1 *Introduction*

1.1 Types in Computer Science

Modern software engineering recognizes a broad range of *formal methods* for helping ensure that a system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior. On one end of the spectrum are powerful frameworks such as Hoare logic, algebraic specification languages, modal logics, and denotational semantics. These can be used to express very general correctness properties but are often cumbersome to use and demand a good deal of sophistication on the part of programmers. At the other end are techniques of much more modest power—modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underlying theories. One well-known instance of this sort of *lightweight formal methods* is *model checkers,* tools that search for errors in finite-state systems such as chip designs or communication protocols. Another that is growing in popularity is *run-time monitoring*, a collection of techniques that allow a system to detect, dynamically, when one of its components is not behaving according to specification. But by far the most popular and best established lightweight formal methods are *type systems*, the central focus of this book.

As with many terms shared by large communities, it is difficult to define "type system" in a way that covers its informal usage by programming language designers and implementors but is still specific enough to have any bite. One plausible definition is this:

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

A number of points deserve comment. First, this definition identifies type systems as tools for reasoning about *programs*. This wording reflects the

orientation of this book toward the type systems found in programming languages. More generally, the term *type systems* (or *type theory*) refers to a much broader field of study in logic, mathematics, and philosophy. Type systems in this sense were first formalized in the early 1900s as ways of avoiding the logical paradoxes, such as Russell's (Russell, 1902), that threatened the foundations of mathematics. During the twentieth century, types have become standard tools in logic, especially in proof theory (see Gandy, 1976 and Hindley, 1997), and have permeated the language of philosophy and science. Major landmarks in this area include Russell's original *ramified theory of types* (Whitehead and Russell, 1910), Ramsey's *simple theory of types* (1925)—the basis of Church's simply typed lambda-calculus (1940)—Martin-Löf's *constructive type theory* (1973, 1984), and Berardi, Terlouw, and Barendregt's *pure type systems* (Berardi, 1988; Terlouw, 1989; Barendregt, 1992).

Even within computer science, there are two major branches to the study of type systems. The more practical, which concerns applications to programming languages, is the main focus of this book. The more abstract focuses on connections between various "pure typed lambda-calculi" and varieties of logic, via the *Curry-Howard correspondence* (§9.4). Similar concepts, notations, and techniques are used by both communities, but with some important differences in orientation. For example, research on typed lambda-calculi is usually concerned with systems in which every well-typed computation is guaranteed to terminate, whereas most programming languages sacrifice this property for the sake of features like recursive function definitions.

Another important element in the above definition is its emphasis on *classification* of terms—syntactic phrases—according to the properties of the values that they will compute when executed. A type system can be regarded as calculating a kind of *static* approximation to the run-time behaviors of the terms in a program. (Moreover, the types assigned to terms are generally calculated *compositionally,* with the type of an expression depending only on the types of its subexpressions.)

The word "static" is sometimes added explicitly—we speak of a "statically typed programming language," for example—to distinguish the sorts of compile-time analyses we are considering here from the *dynamic* or *latent typing* found in languages such as Scheme (Sussman and Steele, 1975; Kelsey, Clinger, and Rees, 1998; Dybvig, 1996), where run-time *type tags* are used to distinguish different kinds of structures in the heap. Terms like "dynamically typed" are arguably misnomers and should probably be replaced by "dynamically checked," but the usage is standard.

Being static, type systems are necessarily also *conservative:* they can categorically prove the *absence* of some bad program behaviors, but they cannot prove their presence, and hence they must also sometimes reject programs

that actually behave well at run time. For example, a program like

```
if <complex test> then 5 else <type error>
```

will be rejected as ill-typed, even if it happens that the `<complex test>` will always evaluate to `true`, because a static analysis cannot determine that this is the case. The tension between conservativity and expressiveness is a fundamental fact of life in the design of type systems. The desire to allow more programs to be typed—by assigning more accurate types to their parts—is the main force driving research in the field.

A related point is that the relatively straightforward analyses embodied in most type systems are not capable of proscribing arbitrary undesired program behaviors; they can only guarantee that well-typed programs are free from *certain* kinds of misbehavior. For example, most type systems can check statically that the arguments to primitive arithmetic operations are always numbers, that the receiver object in a method invocation always provides the requested method, etc., but not that the second argument to the division operation is non-zero, or that array accesses are always within bounds.

The bad behaviors that can be eliminated by the type system in a given language are often called *run-time type errors*. It is important to keep in mind that this set of behaviors is a per-language choice: although there is substantial overlap between the behaviors considered to be run-time type errors in different languages, in principle each type system comes with a definition of the behaviors it aims to prevent. The *safety* (or *soundness*) of each type system must be judged with respect to its own set of run-time errors.

The sorts of bad behaviors detected by type analysis are not restricted to low-level faults like invoking non-existent methods: type systems are also used to enforce higher-level *modularity* properties and to protect the integrity of user-defined *abstractions*. Violations of information hiding, such as directly accessing the fields of a data value whose representation is supposed to be abstract, are run-time type errors in exactly the same way as, for example, treating an integer as a pointer and using it to crash the machine.

Typecheckers are typically built into compilers or linkers. This implies that they must be able to do their job *automatically,* with no manual intervention or interaction with the programmer—i.e., they must embody computationally *tractable* analyses. However, there is still plenty of room for requiring guidance from the programmer, in the form of explicit *type annotations* in programs. Usually, these annotations are kept fairly light, to make programs easier to write and read. But, in principle, a full proof that the program meets some arbitrary specification could be encoded in type annotations; in this case, the typechecker would effectively become a *proof* checker. Technologies like Extended Static Checking (Detlefs, Leino, Nelson, and Saxe, 1998)

are working to settle this territory between type systems and full-scale program verification methods, implementing fully automatic checks for some broad classes of correctness properties that rely only on "reasonably light" program annotations to guide their work.

By the same token, we are most interested in methods that are not just automatable in principle, but that actually come with *efficient* algorithms for checking types. However, exactly what counts as efficient is a matter of debate. Even widely used type systems like that of ML (Damas and Milner, 1982) may exhibit huge typechecking times in pathological cases (Henglein and Mairson, 1991). There are even languages with typechecking or type reconstruction problems that are undecidable, but for which algorithms are available that halt quickly "in most cases of practical interest" (e.g. Pierce and Turner, 2000; Nadathur and Miller, 1988; Pfenning, 1994).

1.2 What Type Systems Are Good For

Detecting Errors

The most obvious benefit of static typechecking is that it allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered much later, when the programmer is in the middle of something else—or even after the program has been deployed. Moreover, errors can often be pinpointed more accurately during typechecking than at run time, when their effects may not become visible until some time after things begin to go wrong.

In practice, static typechecking exposes a surprisingly broad range of errors. Programmers working in richly typed languages often remark that their programs tend to "just work" once they pass the typechecker, much more often than they feel they have a right to expect. One possible explanation for this is that not only trivial mental slips (e.g., forgetting to convert a string to a number before taking its square root), but also deeper conceptual errors (e.g., neglecting a boundary condition in a complex case analysis, or confusing units in a scientific calculation), will often manifest as inconsistencies at the level of types. The strength of this effect depends on the expressiveness of the type system and on the programming task in question: programs that manipulate a variety of data structures (e.g., symbol processing applications such as compilers) offer more purchase for the typechecker than programs involving just a few simple types, such as numerical calculations in scientific applications (though, even here, refined type systems supporting *dimension analysis* [Kennedy, 1994] can be quite useful).

Obtaining maximum benefit from the type system generally involves some

attention on the part of the programmer, as well as a willingness to make good use of the facilities provided by the language; for example, a complex program that encodes all its data structures as lists will not get as much help from the compiler as one that defines a different datatype or abstract type for each. Expressive type systems offer numerous "tricks" for encoding information about structure in terms of types.

For some sorts of programs, a typechecker can also be an invaluable *maintenance* tool. For example, a programmer who needs to change the definition of a complex data structure need not search by hand to find all the places in a large program where code involving this structure needs to be fixed. Once the declaration of the datatype has been changed, all of these sites become type-inconsistent, and they can be enumerated simply by running the compiler and examining the points where typechecking fails.

Abstraction

Another important way in which type systems support the programming process is by enforcing disciplined programming. In particular, in the context of large-scale software composition, type systems form the backbone of the *module languages* used to package and tie together the components of large systems. Types show up in the interfaces of modules (and related structures such as classes); indeed, an interface itself can be viewed as "the type of a module," providing a summary of the facilities provided by the module—a kind of partial contract between implementors and users.

Structuring large systems in terms of modules with clear interfaces leads to a more abstract style of design, where interfaces are designed and discussed independently from their eventual implementations. More abstract thinking about interfaces generally leads to better design.

Documentation

Types are also useful when *reading* programs. The type declarations in procedure headers and module interfaces constitute a form of documentation, giving useful hints about behavior. Moreover, unlike descriptions embedded in comments, this form of documentation cannot become outdated, since it is checked during every run of the compiler. This role of types is particularly important in module signatures.

Language Safety

The term "safe language" is, unfortunately, even more contentious than "type system." Although people generally feel they know one when they see it, their notions of exactly what constitutes language safety are strongly influenced by the language community to which they belong. Informally, though, safe languages can be defined as ones that make it impossible to shoot yourself in the foot while programming.

Refining this intuition a little, we could say that *a safe language is one that protects its own abstractions*. Every high-level language provides abstractions of machine services. Safety refers to the language's ability to guarantee the integrity of these abstractions and of higher-level abstractions introduced by the programmer using the definitional facilities of the language. For example, a language may provide arrays, with access and update operations, as an abstraction of the underlying memory. A programmer using this language then expects that an array can be changed only by using the update operation on it explicitly—and not, for example, by writing past the end of some other data structure. Similarly, one expects that lexically scoped variables can be accessed only from within their scopes, that the call stack truly behaves like a stack, etc. In a safe language, such abstractions can be used *abstractly;* in an unsafe language, they cannot: in order to completely understand how a program may (mis)behave, it is necessary to keep in mind all sorts of low-level details such as the layout of data structures in memory and the order in which they will be allocated by the compiler. In the limit, programs in unsafe languages may disrupt not only their own data structures but even those of the run-time system; the results in this case can be completely arbitrary.

Language safety is not the same thing as static type safety. Language safety can be *achieved* by static checking, but also by run-time checks that trap nonsensical operations just at the moment when they are attempted and stop the program or raise an exception. For example, Scheme is a safe language, even though it has no static type system.

Conversely, unsafe languages often provide "best effort" static type checkers that help programmers eliminate at least the most obvious sorts of slips, but such languages do not qualify as type-safe either, according to our definition, since they are generally not capable of offering any sort of *guarantees* that well-typed programs are well behaved—typecheckers for these languages can suggest the presence of run-time type errors (which is certainly better than nothing) but not prove their absence.

	Statically checked	Dynamically checked
Safe	ML, Haskell, Java, etc.	Lisp, Scheme, Perl, Postscript, etc.
Unsafe	C, C++, etc.	

The emptiness of the bottom-right entry in the preceding table is explained by the fact that, once facilities are in place for enforcing the safety of *most* operations at run time, there is little additional cost to checking *all* operations. (Actually, there are a few dynamically checked languages, e.g., some dialects of Basic for microcomputers with minimal operating systems, that do offer low-level primitives for reading and writing arbitrary memory locations, which can be misused to destroy the integrity of the run-time system.)

Run-time safety is not normally achievable by static typing alone. For example, *all* of the languages listed as safe in the table above actually perform *array-bounds checking* dynamically.[1] Similarly, statically checked languages sometimes choose to provide operations (e.g., the down-cast operator in Java—see §15.5) whose typechecking rules are actually unsound—language safety is obtained by checking each use of such a construct dynamically.

Language safety is seldom absolute. Safe languages often offer programmers "escape hatches," such as foreign function calls to code written in other, possibly unsafe, languages. Indeed, such escape hatches are sometimes provided in a controlled form within the language itself—`Obj.magic` in OCaml (Leroy, 2000), `Unsafe.cast` in the New Jersey implementation of Standard ML, etc. Modula-3 (Cardelli et al., 1989; Nelson, 1991) and C♯ (Wille, 2000) go yet further, offering an "unsafe sublanguage" intended for implementing low-level run-time facilities such as garbage collectors. The special features of this sublanguage may be used only in modules explicitly marked `unsafe`.

Cardelli (1996) articulates a somewhat different perspective on language safety, distinguishing between so-called *trapped* and *untrapped* run-time errors. A trapped error causes a computation to stop immediately (or to raise an exception that can be handled cleanly within the program), while untrapped errors may allow the computation to continue (at least for a while). An example of an untrapped error might be accessing data beyond the end of an array in a language like C. A safe language, in this view, is one that prevents untrapped errors at run time.

Yet another point of view focuses on portability; it can be expressed by the slogan, "A safe language is completely defined by its programmer's manual." Let the *definition* of a language be the set of things the programmer needs to understand in order to predict the behavior of every program in the language. Then the manual for a language like C does not constitute a definition, since the behavior of some programs (e.g., ones involving unchecked array

1. Static elimination of array-bounds checking is a long-standing goal for type system designers. In principle, the necessary mechanisms (based on *dependent types*—see §30.5) are well understood, but packaging them in a form that balances expressive power, predictability and tractability of typechecking, and complexity of program annotations remains a significant challenge. Some recent advances in the area are described by Xi and Pfenning (1998, 1999).

accesses or pointer arithmetic) cannot be predicted without knowing the details of how a particular C compiler lays out structures in memory, etc., and the same program may have quite different behaviors when executed by different compilers. By contrast, the manuals for Java, Scheme, and ML specify (with varying degrees of rigor) the exact behavior of all programs in the language. A well-typed program will yield the same results under any correct implementation of these languages.

Efficiency

The first type systems in computer science, beginning in the 1950s in languages such as Fortran (Backus, 1981), were introduced to improve the efficiency of numerical calculations by distinguishing between integer-valued arithmetic expressions and real-valued ones; this allowed the compiler to use different representations and generate appropriate machine instructions for primitive operations. In safe languages, further efficiency improvements are gained by eliminating many of the dynamic checks that would be needed to guarantee safety (by proving statically that they will always be satisfied). Today, most high-performance compilers rely heavily on information gathered by the typechecker during optimization and code-generation phases. Even compilers for languages without type systems *per se* work hard to recover approximations to this typing information.

Efficiency improvements relying on type information can come from some surprising places. For example, it has recently been shown that not only code generation decisions but also pointer representation in parallel scientific programs can be improved using the information generated by type analysis. The Titanium language (Yelick et al., 1998) uses type inference techniques to analyze the scopes of pointers and is able to make measurably better decisions on this basis than programmers explicitly hand-tuning their programs. The ML Kit Compiler uses a powerful *region inference* algorithm (Gifford, Jouvelot, Lucassen, and Sheldon, 1987; Jouvelot and Gifford, 1991; Talpin and Jouvelot, 1992; Tofte and Talpin, 1994, 1997; Tofte and Birkedal, 1998) to replace most (in some programs, all) of the need for garbage collection by stack-based memory management.

Further Applications

Beyond their traditional uses in programming and language design, type systems are now being applied in many more specific ways in computer science and related disciplines. We sketch just a few here.

An increasingly important application area for type systems is computer and network security. Static typing lies at the core of the security model of Java and of the JINI "plug and play" architecture for network devices (Arnold et al., 1999), for example, and is a critical enabling technology for Proof-Carrying Code (Necula and Lee, 1996, 1998; Necula, 1997). At the same time, many fundamental ideas developed in the security community are being re-explored in the context of programming languages, where they often appear as type analyses (e.g., Abadi, Banerjee, Heintze, and Riecke, 1999; Abadi, 1999; Leroy and Rouaix, 1998; etc.). Conversely, there is growing interest in applying programming language theory directly to problems in the security domain (e.g., Abadi, 1999; Sumii and Pierce, 2001).

Typechecking and inference algorithms can be found in many program analysis tools other than compilers. For example, AnnoDomini, a Year 2000 conversion utility for Cobol programs, is based on an ML-style type inference engine (Eidorff et al., 1999). Type inference techniques have also been used in tools for alias analysis (O'Callahan and Jackson, 1997) and exception analysis (Leroy and Pessaux, 2000).

In automated theorem proving, type systems—usually very powerful ones based on dependent types—are used to represent logical propositions and proofs. Several popular proof assistants, including Nuprl (Constable et al., 1986), Lego (Luo and Pollack, 1992; Pollack, 1994), Coq (Barras et al., 1997), and Alf (Magnusson and Nordström, 1994), are based directly on type theory. Constable (1998) and Pfenning (1999) discuss the history of these systems.

Interest in type systems is also on the increase in the database community, with the explosion of "web metadata" in the form of Document Type Definitions (XML 1998) and other kinds of schemas (such as the new XML-Schema standard [XS 2000]) for describing structured data in XML. New languages for querying and manipulating XML provide powerful static type systems based directly on these schema languages (Hosoya and Pierce, 2000; Hosoya, Vouillon, and Pierce, 2001; Hosoya and Pierce, 2001; Relax, 2000; Shields, 2001).

A quite different application of type systems appears in the field of computational linguistics, where typed lambda-calculi form the basis for formalisms such as *categorial grammar* (van Benthem, 1995; van Benthem and Meulen, 1997; Ranta, 1995; etc.).

1.3 Type Systems and Language Design

Retrofitting a type system onto a language not designed with typechecking in mind can be tricky; ideally, language design should go hand-in-hand with type system design.

One reason for this is that languages without type systems—even safe, dynamically checked languages—tend to offer features or encourage programming idioms that make typechecking difficult or infeasible. Indeed, in typed languages the type system itself is often taken as the foundation of the design and the organizing principle in light of which every other aspect of the design is considered.

Another factor is that the concrete syntax of typed languages tends to be more complicated than that of untyped languages, since type annotations must be taken into account. It is easier to do a good job of designing a clean and comprehensible syntax when all the issues can be addressed together.

The assertion that types should be an integral part of a programming language is separate from the question of where the programmer must physically write down type annotations and where they can instead be inferred by the compiler. A well-designed statically typed language will never require huge amounts of type information to be explicitly and tediously maintained by the programmer. There is some disagreement, though, about how much explicit type information is too much. The designers of languages in the ML family have worked hard to keep annotations to a bare minimum, using type inference methods to recover the necessary information. Languages in the C family, including Java, have chosen a somewhat more verbose style.

1.4 Capsule History

In computer science, the earliest type systems were used to make very simple distinctions between integer and floating point representations of numbers (e.g., in Fortran). In the late 1950s and early 1960s, this classification was extended to structured data (arrays of records, etc.) and higher-order functions. In the 1970s, a number of even richer concepts (parametric polymorphism, abstract data types, module systems, and subtyping) were introduced, and type systems emerged as a field in its own right. At the same time, computer scientists began to be aware of the connections between the type systems found in programming languages and those studied in mathematical logic, leading to a rich interplay that continues to the present.

Figure 1-1 presents a brief (and scandalously incomplete!) chronology of some high points in the history of type systems in computer science. Related developments in logic are included, in italics, to show the importance of this field's contributions. Citations in the right-hand column can be found in the bibliography.

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^ω	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)
1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

Figure 1-1: Capsule history of types in computer science and logic

1.5 Related Reading

While this book attempts to be self contained, it is far from comprehensive; the area is too large, and can be approached from too many angles, to do it justice in one book. This section lists a few other good entry points.

Handbook articles by Cardelli (1996) and Mitchell (1990b) offer quick introductions to the area. Barendregt's article (1992) is for the more mathematically inclined. Mitchell's massive textbook on *Foundations for Programming Languages* (1996) covers basic lambda-calculus, a range of type systems, and many aspects of semantics. The focus is on semantic rather than implementation issues. Reynolds's *Theories of Programming Languages* (1998b), a graduate-level survey of the theory of programming languages, includes beautiful expositions of polymorphism, subtyping, and intersection types. *The Structure of Typed Programming Languages,* by Schmidt (1994), develops core concepts of type systems in the context of language design, including several chapters on conventional imperative languages. Hindley's monograph *Basic Simple Type Theory* (1997) is a wonderful compendium of results about the simply typed lambda-calculus and closely related systems. Its coverage is deep rather than broad.

Abadi and Cardelli's *A Theory of Objects* (1996) develops much of the same material as the present book, de-emphasizing implementation aspects and concentrating instead on the application of these ideas in a foundation treatment of object-oriented programming. Kim Bruce's *Foundations of Object-Oriented Languages: Types and Semantics* (2002) covers similar ground. Introductory material on object-oriented type systems can also be found in Palsberg and Schwartzbach (1994) and Castagna (1997).

Semantic foundations for both untyped and typed languages are covered in depth in the textbooks of Gunter (1992), Winskel (1993), and Mitchell (1996). Operational semantics is also covered in detail by Hennessy (1990). Foundations for the semantics of types in the mathematical framework of *category theory* can also be found in many sources, including the books by Jacobs (1999), Asperti and Longo (1991), and Crole (1994); a brief primer can be found in *Basic Category Theory for Computer Scientists* (Pierce, 1991a).

Girard, Lafont, and Taylor's *Proofs and Types* (1989) treats logical aspects of type systems (the Curry-Howard correspondence, etc.). It also includes a description of System F from its creator, and an appendix introducing linear logic. Connections between types and logic are further explored in Pfenning's *Computation and Deduction* (2001). Thompson's *Type Theory and Functional Programming* (1991) and Turner's *Constructive Foundations for Functional Languages* (1991) focus on connections between functional programming (in the "pure functional programming" sense of Haskell or Miranda) and con-

structive type theory, viewed from a logical perspective. A number of relevant topics from proof theory are developed in Goubault-Larrecq and Mackie's *Proof Theory and Automated Deduction* (1997). The history of types in logic and philosophy is described in more detail in articles by Constable (1998), Wadler (2000), Huet (1990), and Pfenning (1999), in Laan's doctoral thesis (1997), and in books by Grattan-Guinness (2001) and Sommaruga (2000).

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. —*Luca Cardelli (1996)*

2 *Mathematical Preliminaries*

Before getting started, we need to establish some common notation and state a few basic mathematical facts. Most readers should just skim this chapter and refer back to it as necessary.

2.1 Sets, Relations, and Functions

2.1.1 DEFINITION: We use standard notation for sets: curly braces for listing the elements of a set explicitly ($\{\ldots\}$) or showing how to construct one set from another by "comprehension" ($\{x \in S \mid \ldots\}$), $\emptyset$ for the empty set, and $S \setminus \mathcal{T}$ for the set difference of S and $\mathcal{T}$ (the set of elements of S that are not also elements of $\mathcal{T}$). The size of a set S is written $|S|$. The powerset of S, i.e., the set of all the subsets of S, is written $\mathcal{P}(S)$. □

2.1.2 DEFINITION: The set $\{0, 1, 2, 3, 4, 5, \ldots\}$ of *natural numbers* is denoted by the symbol $\mathbb{N}$. A set is said to be *countable* if its elements can be placed in one-to-one correspondence with the natural numbers. □

2.1.3 DEFINITION: An n-place *relation* on a collection of sets S_1, S_2, ..., S_n is a set $R \subseteq S_1 \times S_2 \times \ldots \times S_n$ of tuples of elements from S_1 through S_n. We say that the elements $s_1 \in S_1$ through $s_n \in S_n$ are *related by* R if $(s_1, \ldots, s_n)$ is an element of R. □

2.1.4 DEFINITION: A one-place relation on a set S is called a *predicate* on S. We say that P is true of an element $s \in S$ if $s \in P$. To emphasize this intuition, we often write $P(s)$ instead of $s \in P$, regarding P as a function mapping elements of S to truth values. □

2.1.5 DEFINITION: A two-place relation R on sets S and $\mathcal{T}$ is called a *binary relation*. We often write $s\ R\ t$ instead of $(s, t) \in R$. When S and $\mathcal{T}$ are the same set $\mathcal{U}$, we say that R is a binary relation on $\mathcal{U}$. □

2.1.6 DEFINITION: For readability, three- or more place relations are often written using a "mixfix" concrete syntax, where the elements in the relation are separated by a sequence of symbols that jointly constitute the name of the relation. For example, for the typing relation for the simply typed lambda-calculus in Chapter 9, we write $\Gamma \vdash \mathtt{s} : \mathtt{T}$ to mean "the triple $(\Gamma, \mathtt{s}, \mathtt{T})$ is in the typing relation." □

2.1.7 DEFINITION: The *domain* of a relation R on sets S and T, written $dom(R)$, is the set of elements $s \in S$ such that $(s,t) \in R$ for some t. The *codomain* or *range* of R, written $range(R)$, is the set of elements $t \in T$ such that $(s,t) \in R$ for some s. □

2.1.8 DEFINITION: A relation R on sets S and T is called a *partial function* from S to T if, whenever $(s,t_1) \in R$ and $(s,t_2) \in R$, we have $t_1 = t_2$. If, in addition, $dom(R) = S$, then R is called a *total function* (or just *function*) from S to T. □

2.1.9 DEFINITION: A partial function R from S to T is said to be *defined* on an argument $s \in S$ if $s \in dom(R)$, and undefined otherwise. We write $f(x)\uparrow$, or $f(x) = \uparrow$, to mean "f is undefined on x," and $f(x)\downarrow$" to mean "f is defined on x."

In some of the implementation chapters, we will also need to define functions that may *fail* on some inputs (see, e.g., Figure 22-2). It is important to distinguish failure (which is a legitimate, observable result) from divergence; a function that may fail can be either partial (i.e., it may also diverge) or total (it must always return a result or explicitly fail)—indeed, we will often be interested in proving totality. We write $f(x) = \mathit{fail}$ when f returns a failure result on the input x.

Formally, a function from S to T that may also fail is actually a function from S to $T \cup \{\mathit{fail}\}$, where we assume that *fail* does not belong to T. □

2.1.10 DEFINITION: Suppose R is a binary relation on a set S and P is a predicate on S. We say that P is *preserved by* R if whenever we have $s\ R\ s'$ and $P(s)$, we also have $P(s')$. □

2.2 Ordered Sets

2.2.1 DEFINITION: A binary relation R on a set S is *reflexive* if R relates every element of S to itself—that is, $s\ R\ s$ (or $(s,s) \in R$) for all $s \in S$. R is *symmetric* if $s\ R\ t$ implies $t\ R\ s$, for all s and t in S. R is *transitive* if $s\ R\ t$ and $t\ R\ u$ together imply $s\ R\ u$. R is *antisymmetric* if $s\ R\ t$ and $t\ R\ s$ together imply that $s = t$. □

2.2.2 DEFINITION: A reflexive and transitive relation R on a set S is called a *preorder* on S. (When we speak of "a preordered set S," we always have in mind some particular preorder R on S.) Preorders are usually written using symbols like $\leq$ or $\sqsubseteq$. We write $s < t$ ("s is *strictly less than* t") to mean $s \leq t \wedge s \neq t$.

A preorder (on a set S) that is also antisymmetric is called a *partial order* on S. A partial order $\leq$ is called a *total order* if it also has the property that, for each s and t in S, either $s \leq t$ or $t \leq s$. □

2.2.3 DEFINITION: Suppose that $\leq$ is a partial order on a set S and s and t are elements of S. An element $j \in S$ is said to be a *join* (or *least upper bound*) of s and t if

1. $s \leq j$ and $t \leq j$, and
2. for any element $k \in S$ with $s \leq k$ and $t \leq k$, we have $j \leq k$.

Similarly, an element $m \in S$ is said to be a *meet* (or *greatest lower bound*) of s and t if

1. $m \leq s$ and $m \leq t$, and
2. for any element $n \in S$ with $n \leq s$ and $n \leq t$, we have $n \leq m$. □

2.2.4 DEFINITION: A reflexive, transitive, and symmetric relation on a set S is called an *equivalence* on S. □

2.2.5 DEFINITION: Suppose R is a binary relation on a set S. The *reflexive closure* of R is the smallest reflexive relation R' that contains R. ("Smallest" in the sense that if R'' is some other reflexive relation that contains all the pairs in R, then we have $R' \subseteq R''$.) Similarly, the *transitive closure* of R is the smallest transitive relation R' that contains R. The transitive closure of R is often written R^+. The *reflexive and transitive closure* of R is the smallest reflexive and transitive relation that contains R. It is often written R^*. □

2.2.6 EXERCISE [⋆⋆ ↛]: Suppose we are given a relation R on a set S. Define the relation R' as follows:

$$R' = R \cup \{(s,s) \mid s \in S\}.$$

That is, R' contains all the pairs in R plus all pairs of the form (s,s). Show that R' is the reflexive closure of R. □

2.2.7 EXERCISE [⋆⋆, ↛]: Here is a more constructive definition of the transitive closure of a relation R. First, we define the following sequence of sets of pairs:

$$\begin{array}{lcl} R_0 & = & R \\ R_{i+1} & = & R_i \cup \{(s,u) \mid \text{for some } t,\ (s,t) \in R_i \text{ and } (t,u) \in R_i\} \end{array}$$

That is, we construct each R_{i+1} by adding to R_i all the pairs that can be obtained by "one step of transitivity" from pairs already in R_i. Finally, define the relation R^+ as the union of all the R_i:

$$R^+ = \bigcup_i R_i$$

Show that this R^+ is really the transitive closure of R—i.e., that it satisfies the conditions given in Definition 2.2.5. □

2.2.8 EXERCISE [⋆⋆, ↛]: Suppose R is a binary relation on a set S and P is a predicate on S that is preserved by R. Show that P is also preserved by R^*. □

2.2.9 DEFINITION: Suppose we have a preorder $\leq$ on a set S. A *decreasing chain* in $\leq$ is a sequence $s_1, s_2, s_3, \ldots$ of elements of S such that each member of the sequence is strictly less than its predecessor: $s_{i+1} < s_i$ for every i. (Chains can be either finite or infinite, but we are more interested in infinite ones, as in the next definition.) □

2.2.10 DEFINITION: Suppose we have a set S with a preorder $\leq$. We say that $\leq$ is *well founded* if it contains no infinite decreasing chains. For example, the usual order on the natural numbers, with $0 < 1 < 2 < 3 < \ldots$ is well founded, but the same order on the integers, $\ldots < -3 < -2 < -1 < 0 < 1 < 2 < 3 < \ldots$ is not. We sometimes omit mentioning $\leq$ explicitly and simply speak of S as a *well-founded set*. □

2.3 Sequences

2.3.1 DEFINITION: A *sequence* is written by listing its elements, separated by commas. We use comma as both the "cons" operation for adding an element to either end of a sequence and as the "append" operation on sequences. For example, if a is the sequence $3, 2, 1$ and b is the sequence $5, 6$, then $0, a$ denotes the sequence $0, 3, 2, 1$, while $a, 0$ denotes $3, 2, 1, 0$ and b, a denotes $5, 6, 3, 2, 1$. (The use of comma for both "cons" and "append" operations leads to no confusion, as long as we do not need to talk about sequences of sequences.) The sequence of numbers from 1 to n is abbreviated $1..n$ (with just two dots). We write $|a|$ for the length of the sequence a. The empty sequence is written either as $\bullet$ or as a blank. One sequence is said to be a *permutation* of another if it contains exactly the same elements, possibly in a different order. □

2.4 Induction

Proofs by induction are ubiquitous in the theory of programming languages, as in most of computer science. Many of these proofs are based on one of the following principles.

2.4.1 AXIOM [PRINCIPLE OF ORDINARY INDUCTION ON NATURAL NUMBERS]: Suppose that P is a predicate on the natural numbers. Then:

If P(0)
and, for all i, $P(i)$ implies $P(i+1)$,
then $P(n)$ holds for all n. □

2.4.2 AXIOM [PRINCIPLE OF COMPLETE INDUCTION ON NATURAL NUMBERS]: Suppose that P is a predicate on the natural numbers. Then:

If, for each natural number n,
  given $P(i)$ for all $i < n$
  we can show $P(n)$,
then $P(n)$ holds for all n. □

2.4.3 DEFINITION: The *lexicographic order* (or "dictionary order") on pairs of natural numbers is defined as follows: $(m,n) \le (m',n')$ iff either $m < m'$ or else $m = m'$ and $n \le n'$. □

2.4.4 AXIOM [PRINCIPLE OF LEXICOGRAPHIC INDUCTION]: Suppose that P is a predicate on pairs of natural numbers.

If, for each pair (m,n) of natural numbers,
  given $P(m',n')$ for all $(m',n') < (m,n)$
  we can show $P(m,n)$,
then $P(m,n)$ holds for all m,n. □

The lexicograpic induction principle is the basis for proofs by *nested induction*, where some case of an inductive proof proceeds "by an inner induction." It can be generalized to lexicographic induction on triples of numbers, 4-tuples, etc. (Induction on pairs is fairly common; on triples it is occasionally useful; beyond triples it is rare.)

Theorem 3.3.4 in Chapter 3 will introduce yet another format for proofs by induction, called *structural induction*, that is particularly useful for proofs about tree structures such as terms or typing derivations. The mathematical foundations of inductive reasoning will be considered in more detail in Chapter 21, where we will see that all these specific induction principles are instances of a single deeper idea.

2.5 Background Reading

If the material summarized in this chapter is unfamiliar, you may want to start with some background reading. There are many sources for this, but Winskel's book (1993) is a particularly good choice for intuitions about induction. The beginning of Davey and Priestley (1990) has an excellent review of ordered sets. Halmos (1987) is a good introduction to basic set theory.

A proof is a repeatable experiment in persuasion. —*Jim Horning*