Should Your Specification Language Be Typed?

LESLIE LAMPORT
Compaq
and
LAWRENCE C. PAULSON
University of Cambridge

Most specification languages have a type system. Type systems are hard to get right, and getting them wrong can lead to inconsistencies. Set theory can serve as the basis for a specification language without types. This possibility, which has been widely overlooked, offers many advantages. Untyped set theory is simple and is more flexible than any simple typed formalism. Polymorphism, overloading, and subtyping can make a type system more powerful, but at the cost of increased complexity, and such refinements can never attain the flexibility of having no types at all. Typed formalisms have advantages too, stemming from the power of mechanical type checking. While types serve little purpose in hand proofs, they do help with mechanized proofs. In the absence of verification, type checking can catch errors in specifications. It may be possible to have the best of both worlds by adding typing annotations to an untyped specification language.

We consider only specification languages, not programming languages.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Verification

Additional Key Words and Phrases: Set theory, specification, types

Editors' introduction. We have invited the following paper for publication in TOPLAS in order to stimulate discussion of the topic it covers, which is the question of whether a typed specification language is preferable to an untyped one. This issue, like many similar ones about programming languages, is a matter of taste or experience. This does not mean that the issue is unimportant, only that the usual TOPLAS standards of proof are not easy to satisfy. Therefore this paper is not meant to approximate "peer-reviewed scientific truth," but was submitted to TOPLAS as a polemic, which Webster's defines as "an aggressive attack on or refutation of the opinions or principles of another." We hope it can promote a useful discussion.

Andrew W. Appel and Carl A. Gunter

Authors' addresses: L. Lamport, System Research Center, Compaq, 130 Lytton Avenue, Palo Alto, CA 94301, lamport@pa.dec.com; L. C. Paulson, University of Cambridge, Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, England, lcp@cl.cam.ac.uk.

1. INTRODUCTION

Types have become ubiquitous in computer science. The advantages of typed programming languages are obvious, so most computer scientists assume that they should also be used in languages and logics for specification and verification. Some computer scientists we have talked to even doubted that an untyped formalism can be sound. Types do more good than harm in a programming language: they let the compiler catch errors that would otherwise be found only after hours of debugging. Specification and verification are different from programming; we do not run specifications, and there is no convincing analogy between debugging and verification.

We begin with two examples illustrating that types are not as benign as they may seem to the unwary. We then try to help readers answer the question, should the specification language they use be typed?

As the first example, suppose that a program contains an array A of type $\textsc{Nat} \rightarrow \textsc{Nat}$ and two variables i and j of type $\textsc{Nat}$, where $\textsc{Nat}$ is the type of natural numbers. Consider the following question: is the postcondition $A[i-j] = A[i-j]$ true if the program terminates with $i - j < 0$ (so $A[i-j]$ is not type-correct)? "It's a run-time error" is not a meaningful answer, since our question is whether a mathematical formula is true, and formulas don't run. Indeed, the program might be error-free; we can ask this question even if the expression $A[i-j]$ does not appear in the program. In any conventional logic (including the one we outline below), the answer is clear—the formula $e = e$ is a tautology for any expression e. Now let us examine some popular books that use types and see how they answer this question. The books by Chandy and Misra [1988] and Manna and Pnueli [1991], despite their efforts to be rigorous, do not provide an answer. Gries and Schneider [1993] were more careful in the description of the typed logic in their book. Their explicit typing rules tell us that $A[i-j] = A[i-j]$ is not a legal expression. Unfortunately, those same rules tell us that $(i-j \geq 0) \Rightarrow (A[i-j] = A[i-j])$ is also an illegal expression. It would appear to be rather awkward to use their logic to reason about a program containing the statement **if** $i - j \geq 0$ **then** $A[i-j] := 0$. Few other books cope any better with the interactions between types and definedness. The book by Apt and Olderog [1990] avoids such problems by not allowing the type $\textsc{Nat}$; one has to use the type $\textsc{Int}$ of all integers. It also insists that all functions be total, even defining division by zero to yield zero. Apt and Olderog do not allow you to declare an array indexed by the set $\{0, \ldots, 99\}$, but they do allow you to write $x := 1/0$ in place of $x := 0$.

As the second example, consider an algorithm for computing the greatest common divisor (gcd) of (the initial values of) the variables m and n. The assertion that the result is the gcd of m and n will be expressed by some formula $F(m,n)$. In an untyped formalism such as untyped temporal logic or an untyped version of Dijkstra's *wp* calculus, correctness of the algorithm for all integers is expressed by the formula $(m, n \in \mathbf{Z}) \Rightarrow F(m,n)$, where $\mathbf{Z}$ is the set of integers. In a typed formalism, it is expressed by the validity of $F(m,n)$ when m and n are of type $\textsc{Int}$, an assertion we write $m, n : \textsc{Int} \vdash F(m,n)$. We would expect that an algorithm for computing the gcd of two integers also computes the gcd of two natural numbers. In an untyped formalism, this is the case because $(m, n \in \mathbf{Z}) \Rightarrow F(m,n)$ implies

$(m, n \in \mathbf{N}) \Rightarrow F(m, n)$, since the set $\mathbf{N}$ of natural numbers is a subset of $\mathbf{Z}$. However, in many typed formalisms, $m, n : \text{INT} \vdash F(m, n)$ does not necessarily imply $m, n : \text{NAT} \vdash F(m, n)$. For example, in a typed formulation of the *wp* calculus, $wp(n := n - 1, \text{TRUE})$ equals TRUE if n has type INT and equals $n > 0$ if n has type NAT [Gries 1981].[1] Thus, $n : \text{INT} \vdash wp(n := n - 1, \text{TRUE})$ is valid, but $n : \text{NAT} \vdash wp(n := n - 1, \text{TRUE})$ is not. We believe, that in formal versions of the logics of Chandy and Misra and of Manna and Pnueli, $m, n : \text{INT} \vdash F(m, n)$ will not imply $m, n : \text{NAT} \vdash F(m, n)$ for arbitrary F.

Many type systems have been proposed that handle the first example. We will describe the most popular and point out their costs. The problem posed by the second example seems to have gone unnoticed. Our raising of it has elicited the response that changing the type of a variable in a program changes the program, so there is nothing surprising about the example. But the example is about the specification of an abstract algorithm, not about programs. We expect any informal description of Euclid's algorithm that works for integers to work for natural numbers. It seems reasonable to expect the same of a formal description, but types force us to abandon common sense and think like a programmer.

An untyped formalism based on axiomatic set theory, the standard way of formalizing everyday mathematics, can provide a simple, powerful foundation for writing formal specifications. For readers not familiar with set theory, Section 2 describes such a formalism and explains how it avoids the potential inconsistencies of naive set theory. Readers already familiar with set theory may find this section perfectly obvious.

Some computer scientists are so used to thinking in terms of types that they find untyped set theory completely unnatural. To them, types express a natural classification of objects—a classification that should be enforced by the syntax. They feel that we should not be allowed to write a nonsensical formula like $2 \cap \mathbf{N}$. Some believe that integers and real numbers are completely distinct types, and it should make no sense to assert that the integer 2 equals the real number 2 [Huet 1997].

There is nothing inherently natural or unnatural about types or sets. There are mathematicians and computer scientists who find untyped set theory to be completely natural. To them, not being allowed to write $2 \cap \mathbf{N}$ is a confusion of syntax with semantics—like trying to redefine the grammar of English so that "Rocks are carnivores" is not a well-formed sentence. They are happy with the standard mathematical construction of the real numbers, in which the integers are identified with (declared to be) a subset of the reals. They find types to be an unnecessary and unnatural complication.

We eschew philosophical arguments about what is natural. We believe, that as Dana Scott once said, "Logic is an experimental science." For us, a formalism is a tool, not an end in itself. We are concerned here with *working* formalisms, those intended as a foundation for the varied and often quite large specifications that

[1]We are using Gries' semantic definition of *wp* from Chapter 7 [Gries 1981, page 108]. His rule for computing *wp* of an assignment statement in Definition 9.1.1 could be interpreted to mean that $wp(n := n - 1, \text{TRUE})$ equals either $n > 0$ or TRUE, when n has type NAT. The ambiguity arises because Gries gives no typing rules.

arise in industrial practice, where even a simplified, high-level formal specification of a system can be more than 50 pages. The choice of a working formalism should be based on pragmatism, not philosophy. Types should be used if and only if they help more than they hinder. We explain how they help and how they hinder, so readers can make a more informed choice of whether to use them. We also discuss the possibility of getting the best of both worlds by overlaying type systems atop a basic untyped formalism.

Section 3 describes the general classes of type systems and how they are used. It is followed by a discussion of the pros and cons of typed and typeless formalisms. From this discussion, we draw the following conclusions:

—If a specification language is to be general, it must be expressive. No simple type system is as expressive as untyped set theory. While a simple type system can allow many specifications to be written easily, it will make some impossible to write and others more complicated than they would be in set theory. The constructive type theories described in Section 3.6 may be expressive enough for writing just about any specification, but they are extremely complicated.

—Any error caught by type checking will be found easily when reasoning about a specification. However, large specifications are seldom verified, and type checking can catch errors in them that would otherwise go undetected. Moreover, mechanical theorem proving with a typed formalism may require less human intervention than with untyped set theory.

—Types serve little purpose in practice unless enforced by mechanical type checking.

These conclusions suggest the possibility of using untyped set theory, either by itself or in combination with a type system that poses additional well-formedness conditions on formulas. Different type systems could be used for different specifications, or even for different parts of the same specification. We believe that this approach merits further study.

While types can be helpful for tools that must deal with real applications, they serve little purpose in the kind of textbooks we have discussed, which rely exclusively on hand proofs. In such books, types either unnecessarily restrict the range of applications, or else add complications that are masked only by informal presentations that sweep them under the rug.

2. TYPES ARE NOT NECESSARY

Although specification languages may employ esoteric formalisms like temporal logic or process algebra, those formalisms are generally based on a more mundane assertion language. The formalism's type system, or lack thereof, comes from this underlying language. We now sketch an untyped language, based on ZF set theory, for specifying data structures and operations on them.[2] It is similar to the language mechanized by one of us using Isabelle [Paulson 1993; 1995].

[2]We prefer Zermelo-Fraenkel (ZF) set theory. However, for the purposes of this article, other axiom systems such as Bernays-Gödel (BG) would serve just as well. Implementors of theorem provers might prefer BG to ZF because BG has no axiom schemes.

2.1 Logic

Our language is based on first-order predicate logic with equality. We also use Hilbert's ε operator [Leisenring 1969], which we call **choose**. The expression **choose** $x . P(x)$ denotes an arbitrary value x that satisfies $P(x)$, if one exists; otherwise it denotes a completely arbitrary value. The **choose** operator satisfies the following axiom schemas ($\Rightarrow$ is implication, and $\equiv$ is the boolean operator *if and only if*).

$$\begin{array}{l}(\exists\, x . P(x)) \;\Rightarrow\; P(\textbf{choose}\; x . P(x)) \\ (\forall\, x . P(x) \equiv Q(x)) \;\Rightarrow\; (\textbf{choose}\; x . P(x)) = (\textbf{choose}\; x . Q(x))\end{array} \tag{1}$$

Although **choose** is seldom mentioned in logic texts, mathematicians implicitly use similar operators all the time. Assuming $x \neq 0$, a mathematician might define $1/x$ to be the unique number such that $x \cdot (1/x) = 1$. This can be expressed formally as

$$1/x \;\triangleq\; \textbf{choose}\; y . (y \in \mathbf{R}) \wedge (x \cdot y = 1)$$

where $\mathbf{R}$ is the set of real numbers. To write a specification, we define new operators in terms of the primitive ones provided by the formalism. We take the simple view that definitions are purely syntactic. For example, writing $F(x) \triangleq \exists\, y . G(x, y)$ makes $F(e)$ an abbreviation for $\exists\, y . G(e, y)$, for any expression e. An operator can be defined only in terms of primitive operators and operators that have already been defined. (Recursion is discussed below.) Thus, by replacing defined symbols with their definitions, any expression can be reduced to one containing only the primitive operators. A definition cannot introduce unsoundness, so we never have to prove a theorem in order to make a definition. Of course, we have to prove that the operators we define have the properties we want. For example, we define the **if**/**then**/**else** construct by

$$\textbf{if}\; p \;\textbf{then}\; e_1 \;\textbf{else}\; e_2 \;\triangleq\; \textbf{choose}\; x . (p \wedge (x = e_1)) \vee (\neg p \wedge (x = e_2)) \;.$$

From this definition and the axiom schemas (1), we can prove

$$(\textbf{if true then}\; e_1 \;\textbf{else}\; e_2) \;=\; e_1$$

Since **choose** is permitted in function definitions, the first axiom of (1) yields a strong form of the axiom of choice. One can use a more restricted **choose** operator with weaker axioms, but the unrestricted form is more convenient. There seems to be no practical reason to avoid the axiom of choice in a formalism for specifying and verifying computer systems.

2.2 Set Theory

Figure 1 describes a collection of operators from set theory that we have found valuable in writing specifications. Some of these operators are defined in terms of the others; the rest are primitive. We will not discuss the axioms of set theory. When writing and reasoning about specifications, it makes no difference which of these operators are taken to be primitive. We need only understand their meanings and know that two sets are equal if and only if they have the same elements.

Naive informal reasoning about sets can be unsound. It leads to many paradoxes [Whitehead and Russell 1962, pp. 60–65], the most famous being Russell's

$= \quad \neq \quad \in \quad \notin \quad \emptyset \quad \cup \quad \cap \quad \subseteq \quad \setminus$	[set difference]
$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : P(x)\}$	[Set of elements x in S satisfying $P(x)$]
$\{e(x) : x \in S\}$	[Set of elements $e(x)$ such that x in S]
$\mathcal{P}(S)$	[Set of subsets of S]
$\bigcup S$	[Union of all elements of S]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

Fig. 1. The operators of set theory.

$f[e]$	[Function application]
$\mathbf{dom}\, f$	[Domain of the function f]
$S \to T$	[Set of functions with domain S and range a subset of T]
$[x \in S \mapsto e(x)]$	[Function f such that $f[x] = e(x)$ for $x \in S$]

Fig. 2. Operators for expressing functions.

paradox of the set $\mathcal{R}$ of all sets that are not elements of themselves. This set satisfies $\mathcal{R} \in \mathcal{R}$ if and only if it satisfies $\mathcal{R} \notin \mathcal{R}$. In axiomatic set theory (such as ZF), paradoxes are avoided by preventing the creation of sets that are too big. The Russell set $\mathcal{R}$ might be written as the comprehension $\{x : x \notin x\}$, but ZF allows only comprehension over some previously constructed set S, as shown in Figure 1.[3]

2.3 Functions

A function is usually defined to be a set of ordered pairs. Formally, one can define the operator *Apply* by

$$Apply(f, x) \ \triangleq\ \mathbf{choose}\ y\,.\,\langle x, y\rangle \in f$$

and let $f(x)$ be an abbreviation for $Apply(f, x)$. But, it doesn't matter how functions are defined. We prefer simply to regard the four operators of Figure 2 as primitive, where we write $f[x]$ instead of the customary $f(x)$ to distinguish function application from operator application.[4] A function f has a domain, which is the set written $\mathbf{dom}\, f$. The set $S \to T$ consists of all functions f such that $\mathbf{dom}\, f = S$ and $f[x] \in T$ for all $x \in S$. The notation $[x \in S \mapsto e(x)]$ is used to describe a function explicitly. (We reserve the more familiar λ-notation for other purposes.) For example, $[r \in \mathbf{R}\setminus\{0\} \mapsto 1/r]$ is the reciprocal function *recip*, whose domain is the set $\mathbf{R}\setminus\{0\}$ of nonzero reals. We can define this function by

$$recip[r : \mathbf{R}\setminus\{0\}] \ \triangleq\ 1/r\ .$$

In general, $f[x : S] \triangleq e(x)$ defines f to equal $[x \in S \mapsto e(x)]$. We describe recursive function definitions in Section 2.5.

[3] S is considered to lie outside the scope of the bound variable x in the expression $\{x \in S : P(x)\}$, so $\{x \in \{x\} : x \notin x\}$ equals $\{y \in \{x\} : y \notin y\}$.

[4] We could use $f(x)$ for both; simple syntactic rules can determine which is meant.

2.4 Functions versus Operators

Functions are different from operators. A function f has a domain, and we define the value of $f[x]$ only for elements x in its domain. The expression *recip*[0], which is an abbreviation for *Apply*(*recip*, 0), is syntactically a term, so it denotes a value. However, we don't know what value. It need not equal 1/0. It need not even be a number. But, whatever its value, it must equal *recip*[2 − 2], since 2 − 2 equals 0. Functions are just like other values; for example, *recip* by itself is syntactically a term. We can quantify over sets of functions, writing expressions such as $\forall f \in (\mathbf{R} \rightarrow \mathbf{R}) \,.\, |f|_\infty \geq 0$.

Operators are different from functions. (Set theorists call them *class functions*.) Consider the operator $\bigcup$, where $\bigcup S$ is the union of all elements of S. We cannot define a function *union* so that *union*$[S]$ equals $\bigcup S$ for all sets S. The domain of *union* would have to be a set that contains all sets, and there is no such set. (If there were, we would encounter Russell's paradox.) The symbol $\bigcup$ by itself is not a term, so it does not denote a value.

Higher-order operators, which take operators as arguments, pose no problem. For example, we can define the operator **increasing** so that **increasing**(F) asserts that the operator F is increasing under the partial order $\subseteq$.

$$\mathbf{increasing}(F) \;\triangleq\; \forall A \,.\, A \subseteq F(A)$$

However, we do not allow quantification over operators, which would lead to a higher-order logic. The string $\exists U \,.\, \mathbf{R} \in U(\mathbf{R})$ is not syntactically well-formed, since we can write $\mathbf{R} \in U(\mathbf{R})$ only if U is an operator, and bound variables are terms, not operators. We could combine ZF with higher-order logic, but there is little reason to adopt such a complicated formalism. Because we can quantify over functions, we have not found quantification over operators to be necessary.

The distinction between operators and functions exists in ordinary mathematics. Mathematicians don't think of $\bigcup$ or $\in$ as functions. However, the distinction tends to go unnoticed—perhaps because ordinary mathematicians have no generic name for what we call operators.

2.5 Recursion

We allow recursive function definitions of the form

$$f[x : S] \;\triangleq\; e(x, f)\;. \tag{2}$$

For example, we can define the factorial function *fact* on natural numbers by

$$\mathit{fact}[n : \mathbf{N}] \;\triangleq\; \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \cdot \mathit{fact}[n{-}1]\;. \tag{3}$$

There are several ways to define (2); perhaps the simplest is to let it be an abbreviation for

$$f \;\triangleq\; \mathbf{choose}\ g \,.\, g = [x \in S \mapsto e(x, g)]\;.$$

One can also introduce constructs for defining sets recursively, as well as for defining least and greatest fixed points [Paulson 1994a]. They can all be translated to simple set-theoretic definitions. However, operators, unlike functions, cannot in general be defined recursively.

Of course, one can write silly recursive definitions—for example, replacing $n - 1$ by $n + 1$ in (3). To prove anything about a recursively defined function, we must prove that the recursion is well-founded. The well-foundedness of many recursive definitions is obvious enough to be verified automatically. For some definitions, the proof of well-foundedness may be difficult; the question may even be undecidable. Well-founded or not, a recursive function definition does define some value (though not necessarily a function). A definition can never introduce inconsistency.

2.6 What is 1/0?

Elementary school children and programmers are taught that 1/0 is meaningless, and they are committing an error by even writing it. In set theory, one can give a simple answer to the question of what 1/0 is: we don't know and we don't care.

Let us take 1/0 to be an abbreviation for *recip*[0], where *recip* is the reciprocal function defined in Section 2.3. Since 0 is not in the domain of *recip*, we know nothing about the value of 1/0; it might equal $\sqrt{2}$, it might equal $\mathbf{R}$, or it might equal anything else. We don't care what it equals. For example, consider

$$(x \in \mathbf{R}) \wedge (x \neq 0) \;\Rightarrow\; (x \cdot (1/x) = 1) \,. \tag{4}$$

This formula holds for all values of x. Substituting 0 for x yields the formula $\mathbf{false} \Rightarrow (0 \cdot (1/0) = 1)$, which equals **true** regardless of the value of 1/0, and regardless of whether or not $0 \cdot (1/0)$ equals 1. The subformula $x \cdot (1/x) = 1$ of (4) may or may not hold; we don't know what $0 \cdot (1/0)$ or $\mathbf{R} \cdot (1/\mathbf{R})$ equals, so we don't know whether or not they equal 1.

One drawback of the don't-care approach is that theorems such as $1/0 = 1/0$ can be proved about undefined quantities. This is avoided by more sophisticated approaches. We can introduce a formal notion of definedness and provide axioms for proving that terms are defined [Farmer 1990]. Domain theory goes even further, adding a *more-defined-than* relation between functions [Gunter and Scott 1990]. In our experience [Paulson 1985], the benefits of these approaches do not justify their complexity. Abstract Incorporated's LAMBDA system moved from a definedness logic [Scott 1979] to conventional higher-order logic for similar reasons.

2.7 Examples

The data structures and related operations found in programming and specification languages are easily represented in set theory. We show how to represent three of these structures: finite lists, records, and objects.

2.7.1 *Finite Lists.* We represent a finite list of length n as a function with domain $1 \,..\, n$, the set $\{i \in \mathbf{N} : 1 \leq i \leq n\}$ of natural numbers from 1 through n. The set $\mathit{List}(L)$ of all finite lists with elements in the set L is just equal to $\bigcup\{(1 \,..\, n) \rightarrow L \,:\, n \in \mathbf{N}\}$. The length $\mathit{Len}(s)$ of a finite list s is defined by

$$\mathit{Len}(s) \;\triangleq\; \mathbf{choose}\; n \,.\, (n \in \mathbf{N}) \wedge (\mathbf{dom}\; s = 1 \,..\, n) \,. \tag{5}$$

List and *Len* are operators; they cannot be functions. For them to be functions, their domains would have to consist of all sets and all finite lists, respectively, neither of which forms a set.

2.7.2 *Records.* We represent a record as a function whose domain is a finite set of strings. For example, the set of all records that consist of a *ptr* field that is a natural number and a *sq* field that is a list of natural numbers is

$$\{r \in (\{\text{“}\mathsf{ptr}\text{”}, \text{“}\mathsf{sq}\text{”}\} \rightarrow \mathbf{N} \cup \mathit{List}(\mathbf{N})) \;:\; r[\text{“}\mathsf{ptr}\text{”}] \in \mathbf{N} \wedge r[\text{“}\mathsf{sq}\text{”}] \in \mathit{List}(\mathbf{N})\} \;.$$

We let $r.\mathit{str}$ be an abbreviation for $r[\text{“}\mathsf{str}\text{”}]$, for any string str.

In set theory, one can define many useful operators on records that are not expressible in conventional programming languages. For example, suppose T is a "record type"—a set of records all having the same components—and r an arbitrary record. We can define the operator *Copy* so that $\mathit{Copy}(T, r)$ is a record t in T such that $t.c$ equals $r.c$ for any component c common to both t and r. We first define $\mathit{Any}(T)$ to equal **choose** $t \,.\, t \in T$ and then define $\mathit{Copy}(T, r)$ to equal

$$[c \in \textbf{dom}\ \mathit{Any}(T) \mapsto \textbf{if}\ c \in \textbf{dom}\ r\ \textbf{then}\ r[c]\ \ \textbf{else}\ \mathit{Any}(T)[c]] \;.$$

2.7.3 *Objects.* Objects and classes generalize the concept of records and record types. One can define the class C of all objects with a *cnt* field and the method $\mathit{add}1$ that, for any object o in this class, returns the object that is the same as o except with its *cnt* field incremented by 1.

To represent objects in set theory, we first define some elementary sets of values, including the set of all strings, the set of all integers, and any other desired sets of primitive data values. We next define (by transfinite recursion) a set U of values to be the smallest set containing all of these elementary sets such that if S and T are elements of U, then $S \rightarrow T$, $\mathcal{P}(S)$, and all the elements of S are also elements of U. The set O of objects is then the set of all elements in U that are records.

The class C of all objects with a *cnt* field is represented by the set $\{o \in O : \text{“}\mathsf{cnt}\text{”} \in \textbf{dom}\ o\}$, and the method $\mathit{add}1$ by the function with domain C such that $\mathit{add}1[o]$ equals the function

$$[\, s \in \textbf{dom}\ o \;\mapsto\; \textbf{if}\ s = \text{“}\mathsf{cnt}\text{”}\ \textbf{then}\ o[\text{“}\mathsf{cnt}\text{”}] + 1\ \textbf{else}\ o[s]\,]$$

for all $o \in C$. In general, a class is a set of objects, and a method is a function whose domain is a class. The set M of methods is the union of all sets $S \rightarrow U$ such that $S \subseteq O$. (Since classical mathematics has no notion of assignment, objects defined in this way resemble objects in a functional programming language rather than an imperative one.)

Some object-oriented languages allow methods to be associated with individual objects. Methods are not elements of U, so they cannot appear as fields of an object.[5] Thus, a field $o.m$ of an object o cannot equal $\mathit{add}1$. However, we can define a set N of method names with $N \subseteq U$ and a function μ in $N \rightarrow M$ such that $\mu[o.m]$ equals $\mathit{add}1$. For example, we could let N be a set of strings, define μ such that $\mu[\text{“}\mathsf{add1}\text{”}] = \mathit{add}1$, and let $o.m$ equal "add1". In principle, we could define N to be the set of strings in some language for describing methods, and define μ to be a "compiler" for that language. In practice, any specification will use only a small set of distinct methods, which can be assigned arbitrary names like "add1".

[5] Because the set $S \rightarrow S$ is always bigger than the set S, there is no way to construct a set of objects so that any method can be a field of some object.

3. TYPED FORMAL LANGUAGES

Types in programming languages are valuable but not essential. Two programming languages can be quite similar except for the use of types—for example, C and BCPL, or ML and Lisp. Programming languages use type checking to catch errors at compile time and to improve run-time efficiency. Declaring A to be an array indexed by $1 \mathinner{\ldotp\ldotp} 4$ allows the compiler to assign storage only for four array elements and to detect $A[\text{“two”}]$ to be an error.

Types in a typed logical formalism play an essential role. If we remove type checking, we almost certainly make the logic inconsistent and therefore useless. One reason for adopting a typed logic is indeed to catch errors, but type checking cannot just be disabled when it is inconvenient.

There is usually no obvious translation between the typed and the untyped worlds. Although basic types like $\textsc{Nat}$ and $\textsc{Int}$ can be identified with particular sets, more general types cannot.

One could consider ZF to be a typed formalism with the two basic types $\textsc{Set}$ and $\textsc{Bool}$, and its syntax could be formulated as typing rules. For example, $\cup$ would have type $\textsc{Set} \times \textsc{Set} \rightarrow \textsc{Set}$, and $\mathcal{P}$ would have type $\textsc{Set} \rightarrow \textsc{Set}$, making $S \cup \mathcal{P}$ illegal because it doesn't type check. But practically all expressions would have type $\textsc{Set}$. By a typed formalism, we mean one with many basic types, such as $\textsc{Nat}$ (natural numbers) and $\textsc{Real}$ (real numbers).

The most popular typed formalism is higher-order logic, also known as simple type theory. Versions of it have been mechanized in a number of proof assistants, including HOL [Gordon and Melham 1993], Isabelle/HOL [Paulson 1994b], and PVS [Owre et al. 1995]. We will focus on higher-order logic, but we will also outline alternatives to it.

3.1 Typed Set Theory

Whitehead and Russell invented types early in this century to prevent the paradoxes of naive set theory [Whitehead and Russell 1962]. Their work contains all the elements of modern higher-order logic. The key idea is that a set must have a different type from its elements. If S is a set, then it has a type of the form $\textsc{Set}(\tau)$; we may write $x \in S$ only if x has type τ. The formula $x \notin x$, which occurs in the definition of the Russell set $\mathcal{R}$, is illegal because x cannot simultaneously have types τ and $\textsc{Set}(\tau)$.

All elements of a set must have the same type. Many constructions used in untyped set theory violate this restriction. They mostly have the flavor of encodings. For example, in ZF one often defines the ordered pair $\langle a, b\rangle$ to be $\{\{a\}, \{a, b\}\}$, and the natural number n to be the set $\{0, \ldots, n-1\}$. Higher-order logic uses different definitions and can express much of mathematics easily. Occasionally, cumbersome constructions are needed to get around its type constraints.

3.2 Polymorphism

As programmers know, an unduly restrictive type system can make it hard to write perfectly reasonable expressions. Whitehead and Russell realized that a type discipline has to be flexible. The proof of a theorem like $x \in \{x\}$ must not depend on the type of x. They invented (in 1910!) the concept we now call *polymorphism*, which they called *typical ambiguity*. The premise of polymorphism is that we should not have to think about types that are irrelevant, and that most type constraints should be implicit. Whitehead and Russell never even bothered to invent a notation for types [Gödel 1983].

Polymorphism uses type variables α, β, ... as placeholders for irrelevant types. We can prove $x \in \{x\}$ where x has type α (so $\{x\}$ has type $\textsc{Set}(\alpha)$) and use the instances of this theorem obtained by replacing α with any type. The length operator *Len* of untyped set theory becomes a polymorphic function with type $\textsc{List}(\alpha) \rightarrow \textsc{Nat}$; we can think of *Len* as a collection of separate functions, one for each type α. Polymorphic equations like $Len(Reverse(L)) = Len(L)$ can be proved without specifying the type of L's elements.

A more interesting example involves the powerset operator, which has type $\textsc{Set}(\alpha) \rightarrow \textsc{Set}(\textsc{Set}(\alpha))$. (Recall that the simple type system for ZF gives powerset the type $\textsc{Set} \rightarrow \textsc{Set}$.) Polymorphism lets us write terms like $\mathcal{P}(\mathcal{P}(S))$ in which the operator appears with two different types. Most proof assistants for higher-order logic automatically type check such terms [Gordon and Melham 1993; Owre et al. 1995; Paulson 1994b].

3.3 Disjoint Sums and Data Types

Properly implemented, polymorphism lets us write specifications that hardly ever mention types. But the types are still there, and they constrain what we may write. In $A \cup B$, the sets A and B must have the same type. Sometimes this restriction is reasonable, but often it is not. If A is a set of apples and B a set of bananas, then it is unreasonable to prohibit the set $A \cup B$ of fruit. The standard way to write this set in higher-order logic is to define the new type FRUIT to be the disjoint union of the existing types APPLE and BANANA:

$$\textbf{datatype}\ \textsc{Fruit}\ \triangleq\ Apple\ \textsc{Apple}\ \mid\ Banana\ \textsc{Banana}$$

In addition to declaring the type FRUIT, this declaration introduces the constructor functions $Apple : \textsc{Apple} \rightarrow \textsc{Fruit}$ and $Banana : \textsc{Banana} \rightarrow \textsc{Fruit}$, as well as other functions for case analysis.

The function *Apple* maps from apples to fruit, but we also need to map from sets of apples to sets of fruits. For this purpose, we can use the image operator “, defined informally by

$$f\text{“}S\ \triangleq\ \{f(x) \mid x \in S\}.$$

(This definition is easily formalized in higher-order logic.) If A has type $\textsc{Set}(\textsc{Apple})$ and B has type $\textsc{Set}(\textsc{Banana})$, then $Apple\text{“}A \cup Banana\text{“}B$ has type $\textsc{Set}(\textsc{Fruit})$.

Data type declarations can be recursive. Here is the definition of lists in terms of two primitive functions, the empty list *Nil* and the constructor function *Cons* that takes arguments of types α and $\textsc{List}(\alpha)$:

$$\textbf{datatype}\ \textsc{List}(\alpha)\ \triangleq\ Nil\ \mid\ Cons(\alpha,\ \textsc{List}(\alpha))$$

Data type declarations can be reduced to the underlying logic [Melham 1989].

3.4 Sets in Higher-Order logic

The simple types of higher-order logic are too restricted a notion of collection to replace sets. To write practical specifications, we need set-theoretic operators such as union and set comprehension. We could add appropriate set theory axioms to typed first-order logic and then develop mathematics more or less as in ZF. However, it is more convenient to develop typed set theory within higher-order logic, adopting functions as a primitive concept instead of coding them as sets of pairs. This permits quantification over predicates, which are variables of type $\tau \rightarrow$ Bool.

A practical type system for higher-order logic should provide several ways of expressing types:

—**Type variables** α, β, γ, ... for polymorphism.

—**Basic types** such as Nat and Real, including the type Bool of logical formulas.

—**Type operators** including the operator $\rightarrow$ such that $\sigma \rightarrow \tau$ is the type of functions from type σ to type τ.

—**Data type declarations**.

Type checking is decidable, using the Hindley-Milner algorithm [Milner 1978]. The algorithm even infers the types of variables occurring in expressions. This kind of type system is used in the functional programming languages Haskell [Hudak et al. 1992] and ML [Paulson 1996], since one may write code that is not only polymorphic, but almost entirely free of type declarations. It works well in logic too.

Sets of elements of type τ are represented as predicates over type τ. We define $\text{Set}(\alpha)$ to be $\alpha \rightarrow \text{Bool}$ and make the following polymorphic definitions of set operations: comprehension $\{x \mid P(x)\}$ equals $\lambda x.P(x)$, $x \in S$ equals $S(x)$, and $\mathcal{P}$ and $\bigcup$ are defined by

$$\begin{array}{rcl} \mathcal{P}(S) & \triangleq & \{T \mid T \subseteq S\} \\ \bigcup S & \triangleq & \{x \mid \exists\, y \in S \,.\, x \in y\}\,. \end{array}$$

The operator $\bigcup$ has type $\text{Set}(\text{Set}(\alpha)) \rightarrow \text{Set}(\alpha)$. The other set-theoretic operators described in Section 2 have similar counterparts in higher-order logic.

In set theory, operators such as $\mathcal{P}$ are different from functions. In higher-order logic they are (polymorphic) functions; there is no need to distinguish between functions and operators.

Much of the discussion of sets in Section 2 carries over to their representation in higher-order logic. Higher-order logic traditionally includes the operator **choose**. It can adopt the same treatment of recursive functions and recursively defined sets. As in untyped set theory, we can let 1/0 have some unspecified value. Since 1/0 has type Real, its value is a real number and thus is not completely unspecified. In principle, this can be a problem—for example, it could allow us to prove the correctness of an algorithm that evaluates 1/0 during its execution. In practice, this is seldom an issue.

3.5 More Sophisticated Type Theories

The simple type theory we have just described is a starting point. We now consider some enhancements that have been added to try to create a more powerful working formalism.

3.5.1 *Subtyping.* In simple type theory, a term has at most one type. We can't consider a value of type NAT also to be of type INT; we can only define an injection $\iota : \text{NAT} \rightarrow \text{INT}$ that converts naturals into the corresponding integers. Addition of natural numbers and addition of integers are different operators with different types. (Overloading, discussed below, does allow us to use the same symbol "+" for both of them.)

An obvious extension to simple type theory is to allow one type to be a subtype of another. Naturals can be a subtype of integers, and text files a subtype of files. Then, $n : \text{NAT}$ implies $n : \text{INT}$, and we don't need the injection ι. But such simple subtyping is not enough. It does not solve the problem, posed in the introduction, of the expression $(i - j \geq 0) \Rightarrow (A[i - j] = A[i - j])$, where A is an array of type $\text{NAT} \rightarrow \text{NAT}$, and i and j are variables of type NAT. We tacitly assumed that "$-$" is ordinary subtraction on integers, so it has type $\text{INT} \times \text{INT} \rightarrow \text{INT}$.[6] Simple subtyping does not allow us to type check this expression. Declaring A to have type $\text{NAT} \rightarrow \text{NAT}$ does not imply that it has type $\text{INT} \rightarrow \text{NAT}$.

This problem can be solved with *predicate subtyping*, a strong form of subtyping used in the proof assistant PVS. Predicate subtyping allows us to declare NAT to be the subtype of INT such that $n : \text{NAT}$ if and only if $n : \text{INT}$ and $n \geq 0$. The expression $(i - j \geq 0) \Rightarrow (A[i - j] = A[i - j])$ then type checks with the original declarations of i, j, and A.

In general, predicate subtyping allows type expressions to contain arbitrary predicates, so they essentially become set comprehensions. It enables us to define the subtype $\text{REAL}_{\neq 0}$ of nonzero real numbers and give the reciprocal function *recip* the type $\text{REAL}_{\neq 0} \rightarrow \text{REAL}$. The question of what $1/0$ means never arises; an expression is not type correct if its meaning depends on the meaning of $1/0$. Type checking of any expression would include proving $x \neq 0$ for every occurrence of $1/x$. Context can be used, so $(x \neq 0) \Rightarrow (x \cdot (1/x) = 1)$ is type correct. But, with predicate subtypes, type checking is undecidable; the user must prove the type-correctness theorems that the type checker cannot.

3.5.2 *Overloading. Overloading* means letting one symbol stand for many different functions, using types to determine which function is intended. The symbol "+" could denote addition over types NAT, INT, and REAL. Because it eliminates the need for different versions of the operators over the numeric types, overloading may be seen as an alternative to subtyping. However, injections among the types are still required.

Haskell's type classes [Wadler and Blott 1989] support overloading in a controlled

[6] There is no problem if one defines "$-$" to have type $\text{NAT} \times \text{NAT} \rightarrow \text{NAT}$; but declaring $1 - 2$ to be a natural number is a way of pretending the problem doesn't exist, not of solving it. A system that does meaningful type checking and allows the type NAT should not only allow $(i - j \geq 0) \Rightarrow (A[i - j] = A[i - j])$, it should also disallow $(i - j \geq 0) \Rightarrow (A[j - i] > 0)$ when A has type $\text{NAT} \rightarrow \text{NAT}$.

fashion, ensuring that symbols are shared only among suitably related types. They treat overloading as a generalization of polymorphism. Type classes were invented for use in functional programming and are implemented in the proof assistant Isabelle [Paulson 1994b].

However, overloading is probably not the best way to promote flexibility in notation because it can lead to confusion. In any case, it is not a decisive reason to prefer a typed language to an untyped one, so we will not discuss it further.

3.6 Constructive Type Theories

A number of type theories, such as the Calculus of Constructions [Coquand 1990], have been designed as constructive alternatives to classical set theory. Constructive reasoning—whether typed or not—is concerned with what we can know, as opposed to what might be true "out there" [Dummett 1977]. This shift of emphasis rejects basic laws of classical logic, even the "obvious" tautology $P \vee \neg P$. Constructive logic accepts the truth of *every integer is either even or odd*, but only because we have an effective means of determining which alternative holds for any integer. It does not accept the statement *every real number is either rational or irrational*; given a real number, say as a convergent series, we have no effective means of determining whether or not it is rational. Constructive logic makes distinctions that are lost in classical logic. For instance, $\exists x \,.\, P(x)$ is a stronger assertion than $\neg(\forall x \,.\, \neg P(x))$, since the former implies that we can compute the value claimed to exist.

Formally, we do not say "A is true," but "a is a proof of A," and write $a \in A$. A proof of the conjunction $A \wedge B$ consists of a proof a of A and a proof b of B; thus, a proof of $A \wedge B$ has the form (a, b) for $a \in A$ and $b \in B$. Clearly, if we regard A and B as sets or types, then $A \wedge B$ is precisely the Cartesian product $A \times B$. Constructive type theories identify each formula with the type of its proofs.

Similarly, a proof of $A \vee B$ either has the form $\mathit{Inl}(a)$, for $a \in A$, or $\mathit{Inr}(b)$, for $b \in B$. The disjunction is simply the disjoint sum $A + B$. (The $\mathit{Inl}/\mathit{Inr}$ tag indicates whether the attached proof verifies A or B.) A proof of $A \Rightarrow B$ must provide a proof of B given a proof of A. It is a function f such that $f(x) \in B$ if $x \in A$. Constructive type theories represent implication by the type $A \to B$ of functions from A to B.

Quantifiers, viewed constructively, yield dependent types. A proof of $\exists x \in A \,.\, B(x)$ consists of some element a of A paired with a proof of $B(a)$. The corresponding set or type is written $\sum x \in A \,.\, B(x)$ and consists of all pairs (a, b) such that $a \in A$ and $b \in B(a)$; it generalizes the Cartesian product $A \times B$ by letting B depend upon elements of A. A proof of $\forall x \in A \,.\, B(x)$ consists of a function f that gives a proof $f(x)$ of $B(x)$ if $x \in A$. The collection of all such functions is written $\prod x \in A \,.\, B(x)$; it generalizes the function space $A \to B$ by letting B depend upon elements of A. For example, if $\mathrm{NL{\small IST}}(n)$ is the type of lists of length n, then the function f that maps each natural number n to the list $[1, 2, \ldots, n]$ has the dependent type $\prod n \in \mathit{Nat} \,.\, \mathrm{NL{\small IST}}(n)$.

Constructive type theories achieve conceptional economy by identifying $\wedge$ with $\times$, $\vee$ with $+$, $\exists$ with Σ, $\forall$ with Π, etc. They can use the same primitives on collections as they do on logical propositions. Their lore is too deep for us to examine here; look elsewhere for explanations of universes, impredicativity, intensional equality, and other mysteries.

For expressing specifications, constructive type theories are no more powerful than the classical systems we have examined above. The Σ and Π constructions can also be defined in both untyped set theory and in the typed set theory of higher-order logic. Classic ZF texts define Π [Halmos 1960, p. 36], and Σ has a simple definition. PVS's predicate subtypes provide the effect of Σ and Π at the level of types.

The main virtue of these type theories is precisely that they are constructive. A constructive proof that two arbitrary numbers always have a greatest common divisor provides an algorithm for computing it [Thompson 1991]. Researchers, using tools such as Coq [Barras et al. 1997] and Nuprl [Constable et al. 1986], are investigating whether this can lead to a practical method of synthesizing programs.

You can perform classical reasoning in a constructive type theory by adding $P \vee \neg P$ as an axiom. The resulting system will probably be strong enough to handle any specification problem likely to arise. However, it will be no stronger than ZF, and it will be much more cumbersome to use. If you want classical reasoning, use a system designed for that purpose. Since most computer scientists do prefer classical reasoning, constructive type theories are not widely used. We will not consider them further.

4. SETS VERSUS TYPES

Having described set theory and typed formalisms, we now compare them—first for writing specifications, then for reasoning about them. We also cast a more critical eye on predicate subtyping.

4.1 Specification

Our comparison of set theory and typed formalisms for writing specifications is partitioned into four rather arbitrary categories: flexibility, convenience, pitfalls, and abstractness.

4.1.1 *Flexibility.* Set theory is more flexible than typed systems. This flexibility is evident in the ability to model objects with sets, described in Section 2.7.3. To construct a set of all objects, we need to write sets like $\bigcup\{B_i : i \in S\}$, the union of all sets B_i with $i \in S$. No simple typed formalism that we know of admits $\bigcup\{B_i : i \in S\}$ as a type. It is a set in a typed set theory only if B_i has the same type for all i in S, which might require the use of disjoint sums.

Object-oriented type theories are being investigated [Fisher and Mitchell 1995], and perhaps a simple, elegant one can be found. However, objects are just one example of the diverse mathematical concepts that arise in real specifications. (In the application that led to this example, objects were needed to represent the system, not because we wanted to write an object-oriented specification.) We cannot expect to find type systems ready-made for each new concept—let alone for combinations of them. But set theory's flexibility should enable it to take new developments in stride. Indeed, one application of set theory is in modeling novel recursive structuring principles that can be used in the design of new type theories [Paulson 1994a; 1995].

The flexibility of set theory is also useful in more mundane circumstances. The *Copy* operator defined in Section 2.7.2 appears in a specification of a distributed system written in part by the first author. The specification describes actions in which a node receives a message m of one type and sends one or more messages of different types containing many of the fields from m. Using *Copy* instead of listing the fields to be copied makes the specification shorter and clearer. In a typed system, one would need a separate *Copy* operator for each pair of types, which is not feasible. Instead, one would define a single message type containing all the fields from all messages, and would ignore irrelevant fields in specific messages.

The typed specification is easy enough to write, but having a single type for all messages makes it essentially untyped. The set-theoretic specification is, in effect, strongly typed: it distinguishes among the individual message types. This example is typical of large specifications. It can be written in a typed formalism, but set theory permits simplifications that would probably not even occur to someone who has used only typed formalisms.

4.1.2 *Convenience.* One argument against typed formalisms is the inconvenience of having to attach type constraints to all variables. This is at most a minor point, and it does not apply to a well-designed language based on higher-order logic with type inference. Type inference propagates type information, rendering most type declarations unnecessary. From the single declaration $0 : \text{NAT}$ and the expression $0 \in A \cup \{x, y\}$, we can infer automatically $x, y : \text{NAT}$ and $A : \text{SET}(\text{NAT})$. The standard type inference algorithm has been proved to be sound [Milner 1978] and enjoys other strong properties; for instance, it always finds the most general type possible. Subtyping complicates matters considerably. If NAT is a subtype of INT, the declaration $0 : \text{NAT}$ does not determine the type of any variable in $0 \in A \cup \{x, y\}$. Mitchell [1991] has proposed a type inference algorithm to handle coercions between atomic types, but subtyping clearly limits what can be inferred automatically.

Conversely, it can be argued that types make writing specifications more convenient because they make parts of the specification implicit that must be expressed explicitly in an untyped formalism. Automatic type inference can deduce that x is of type NAT in cases where a set-theoretic specification would need the explicit assumption $x \in Nat$. However, the simple type system that makes type inference possible would force us to write $\iota(x) - \iota(y)$, where ι is the injection of type $\text{NAT} \rightarrow \text{INT}$, in cases where the set-theoretic specification would let us simply write $x - y$. Whether type inference helps more than injections hurt will depend on the particular specification.

4.1.3 *Pitfalls.* By pitfalls, we mean subtle aspects of a formalism that can lead unwary users to write specifications that don't mean what the users think they do. We illustrate some pitfalls using an *action* formalism, in which a system is specified by an initial predicate and a next-state relation, which is a predicate relating old and new values [Hehner 1984; Lam and Shankar 1984]. For example, a nonterminating program in which n is initially 0 and is continually decremented by 1 is specified by the initial predicate $n = 0$ and the next-state relation $n' = n - 1$. This next-state relation is equivalent to the programming-language statement $n := n - 1$, but action formalisms allow you to write next-state relations such as $n = n' + 1$ that have no counterpart in a conventional programming language.

One would expect the next-state relations $n' = n - 1$ and $n = n' + 1$ to be equivalent. They are in a typed formalism, if n is declared to have a numeric type such as NAT or INT. They are not equivalent in an untyped formalism like ZF. For example, there could be some nonnumeric value v, different from 3, such that $4 = v + 1$, so $4 = n' + 1$ does not imply $n' = 4 - 1$. The next-state relation for a program that continually decrements n by 1 can be written in set theory as $n' = n - 1$ or $(n = n' + 1) \wedge (n' \in \mathit{Int})$, but not as $n = n' + 1$.[7] The inequivalence of $n' = n - 1$ and $n = n' + 1$ is a minor nuisance. But, it could lead unwary users to write $n = n' + 1$ when they mean $n' = n - 1$.

This pitfall is avoided in a typed system because declaring n to have type INT asserts the assumption that n and n' are integers. However, such assumptions lead to a different pitfall. If we declare n to have type NAT, then we are assuming $n \geq 0$ and $n' \geq 0$. Hence, $n' = n - 1$ is equivalent to $(n' = n - 1) \wedge (n > 0)$. Thus, $n' = n - 1$ represents not the usual assignment statement $n := n - 1$, but the semaphore operation $P(n)$. This means that $\text{INT} \vdash F$ does not imply $\text{NAT} \vdash F$, if F is the formula asserting that $n' = n - 1$ is enabled. It is quite easy to forget that the next-state relation $n' = n - 1$ is not always enabled, and this can lead to errors. Indeed, the first author once fell into this trap and wrote incorrect proofs for a few algorithms.

Subtlety is in the mind of the beholder. A subtle trap for the naive user is an obvious error to the expert. An experienced user of a formalism may find it perfectly simple and think that only other formalisms have subtle pitfalls. Set theory and higher-order logic both have their pitfalls; there is no reason to believe that either has fewer than the other. However, complexity usually leads to subtle problems, so we might expect more pitfalls in a more complicated type system.

4.1.4 *Abstractness.* Mathematicians typically define objects by explicitly constructing them. For example, a standard way of defining $\mathbf{N}$ inductively is to let 0 be the empty set and n be the set $\{0, \ldots, n - 1\}$, for $n > 0$. This makes the strange-looking formula $3 \in 4$ a theorem.

Such definitions are often rejected in favor of more abstract ones. For example, de Bruijn [1995, Sect. 3] writes

> If we have a rational number and a set of points in the Euclidean plane, we cannot even imagine what it means to form the intersection. The idea that both might have been coded in ZF with a coding so crazy that the intersection is *not empty* seems to be ridiculous.

In the abstract data type approach [Guttag and Horning 1978], one defines data structures in terms of their properties, without explicitly constructing them.

The argument that abstract definitions are better than concrete ones is a philosophical one. It makes no practical difference how the natural numbers are defined. We can either define them abstractly in terms of Peano's axioms, or define them concretely and prove Peano's axioms. What matters is how we reason about them. If we use only Peano's axioms, then we will never prove $3 \in 4$, even if it should happen to follow from our definition of the natural numbers.

[7] Defining "+" so $m+n$ is a number if and only if m and n are both numbers does make $n' = n-1$ and $n = n' + 1$ equivalent as next-state relations for a system in which n is initially a number.

Experience with abstract data types has shown that defining data structures abstractly in terms of their properties is error-prone. It is easy to write inconsistent or incomplete lists of properties. When there does not already exist a well-established mathematical characterization of a data structure, one is better off defining it explicitly in terms of sets and functions.

Even though we define data types explicitly, experience with large programs shows that it is important not to "break" an abstraction by making use of its underlying representation. An abstraction can be enforced by some form of modularity that hides the representation from users of the abstraction. Such hiding works for both typed and untyped specifications.

4.2 Verification

One reason for writing a specification is to allow a subsequent verification. To say that a program (or hardware design) is correct means that it meets its specification. Some programs are verified by hand. Others are verified using proof tools. But most are not verified at all, even those that have been specified formally. We now consider what these alternatives imply about the choice of a formalism.

4.2.1 *Specification without Verification.* Verification is difficult and time-consuming. For most real systems, it is prohibitively expensive. But the very act of writing a formal specification catches errors, omissions, and ambiguities early in the design process [Fitzgerald et al. 1995]. This is the main objective of the popular specification languages Z and VDM. Such specifications are seldom intended for use in proofs.

Type checking can find errors in these specifications. This is a good reason for choosing a typed formalism. However, there may be other ways of finding errors. We envision a system in which an untyped specification can be augmented with typing annotations that are checked by machine. This approach is highly flexible. The "type system" could exploit the full power of set theory, perhaps in unusual ways. We could safely ignore "type errors"; our set theory is untyped, after all.

Analogous approaches have already been adopted for programming languages. Soft typing [Wright and Cartwright 1997] is one attempt to combine the advantages of untyped and typed programming languages. Modula-3 is strongly typed but provides loopholes in order to achieve the flexibility needed for writing systems programs [Nelson 1991].

A checker for typing annotations could combine the advantages of type checking with the generality of set theory, but building it is a research project. Now, if one wants the advantages of type checking, one must use a typed formalism.

4.2.2 *Verification by Machine.* Although not all specifications will serve as a basis for mechanical verification, the desire for machine-checked proofs may affect the choice of a formalism.

Type checking finds errors that, in an untyped system, must be caught when writing a proof. It can be argued that type checking saves work by catching errors earlier. However, type errors are generally trivial compared to the subtle errors that the proof process is designed to catch, and they are usually caught early in the proof. Indeed, one can argue that type checking wastes time by forcing the user to correct type errors in formulas that are later discarded because they turn out to be wrong or unnecessary. Neither argument carries much weight. Type checking can save work by catching an error early, and it can create extra work by forcing one to type check an unnecessary formula; but the amount of time saved or wasted is almost always negligible.

A more compelling argument in favor of typed formalisms for mechanical verification is that the type checker can automatically deduce facts that have to be asserted as lemmas with an untyped formalism. Suppose we want to prove that $(x+1)+y = y+(x+1)$ for real numbers x and y. In an untyped system, we would use the proof rule

$$\forall r, s \in \mathbf{R} \,.\, r + s = s + r \;. \tag{6}$$

To apply that rule, we must first prove $x + 1 \in \mathbf{R}$, using the rule

$$\forall r, s \in \mathbf{R} \,.\, r + s \in \mathbf{R} \;.$$

In a typed system, we would be proving $(x + 1) + y = y + (x + 1)$ when x and y are of type Real. The analog of (6) in a typed system is simply

$$r + s = s + r \tag{7}$$

where "+" has type $\text{Real} \times \text{Real} \rightarrow \text{Real}$. One can apply (7) directly to deduce $(x + 1) + y = y + (x + 1)$.

Logically, there is no difference between the untyped and typed proof. To apply (7), the typed system must type check the expression $(x + 1) + y$, which requires checking that $x + 1$ is of type Real. Hence, it has to prove the same lemma that must be proved in the untyped proof. The two proofs are completely isomorphic, where one writes $r \in \mathbf{R}$ in the untyped proof and $r : \text{Real}$ in the typed proof. An untyped prover could automatically prove theorems that correspond to type correctness. In fact, the Nqthm theorem prover [Boyer and Moore 1988] uses type information internally, even though the logic is untyped. Still, the untyped prover ends up doing extra work to prove type-correctness lemmas like $x + 1 \in \mathbf{R}$; unless precautions are taken, it may prove the same lemmas repeatedly. In ACL2 [Kaufmann and Moore 1996], an untyped theorem prover for an applicative subset of Common Lisp, the user can provide type declarations as hints to get the system to automatically deduce the same facts that a type checker does in a typed system.

Applying the conditional rewrite rule implicit in (6) is sufficiently difficult with current untyped systems that one often tries to avoid the problem by artificially extending the definition of "+" so that (6) holds unconditionally:[8]

$$\forall r, s \,.\, r + s = s + r \tag{8}$$

For example, we can define

$$r + s \;\stackrel{\Delta}{=}\; \textbf{if } r, s \in \mathbf{R} \textbf{ then } \ldots \textbf{ else } \textit{nonreal} \tag{9}$$

where *nonreal* is some arbitrary value not in $\mathbf{R}$. However, this approach cannot completely avoid the need to use type information in proofs. For example, redefining "+" and "−" in this way will not make $n' = n - 1$ and $n = n' + 1$ equivalent for nonnumeric values of n and n'.

[8]Boyer and Moore use this technique extensively to ensure that all functions are total.

The mechanical verification systems that have so far been most successful are probably HOL [Gordon and Melham 1993] and the Boyer-Moore prover [Boyer and Moore 1988]. Both systems have been used for over a decade to verify many systems. HOL uses higher-order logic, while the Boyer-Moore system is untyped, though it is not based on set theory. PVS [Owre et al. 1995], based on predicate subtyping, has recently become quite popular. Tools for ZF, such as EVES and Isabelle/ZF [Paulson 1993], are emerging; other axiom systems for set theory, such as Bernays-Gödel, are also suitable for automation [Quaife 1992]. It seems impossible to draw any conclusions about the superiority of typed or untyped formalisms from experience with existing verification systems. The most significant differences among them lie in such issues as the user interface, decision procedures, extensibility, and the ability to write proof tactics—not in whether the formalism is typed.

4.2.3 *Verification by Hand.* Any advantages that typed formalisms might have for mechanized proofs do not apply to hand proofs. One might try to argue that, even if the proof is done by hand, one could still use a type checker to catch some errors. However, automatic type checking is possible only for simple type systems. The errors caught by such type checking are mathematically trivial; they would be easily caught by any reasoning rigorous enough to be called a proof. Type checking will catch the error sooner, but our experience writing hand proofs indicates that this would not save a significant amount of time.

When reasoning by hand, it makes little difference if the formalism is typed or untyped. What matters is how simple the theorem is that one is trying to prove. Because of its greater flexibility, set theory can allow a simpler statement of a theorem than a typed formalism.

4.3 The Trouble with Predicate Subtypes

Predicate subtypes provide an appealing solution to the problem of the expression $(i - j \geq 0) \Rightarrow (A[i-j] = A[i-j])$. They seem to add the advantages of sets to a typed formalism. However, they introduce their own problems.

One problem with predicate subtypes is that they restrict how one can decompose definitions. They permit the definition

$$P \triangleq (i - j \geq 0) \Rightarrow (A[i-j] = A[i-j])$$

but not the pair of definitions

$$\begin{aligned} Q &\triangleq A[i-j] = A[i-j] \\ P &\triangleq (i - j \geq 0) \Rightarrow Q\ . \end{aligned}$$

The definition of Q does not type check. (Note that i and j are variables declared elsewhere, not parameters of the definition.) To write this, one would need a separate kind of "macro" definition (perhaps akin to C's `#define` directive) that defers type checking until the definition is used. Formulas in specifications can be very large. Reasoning about a large formula requires defining it in terms of subformulas whose definitions are expanded only as needed. Restrictions on what subformulas can be defined may be burdensome.

$$
\begin{array}{l}
\textbf{initially } n = 0;\ s = [\,] \\
\textbf{do}\ \ \textbf{true} \quad \rightarrow \quad n := n + 1;\ \ s := s \oplus [42] \\
\qquad \square \\
\qquad n > 0 \quad \rightarrow \quad n := n - 1;\ \ s := \mathit{Tail}(s)\ \ \textbf{od}
\end{array}
$$

Fig. 3. A simple algorithm.

Another problem with predicate subtypes is illustrated by the program of Figure 3. The program uses Dijkstra's **do** construct

$$\textbf{do } g_1 \rightarrow s_1 \,\square\, \ldots \,\square\, g_n \rightarrow s_n \textbf{ od}$$

which is executed by repeatedly choosing an arbitrary i such that g_i is true, and executing s_i. The statement terminates when all the g_i are false. *Tail* and $\oplus$ (concatenation) are the usual operations on sequences, and $[\,]$ is the empty sequence. The program of Figure 3 loops forever, nondeterministically adding and removing 42s from the sequence s, while keeping n equal to the length of s. We consider the proof that the program never sets s to *Tail* of the empty sequence. In a formalism based on set theory, we prove that the assertion $(s \in \mathit{List}(\mathbf{N})) \wedge (n = \mathit{Len}(s))$ is an invariant of the program. What do we do in a formalism based on predicate subtypes?

With predicate subtyping, *Tail* would have type $\textsc{List}_{ne}(\alpha) \rightarrow \textsc{List}(\alpha)$, where $\textsc{List}_{ne}(\alpha)$ is the subtype of $\textsc{List}(\alpha)$ consisting of nonempty lists of elements of α. If we let n have type $\textsc{Nat}$ and s have type $\textsc{List}(\textsc{Nat})$, then the program of Figure 3 does not type check because *Tail* is applied to an expression of the wrong type in the last clause of the **do** statement. The problem can be made to go away by adding the extra test $s \neq [\,]$ to the second guard. But verification means proving the correctness of a given implementation, not finding an implementation whose correctness one can prove.

In general, the type declarations of the program variables will have to encode an invariant of the program. In the worst case—which is probably not uncommon for algorithms that employ "partial functions" like *Tail* and division—the type declaration will have to include a large part of the invariant needed to prove the algorithm correct. Type checking requires performing a major part of the correctness proof and can be quite difficult. Type declarations are likely to be an awkward way of expressing an invariant.

With predicate subtyping, type checking is undecidable and often requires human intelligence. A well-designed verifier will handle the easy cases automatically and generate proof obligations for the rest. Predicate subtypes can be useful for increasing the flexibility of the type system in a theorem prover. However, the extra flexibility comes at the price of making some specifications awkward to write. Even with predicate subtyping, a typed formalism is significantly less flexible than set theory. Moreover, the subtyping rules of PVS are not simple; they have caused several bugs that violate soundness. For example, a recent PVS release note reads in part [Owre 1998]:

> Soundness bug 160 is due to subtype constraints being asserted out of context. The subtype information in B for the expression A AND B

should not be asserted globally since the subtyping might depend on the context A.

5. CONCLUSIONS

The advantages of types in programming languages are well known. Few people are aware of the problems they can introduce in a working formalism for specifying and reasoning about computer systems. A simple type system prohibits the simple formula $(i > 0) \Rightarrow (A[i] > 0)$ if i has type $\textsc{Int}$ and A has type $\textsc{Nat} \rightarrow \textsc{Nat}$. Predicate subtypes constrain how we can decompose formulas and can require quite complicated type declarations. Set theory provides a simple, powerful alternative that avoids these problems.

A working formalism should be designed on practical grounds. For types to be worth using, they must offer some benefit. That benefit can lie only in the realm of computerized tools. Without mechanical support, types have nothing to offer; set theory's greater flexibility makes it better suited to writing hand proofs.

The most obvious tool is a type checker. Simple type checking can catch errors in specifications. Those errors are easy to catch with rigorous proofs. However, many specifications are never verified; they can benefit from automatic type checking. But typed formalisms that permit automatic type checking are less flexible than set theory. The best way to catch errors may be to treat type declarations as annotations that do not affect the meaning of the specification. If the type system proved to be too inflexible, one could replace it by a different one or simply ignore certain type errors.

Another important class of tool is a mechanical theorem prover. Mechanical proofs in set theory tend to require more human guidance than proofs in a typed formalism. Type checking establishes results that otherwise have to be proved as theorems about set membership. However, untyped provers can already prove some "type-checking" results automatically, and we can expect such implementations to improve. Eventually, provers based on set theory may provide the benefits of type checking together with the ability to write specifications that cannot be type checked.

Model checkers are becoming increasingly popular, since they provide the guarantees of theorem proving with little human effort. Model checking, which in principle involves exhaustively checking all possibilities, can work only on a restricted class of specifications. This class of specifications can be defined by adding a quite restrictive type system to an untyped formalism based on ZF, so the class consists of all type-correct specifications. This approach is currently being pursued by colleagues of the first author.

Set theory is particularly appealing as a single formalism that can be used for a range of diverse and unforeseen applications. For each application, there may exist a type theory that is ideal for it. But it is unlikely that any type theory can be good for all applications. Universal formalisms do not exist in the real world; set theory is as close to one as we are likely to get. Now, mathematical theories must be redeveloped from scratch for each new verification system. The use of set theory as a common foundation could make possible the sharing of results between these different systems.

We believe that the generality of set theory can be combined with the benefit of type-based tools by viewing a type system as just a way of imposing well-formedness conditions on formulas. We can overlay different type systems atop untyped set theory, choosing the one that is suited to the particular tool we want to use. This is equivalent to defining a sublanguage of set theory to be translated into the language of the tool. The approach was used in TLP [Engberg et al. 1992], which translated from an untyped, ZF-based first-order language into LP [Garland and Guttag 1989], a typed logic that (at the time) lacked quantifiers. We believe this technique can be applied more generally and merits further research.

ACKNOWLEDGEMENTS

This work began as a diatribe against types by the first author. Martín Abadi, Robert Boyer, Luca Cardelli, Peter Hancock, Peter Ladkin, Denis Roegel, Fred Schneider, and Andrzej Trybulec suggested improvements to early versions of that paper. Andrew Appel, then editor-in-chief of this journal, suggested that it be published together with a rebuttal by the second author. We felt that presenting both sides of the issue in a single article would be more fruitful. The collaboration allowed us to explore our initial differences and eventually to reach agreement. The views presented here are shared by both of us. Their exposition benefited from comments by Martín Abadi, Nikolaj Bjørner, Robert Boyer, Michael Gordon, Jim Horning, Florian Kammüller, Gary Leavens, J Moore, Fred Schneider, and Natarajan Shankar.

REFERENCES

Apt, K. R. and Olderog, E.-R. 1990. *Verification of Sequential and Concurrent Programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, London, Paris, Tokyo, Hong Kong, Barcelona.

Barras, B., Boutin, S., Cornes, C., Courant, J., Fillibtre, J.-C., Giminez, E., Herbelin, H., Huet, G., Muqoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saobi, A., and Werner, B. 1997. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203 (May), INRIA-Rocquencourt. Version 5.8.

Boyer, R. S. and Moore, J. S. 1988. *A Computational Logic Handbook.* Academic Press.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panagaden, P., Sasaki, J. T., and Smith, S. F. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall.

Coquand, T. 1990. Metamathematical investigations of a calculus of constructions. In P. Odifreddi (Ed.), *Logic and Computer Science*, pp. 91–122. Academic Press.

de Bruijn, N. G. 1995. On the roles of types in mathematics. In P. de Groote (Ed.), *The Curry-Howard isomorphism*, pp. 27–54. Academia.

Dummett, M. 1977. *Elements of Intuitionism.* Oxford University Press.

Engberg, U., Grønning, P., and Lamport, L. 1992. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst (Eds.), *Proceedings of the Fourth International Conference on Computer Aided Verification*, Volume 663 of *Lecture Notes in Computer Science*, Berlin, pp. 44–55. Springer-Verlag. Proceedings of the Fourth International Conference, CAV'92.

Farmer, W. M. 1990. A partial functions version of church's simple theory of types. *Journal of Symbolic Logic 55,* 3, 1269–1291.

FISHER, K. AND MITCHELL, J. C. 1995. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems 1,* 3, 189–220.

FITZGERALD, J. S., LARSEN, P. G., BROOKES, T. M., AND GREEN, M. A. 1995. Developing a security-critical system using formal and conventional methods. In M. HINCHEY AND J. P. BOWEN (Eds.), *Applications of Formal Methods*, pp. 333–356. Prentice-Hall.

GARLAND, S. J. AND GUTTAG, J. V. 1989. An overview of LP, the Larch Prover. In N. DERSHOWITZ (Ed.), *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Volume 355 of *Lecture Notes on Computer Science*, pp. 137–151. Springer-Verlag.

GÖDEL, K. 1983. Russell's mathematical logic. In P. BENACERRAF AND H. PUTNAM (Eds.), *Philosophy of Mathematics: Selected Readings* (2nd ed.). Cambridge University Press. First published in 1944.

GORDON, M. J. C. AND MELHAM, T. F. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press.

GRIES, D. 1981. *The Science of Programming.* Springer-Verlag.

GRIES, D. AND SCHNEIDER, F. B. 1993. *A Logical Approach to Discrete Math*. Springer-Verlag, New York.

GUNTER, C. A. AND SCOTT, D. S. 1990. Semantic domains. In J. VAN LEEUWEN (Ed.), *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pp. 633–674. Elsevier.

GUTTAG, J. V. AND HORNING, J. J. 1978. The algebraic specification of abstract data types. *Acta Informatica 10*, 27–52.

HALMOS, P. R. 1960. *Naive Set Theory.* Van Nostrand.

HEHNER, E. C. R. 1984. Predicative programming. *Commun. ACM 27,* 2 (Feb.), 134–151.

HUDAK, P., JONES, S. P., AND WADLER, P. 1992. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices 27,* 5 (May). Version 1.2.

HUET, G. 1997. Re: types and extremism. Email to Leslie Lamport. Internet message sent on April 25, 1997 23:11:37 MET DST, number 199704252111.XAA19096@pauillac.inria.fr.

KAUFMANN, M. AND MOORE, J. S. 1996. ACL2: An industrial strength version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pp. 23–34. IEEE Computer Society Press.

LAM, S. S. AND SHANKAR, A. U. 1984. Protocol verification via projections. *IEEE Transactions on Software Engineering SE-10,* 4 (July), 325–342.

LEISENRING, A. C. 1969. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York.

MANNA, Z. AND PNUELI, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York.

MELHAM, T. F. 1989. Automating recursive type definitions in higher order logic. In G. BIRTWISTLE AND P. A. SUBRAHMANYAM (Eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 341–386. Springer.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MITCHELL, J. C. 1991. Type inference with simple subtypes. *Journal of Functional Programming 1,* 3 (July), 245–285.

NELSON, G. (Ed.) 1991. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

OWRE, S. 1998. PVS 2.1 patches (2.417). Email to pvs@csl.sri.com. Internet message sent on Sat, Feb 7, 1998 02:53:40 -0800, number 199802071053.CAA02273@lotus.csl.sri.com.

OWRE, S., RUSHBY, J., SHANKAR, N., AND VON HENKE, F. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21,* 2 (Feb.), 107–125.

PAULSON, L. C. 1985. Verifying the unification algorithm in LCF. *Science of Computer Programming 5*, 143–170.

PAULSON, L. C. 1993. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning 11,* 3, 353–389.

PAULSON, L. C. 1994a. A fixedpoint approach to implementing (co)inductive definitions. In A. BUNDY (Ed.), *12th International Conference on Automated Deduction*, LNAI 814, pp. 148–161. Springer.

PAULSON, L. C. 1994b. *Isabelle: A Generic Theorem Prover*. Springer. LNCS 828.

PAULSON, L. C. 1995. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning 15,* 2, 167–215.

PAULSON, L. C. 1996. *ML for the Working Programmer* (2nd ed.). Cambridge University Press.

QUAIFE, A. 1992. Automated deduction in von Neumann-Bernays-Gödel set theory. *Journal of Automated Reasoning 8,* 1, 91–147.

SCOTT, D. 1979. Identity and existence in intuitionistic logic. In M. P. FOURMAN (Ed.), *Applications of Sheaves*, pp. 660–696. Springer. Lecture Notes in Mathematics 753.

THOMPSON, S. 1991. *Type Theory and Functional Programming.* Addison-Wesley.

WADLER, P. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *16th Annual Symposium on Principles of Programming Languages*, pp. 60–76. ACM Press.

WHITEHEAD, A. N. AND RUSSELL, B. 1962. *Principia Mathematica.* Cambridge University Press. Paperback edition to *56, abridged from the 2nd edition (1927).

WRIGHT, A. K. AND CARTWRIGHT, R. 1997. A practical soft type system for Scheme. *ACM Trans. on Programm. Lang. Syst. 19,* 1 (Jan.), 87–152.

Received November 1995; revised April 1998; accepted February 1999.