment (which involves specifying how variables are bound by declarations). The associated notion of *substitution* of types or terms for free variables can then be defined.

When this much is settled, one can proceed to define the type rules of the language. These describe a relation *has-type* of the form $M{:}A$ between terms M and types A. Some languages also require a relation *subtype-of* of the form $A{<:}B$ between types, and often a relation *equal-type* of the form $A{=}B$ of type equivalence. The collection of type rules of a language forms its type system. A language that has a type system is called a typed language.

The type rules cannot be formalized without first introducing another fundamental ingredient that is not reflected in the syntax of the language: *static typing environments*. These are used to record the types of free variables during the processing of program fragments; they correspond closely to the symbol table of a compiler during the typechecking phase. The type rules are always formulated with respect to a static environment for the fragment being typechecked. For example, the has-type relation $M{:}A$ is associated with a static typing environment Γ that contains information about the free variables of M and A. The relation is written in full as $\Gamma \vdash M{:}A$, meaning that M has type A in environment Γ.

The final step in formalizing a language is to define its semantics as a relation *has-value* between terms and a collection of *results*. The form of this relation depends strongly on the style of semantics that is adopted. In any case, the semantics and the type system of a language are interconnected: the types of a term and of its result should be the same (or appropriately related); this is the essence of the soundness theorem.

The fundamental notions of type system are applicable to virtually all computing paradigms (functional, imperative, concurrent, etc.). Individual type rules can often be adopted unchanged for different paradigms. For example, the basic type rules for functions are the same whether the semantics is call-by-name or call-by-value or, orthogonally, functional or imperative.

In this chapter we discuss type systems independently of semantics. It should be understood, though, that ultimately a type system must be related to a semantics, and that soundness should hold for that semantics. Suffice it to say that the techniques of structural operational semantics deal uniformly with a large collection of programming paradigms, and fit very well with the treatment found in this chapter.

Type equivalence

As mentioned above, most nontrivial type systems require the definition of a relation *equal type* of type equivalence. This is an important issue when defining a programming language: when

are separately written type expressions equivalent? Consider, for example, two distinct type names that have been associated with similar types:

type X = Bool
type Y = Bool

If the type names *X* and *Y* match by virtue of being associated with similar types, we have *structural equivalence*. If they fail to match by virtue of being distinct type names (without looking at the associated types), we have *by-name equivalence*.

In practice, a mixture of structural and by-name equivalence is used in most languages. Pure structural equivalence can be easily and precisely defined by means of type rules, while by-name equivalence is harder to pin down, and often has an algorithmic flavor. Structural equivalence has unique advantages when typed data has to be stored or transmitted over a network; in contrast, by-name equivalence cannot deal easily with interacting programs that have been developed and compiled separately in time or space.

We assume structural equivalence in what follows (although this issue does not arise often). Satisfactory emulation of by-name equivalence can be obtained within structural equivalence, as demonstrated by the Modula-3 *branding* mechanism.

2 The language of type systems

A type system specifies the type rules of a programming language independently of particular typechecking algorithms. This is analogous to describing the syntax of a programming language by a formal grammar, independently of particular parsing algorithms.

It is both convenient and useful to decouple type systems from typechecking algorithms: type systems belong to language definitions, while algorithms belong to compilers. It is easier to explain the typing aspects of a language by a type system, rather than by the algorithm used by a given compiler. Moreover, different compilers may use different typechecking algorithms for the same type system.

As a minor problem, it is technically possible to define type systems that admit only unfeasible typechecking algorithms, or no algorithms at all. The usual intent, however, is to allow for efficient typechecking algorithms.

Judgments

Type systems are described by a particular formalism, which we now introduce. The description of a type system starts with the description of a collection of formal utterances called judgments. A typical judgment has the form:

$\Gamma \vdash \mathfrak{J}$ $\mathfrak{J}$ is an assertion; the free variables of $\mathfrak{J}$ are declared in Γ

We say that Γ *entails* $\mathfrak{J}$. Here Γ is a *static typing environment*; for example, an ordered list of distinct variables and their types, of the form $\emptyset$, $x_1{:}A_1$, ..., $x_n{:}A_n$. The empty environment is denoted by $\emptyset$, and the collection of variables $x_1 \ldots x_n$ declared in Γ is indicated by $dom(\Gamma)$, the domain of Γ. The form of the *assertion* $\mathfrak{J}$ varies from judgment to judgment, but all the free variables of $\mathfrak{J}$ must be declared in Γ.

The most important judgment, for our present purposes, is the *typing judgment*, which asserts that a term M has a type A with respect to a static typing environment for the free variables of M. It has the form:

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

Examples.

$\emptyset \vdash true : Bool$ *true* has type *Bool*

$\emptyset, x{:}Nat \vdash x{+}1 : Nat$ x+1 has type *Nat*, provided that x has type *Nat*

Other judgment forms are often necessary; a common one asserts simply that an environment is **well formed**:

$\Gamma \vdash \diamond$ Γ is well-formed (i.e., it has been properly constructed)

Any given judgment can be regarded as *valid* (e.g., $\Gamma \vdash true : Bool$) or *invalid* (e.g., $\Gamma \vdash true : Nat$). Validity formalizes the notion of well typed programs. The distinction between valid and invalid judgements could be expressed in a number of ways; however, a highly stylized way of presenting the set of valid judgments has emerged. This presentation style, based on type rules, facilitates stating and proving technical lemmas and theorems about type systems. Moreover, type rules are highly modular: rules for different constructs can be written separately (in contrast to a monolithic typechecking algorithm). Therefore, type rules are comparatively easy to read and understand.

Type rules

Type rules assert the validity of certain judgments on the basis of other judgments that are already known to be valid. The process gets off the ground by some intrinsically valid judgment (usually: $\emptyset \vdash \diamond$, stating that the empty environment is well formed).

$$\frac{\text{(Rule name)} \quad \text{(Annotations)} \qquad \Gamma_1 \vdash \Im_1 \quad \ldots \quad \Gamma_n \vdash \Im_n \quad \text{(Annotations)}}{\Gamma \vdash \Im}$$

General form of a type rule.

Each type rule is written as a number of *premise* judgments $\Gamma_i \vdash \Im_i$ above a horizontal line, with a single *conclusion* judgment $\Gamma \vdash \Im$ below the line. When all of the premises are satisfied, the conclusion must hold; the number of premises may be zero. Each rule has a name. (By convention, the first word of the name is determined by the conclusion judgment; for example, rule names of the form "(Val ...)" are for rules whose conclusion is a value typing judgment.) When needed, conditions restricting the applicability of a rule, as well as abbreviations used within the rule, are annotated next to the rule name or the premises.

For example, the first of the following two rules states that any numeral is an expression of type *Nat*, in any well-formed environment Γ. The second rule states that two expressions M and N denoting natural numbers can be combined into a larger expression M+N, which also denotes a natural number. Moreover, the environment Γ for M and N, which declares the types of any free variable of M and N, carries over to M+N.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : Nat}\ (\text{Val } n)\ \ (n = 0, 1, ...) \qquad \frac{\Gamma \vdash M : Nat \qquad \Gamma \vdash N : Nat}{\Gamma \vdash M{+}N : Nat}\ (\text{Val } +)$$

A fundamental rule states that the empty environment is well formed, with no assumptions:

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset)$$

A collection of type rules is called a *(formal) type system*. Technically, type systems fit into the general framework of *formal proof systems*: collections of rules used to carry out step-by-step deductions. The deductions carried out in type systems concern the typing of programs.

Type derivations

A derivation in a given type system is a tree of judgments with leaves at the top and a root at the bottom, where each judgment is obtained from the ones immediately above it by some rule of the system. A fundamental requirement on type systems is that it must be possible to check whether or not a derivation is properly constructed.

A **valid judgment** is one that can be obtained as the root of a derivation in a given type system. That is, a valid judgment is one that can be obtained by correctly applying the type rules. For example, using the three rules given previously we can build the following derivation, which establishes that $\emptyset \vdash 1{+}2 : Nat$ is a valid judgment. The rule applied at each step is displayed to the right of each conclusion:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset)}{\emptyset \vdash 1 : Nat}\ \text{by (Val } n) \qquad \dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset)}{\emptyset \vdash 2 : Nat}\ \text{by (Val } n)}{\emptyset \vdash 1{+}2 : Nat}\ \text{by (Val } +)$$

Well typing and type inference

In a given type system, a term M is well typed for an environment Γ, if there is a type A such that $\Gamma \vdash M : A$ is a valid judgment; that is, if the term M can be given some type.

The discovery of a derivation (and hence of a type) for a term is called the **type inference** problem. In the simple type system consisting of the rules (Env $\emptyset$), (Val n), and (Val +), a type can be *inferred* for the term 1+2 in the empty environment. This type is *Nat*, by the preceding derivation.

Suppose we now add a type rule with premise $\Gamma \vdash \diamond$ and conclusion $\Gamma \vdash true : Bool$. In the resulting type system we cannot infer any type for the term 1+*true*, because there is no rule for summing a natural number with a boolean. Because of the absence of any derivations for 1+*true*, we say that 1+*true* is *not typeable*, or that it is ill typed, or that it has a **typing error**.

We could further add a type rule with premises $\Gamma \vdash M : Nat$ and $\Gamma \vdash N : Bool$, and with conclusion $\Gamma \vdash M+N : Nat$ (e.g., with the intent of interpreting *true* as 1). In such a type system, a type could be inferred for the term 1+*true*, which would now be well typed.

Thus, the type inference problem for a given term is very sensitive to the type system in question. An algorithm for type inference may be very easy, very hard, or impossible to find, depending on the type system. If found, the best algorithm may be very efficient, or hopelessly slow. Although type systems are expressed and often designed in the abstract, their practical utility depends on the availability of good type inference algorithms.

The type inference problem for explicitly typed procedural languages such as Pascal is fairly easily solved; we treat it in section 8. The type inference problem for implicitly typed languages such as ML is much more subtle, and we do not treat it here. The basic algorithm is well understood (several descriptions of it appear in the literature) and is widely used. However, the versions of the algorithm that are used in practice are complex and are still being investigated.

The type inference problem becomes particularly hard in the presence of **polymorphism** (discussed in section 5). The type inference problems for the explicitly typed polymorphic features of Ada, CLU, and Standard ML are treatable in practice. However, these problems are typically solved by algorithms, without first describing the associated type systems. The purest and most general type system for polymorphism is embodied by a λ-calculus discussed in section 5. The type inference algorithm for this polymorphic λ-calculus is fairly easy, and we present it in section 8. The simplicity of the solution, however, depends on impractically verbose typing annotations. To make this general polymorphism practical, some type information has to be omitted. Such type inference problems are still an area of active research.

Type soundness

We have now established all of the general notions concerning type systems, and we can begin examining particular type systems. Starting in section 3, we review some very powerful but rather theoretical type systems. The idea is that by first understanding these few systems, it becomes easier to write the type rules for the varied and complex features that one may encounter in programming languages.

When immersing ourselves in type rules, we should keep in mind that a sensible type system is more than just an arbitrary collection of rules. Well typing is meant to correspond to a semantic notion of good program behavior. It is customary to check the internal consistency of a type system by proving a type soundness theorem. This is where type systems meet semantics. For denotational semantics we expect that if $\emptyset \vdash M : A$ is valid, then $[\![M]\!] \in [\![A]\!]$ holds (the value of M belongs to the set of values denoted by the type A), and for operational semantics, we expect that if $\emptyset \vdash M : A$ and M reduces to M' then $\emptyset \vdash M' : A$. In both cases the type soundness theorem asserts that well typed programs compute without execution errors. See [11, 34] for surveys of techniques, as well as state-of-the-art soundness proofs.

3 First-order Type Systems

The type systems found in most common procedural languages are called **first order**. In type-theoretical jargon this means that they lack type parameterization and type abstraction, which

checking, dynamic checking, and absence of checking for program errors in various kinds of languages.

The most important lesson to remember from this chapter is the general framework for formalizing type systems. Understanding type systems, in general terms, is as fundamental as understanding BNF (Backus-Naur Form): it is hard to discuss the typing of programs without the precise language of type systems, just as it is hard to discuss the syntax of programs without the precise language of BNF. In both cases, the existence of a formalism has clear benefits for language design, compiler construction, language learning, and program understanding. We described the formalism of type systems, and how it captures the notions of type soundness and type errors.

Armed with formal type systems, we embarked on the description of an extensive list of program constructions and of their type rules. Many of these constructions are slightly abstracted versions of familiar features, whereas others apply only to obscure corners of common languages. In both cases, our collection of typing constructions is meant as a key for interpreting the typing features of programming languages. Such an interpretation may be nontrivial, particularly because most language definitions do not come with a type system, but we hope to have provided sufficient background for independent study. Some of the advanced type constructions will appear, we expect, more fully, cleanly, and explicitly in future languages.

In the latter part of the chapter, we reviewed some fundamental type inference algorithms: for simple languages, for polymorphic languages, and for languages with subtyping. These algorithms are very simple and general, but are mostly of an illustrative nature. For a host of pragmatic reasons, type inference for real languages becomes much more complex. It is interesting, though, to be able to describe concisely the core of the type inference problem and some of its solutions.

Future directions

The formalization of type systems for programming languages, as described in this chapter, evolved as an application of type theory. Type theory is a branch of formal logic. It aims to replace predicate logics and set theory (which are untyped) with typed logics, as a foundation for mathematics.

One of the motivations for these logical type theories, and one of their more exciting applications, is in the mechanization of mathematics via proof checkers and theorem provers. Typing is useful in theorem provers for exactly the same reasons it is useful in programming. The mechanization of proofs reveals striking similarities between proofs and programs: the structuring problems found in proof construction are analogous to the ones found in program construction. Many of the arguments that demonstrate the need for typed programming languages also demonstrate the need for typed logics.

Comparisons between the type structures developed in type theory and in programming are, thus, very instructive. Function types, product types, (disjoint) union types, and quantified types occur in both disciplines, with similar intents. This is in contrast, for example, to structures used in set theory, such as unions and intersections of sets, and the encoding of functions as sets of pairs, that have no correspondence in the type systems of common programming languages.

Beyond the simplest correspondences between type theory and programming, it turns out that the structures developed in type theory are far more expressive than the ones commonly used in programming. Therefore type theory provides a rich environment for future progress in programming languages.

Conversely, the size of systems that programmers build is vastly greater than the size of proofs that mathematicians usually handle. The management of large programs, and in particular the type structures needed to manage large programs, is relevant to the management of mechanical proofs. Certain type theories developed in programming, for example, for object-orientation and for modularization, go beyond the normal practices found in mathematics, and should have something to contribute to the mechanization of proofs.

Therefore, the cross fertilization between logic and programming will continue, within the common area of type theory. At the moment, some advanced constructions used in programming escape proper type-theoretical formalization. This could be happening 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.

Defining Terms

Abstract type: A data type whose nature is kept hidden, in such a way that only a predetermined collection of operations can operate on it.

Contravariant: A type that varies in the inverse direction from one of its parts with respect to subtyping. The main example is the contravariance of function types in their domain. For example, assume $A<:B$ and vary X from A to B in $X \to C$; we obtain $A \to C :> B \to C$. Thus $X \to C$ varies in the inverse direction of X.

Covariant: A type that varies in the same direction as one of its parts with respect to subtyping. For example, assume $A<:B$ and vary X from A to B in $D \to X$; we obtain $D \to A <: D \to B$. Thus $D \to X$ varies in the same direction as X.

Derivation: A tree of judgments obtained by applying the rules of a type system.

Dynamic checking. A collection of run time tests aimed at detecting and preventing forbidden errors.

Dynamically checked language: A language where good behavior is enforced during execution.

Explicitly typed language: A typed language where types are part of the syntax.

First-order type system: One that does not include quantification over type variables.

Forbidden error: The occurrence of one of a predetermined class of execution errors; typically the improper application of an operation to a value, such as *not*(3).

Good behavior: Same as being well behaved.

Ill typed: A program fragment that does not comply with the rules of a given type system.

Implicitly typed language: A typed language where types are not part of the syntax.

Judgment: A formal assertion relating entities such as terms, types, and environments. Type systems prescribe how to produce valid judgments from other valid judgements.

Polymorphism: The ability of a program fragment to have multiple types (opposite of monomorphism).

Safe language: A language where no untrapped errors can occur.

Second-order type system: One that includes quantification over type variables, either universal or existential.

Static checking. A collection of compile time tests, mostly consisting of typechecking.

Statically checked language: A language where good behavior is determined before execution.

Strongly checked language: A language where no forbidden errors can occur at run time (depending on the definition of forbidden error).

Subsumption: A fundamental rule of subtyping, asserting that if a term has a type A, which is a subtype of a type B, then the term also has type B.

Subtyping: A reflexive and transitive binary relation over types that satisfies subsumption; it asserts the inclusion of collections of values.

Trapped error: An execution error that immediately results in a fault.

Type: A collection of values. An estimate of the collection of values that a program fragment can assume during program execution.

Type inference: The process of finding a type for a program within a given type system.

Type reconstruction: The process of finding a type for a program where type information has been omitted, within a given type system.

Type rule: A component of a type system. A rule stating the conditions under which a particular program construct will not cause forbidden errors.

Type safety: The property stating that programs do not cause untrapped errors.

Type soundness: The property stating that programs do not cause forbidden errors.

Type system: A collection of type rules for a typed programming language. Same as static type system.

Typechecker: The part of a compiler or interpreter that performs typechecking.

Typechecking: The process of checking a program before execution to establish its compliance with a given type system and therefore to prevent the occurrence of forbidden errors.

Typed language: A language with an associated (static) type system, whether or not types are part of the syntax.

Typing error: An error reported by a typechecker to warn against possible execution errors.

Untrapped error: An execution error that does not immediately result in a fault.

Untyped language: A language that does not have a (static) type system, or whose type system has a single type that contains all values.

Valid judgment: A judgment obtained from a derivation in a given type system.

Weakly checked language: A language that is statically checked but provides no clear guarantee of absence of execution errors.

Well behaved: A program fragment that will not produce forbidden errors at run time.

Well formed: Properly constructed according to formal rules.

Well-typed program: A program (fragment) that complies with the rules of a given type system.

References

[1] Aiken, A. and E.L. Wimmers, **Type inclusion constraints and type inference**, *Proc. ACM Conference on Functional Programming and Computer Architecture*, 31-41. 1993.

[2] Amadio, R.M. and L. Cardelli, **Subtyping recursive types**. *ACM Transactions on Programming Languages and Systems* **15**(4), 575-631. 1993.

[3] Birtwistle, G.M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, **Simula Begin**. Studentlitteratur. 1979.

[4] Böhm, C. and A. Berarducci, **Automatic synthesis of typed λ-programs on term algebras**. *Theoretical Computer Science* **39**, 135-154. 1985.

[5] Cardelli, L., **Basic polymorphic typechecking**. *Science of Computer Programming* **8**(2). 147-172. 1987.

[6] Cardelli, L., **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming,* C.A. Gunter and J.C. Mitchell, ed. MIT Press. 373-425. 1994.

[7] Cardelli, L. and P. Wegner, **On understanding types, data abstraction and polymorphism**. *ACM Computing Surveys* **17**(4), 471-522. 1985.

[8] Curien, P.-L. and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$**. *Mathematical Structures in Computer Science* **2**(1), 55-91. 1992.

[9] Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare, **Structured programming**. *Academic Press*, 1972.

[10] Eifrig, J., S. Smith, and V. Trifonov, **Sound polymorphic type inference for objects**. *Proc. OOPSLA'95*, 169-184. 1995.

[11] Gunter, C.A., **Semantics of programming languages: structures and techniques**. Foundations of computing, M. Garey and A. Meyer ed. MIT Press. 1992.

[12] Girard, J.-Y., Y. Lafont, and P. Taylor, **Proofs and types**. Cambridge University Press. 1989.

[13] Gunter, C.A. and J.C. Mitchell, ed., **Theoretical Aspects of Object-Oriented Programming**. MIT Press. 1994.

[14] Huet, G. ed., **Logical foundations of functional programming**. Addison-Wesley. 1990.

[15] Jensen, K., **Pascal user manual and report, second edition**. *Springer Verlag*, 1978.

[16] Liskov, B.H., **CLU Reference Manual**. Lecture Notes in Computer Science 114. Springer-Verlag. 1981.

[17] Milner, R., **A theory of type polymorphism in programming**. *Journal of Computer and System Sciences* **17**, 348-375. 1978.

[18] Milner, R., M. Tofte, and R. Harper, **The definition of Standard ML**. MIT Press. 1989.

[19] Mitchell, J.C., **Coercion and type inference**. *Proc. 11th Annual ACM Symposium on Principles of Programming Languages*, 175-185. 1984.

[20] Mitchell, J.C., **Type systems for programming languages**. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed. North Holland. 365-458. 1990.

[21] Mitchell, J.C.: **Foundations for programming languages**. MIT Press, 1996.

[22] Mitchell, J.C. and G.D. Plotkin, **Abstract types have existential type**. *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*. 37-51. 1985.

[23] Nordström, B., K. Petersson, and J.M. Smith, **Programming in Martin-Löf's type theory**. Oxford Science Publications. 1990.

[24] Palsberg, J., **Efficient inference for object types**. *Information and Computation.* **123**(2), 198-209. 1995.

[25] Pierce, B.C., **Bounded quantification is undecidable**. *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*. 305-315. 1992.

[26] Pierce, B.C., **Types and Programming Languages**. MIT Press, 2002.

[27] Reynolds, J.C., **Towards a theory of type structure**. *Proc. Colloquium sur la programmation*, 408-423. Lecture Notes in Computer Science 19. Springer-Verlag. 1974.

[28] Reynolds, J.C., **Types, abstraction, and parametric polymorphism**. In *Information Processing,* R.E.A. Mason, ed. North Holland. 513-523. 1983.

[29] Schmidt, D.A., **The structure of typed programming languages**. MIT Press. 1994.

[30] Spencer, H., **The ten commandments for C programmers (annotated edition)**. Available on the World Wide Web.

[31] Tofte, M., **Type inference for polymorphic references**. *Information and Computation* **89**, 1-34. 1990.

[32] Wells, J.B., **Typability and type checking in the second-order λ-calculus are equivalent and undecidable.** *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, 176-185. 1994.

[33] Wijngaarden, V., ed. **Revised report on the algorithmic language Algol68**. 1976.

[34] Wright, A.K. and M. Felleisen, **A syntactic approach to type soundness**. *Information and Computation* **115**(1), 38-94. 1994.

Further Information

For a complete background on type systems one should read (1) some material on type theory, which is usually rather hard, (2) some material connecting type theory to computing, and (3) some material about programming languages with advanced type systems.

The book edited by Huet [14] covers a variety of topics in type theory, including several tutorial articles. The book edited by Gunter and Mitchell [13] contains a collection of papers on object-oriented type theory. The book by Nordström, Petersson, and Smith [23] is a recent summary of Martin-Löf's work. Martin-Löf proposed type theory as a general logic that is firmly grounded in computation. He introduced the systematic notation for judgments and type rules used in this chapter. Girard and Reynolds [12, 27] developed the polymorphic λ-calculus (F_2), which inspired much of the work covered in this chapter.

A modern exposition of technical issues that arise from the study of type systems can be found in Pierce's book [26], in Gunter's book [11], in Mitchell's article in the Handbook of Theoretical Computer Science [20], and in Mitchell's book [21].

Closer to programming languages, rich type systems were pioneered in the period between the development of Algol and the establishment of structured programming [9], and were developed into a new generation of richly-typed languages, including Pascal [15], Algol68 [33], Simula [3], CLU [16], and ML [18]. Reynolds gave type-theoretical explanations for polymorphism and data abstraction [27, 28]. (On that topic, see also [7, 22].) The book by Schmidt [29] covers several issues discussed in this chapter, and provides more details on common language constructions.

Milner's paper on type inference for ML [17] brought the study of type systems and type inference to a new level. It includes an algorithm for polymorphic type inference, and the first proof of type soundness for a (simplified) programming language, based on a denotational tech-

nique. A more accessible exposition of the algorithm described in that paper can be found in [5]. Proofs of type soundness are now often based on operational techniques [31, 34]. Currently, Standard ML is the only widely used programming language with a formally specified type system [18], although similar work has now been carried out for large fragments of Java.