Type Systems

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

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.

非正式的陈述
澄清
什么构成执行错误

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 (see Chapter 139) 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 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 and Modula-3's TYPECASE 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 [28] 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	LISP
Unsafe	C	Assembler

Should languages be safe?

Lack of safety in a language design is motivated by performance considerations (when not introduced by mistake). 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.

Safety, however, is cost effective according to different measures. Safety produces fail-stop behavior in case of execution errors, reducing debugging time. Moreover, 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.

Thus, the choice between a safe and unsafe language may be ultimately related to a tradeoff between development time and execution time. Although undeniable, the advantages of safety have not yet caused a widespread adoption of safe languages. Instead of regarding lack of safety as bad, many developers consider almost safety as almost good, and live with the consequences.

Should languages be typed?

The issue of whether programming languages should have types (even with weak checking) is still subject to some debate. There is little doubt, though, 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 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 (see Chapter 139).

When this much is settled one can proceed to define the type rules of the language, which 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 (see Chapters 139 and 141) 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 compilers, but the precise mixture is rarely prescribed in the corresponding language definition. In contrast, pure structural equivalence can be easily and precisely defined by means of type rules. Moreover, structural equivalence has unique advantages when typed data has to be stored or transmitted over a network (as in Modula-3). By-name equivalence cannot deal easily with interacting program sources 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). If by-name equivalence is desired for a language, one should attempt to write the appropriate type rules: the arbitrary nature of by-name equivalence then becomes apparent. Moreover, 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:

$\Gamma \vdash \Im$ $\Im$ is an assertion; the free variables of $\Im$ are declared in Γ

We say that Γ *entails* $\Im$. 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 $x_1 ... x_n$ declared in Γ is indicated by *dom*(Γ). The form of the *assertion* $\Im$ varies from judgment to judgment, but all the free variables of $\Im$ 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.

$\emptyset \vdash \mathit{true} : \mathit{Bool}$ *true* has type *Bool*

$\emptyset, x{:}\mathit{Nat} \vdash x{+}1 : \mathit{Nat}$ 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., $\Gamma \vdash \mathit{true} : \mathit{Bool}$) or *invalid* (e.g., $\Gamma \vdash \mathit{true} : \mathit{Nat}$). 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).

(Rule name) (Annotations)

$$\frac{\Gamma_1 \vdash \Im_1 \quad ... \quad \Gamma_n \vdash \Im_n \quad \text{(Annotations)}}{\Gamma \vdash \Im}$$

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 $M+N$.

(Val n) ($n = 0, 1, ...$)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : Nat}$$

(Val +)

$$\frac{\Gamma \vdash M : Nat \qquad \Gamma \vdash N : Nat}{\Gamma \vdash M{+}N : Nat}$$

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

(Env $\emptyset$)

$$\frac{}{\emptyset \vdash \diamond}$$

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 $\varnothing \vdash 1{+}2 : \mathit{Nat}$ is a valid judgment. The rule applied at each step is displayed to the right of each conclusion:

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

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 $\varnothing$), (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 $\Gamma \vdash \mathit{true} : \mathit{Bool}$. 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 $\Gamma \vdash M : \mathit{Nat}$ and $\Gamma \vdash N : \mathit{Bool}$, and with conclusion $\Gamma \vdash M{+}N : \mathit{Nat}$ (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. In the notation of Chapter 141, for denotational semantics we expect that if $\phi \vdash M : A$ 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 $\phi \vdash M : A$ and M reduces to M' then $\phi \vdash M' : A$. In both cases the type soundness theorem asserts that well typed programs compute without execution errors. See [11, 33] 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 are **second order** features. First-order type systems include (rather confusingly) higher order functions. Pascal and Algol68 have rich first-order type systems, whereas FORTRAN and Algol60 have very poor ones.

A minimal first-order type system can be given for the untyped λ-calculus described in Chapter 139. There, the untyped λ-abstraction $\lambda x.M$ represents a function of parameter x and result M. Typing for this calculus requires only function types and some base types; we will see later how to add other common type structures.

The first-order typed λ-calculus is called system F_1. The main change from the untyped λ-calculus is the addition of type annotations for λ-abstractions, using the syntax $\lambda x{:}A.M$, where x is the function parameter, A is its type, and M is the body of the function. (In a typed programming language we would likely include the type of the result, but this is not necessary here.) The step from $\lambda x.M$ to $\lambda x{:}A.M$ is typical of any progression from an untyped to a typed language: bound variables acquire type annotations.

Since F_1 is based mainly on function values, the most interesting types are function types: $A \rightarrow B$ is the type of functions with arguments of type A and results of type B. To get started, though, we also need some basic types over which to build function types. We indicate by *Basic* a collection of such types, and by $K \in Basic$ any such type. At this point basic types are purely a technical necessity, but shortly we will consider interesting basic types such as *Bool* and *Nat*.

The syntax of F_1 is given in Table 2. It is important to comment briefly on the role of syntax in typed languages. In the case of the untyped λ-calculus, the context-free syntax describes exactly the legal programs. This is not the case in typed calculi, since good behavior is not (usually) a context-free property. The task of describing the legal programs is taken over by the type system. For example, $\lambda x{:}K.x(y)$ respects the syntax of F_1 given in Table 2, but is not a program of F_1 because it is not well typed, since K is not a function type. The context-free syntax is still needed, but only in order to define the notions of free and bound variables; that is, to define the scoping rules of the language. Based on the scoping rules, terms that differ only in their bound variables, such as $\lambda x{:}K.x$ and $\lambda y{:}K.y$, are considered *syntactically identical*. This convenient identification is implicitly assumed in the type rules (one may have to rename bound variables in order to apply certain type rules).

Table 2. Syntax of F_1

A,B ::=		types
K	$K \in Basic$	basic types
$A \to B$		function types
M,N ::=		terms
x		variable
$\lambda x{:}A.M$		function
$M\ N$		application

The definition of free variables for F_1 is the same as for the untyped λ-calculus from Chapter 139, simply ignoring the typing annotations.

We need only three simple judgments for F_1; they are shown in Table 3. The judgment $\Gamma \vdash A$ is in a sense redundant, since all syntactically correct types A are automatically well formed in any environment Γ. In second order systems, however, the well formedness of types is not captured by grammar alone, and the judgement $\Gamma \vdash A$ becomes essential. It is convenient to adopt this judgment now, so that later extensions are easier.

Table 3. Judgments for F_1

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash M : A$	M is a well-formed term of type A in Γ

Validity for these judgments is defined by the rules in Table 4. The rule (Env $\emptyset$) is the only one that does not require assumptions (i.e., it is the only *axiom*). It states that the empty environment is a valid environment. The rule (Env x) is used to extend an environment Γ to a longer environment $\Gamma, x{:}A$, provided that A is a valid type in Γ.

Note that the assumption $\Gamma \vdash A$ implies, inductively, that Γ is valid. That is, in the process of deriving $\Gamma \vdash A$ we must have derived $\Gamma \vdash \diamond$. Another requirement of this rule is that the variable x must not be defined in Γ. We are careful to keep variables distinct in environments, so that when $\Gamma, x{:}A \vdash M : B$ has been derived, as in the assumption of (Val Fun), we know that x cannot occur in $dom(\Gamma)$.

Table 4. Rules for $\mathbf{F_1}$

$$\begin{array}{lll}
\text{(Env } \varnothing) & \text{(Env } x) & \\
\dfrac{}{\varnothing \vdash \diamond} & \dfrac{\Gamma \vdash A \qquad x \notin dom(\Gamma)}{\Gamma, x{:}A \vdash \diamond} & \\[2ex]
\text{(Type Const)} & \text{(Type Arrow)} & \\
\dfrac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K} & \dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} & \\[2ex]
\text{(Val } x) & \text{(Val Fun)} & \text{(Val Appl)} \\
\dfrac{\Gamma', x{:}A, \Gamma'' \vdash \diamond}{\Gamma', x{:}A, \Gamma'' \vdash x{:}A} & \dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B} & \dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}
\end{array}$$

The rules (Type Const) and (Type Arrow) construct types. The rule (Val x) extracts an assumption from an environment: we use the notation $\Gamma', x{:}A, \Gamma''$, rather informally, to indicate that $x{:}A$ occurs somewhere in the environment. The rule (Val Fun) gives the type $A \to B$ to a function, provided that the function body receives the type B under the assumption that the formal parameter has type A. Note how the environment changes length in this rule. The rule (Val Appl) applies a function to an argument: the same type A must appear twice when verifying the premises.

Table 5 shows a rather large derivation where all of the rules of F_1 are used.

Table 5. A derivation in $\mathbf{F_1}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \diamond}\,\text{by (Env }\varnothing)}{\varnothing \vdash K}\,\text{by (Type Const)} \quad \dfrac{\dfrac{}{\varnothing \vdash \diamond}\,\text{by (Env }\varnothing)}{\varnothing \vdash K}\,\text{by (Type Const)}}{\varnothing \vdash K \to K}\,\text{by (Type Arrow)}}{\varnothing, y{:}K \to K \vdash \diamond}\,\text{by (Env } x)}{\varnothing, y{:}K \to K \vdash K}\,\text{by (Type Const)}}{\varnothing, y{:}K \to K, z{:}K \vdash \diamond}\,\text{by (Env } x)}{\varnothing, y{:}K \to K, z{:}K \vdash y : K \to K}\,\text{by (Val } x) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \diamond}\,\text{by (Env }\varnothing)}{\varnothing \vdash K}\,\text{by (Type Const)} \quad \dfrac{\dfrac{}{\varnothing \vdash \diamond}\,\text{by (Env }\varnothing)}{\varnothing \vdash K}\,\text{by (Type Const)}}{\varnothing \vdash K \to K}\,\text{by (Type Arrow)}}{\varnothing, y{:}K \to K \vdash \diamond}\,\text{by (Env } x)}{\varnothing, y{:}K \to K \vdash K}\,\text{by (Type Const)}}{\varnothing, y{:}K \to K, z{:}K \vdash \diamond}\,\text{by (Env } x)}{\varnothing, y{:}K \to K, z{:}K \vdash z : K}\,\text{by (Val } x)}{\dfrac{\varnothing, y{:}K \to K, z{:}K \vdash y(z) : K \quad \text{by (Val Appl)}}{\varnothing, y{:}K \to K \vdash \lambda z{:}K.y(z) : K \to K \quad \text{by (Val Fun)}}}$$

Now that we have examined the basic structure of a simple first-order type system, we can begin enriching it to bring it closer to the type structure of actual programming languages. We are going to add a set of rules for each new type construction, following a fairly regular pattern. We begin with some basic data types: the type *Unit*, whose only value is the constant *unit*; the type *Bool*, whose values are *true* and *false*; and the type *Nat*, whose values are the natural numbers.

The *Unit* type is often used as a filler for uninteresting arguments and results; it is called *Void* or *Null* in some languages. There are no operations on *Unit*, so we need only a rule stating that *Unit* is a legal type, and one stating that *unit* is a legal value of type *Unit* (Table 6).

Table 6. Unit Type

(Type Unit)	(Val Unit)
$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Unit}}$	$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{unit} : \mathit{Unit}}$

We have a similar pattern of rules for *Bool*, but booleans also have a useful operation, the conditional, that has its own typing rule (Table 7). In the rule (Val Cond) the two branches of the conditional must have the same type *A*, because either may produce the result.

The rule (Val Cond) illustrates a subtle issue about the amount of type information needed for typechecking. When encountering a conditional expression, a typechecker has to infer separately the types of N_1 and N_2, and then find a single type *A* that is compatible with both. In some type systems it might not be easy or possible to determine this single type from the types of N_1 and N_2. To account for this potential typechecking difficulty, we use a subscripted type to express additional type information: if_A is a hint to the typechecker that the result type should be *A*, and that types inferred for N_1 and N_2 should be separately compared with the given *A*. In general, we use subscripted types to indicate information that may be useful or necessary for typechecking, depending on the whole type system under consideration. It is often the task of a typechecker to synthesize this additional information. When it is possible to do so, subscripts may be omitted. (Most common languages do not require the annotation if_A.)

Table 7. Bool Type

(Type Bool)	(Val True)	(Val False)
$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Bool}}$	$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{true} : \mathit{Bool}}$	$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{false} : \mathit{Bool}}$

(Val Cond)

$$\frac{\Gamma \vdash M : \mathit{Bool} \qquad \Gamma \vdash N_1 : A \qquad \Gamma \vdash N_2 : A}{\Gamma \vdash (\mathit{if}_A\ M\ \mathit{then}\ N_1\ \mathit{else}\ N_2) : A}$$

The type of natural numbers, *Nat* (Table 8), has 0 and *succ* (successor) as generators. Alternatively, as we did earlier, a single rule could state that all numeric constants have type *Nat*. Computations on *Nat* are made possible by the *pred* (predecessor) and *isZero* (test for zero) primitives; other sets of primitives can be chosen.

Table 8. Nat Type

$$\begin{array}{lll}
\text{(Type Nat)} & \text{(Val Zero)} & \text{(Val Succ)} \\
\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Nat}} & \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash 0 : \mathit{Nat}} & \dfrac{\Gamma \vdash M : \mathit{Nat}}{\Gamma \vdash \mathit{succ}\ M : \mathit{Nat}}
\end{array}$$

$$\begin{array}{ll}
\text{(Val Pred)} & \text{(Val IsZero)} \\
\dfrac{\Gamma \vdash M : \mathit{Nat}}{\Gamma \vdash \mathit{pred}\ M : \mathit{Nat}} & \dfrac{\Gamma \vdash M : \mathit{Nat}}{\Gamma \vdash \mathit{isZero}\ M : \mathit{Bool}}
\end{array}$$

Now that we have a collection of basic types, we can begin looking at structured types, starting with *product types* (Table 9). A product type $A_1 \times A_2$ is the type of pairs of values with first component of type A_1 and second component of type A_2. These components can be extracted with the projections *first* and *second*, respectively. Instead of (or in addition to) the projections, one can use a *with* statement that decomposes a pair M and binds its components to two separate variables x_1 and x_2 in the scope N. The *with* notation is related to pattern matching in ML, but also to Pascal's *with*; the connection with the latter will become clearer when we consider record types.

Product types can be easily generalized to *tuple types* $A_1 \times ... \times A_n$, with corresponding generalized projections and generalized *with*.

Table 9. Product Types

$$\begin{array}{ll}
\text{(Type Product)} & \text{(Val Pair)} \\
\dfrac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2} & \dfrac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}
\end{array}$$

$$\begin{array}{ll}
\text{(Val First)} & \text{(Val Second)} \\
\dfrac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathit{first}\ M : A_1} & \dfrac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathit{second}\ M : A_2}
\end{array}$$

$$\begin{array}{l}
\text{(Val With)} \\
\dfrac{\Gamma \vdash M : A_1 \times A_2 \qquad \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash N : B}{\Gamma \vdash (\mathit{with}\ (x_1{:}A_1, x_2{:}A_2) := M\ \mathit{do}\ N) : B}
\end{array}$$

Union types (Table 10) are often overlooked, but are just as important as product types for expressiveness. An element of a union type $A_1 + A_2$ can be thought of as an element of A_1 tagged with a *left* token (created by *inLeft*), or an element of A_2 tagged with a *right* token (created by *inRight*). The tags can be tested by *isLeft* and *isRight*, and the corresponding value extracted with *asLeft* and *asRight*. If *asLeft* is mistakenly applied to a right-tagged value, a trapped error or exception is produced; this trapped error is not considered a forbidden error. Note that it is safe to assume that any result of *asLeft* has type A_1, because either the argument is left tagged, in which case the result is indeed of type A_1, or it is right tagged, in which case there is no result. Subscripts are used to disambiguate some of the rules, as we discussed in the case of the conditional.

The rule (Val Case) describes an elegant construct that can replace *isLeft*, *isRight*, *asLeft*, *asRight*, and the related trapped errors. (It also eliminates any dependence of union operations on the *Bool* type). The *case* construct executes one of two branches depending on the tag of M, with the untagged contents of M bound to x_1 or x_2 in the scope of N_1 or N_2, respectively. A vertical bar separates the branches.

Table 10. Union Types

(Type Union)

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1{+}A_2}$$

(Val inLeft)

$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash \mathit{inLeft}_{A_2}\, M_1 : A_1{+}A_2}$$

(Val inRight)

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathit{inRight}_{A_1}\, M_2 : A_1{+}A_2}$$

(Val isLeft)

$$\frac{\Gamma \vdash M : A_1{+}A_2}{\Gamma \vdash \mathit{isLeft}\ M : \mathit{Bool}}$$

(Val isRight)

$$\frac{\Gamma \vdash M : A_1{+}A_2}{\Gamma \vdash \mathit{isRight}\ M : \mathit{Bool}}$$

(Val asLeft)

$$\frac{\Gamma \vdash M : A_1{+}A_2}{\Gamma \vdash \mathit{asLeft}\ M : A_1}$$

(Val asRight)

$$\frac{\Gamma \vdash M : A_1{+}A_2}{\Gamma \vdash \mathit{asRight}\ M : A_2}$$

(Val Case)

$$\frac{\Gamma \vdash M : A_1{+}A_2 \qquad \Gamma, x_1{:}A_1 \vdash N_1 : B \qquad \Gamma, x_2{:}A_2 \vdash N_2 : B}{\Gamma \vdash (\mathit{case}_B\ M\ \mathit{of}\ x_1{:}A_1\ \mathit{then}\ N_1 \mid x_2{:}A_2\ \mathit{then}\ N_2) : B}$$

In terms of expressiveness (if not of implementation) note that the type *Bool* can be defined as *Unit* + *Unit*, in which case the *case* construct reduces to the conditional. The type *Int* can be defined as *Nat* + *Nat*, with one copy of *Nat* for the nonnegative integers and the other for the negative ones. We can define a prototypical trapped error as $\mathit{error}_A = \mathit{asRight}(\mathit{inLeft}_A(\mathit{unit})) : A$. Thus, we can build an error expression for each type.

Product types and union types can be iterated to produce tuple types and multiple unions. However, these derived types are rather inconvenient, and are rarely seen in languages. Instead, *labeled* products and unions are used: they go under the names of *record types* and *variant types*, respectively.

A record type is the familiar named collection of types, with a value-level operation for extracting components by name. The rules in Table 11 assume the syntactic identification of record types and records up to reordering of their labeled components;

this is analogous to the syntactic identification of functions up to renaming of bound variables.

The *with* statement of product types is generalized to record types in (Val Record With). The components of the record M labeled l_1, ..., l_n are bound to the variables x_1, ..., x_n in the scope of N. Pascal has a similar construct, also called *with*, but where the binding variables are left implicit. (This has the rather unfortunate consequence of making scoping depend on typechecking, and of causing hard-to-trace bugs due to hidden variable clashes.)

Product types $A_1 \times A_2$ can be defined as $Record(first{:}A_1, second{:}A_2)$.

Table 11. Record Types

(Type Record) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n}{\Gamma \vdash Record(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Record) (l_i distinct)

$$\frac{\Gamma \vdash M_1 : A_1 \quad ... \quad \Gamma \vdash M_n : A_n}{\Gamma \vdash record(l_1{=}M_1, ..., l_n{=}M_n) : Record(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Record Select)

$$\frac{\Gamma \vdash M : Record(l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M.l_j : A_j}$$

(Val Record With)

$$\frac{\Gamma \vdash M : Record(l_1{:}A_1, ..., l_n{:}A_n) \qquad \Gamma, x_1{:}A_1, ..., x_n{:}A_n \vdash N : B}{\Gamma \vdash (with\ (l_1{=}x_1{:}A_1, ..., l_n{=}x_n{:}A_n) := M\ do\ N) : B}$$

Variant types (Table 12) are named disjoint unions of types; they are syntactically identified up to reordering of components. The *is l* construct generalizes *isLeft* and *isRight*, and the *as l* construct generalizes *asLeft* and *asRight*. As with unions, these constructs may be replaced by a *case* statement, which has now multiple branches.

Union types A_1+A_2 can be defined as $Variant(left{:}A_1, right{:}A_2)$. Enumeration types, such as $\{red, green, blue\}$, can be defined as $Variant(red{:}Unit, green{:}Unit, blue{:}Unit)$.

Table 12. Variant Types

(Type Variant) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n}{\Gamma \vdash Variant(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Variant) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n \qquad \Gamma \vdash M_{\mathrm{j}} : A_{\mathrm{j}} \qquad j \in 1..n}{\Gamma \vdash variant_{(l_1:A_1, ..., l_n:A_n)}(l_{\mathrm{j}}{=}M_{\mathrm{j}}) : Variant(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Variant Is)

$$\frac{\Gamma \vdash M : Variant(l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M\ is\ l_j : Bool}$$

(Val Variant As)

$$\frac{\Gamma \vdash M : Variant(l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M\ as\ l_j : A_j}$$

(Val Variant Case)

$$\frac{\Gamma \vdash M : \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n) \qquad \Gamma, x_1{:}A_1 \vdash N_1 : B \;\; ... \;\; \Gamma, x_n{:}A_n \vdash N_n : B}{\Gamma \vdash (\mathit{case}_B\ M\ \mathit{of}\ l_1{=}x_1{:}A_1\ \mathit{then}\ N_1 \mid ... \mid l_n{=}x_n{:}A_n\ \mathit{then}\ N_n) : B}$$

Reference types (Table 13) can be used as the fundamental type of mutable locations in imperative languages. An element of *Ref*(A) is a mutable cell containing an element of type A. A new cell can be allocated by (Val Ref), updated by (Val Assign), and explicitly dereferenced by (Val Deref). Since the main purpose of an assignment is to perform a side effect, its resulting value is chosen to be *unit*.

Table 13. Reference Types

(Type Ref)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathit{Ref}\ A}$$

(Val Ref)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathit{ref}\ M : \mathit{Ref}\ A}$$

(Val Deref)

$$\frac{\Gamma \vdash M : \mathit{Ref}\ A}{\Gamma \vdash \mathit{deref}\ M : A}$$

(Val Assign)

$$\frac{\Gamma \vdash M : \mathit{Ref}\ A \qquad \Gamma \vdash N : A}{\Gamma \vdash M := N : \mathit{Unit}}$$

Common mutable types can be derived from *Ref*. Mutable record types, for example, can be modeled as record types containing *Ref* types.

Table 14. An implementation of arrays

$\mathit{Array}(A) \triangleq$ $\mathit{Nat} \times (\mathit{Nat} \to \mathit{Ref}(A))$	Array type a bound plus a map from indices less than the bound to refs
$\mathit{array}_A(N,M) \triangleq$ $\mathit{let}\ \mathit{cell}_0 : \mathit{Ref}(A) = \mathit{ref}(M)\ \mathit{and}\ ...\ \mathit{and}\ \mathit{cell}_{N\text{-}1} : \mathit{Ref}(A) = \mathit{ref}(M)$ $\mathit{in}\ \langle N, \lambda x{:}\mathit{Nat}.\ \mathit{if}\ x{=}0\ \mathit{then}\ \mathit{cell}_0\ \mathit{else}\ \mathit{if}\ ...\ \mathit{else}\ \mathit{if}\ x{=}N\text{-}1\ \mathit{then}\ \mathit{cell}_{N\text{-}1}\ \mathit{else}\ \mathit{error}_{\mathit{Ref}(A)}\rangle$	Array constructor (for N refs initialized to M)
$\mathit{bound}(M) \triangleq$ $\mathit{first}\ M$	Array bound
$M[N]_A \triangleq$ $\mathit{if}\ N < \mathit{first}\ M$ $\mathit{then}\ \mathit{deref}\ ((\mathit{second}\ M)(N))$ $\mathit{else}\ \mathit{error}_A$	Array indexing
$M[N] := P \triangleq$ $\mathit{if}\ N < \mathit{first}\ M$ $\mathit{then}\ ((\mathit{second}\ M)(N)) := P$ $\mathit{else}\ \mathit{error}_{\mathit{Unit}}$	Array update

More interestingly, arrays and array operations can be modeled as in Table 14, where *Array*(A) is the type of arrays of elements of type A of some length. (The code uses some arithmetic primitives and local *let* declarations.) The code in Table 14 is of course an inefficient implementation of arrays, but it illustrates a point: the type rules for more complex constructions can be derived from the type rule for simpler constructions. The typing rules for array operations shown in Table 15 can be easily derived from Table 14, according to the rules for products, functions, and refs.

Table 15. Array Types (derived rule)

(Type Array)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathit{Array}(A)}$$

(Val Array)

$$\frac{\Gamma \vdash N : \mathit{Nat} \qquad \Gamma \vdash M : A}{\Gamma \vdash \mathit{array}(N{,}M) : \mathit{Array}(A)}$$

(Val Array Bound)

$$\frac{\Gamma \vdash M : \mathit{Array}(A)}{\Gamma \vdash \mathit{bound}\ M : \mathit{Nat}}$$

(Val Array Index)

$$\frac{\Gamma \vdash N : \mathit{Nat} \qquad \Gamma \vdash M\!: \mathit{Array}(A)}{\Gamma \vdash M[N] : A}$$

(Val Array Update)

$$\frac{\Gamma \vdash N : \mathit{Nat} \qquad \Gamma \vdash M : \mathit{Array}(A) \qquad \Gamma \vdash P : A}{\Gamma \vdash M[N] := P : \mathit{Unit}}$$

In most programming language, types can be defined recursively. Recursive types are important, since they make all of the other type constructions more useful. They are often introduced implicitly, or without precise explanation, and their characteristics are rather subtle. Hence, their formalization deserves particular care.

The treatment of recursive types requires a rather fundamental addition to F_1: environments are extended to include type variables X. These type variables are used in recursive types of the form $\mu X.A$ (Table 16), which intuitively denote solutions to recursive equations of the form X=A where X may occur in A. The operations *unfold* and *fold* are explicit coercions that map between a recursive type $\mu X.A$ and its unfolding $[\mu X.A/X]A$ (where $[B/X]A$ is the substitution of B for all free occurrences of X in A), and vice versa. These coercions do not have any run time effect (in the sense that *unfold*(*fold*(M))=M and *fold*(*unfold*(M'))=M'). They are usually omitted from the syntax of practical programming languages, but their existence makes formal treatment easier.

Table 16. Recursive Types

(Env X)

$$\frac{\Gamma \vdash \diamond \qquad X \notin \mathit{dom}(\Gamma)}{\Gamma, X \vdash \diamond}$$

(Type Rec)

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \mu X.A}$$

(Val Fold) (Val Unfold)

$$\frac{\Gamma \vdash M : [\mu X.A/X]A}{\Gamma \vdash \mathit{fold}_{\mu X.A}\, M : \mu X.A} \qquad \frac{\Gamma \vdash M : \mu X.A}{\Gamma \vdash \mathit{unfold}_{\mu X.A}\, M : [\mu X.A/X]A}$$

A standard application of recursive types is in defining types of lists and trees, in conjunction with products and union types. The type $List_A$ of lists of elements of type A is defined in Table 17, together with the list constructors *nil* and *cons*, and the list analyzer *listCase*.

Table 17. List Types

$$List_A \triangleq \mu X.Unit+(A\times X)$$

$$nil_A : List_A \triangleq fold(inLeft\ unit)$$

$$cons_A : A\to List_A\to List_A \triangleq \lambda hd{:}A.\ \lambda tl{:}List_A.\ fold(inRight\ \langle hd,tl\rangle)$$

$$\begin{aligned} & listCase_{A,B} : List_A\to B\to(A\times List_A\to B)\to B \triangleq \\ & \quad \lambda l{:}List_A.\ \lambda n{:}B.\ \lambda c{:}A\times List_A\to B. \\ & \qquad case\ (unfold\ l)\ of\ unit{:}Unit\ then\ n \mid p{:}A\times List_A\ then\ c\ p \end{aligned}$$

Recursive types can be used together with record and variant types, to define complex tree structures such as abstract syntax trees. The *case* and *with* statements can then be used to analyze these trees conveniently.

When used in conjunction with function types, recursive types are surprisingly expressive. Via clever encodings, one can show that recursion at the value level is already implicit in recursive types: there is no need to introduce recursion as a separate construct. Moreover, in the presence of recursive types, untyped programming can be carried out within typed languages. More precisely, Table 18 shows how to define, for any type A, a divergent element $\bot_A$ of that type, and a fixpoint operator $\mathbf{Y}_A$ for that type. Table 19 shows how to encode the untyped λ-calculus within typed calculi. (These encoding are for call-by-name; they take slightly different forms in call-by-value.)

Table 18. Encoding of Divergence and Recursion via Recursive Types

$$\bot_A : A \triangleq (\lambda x{:}B.\ (unfold_B\ x)\ x)\ (fold_B\ (\lambda x{:}B.\ (unfold_B\ x)\ x))$$

$$\mathbf{Y}_A : (A\to A)\to A \triangleq \lambda f{:}A\to A.\ (\lambda x{:}B.\ f\,((unfold_B\ x)\ x))\ (fold_B\ (\lambda x{:}B.\ f\,((unfold_B\ x)\ x)))$$

where $B \equiv \mu X.X\to A$, for an arbitrary A

Table 19. Encoding the Untyped λ-calculus via Recursive Types

$V \triangleq \mu X.X\to X$	the type of untyped λ-terms
$\langle\!\langle x\rangle\!\rangle \triangleq x$ $\langle\!\langle \lambda x.M\rangle\!\rangle \triangleq fold_V\ (\lambda x{:}V.\langle\!\langle M\rangle\!\rangle)$ $\langle\!\langle M\ N\rangle\!\rangle \triangleq (unfold_V\ \langle\!\langle M\rangle\!\rangle)\ \langle\!\langle N\rangle\!\rangle$	translation $\langle\!\langle$-$\rangle\!\rangle$ from untyped λ-terms to V elements

Type equivalence becomes particularly interesting in the presence of recursive types. We have sidestepped several problems here by not dealing with type definitions, by requiring explicit *fold unfold* coercions between a recursive type and its unfolding, and by not assuming any identifications between recursive types except for renaming of bound variables. In the current formulation we do not need to define a formal judgment for type equivalence: two recursive types are equivalent simply if they are structurally identical (up to renaming of bound variables). This simplified approach can be extended to include type definitions and type equivalence up to unfolding of recursive types [2].

4 First-order Type Systems for Imperative Languages

Imperative languages have a slightly different style of type systems, mostly because they distinguish commands, which do not produce values, from expressions, which do produce values. (It is quite possible to reduce commands to expressions by giving them type *Unit*, but we prefer to remain faithful to the natural distinction.)

As an example of a type system for an imperative language, we consider the untyped imperative language of Chapter 141 extended with variable declarations. This language permits us to study type rules for declarations, which we have not considered so far. The treatment of procedures and data types is very rudimentary in this language, but the rules for functions and data described in section 3 can be easily adapted.

The meaning of the features of the imperative language should be self-evident. If not, the reader is advised to skip section 4, and come back to it after reading a portion of Chapter 141.

Table 20. Syntax of the imperative language

$A ::=$	types
$Bool$	boolean type
Nat	natural numbers type
$Proc$	procedure type (no arguments, no result)
$D ::=$	declarations
proc $I = C$	procedure declaration
var $I : A = E$	variable declaration
$C ::=$	commands
$I := E$	assignment
$C_1; C_2$	sequential composition
begin D in C end	block
call I	procedure call
while E do C end	while loop

E ::=	expressions
I	identifier
N	numeral
$E_1 + E_2$	sum of two numbers
E_1 not= E_2	inequality of two numbers

The judgments for our imperative language are listed in Table 21. The judgments $\Gamma \vdash C$ and $\Gamma \vdash E : A$ correspond to the single judgment $\Gamma \vdash M : A$ of F_1, since we now have a distinction between commands C and expressions E. The judgment $\Gamma \vdash D \therefore S$ assigns a *signature* S to a *declaration* D; a signature is essentially the type of a declaration. In this simple language a signature consists of a single component, for example x : *Nat*, and a matching declaration could be var $x : Nat = 3$. In general, signatures would consist of lists of such components, and would look very similar or identical to environments Γ.

Table 21. Judgments for the imperative language

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash C$	C is a well-formed command in Γ
$\Gamma \vdash E : A$	E is a well-formed expression of type A in Γ
$\Gamma \vdash D \therefore S$	D is a well-formed declaration of signature S in Γ

Table 22 lists the type rules for the imperative language.

Table 22. Type rules for the imperative language

(Env $\emptyset$) (Env I)

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I{:}A \vdash \diamond}$$

(Type Bool) (Type Nat) (Type Proc)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Nat} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Proc}$$

(Decl Proc) (Decl Var)

$$\frac{\Gamma \vdash C}{\Gamma \vdash (\mathrm{proc}\ I = C) \therefore (I : Proc)} \qquad \frac{\Gamma \vdash E : A \qquad A \in \{Bool, Nat\}}{\Gamma \vdash (\mathrm{var}\ I : A = E) \therefore (I : A)}$$

(Comm Assign) (Comm Sequence)

$$\frac{\Gamma \vdash I : A \qquad \Gamma \vdash E : A}{\Gamma \vdash I := E} \qquad \frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1 ; C_2}$$

(Comm Block)
$$\frac{\Gamma \vdash D \therefore (I : A) \qquad \Gamma, I{:}A \vdash C}{\Gamma \vdash \text{begin } D \text{ in } C \text{ end}}$$

(Comm Call)
$$\frac{\Gamma \vdash I : \mathit{Proc}}{\Gamma \vdash \text{call } I}$$

(Comm While)
$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash C}{\Gamma \vdash \text{while } E \text{ do } C \text{ end}}$$

(Expr Identifier)
$$\frac{\Gamma_1, I{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, I{:}A, \Gamma_2 \vdash I : A}$$

(Expr Numeral)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N : \mathit{Nat}}$$

(Expr Plus)
$$\frac{\Gamma \vdash E_1 : \mathit{Nat} \qquad \Gamma \vdash E_2 : \mathit{Nat}}{\Gamma \vdash E_1 + E_2 : \mathit{Nat}}$$

(Expr NotEq)
$$\frac{\Gamma \vdash E_1 : \mathit{Nat} \qquad \Gamma \vdash E_2 : \mathit{Nat}}{\Gamma \vdash E_1 \text{ not= } E_2 : \mathit{Bool}}$$

The rules (Env ...), (Type ...), and (Expr ...) are straightforward variations on the rules we have seen for F_1. The rules (Decl ...) handle the typing of declarations. The rules (Comm ...) handle commands; notice how (Comm Block) converts a signature to a piece of an environment when checking the body of a block.

5 Second-order Type Systems

Many modern languages include constructs for type parameters, type abstraction, or both. Type parameters can be found in the module system of several languages, where a generic module or interface is parameterized by a type to be supplied later. Polymorphic languages such as ML use type parameters more pervasively, at the function level. Type abstraction can similarly be found in conjunction with modules, where it appears as opaque types in interfaces. Languages such as CLU use type abstraction at the data level, to obtain abstract data types. These advanced features can be modeled by so-called second-order type systems.

Second-order type systems extend first-order type systems with the notion of *type parameters*. A new kind of term, written $\lambda X.M$, indicates a program M that is parameterized with respect to a type variable X that stands for an arbitrary type. For example, the identity function for a fixed type A, written $\lambda x{:}A.x$, can be turned into a parametric identity function by abstracting over A and writing $id \triangleq \lambda X.\lambda x{:}X.x$. One can then instantiate such a parametric function to any given type A by a *type instantiation id A*, which produces back $\lambda x{:}A.x$.

Corresponding to the new terms $\lambda X.M$ we need new *universally quantified* types. The type of a term such as $\lambda X.M$ is written $\forall X.A$, meaning that *forall X*, the body M has type A (here M and A may contain occurrences of X). For example, the type of the parametric identity is $id : \forall X.X{\to}X$.

The pure second-order system F_2 (Table 23) is based exclusively on type variables, function types, and quantified types. Note that we are dropping the basic types K, since we can now use type variables as the basic case. It turns out that virtually any basic type of interest can be encoded within F_2 [4]. Similarly, product types, sum types, existential types, and some recursive types, can be encoded within F_2: polymorphism has an amazing expressive power. Thus there is little need, technically, to deal with these type constructions directly.

Table 23. Syntax of F_2

A,B ::=	types
X	type variable
$A{\rightarrow}B$	function type
$\forall X.A$	universally quantified type
M,N ::=	terms
x	variable
$\lambda x{:}A.M$	function
$M\ N$	application
$\lambda X.M$	polymorphic abstraction
$M\ A$	type instantiation

Free variables for F_2 types and terms can be defined in the usual fashion; suffice it to say that $\forall X.A$ binds X in A and $\lambda X.M$ binds X in M. An interesting aspect of F_2 is the substitution of a type for a type variable that is carried out in the type rule for type instantiation, (Val Appl2).

Table 24. Judgments for F_2

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash M : A$	M is a well-formed term of type A in Γ

Table 25. Rules for F_2

(Env $\varnothing$)

$$\frac{}{\varnothing \vdash \diamond}$$

(Env x)

$$\frac{\Gamma \vdash A \qquad x \notin dom(\Gamma)}{\Gamma, x{:}A \vdash \diamond}$$

(Env X)

$$\frac{\Gamma \vdash \diamond \qquad X \notin dom(\Gamma)}{\Gamma, X \vdash \diamond}$$

(Type X)

$$\frac{\Gamma', X, \Gamma'' \vdash \diamond}{\Gamma', X, \Gamma'' \vdash X}$$

(Type Arrow)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A{\rightarrow}B}$$

(Type Forall)

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X.A}$$

(Val x)

$$\frac{\Gamma', x{:}A, \Gamma'' \vdash \diamond}{\Gamma', x{:}A, \Gamma'' \vdash x{:}A}$$

(Val Fun)

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A{\rightarrow}B}$$

(Val Appl)

$$\frac{\Gamma \vdash M : A{\rightarrow}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

(Val Fun2) (Val Appl2)

$$\frac{\Gamma, X \vdash M : A}{\Gamma \vdash \lambda X.M : \forall X.A} \qquad \frac{\Gamma \vdash M : \forall X.A \qquad \Gamma \vdash B}{\Gamma \vdash M\,B : [B/X]A}$$

The judgments for F_2 (Table 24) are the same ones as for F_1, but the environments are richer. With respect to F_1, the new rules (Table 25), are: (Env X), which adds a type variable to the environment; (Type Forall), which constructs a quantified type $\forall X.A$ from a type variable X and a type A where X may occur; (Val Fun2), which builds a polymorphic abstraction; and (Val Appl2), which instantiates a polymorphic abstraction to a given type, where $[B/X]A$ is the substitution of B for all the free occurrences of X in A. For example, if *id* has type $\forall X.X\rightarrow X$ and A is a type, then by (Val Appl2) we have that *id* A has type $[A/X](X\rightarrow X) \equiv A\rightarrow A$. As a simple but instructive exercise, the reader may want to build the derivation for $id(\forall X.X\rightarrow X)(id)$.

As extensions of F_2 we could adopt all the first-order constructions that we already discussed for F_1. A more interesting extension to consider is *existentially quantified* types, also known as type abstractions:

Table 26. Existential types

(Type Exists) (Val Pack)

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \exists X.A} \qquad \frac{\Gamma \vdash [B/X]M : [B/X]A}{\Gamma \vdash (pack_{\exists X.A}\ X{=}B\ with\ M) : \exists X.A}$$

(Val Open)

$$\frac{\Gamma \vdash M : \exists X.A \qquad \Gamma, X, x{:}A \vdash N : B \qquad \Gamma \vdash B}{\Gamma \vdash (open_B\ M\ as\ X, x{:}A\ in\ N) : B}$$

To illustrate the use of existentials, we consider an **abstract type** for booleans. As we said earlier, booleans can be represented as the type *Unit+Unit*. We can now show how to hide this representation detail from a client who does not care how booleans are implemented, but who wants to make use of *true*, *false* and *cond* (conditional). We first define an interface for such a client to use,

$$BoolInterface \triangleq \exists Bool.\ Record(true{:}\ Bool, false{:}\ Bool, cond{:}\ \forall Y.\ Bool\rightarrow Y\rightarrow Y\rightarrow Y)$$

This interface declares that there exists a type *Bool* (without revealing its identity) that supports the operations *true*, *false* and *cond* of appropriate types. The conditional is parameterized with respect to its result type Y, which may vary depending of the context of usage.

Next we define a particular implementation of this interface; one that represents *Bool* as *Unit+Unit*, and that implements the conditional via a case statement. The boolean representation type and the related boolean operations are packaged together by the *pack* construct.

$$\begin{array}{l}
\mathit{boolModule} : \mathit{BoolInterface} \ \triangleq \\
\quad \mathit{pack}_{\mathit{BoolInterface}}\ \mathit{Bool}{=}\mathit{Unit}{+}\mathit{Unit} \\
\quad \mathit{with\ record}(\\
\qquad\qquad \mathit{true} = \mathit{inLeft}(\mathit{unit}), \\
\qquad\qquad \mathit{false} = \mathit{inRight}(\mathit{unit}), \\
\qquad\qquad \mathit{cond} = \lambda Y.\ \lambda x{:}\mathit{Bool}.\ \lambda y_1{:}Y.\ \lambda y_2{:}Y. \\
\qquad\qquad\qquad \mathit{case}_Y\ x\ \mathit{of}\ x_1{:}\mathit{Unit}\ \mathit{then}\ y_1 \mid x_2{:}\mathit{Unit}\ \mathit{then}\ y_2)
\end{array}$$

Finally, a client could make use of this module by opening it, and thus getting access to an abstract name *Bool* for the boolean type, and a name *boolOp* for the record of boolean operations. These names are used in the next example for a simple computation that returns a natural number. (The computation following *in* is, essentially, *if boolOp.true then* 1 *else* 0.)

$$\begin{array}{l}
\mathit{open}_{\mathit{Nat}}\ \mathit{boolModule} \\
\mathit{as\ Bool},\ \mathit{boolOp}{:}\mathit{Record}(\mathit{true}{:}\ \mathit{Bool}, \mathit{false}{:}\ \mathit{Bool}, \mathit{cond}{:}\ \forall Y.\ \mathit{Bool}{\to}Y{\to}Y{\to}Y) \\
\mathit{in}\ \mathit{boolOp}.\mathit{cond}(\mathit{Nat})(\mathit{boolOp}.\mathit{true})(1)(0)
\end{array}$$

The reader should verify that these examples typecheck according to the rules previously given. Note the critical third assumption of (Val Open) that forbids writing, for example, *boolOp.true* as the body of *open* in the preceeding example. Because of that assumption, the abstract name of the representation type (*Bool*) cannot escape the scope of *open*, and therefore values having the representation type cannot escape either. A restriction of this kind is necessary, otherwise the representation type might become known to clients.

6 Subtyping

Typed object-oriented languages have particularly interesting and complex type systems. There is little consensus about what characterizes these languages, but at least one feature is almost universally present: **subtyping**. Subtyping captures the intuitive notion of inclusion between types, where types are seen as collections of values. An element of a type can be considered also as an element of any of its supertypes, thus allowing a value (object) to be used flexibly in many different typed contexts.

When considering a subtyping relation, such as the one found in object-oriented programming languages, it is customary to add a new judgment $\Gamma \vdash A <: B$ stating that A is a subtype of B. The intuition is that any element of A is an element of B or, more appropriately, any program of type A is also a program of type B.

One of the simplest type systems with subtyping is an extension of F_1 called $F_{1<:}$. The syntax of F_1 is unchanged, except for the addition of a type *Top* that is a supertype of all types. The existing type rules are also unchanged. The subtyping judgment is independently axiomatized, and a single type rule, called **subsumption**, is added to connect the typing judgment to the subtyping judgment.

Table 27. Judgments for type systems with subtyping

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash A <: B$	A is a subtype of B in Γ
$\Gamma \vdash M : A$	M is a well-formed term of type A in Γ

The subsumption rule states that if a term has type *A*, and *A* is a subtype of *B*, then the term also has type *B*. That is, subtyping behaves very much like set inclusion, when type membership is seen as set membership.

The subtyping relation in Table 28 is defined as a reflexive and transitive relation with a maximal element called *Top*, which is therefore interpreted as the type of all well typed terms.

The subtype relation for function types says that $A \rightarrow B$ is a subtype of $A' \rightarrow B'$ if A' is a subtype of *A*, and *B* is a subtype of B'. Note that the inclusion is inverted (**contravariant**) for function arguments, while it goes in the same direction (**covariant**) for function results. Simple-minded reasoning reveals that this is the only sensible rule. A function *M* of type $A \rightarrow B$ accepts elements of type *A*; obviously it also accepts elements of any subtype A' of *A*. The same function *M* returns elements of type *B*; obviously it returns elements that belong to any supertype B' of *B*. Therefore, any function *M* of type $A \rightarrow B$, by virtue of accepting arguments of type A' and returning results of type B', has also type $A' \rightarrow B'$. The latter is compatible with saying that $A \rightarrow B$ is a subtype of $A' \rightarrow B'$.

In general, we say that a type variable occurs contravariantly within another type of F_1, if it always occurs on the left of an odd number of arrows (double contravariance equals covariance). For example, $X \rightarrow Unit$ and $(Unit \rightarrow X) \rightarrow Unit$ are contravariant in *X*, whereas $Unit \rightarrow X$ and $(X \rightarrow Unit) \rightarrow X$ are covariant in *X*.

Table 28. Additional rules for $F_{1<:}$

$$\text{(Sub Refl)}\quad \frac{\Gamma \vdash A}{\Gamma \vdash A <: A} \qquad \text{(Sub Trans)}\quad \frac{\Gamma \vdash A <: B \qquad \Gamma \vdash B <: C}{\Gamma \vdash A <: C} \qquad \text{(Val Subsumption)}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A <: B}{\Gamma \vdash a : B}$$

$$\text{(Type Top)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Top} \qquad \text{(Sub Top)}\quad \frac{\Gamma \vdash A}{\Gamma \vdash A <: Top} \qquad \text{(Sub Arrow)}\quad \frac{\Gamma \vdash A' <: A \qquad \Gamma \vdash B <: B'}{\Gamma \vdash A \rightarrow B <: A' \rightarrow B'}$$

Ad hoc subtyping rules can be added on basic types, such as $Nat <: Int$ [19].

All of the structured types we considered as extensions of F_1 admit simple subtyping rules; therefore, these structured types can be added to $F_{1<:}$ as well (Table 29). Typically, we need to add a single subtyping rule for each type constructor, taking care that the subtyping rule is sound in conjunction with subsumption. The subtyping rules for products and unions work componentwise. The subtyping rules for records and variants operate also lengthwise: a longer record type is a subtype of a shorter record type (additional fields can be forgotten by subtyping), whereas a shorter variant type is a subtype of a longer variant type (additional cases can be introduced by subtyping). For example,

$$\begin{array}{l}
\mathit{WorkingAge} \;\triangleq\; \mathit{Variant}(\mathit{student}{:}\ \mathit{Unit},\ \mathit{adult}{:}\ \mathit{Unit}) \\
\mathit{Age} \;\triangleq\; \mathit{Variant}(\mathit{child}{:}\ \mathit{Unit},\ \mathit{student}{:}\ \mathit{Unit},\ \mathit{adult}{:}\ \mathit{Unit},\ \mathit{senior}{:}\ \mathit{Unit}) \\
\mathit{Worker} \;\triangleq\; \mathit{Record}(\mathit{name}{:}\ \mathit{String},\ \mathit{age}{:}\ \mathit{WorkingAge},\ \mathit{profession}{:}\ \mathit{String}) \\
\mathit{Person} \;\triangleq\; \mathit{Record}(\mathit{name}{:}\ \mathit{String},\ \mathit{age}{:}\ \mathit{Age})
\end{array}$$

Then,

$$\begin{array}{l}
\mathit{WorkingAge} <: \mathit{Age} \\
\mathit{Worker} <: \mathit{Person}
\end{array}$$

Reference types do not have any subtyping rule: $\mathit{Ref}(A) <: \mathit{Ref}(B)$ holds only if $A{=}B$ (in which case $\mathit{Ref}(A) <: \mathit{Ref}(B)$ follows from reflexivity). This strict rule is necessary because references can be both read and written, and hence behave both covariantly and contravariantly. For the same reason, array types have no additional subtyping rules.

Table 29. Additional rules for extensions of $F_{1<:}$

(Sub Product)

$$\frac{\Gamma \vdash A_1 <: B_1 \qquad \Gamma \vdash A_2 <: B_2}{\Gamma \vdash A_1{\times}A_2 <: B_1{\times}B_2}$$

(Sub Union)

$$\frac{\Gamma \vdash A_1 <: B_1 \qquad \Gamma \vdash A_2 <: B_2}{\Gamma \vdash A_1{+}A_2 <: B_1{+}B_2}$$

(Sub Record) (l_i distinct)

$$\frac{\Gamma \vdash A_1 <: B_1 \;\;\ldots\;\; \Gamma \vdash A_n <: B_n \qquad \Gamma \vdash A_{n+1} \;\;\ldots\;\; \Gamma \vdash A_{n+m}}{\Gamma \vdash \mathit{Record}(l_1{:}A_1, \ldots, l_{n+m}{:}A_{n+m}) <: \mathit{Record}(l_1{:}B_1, \ldots, l_n{:}B_n)}$$

(Sub Variant) (l_i distinct)

$$\frac{\Gamma \vdash A_1 <: B_1 \;\;\ldots\;\; \Gamma \vdash A_n <: B_n \qquad \Gamma \vdash B_{n+1} \;\;\ldots\;\; \Gamma \vdash B_{n+m}}{\Gamma \vdash \mathit{Variant}(l_1{:}A_1, \ldots, l_n{:}A_n) <: \mathit{Variant}(l_1{:}B_1, \ldots, l_{n+m}{:}B_{n+m})}$$

As was the case for F_1, a change to the structure of environments is necessary when considering recursive types. This time, we must add *bounded variables* to environments (Table 30). Variables bound by *Top* correspond to our old unconstrained variables. The soundness of the subtyping rule (Sub Rec) for recursive types (Table 31) is not obvious, but the intuition is fairly straightforward. To check whether $\mu X.A <: \mu Y.B$ we assume $X{<:}Y$ and we check $A <: B$; the assumption helps us when finding matching occurrences of X and Y in A and B, as long as they are in covariant contexts. A simpler rule asserts that $\mu X.A <: \mu X.B$ whenever $A <: B$ for any X, but this rule is unsound when X occurs in contravariant contexts (e.g., immediately on the left of an arrow).

Table 30. Environments with bounded variables

(Env X<:)	(Type X<:)	(Sub X<:)
$\dfrac{\Gamma \vdash A \qquad X \notin dom(\Gamma)}{\Gamma, X{<:}A \vdash \diamond}$	$\dfrac{\Gamma', X{<:}A, \Gamma'' \vdash \diamond}{\Gamma', X{<:}A, \Gamma'' \vdash X}$	$\dfrac{\Gamma', X{<:}A, \Gamma'' \vdash \diamond}{\Gamma', X{<:}A, \Gamma'' \vdash X{<:}A}$

Table 31. Subtyping recursive types

(Type Rec)	(Sub Rec)
$\dfrac{\Gamma, X{<:}Top \vdash A}{\Gamma \vdash \mu X.A}$	$\dfrac{\Gamma \vdash \mu X.A \qquad \Gamma \vdash \mu Y.B \qquad \Gamma, Y{<:}Top, X{<:}Y \vdash A <: B}{\Gamma \vdash \mu X.A <: \mu Y.B}$

The bounded variables in environments are also the basis for the extension of F_2 with subtyping, which gives a system called $F_{2<:}$. In this system the term $\lambda X{<:}A.M$ indicates a program M parameterized with respect to a type variable X that stands for an arbitrary subtype of A. This is a generalization of F_2, since the F_2 term $\lambda X.M$ can be represented as $\lambda X{<:}Top.M$. Corresponding to the terms $\lambda X{<:}A.M$, we have bounded type quantifiers of the form $\forall X{<:}A.B$.

Table 32. Syntax of $F_{2<:}$

A,B ::=	types
X	type variable
Top	the biggest type
$A \to B$	function type
$\forall X{<:}A.B$	bounded universally quantified type
M,N ::=	terms
x	variable
$\lambda x{:}A.M$	function
$M\ N$	application
$\lambda X{<:}A.M$	bounded polymorphic abstraction
$M\ A$	type instantiation

Scoping for $F_{2<:}$ types and terms is defined similarly to F_2, except that $\forall X{<:}A.B$ binds X in B but not in A, and $\lambda X{<:}A.M$ binds X in M but not in A.

The type rules for $F_{2<:}$ consist of most of the type rules for $F_{1<:}$ (namely, (Env ϕ), (Env x), (Type Top), (Type Arrow), (Sub Refl), (Sub Trans), (Sub Top), (Sub Arrow), (Val Subsumption), (Val x), (Val Fun), and (Val Appl)), plus the rules for bounded variables (namely, (Env X<:), (Type X<:), and (Sub X<:)), and the ones listed in Table 33 for bounded polymorphism.

Table 33. Rules for bounded universal quantifiers

(Type Forall<:)

$$\frac{\Gamma, X{<:}A \vdash B}{\Gamma \vdash \forall X{<:}A.B}$$

(Sub Forall<:)

$$\frac{\Gamma \vdash A' <: A \qquad \Gamma, X{<:}A' \vdash B <: B'}{\Gamma \vdash (\forall X{<:}A.B) <: (\forall X{<:}A'.B')}$$

(Val Fun2<:)

$$\frac{\Gamma, X{<:}A \vdash M : B}{\Gamma \vdash \lambda X{<:}A.M : \forall X{<:}A.B}$$

(Val Appl2<:)

$$\frac{\Gamma \vdash M : \forall X{<:}A.B \qquad \Gamma \vdash A' <: A}{\Gamma \vdash M\,A' : [A'/X]B}$$

As for F_2, we do not need to add other type constructions to $F_{2<:}$, since all of the common ones can be expressed within it (except for recursion). Moreover, it turns out that the encodings used for F_2 satisfy the expected subtyping rules. For example, it is possible to encode bounded existential types so that the rules described in Table 34 are satisfied. The type $\exists X{<:}A.B$ represents a *partially abstract type*, whose representation type X is not completely known, but is known to be a subtype of A. This kind of partial abstraction occurs in some languages based on subtyping (e.g., in Modula-3).

Table 34. Rules for bounded existential quantifiers (derivable)

(Type Exists<:)

$$\frac{\Gamma, X{<:}A \vdash B}{\Gamma \vdash \exists X{<:}A.B}$$

(Sub Exists<:)

$$\frac{\Gamma \vdash A <: A' \qquad \Gamma, X{<:}A \vdash B <: B'}{\Gamma \vdash (\exists X{<:}A.B) <: (\exists X{<:}A'.B')}$$

(Val Pack<:)

$$\frac{\Gamma \vdash C <: A \qquad \Gamma \vdash [C/X]M : [C/X]B}{\Gamma \vdash (\mathit{pack}_{\exists X<:A.B}\ X{<:}A{=}C\ \mathit{with}\ M) : \exists X{<:}A.B}$$

(Val Open<:)

$$\frac{\Gamma \vdash M : \exists X{<:}A.B \qquad \Gamma \vdash D \qquad \Gamma, X{<:}A, x{:}B \vdash N : D}{\Gamma \vdash (\mathit{open}_D\ M\ \mathit{as}\ X{<:}A, x{:}B\ \mathit{in}\ N) : D}$$

Some nontrivial work is needed to obtain encodings of record and variant types in $F_{2<:}$ that satisfy the expected subtyping rules, but even those can be found [6].

7 Equivalence

For simplicity, we have avoided describing certain judgments that are necessary when type systems become complex and when one wishes to capture the semantics of programs in addition to their typing. We briefly discuss some of these judgments.

A *type equivalence* judgment, of the form $\Gamma \vdash A = B$, can be used when type equivalence is nontrivial and requires precise description. For example, some type systems identify a recursive type and its unfolding, in which case we would have $\Gamma \vdash \mu X.A =$

$[\mu X.A/X]A$ whenever $\Gamma \vdash \mu X.A$. As another example, type systems with type operators $\lambda X.A$ (functions from types to types) have a reduction rule for operator application of the form $\Gamma \vdash (\lambda X.A)\ B = [A/X]B$. The type equivalence judgment is usually employed in a *retyping rule* stating that if $\Gamma \vdash M : A$ and $\Gamma \vdash A = B$ then $\Gamma \vdash M : B$.

A *term equivalence* judgment determines which programs are equivalent with respect to a common type. It has the form $\Gamma \vdash M = N : A$. For example, with appropriate rules we could determine that $\Gamma \vdash 2{+}1 = 3 : \mathit{Int}$. The term equivalence judgment can be used to give a typed semantics to programs: if N is an irreducible expression, then we can consider N as the resulting value of the program M.

8 Type inference

Type inference is the problem of finding a type for a term within a given type system, if any type exists. In the type systems we have considered earlier, programs have abundant type annotations. Thus, the type inference problem often amounts to little more than checking the mutual consistency of the annotations. The problem is not always trivial but, as in the case of F_1, simple typechecking algorithms may exist.

A harder problem, called *typability* or **type reconstruction**, consists in starting with an untyped program M, and finding an environment Γ, a type-annotated version M' of M, and a type A such that A is a type for M' with respect to Γ. (A type-annotated program M' is simply one that stripped of all type annotations reduces back to M.) The type reconstruction problem for the untyped λ-calculus is solvable within F_1 by the Hindley-Milner algorithm used in ML [17]; in addition, that algorithm has the property of producing a unique representation of all possible F_1 typings of a λ-term. The type reconstruction problem for the untyped λ-calculus, however, is not solvable within F_2 [31]. Type reconstruction within systems with subtyping is still largely an open problem, although special solutions are beginning to emerge [1, 10, 13, 24].

We concentrate here on the type inference algorithms for some representative systems: F_1, F_2, and $F_{2<:}$. The first two systems have the unique type property: if a term has a type it has only one type. In $F_{2<:}$ there are no unique types, simply because the subsumption rule assigns all of the supertypes of a type to any term that has that type. However, a minimum type property holds: if a term has a collection of types, that collection has a least element in the subtype order [8]. The minimum type property holds for many common extensions of $F_{2<:}$ and of $F_{1<:}$ but may fail in the presence of ad-hoc subtypings on basic types.

The type inference problem

In a given type system, given an environment Γ and a term M is there a type A such that $\Gamma \vdash M : A$ is valid? The following are examples:

- In F_1, given $M \equiv \lambda x{:}K.x$ and any well-formed Γ we have that $\Gamma \vdash M : K{\to}K$.
- In F_1, given $M \equiv \lambda x{:}K.y(x)$ and $\Gamma \equiv \Gamma', y{:}K{\to}K$ we have that $\Gamma \vdash M : K{\to}K$.
- In F_1, there is no typing for $\lambda x{:}B.x(x)$, for any type B.

- However, in $F_{1<:}$ there is the typing $\Gamma \vdash \lambda x{:}Top{\to}B.x(x) : (Top{\to}B){\to}B$, for any type B, since x can also be given type Top.
- Moreover, in F_1 with recursive types, there is the typing $\Gamma \vdash \lambda x{:}B.(unfold_B\ x)(x) : B{\to}B$, for $B \equiv \mu X.X{\to}X$, since $unfold_B\ x$ has type $B{\to}B$.
- Finally, in F_2 there is the typing $\Gamma \vdash \lambda x{:}B.\ x(B)(x) : B{\to}B$, for $B \equiv \forall X.X{\to}X$, since $x(B)$ has type $B{\to}B$.

(An alternative formulation of the type inference problem requires Γ to be found, instead of given. However, in programming practice one is interested only in type inference for programs embedded in a complete programming context, where Γ is therefore given.)

We begin with the type inference algorithm for pure F_1, given in Table 35. The algorithm can be extended in straightforward ways to all of the first-order type structures studied earlier. This is the basis of the typechecking algorithms used in Pascal and all similar procedural languages.

The main routine $Type(\Gamma, M)$, takes an environment Γ and a term M and produces the unique type of M, if any. The instruction *fail* causes a global failure of the algorithm: it indicates a typing error. In this algorithm, as in the ones that follow, we assume that the initial environment parameter Γ is well formed so as to rule out the possibility of feeding invalid environments to internal calls. (For example, we may start with the empty environment when checking a full program.) In any case, it is easy to write a subroutine that checks the well formedness of an environment, from the code we provide. The case for $\lambda x{:}A.M$ should have a restriction requiring that $x \notin dom(\Gamma)$, since x is used to extend Γ. However, this restriction can be easily sidestepped by renaming, e.g., by making all binders unique before running the algorithm. We omit this kind of restrictions from Tables 35, 36, and 37.

Table 35. Type inference algorithm for F_1

```
$Type(\Gamma, x) \triangleq$
    $if\ x{:}A \in \Gamma\ for\ some\ A\ then\ A\ else\ fail$
$Type(\Gamma, \lambda x{:}A.M) \triangleq$
    $A{\to}Type((\Gamma, x{:}A), M)$
$Type(\Gamma, M\ N) \triangleq$
    $if\ Type(\Gamma, M) \equiv Type(\Gamma, N){\to}B\ for\ some\ B\ then\ B\ else\ fail$
```

As an example, let us consider the type inference problem for term $\lambda z{:}K.y(z)$ in the environment $\emptyset, y{:}K{\to}K$, for which we gave a full F_1 derivation in section 3. The algorithm proceeds as follows:

$$\begin{aligned}
&Type((\emptyset, y{:}K{\to}K), \lambda z{:}K.y(z))\\
&= K{\to}Type((\emptyset, y{:}K{\to}K, z{:}K), y(z))\\
&= K{\to}(if\ Type((\emptyset, y{:}K{\to}K, z{:}K), y) \equiv Type((\emptyset, y{:}K{\to}K, z{:}K), z){\to}B\ for\ some\ B\\
&\qquad\qquad then\ B\ else\ fail)
\end{aligned}$$

$$= K \to (\mathit{if}\ K \to K \equiv K \to B\ \mathit{for\ some}\ B\ \mathit{then}\ B\ \mathit{else\ fail}) \qquad \text{(taking } B \equiv K\text{)}$$
$$= K \to K$$

The type inference algorithm for F_2 (Table 36) is not much harder than the one for F_1, but it requires a subroutine *Good*(Γ, *A*) to verify that the types encountered in the source program are well formed. This check is necessary because types in F_2 contain type variables that might be unbound. A substitution subroutine must also be used in the type instantiation case, *M A*.

Table 36. Type inference algorithm for F_2

```
$Good(\Gamma, X) \triangleq X \in dom(\Gamma)$
$Good(\Gamma, A \to B) \triangleq Good(\Gamma, A)\ and\ Good(\Gamma, B)$
$Good(\Gamma, \forall X.\ A) \triangleq Good((\Gamma, X), A)$

$Type(\Gamma, x) \triangleq$
    $if\ x{:}A \in \Gamma\ for\ some\ A\ then\ A\ else\ fail$
$Type(\Gamma, \lambda x{:}A.M) \triangleq$
    $if\ Good(\Gamma, A)\ then\ A \to Type((\Gamma, x{:}A), M)\ else\ fail$
$Type(\Gamma, M\ N) \triangleq$
    $if\ Type(\Gamma, M) \equiv Type(\Gamma, N) \to B\ for\ some\ B\ then\ B\ else\ fail$
$Type(\Gamma, \lambda X.M) \triangleq$
    $\forall X.\ Type((\Gamma, X), M)$
$Type(\Gamma, M\ A) \triangleq$
    $if\ Type(\Gamma, M) \equiv \forall X.B\ for\ some\ X,B\ and\ Good(\Gamma, A)\ then\ [A/X]B\ else\ fail$
```

The type inference algorithm for $F_{2<:}$, given in Table 37, is more subtle. The subroutine *Subtype*(Γ, *A, B*) attempts to decide whether *A* is a subtype of *B* in Γ, and is at first sight straightforward. It has been shown, though, that *Subtype* is only a semialgorithm: it may diverge on certain pairs *A*,*B* that are not in subtype relation. That is, the typechecker for $F_{2<:}$ may diverge on ill typed programs, although it will still converge and produce a minimum type for well typed programs. More generally, there is no decision procedure for subtyping: the type system for $F_{2<:}$ is undecidable [25]. Several attempts have been made to cut $F_{2<:}$ down to a decidable subset; the simplest solution at the moment consists in requiring equal quantifiers bounds in (Sub Forall<:). In any case, the bad pairs *A*,*B* are extremely unlikely to arise in practice. The algorithm is still sound in the usual sense: if it finds a type, the program will not go wrong. The only troublesome case is in the subtyping of quantifiers; the restriction of the algorithm to $F_{1<:}$ is decidable and produces minimum types.

Table 37. Type inference algorithm for $F_{2<:}$

```
$Good(\Gamma, X) \triangleq X \in dom(\Gamma)$
$Good(\Gamma, Top) \triangleq true$
$Good(\Gamma, A \to B) \triangleq Good(\Gamma, A)\ and\ Good(\Gamma, B)$
$Good(\Gamma, \forall X{<:}A.\ B) \triangleq Good(\Gamma, A)\ and\ Good((\Gamma, X{<:}A), B)$
```

```
$Subtype(\Gamma, A, Top) \triangleq$ true
$Subtype(\Gamma, X, X) \triangleq$ true
$Subtype(\Gamma, X, A) \triangleq$                                          for $A \neq X, Top$
    if $X{<:}B \in \Gamma$ for some $B$ then $Subtype(\Gamma, B, A)$ else false
$Subtype(\Gamma, A \to B, A' \to B') \triangleq$
    $Subtype(\Gamma, A', A)$ and $Subtype(\Gamma, B, B')$
$Subtype(\Gamma, \forall X{<:}A.\ B, \forall X'{<:}A'.\ B') \triangleq$
    $Subtype(\Gamma, A', A)$ and $Subtype((\Gamma, X'{<:}A'), [X'/X]B, B')$
$Subtype(\Gamma, A, B) \triangleq$ false                                    otherwise

$Expose(\Gamma, X) \triangleq$ if $X{<:}A \in \Gamma$ for some $A$ then $Expose(\Gamma, A)$ else fail
$Expose(\Gamma, A) \triangleq A$                                            otherwise

$Type(\Gamma, x) \triangleq$
    if $x{:}A \in \Gamma$ for some $A$ then $A$ else fail
$Type(\Gamma, \lambda x{:}A.M) \triangleq$
    if $Good(\Gamma, A)$ then $A \to Type((\Gamma, x{:}A), M)$ else fail
$Type(\Gamma, M\ N) \triangleq$
    if $Expose(\Gamma, Type(\Gamma, M)) \equiv A \to B$ for some $A,B$
        and $Subtype(\Gamma, Type(\Gamma, N), A)$ then $B$ else fail
$Type(\Gamma, \lambda X{<:}A.M) \triangleq$
    if $Good(\Gamma, A)$ then $\forall X{<:}A.Type((\Gamma, X{<:}A), M)$ else fail
$Type(\Gamma, M\ A) \triangleq$
    if $Expose(\Gamma, Type(\Gamma, M)) \equiv \forall X{<:}A'.B$ for some $X,A',B$
        and $Good(\Gamma, A)$ and $Subtype(\Gamma, A, A')$ then $[A/X]B$ else fail
```

$F_{2<:}$ provides an interesting example of the anomalies one may encounter in type inference. The type inference algorithm given above is theoretically undecidable but is practically applicable. It is convergent and efficient on virtually all programs one may encounter; it diverges only on some ill typed programs, which should be rejected anyway. Therefore, $F_{2<:}$ sits close to the boundary between acceptable and unacceptable type systems, according to the criteria enunciated in the introduction.

9 Summary and Research Issues

What we learned

Natural questions for a beginner programmer are: What is an error? What is type safety? What is type soundness? (perhaps phrased, respectively, as Which errors will the computer tell me about? Why did my program crash? Why does the computer refuse to run my program?). The answers, even informal ones, are surprisingly intricate. We have paid particular attention to the distinction between type safety and type soundness, and we have reviewed the varieties of static 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 unrestricted 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 $X{\rightarrow}C$; we obtain $A{\rightarrow}C :> B{\rightarrow}C$. Thus $X{\rightarrow}C$ 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 $D{\rightarrow}X$; we obtain $D{\rightarrow}A <: D{\rightarrow}B$. Thus $D{\rightarrow}X$ 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). 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 typechecking 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*. 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**. *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, 186-195. 1994. *(To appear in Information and Computation.)*

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

[26] 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.

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

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

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

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

[31] 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.

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

[33] 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, 26] 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 Gunter's book [11], in Mitchell's article in the Handbook of Theoretical Computer Science [20], in Mitchell's book [21], and in the paper by Wright and Felleisen [33].

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 [32], Simula [3], CLU [16], and ML [18]. Reynolds gave type-theoretical explanations for polymorphism and data abstraction [26, 27]. (On that topic, see also [7, 22].) The book by Schmidt [28] 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 technique. 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 [30, 33]. Currently, Standard ML is the only widely used programming language with a formally specified type system [18].