Types Are Not Harmless

Leslie Lamport
lamport@src.dec.com

Tue 18 Jul 1995 [16:03]

Contents

1 Introduction

Types have become ubiquitous in computer science. Because the advantages of typed programming languages are so obvious, most computer scientists assume that mathematics should be formalized with types. They seem unaware of the harm that types inflict on the simplicity and elegance of mathematics. Some even feel that mathematics cannot be formalized without types. I believe that types do more harm than good in a formal system for mathematical reasoning. I do not expect many readers to agree with this conclusion. But, I hope to demonstrate that types are neither harmless nor necessary, and that their use in mathematics needs to be justified.

It is quite easy to formalize mathematics using sets and operators. Section 2 describes a formalism that I call ZFM (ZF for Mathematics). This section will seem excruciatingly obvious to some readers, since ZFM is a straightforward formalization of everyday mathematics. However, I have found that many computer scientists think that a typeless set theory must harbor logical inconsistencies.

Section 3 describes how the concepts of mathematics are expressed in typed formalisms, and Section 4 discusses some problems that types introduce. These problems arise in an of area of particular concern to computer scientists—formal reasoning about programs. I expect them to occur in more conventional branches of mathematics as well.

The problems raised by types are not insurmountable. Each problem that I describe can be solved in most type systems. But, the solutions add complexity and make a formalism harder to use. The question I wish to raise is not how type systems can solve these problems, but whether the benefits types provide offset the harm they do. Section 5 discusses what those benefits might be.

2 Math Without Types

There are a number of ways to formalize mathematics without using types. The formalism I call ZFM is the one I find to be the simplest and most natural way to capture ordinary mathematical reasoning. I describe it informally; the formal definition of ZFM should be a straightforward exercise for a mathematician familiar with the foundations of logic and set theory.

2.1 Logic

ZFM is based on first-order predicate logic with equality. Its formulas contain variables, quantifiers, and operators. Each operator has a signature, defined in terms of the primitive signatures *term* and *formula*. For example, the equality operator ($=$) has signature $term \times term \rightarrow formula$. Quantification is only over variables, which have signature *term*.[1]

ZFM also includes Hilbert's ε operator [1], which I call **choose**. The expression $\textbf{choose } x : P(x)$ 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").

$$\vdash (\exists\, x : P(x)) \;\Rightarrow\; P(\textbf{choose } x : P(x))$$
$$\vdash (\forall\, x : P(x) \equiv Q(x)) \;\Rightarrow\; (\textbf{choose } x : P(x)) = (\textbf{choose } x : Q(x))$$

Mathematicians need to define new operators in terms of the primitive ones provided by a formalism. ZFM takes 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 of ZFM. A definition cannot introduce unsoundness, so we never have to prove a theorem in order to make a definition. However, we may have to prove that a defined operator has the properties we want.

One useful operator is the **if/then/else** construct, which has signature $formula \times term \times term \rightarrow term$. It is defined by

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

2.2 Set Theory

The primitive operators of ZFM include the ones of Zermelo-Fraenkel set theory [2]. ZFM uses the simplest form of set theory, in which everything

[1] Bound variables introduce complications. A simpler formalism can be obtained by eliminating them. For example, the formula $\exists\, x : \exists\, y : P(x, y)$ can be replaced by $\exists\,(\exists\,(Q))$, where Q is the operator of sort $term \rightarrow (term \rightarrow formula)$ such that $Q(x)(y)$ equals $P(x, y)$. Such a formalism is less convenient in practice, so ZFM uses bound variables. However, thinking of ZFM as based on an underlying formalism with no bound variables can help clarify some subtle points, such as the meaning of substitution.

$= \quad \neq \quad \in \quad \notin \quad \emptyset \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [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]
2^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]

Figure 1: The operators of set theory.

is a set. Thus, the number 2 is a set, though we usually don't think of it as one because we don't care what its elements are. The phrase "the set 2" seems strange, so I let *value* be synonymous with *set* and refer to 2 as a value.

Figure 1 describes a collection of operators from set theory that I have found adequate for ordinary mathematics. Some of these operators are defined in terms of the others; the rest are primitive and are effectively defined by axioms. For example, the Separation Axiom [2, page 325] asserts that, for any set S, the collection $\{x \in S : P(x)\}$ of all elements of S satisfying $P(x)$ is a set.[2] It makes no practical difference which of these operators are taken to be primitive. (They can all be defined in terms of $\in$.)

For most uses of ZFM, it suffices to understand the meanings of the operators of Figure 1, and to know that two sets are equal iff they have the same elements. Formally, we need axioms to guarantee the existence of "enough" distinct sets. For computer science, it seems necessary only to assume an axiom of infinity that implies the existence of the natural numbers. More sophisticated mathematical applications may need the Axiom of Choice [2, page 335].[3]

Naive informal reasoning about sets can be unsound. In ZFM, soundness is guaranteed by restricting how one can construct sets. First order logic with the operators of Figure 1, and the corresponding axioms, is sound. For example, Russell's paradox is avoided because the "set" $\mathcal{R}$ of all sets that are not elements of themselves, which satisfies $\mathcal{R} \in \mathcal{R}$ iff it satisfies $\mathcal{R} \notin \mathcal{R}$, cannot be expressed with the operators of Figure 1.

[2] This axiom is valid only because S is considered to lie outside the scope of the bound variable x. Similarly, S lies outside the scope of x in $\{e(x) : x \in S\}$.

[3] The axioms for the **choose** operator are independent of the Axiom of Choice.

2.3 Functions

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

$$\mathit{Apply}(f, x) \;\triangleq\; \textbf{choose } y : \langle x, y \rangle \in f$$

and let $f(x)$ be an abbreviation for $\mathit{Apply}(f, x)$. It doesn't matter how functions are defined. I prefer simply to regard the four operators of Figure 2 as primitive, where I 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 $\textbf{dom}\, f$. The set $[S \to T]$ consists of all functions f such that $\textbf{dom}\, f = S$ 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. For example, if $\mathbf{R}$ is the set of real numbers, then $[r \in \mathbf{R}\backslash\{0\} \mapsto 1/r]$ is the *reciprocal* function, whose domain is the set $\mathbf{R}\backslash\{0\}$ of nonzero reals.

Functions can be defined recursively using the **choose** operator. For example, the factorial function *fact* is defined by

$$\mathit{fact} \;\triangleq\; \textbf{choose } f : f = [n \in \mathbf{N} \mapsto \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot f[n-1]]$$

where $\mathbf{N}$ is the set of natural numbers. I write this definition as

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

In general, $f[x : S] \triangleq e(x)$ defines f to equal $\textbf{choose } f : f = [x \in S \mapsto e(x)]$. To deduce that f is a function with domain S, we must prove that there exists a function f that equals $[x \in S \mapsto e(x)]$. This is the same proof that is often cited as a requirement for the definition to be legal. In ZFM, definitions are always legal; proofs are needed only to deduce properties of the definiendum.

[4] One could use $f(x)$ for both, since the signature of f determines which is meant.

$f[e]$	[Function application]
$\textbf{dom}\, f$	[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 $x \in S$]

Figure 2: Operators for expressing functions

Functions are different from operators.[5] A function f has a domain, and we can define the value of $f[x]$ only for elements x in its domain. The expression $\mathit{fact}[\sqrt{2}]$, which is an abbreviation for $\mathit{Apply}(\mathit{fact}, \sqrt{2})$, is syntactically a term, so it denotes a value. However, we don't know what value. It need not equal $\sqrt{2} \cdot \mathit{fact}[\sqrt{2} - 1]$. It need not even be a number. But, whatever its value, it must equal $\mathit{fact}[2/\sqrt{2}]$, since $\sqrt{2}$ equals $2/\sqrt{2}$. Functions are just like other values; for example, *fact* by itself is syntactically a term. We can quantify over sets of functions, writing expressions such as $\forall f \in [\mathbf{R} \to \mathbf{R}] : |f|_\infty \geq 0$.[6]

Operators are different from functions. Consider the operator $\bigcup$, where $\bigcup S$ is the union of all elements of S. We cannot define a function *union* so that $\mathit{union}[S]$ 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 could express Russell's paradox.) The symbol $\bigcup$ by itself is not a term, so it does not denote a value. We cannot quantify over operators. 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 the signature of U is $\mathit{term} \to \mathit{term}$, and bound variables are terms.

2.4 Examples

ZFM is easy to use in practice. As an example of something that can pose difficulties for a typed system but is trivial in ZFM, let us define $\mathit{len}(s)$ to be the length of s, for any finite sequence s. Formally, a sequence $s_1, \ldots, s_n$ is a function s whose domain is the set $\{1, \ldots, n\}$ such that $s[i] = s_i$, for $1 \leq i \leq n$. It is impossible to define *len* to be a function, since its domain would have to be a set consisting of all finite sequences, and there is no such set in ZFM. But, the operator *len* is easily defined by

$$\mathit{len}(s) \;\triangleq\; \mathbf{choose}\ n : (n \in \mathbf{N}) \wedge (\mathbf{dom}\ s = \{i \in \mathbf{N} : 1 \leq i \leq n\})$$

This defines $\mathit{len}(s)$ for all s, even if s is not a sequence. In that case, $\mathbf{dom}\ s$ is not a set of the form $\{1, \ldots, n\}$ for any n in $\mathbf{N}$, and the value of $\mathit{len}(s)$ is some unspecified value.

As a further example of how easily mathematics can be formalized with ZFM, Figure 3 defines the Riemann integral of elementary calculus. The

[5] More precisely, a function is an operator of signature *term*, so it has no (operator) arguments.

[6] $\forall x \in S : P(x)$ and $\exists x \in S : P(x)$ are abbreviations for $\forall x : (x \in S) \Rightarrow P(x)$ and $\exists x : (x \in S) \wedge P(x)$, respectively.

$$
\begin{array}{lcl}
seq(S) & \triangleq & \bigcup \{ [\{i \in \mathbf{N} : 1 \leq i \leq n\} \rightarrow S] : n \in \mathbf{N}\} \\
\sum\limits_{m}^{n \,:\, \mathbf{N}} f & \triangleq & \textbf{if } n < m \textbf{ then } 0 \textbf{ else } f[n] + \sum\limits_{m}^{n-1} f \\
\mathbf{R}^{+} & \triangleq & \{r \in \mathbf{R} : 0 < r\} \\
|r| & \triangleq & \textbf{if } r < 0 \textbf{ then } -r \textbf{ else } r \\
\mathcal{P}_{a}^{b}(\delta) & \triangleq & \{\, p \in seq(\mathbf{R}) : (p[1] = a) \wedge (p[len(p)] = b) \\
 & & \qquad \wedge\ \forall\, i \in \{1 \ldots len(p) - 1\} : \\
 & & \qquad\qquad (\,|p[i+1] - p[i]| < \delta) \\
 & & \qquad\qquad\ \wedge\ \textbf{if } a \leq b \textbf{ then } p[i] \leq p[i+1] \\
 & & \qquad\qquad\qquad\qquad \textbf{else } \ p[i] \geq p[i+1] \ \} \\
\mathcal{S}_{p}(f) & \triangleq & \sum\limits_{1}^{len(p)-1} [\, i \in \mathbf{N} \mapsto (p[i+1] - p[i]) \cdot (f[\,p[i]\,] + f[\,p[i+1]\,]\,)\,/\,2] \\
\int_{a}^{b} f & \triangleq & \textbf{choose } r : \\
 & & \quad (r \in \mathbf{R}) \wedge (\forall\, \epsilon \in \mathbf{R}^{+} : \exists\, \delta \in \mathbf{R}^{+} : \forall\, p \in \mathcal{P}_{a}^{b}(\delta) : \\
 & & \qquad\qquad\qquad\qquad |\, r - \mathcal{S}_{p}(f)\,| < \epsilon\,)
\end{array}
$$

Figure 3: The definition of the Riemann integral.

definition uses the operators of Figures 1 and 2, the set $\mathbf{R}$ of real numbers, the subset $\mathbf{N}$ of naturals, the usual arithmetic operations and ordering relations on numbers, and the operator *len*. The integral $\int_a^b f$ is defined to be the limit of trapezoidal approximations $\mathcal{S}_p(f)$ determined by partitions p of the interval $[a,\ b]$, as the diameter (maximum subinterval length) of p goes to zero. Figure 3 makes the following preliminary definitions: $seq(S)$ is the set of sequences of elements of the set S; $\sum\limits_m^n f$, an abbreviation for $\sum(m, f)[n]$, equals $f[m] + f[m+1] + \cdots + f[n]$; $\mathbf{R}^+$ is the set of positive reals; $|r|$ is the absolute value of r; and $\mathcal{P}_a^b(\delta)$ is the set of partitions of $[a,\ b]$ (sequences $a = p[1],\ p[2],\ \ldots,\ b$) of diameter less than δ.

It is often best to define something axiomatically rather than constructing it explicitly. The **choose** operator makes this easy. For example, to define the natural numbers $\mathbf{N}$ with successor function *succ* and zero element 0, we first define $Peano(N, s, z)$ to be the formula asserting that the set N with successor function s and zero element z satisfies Peano's axioms.[7]

[7] With first-order logic alone, one cannot rule out nonstandard models of the naturals. This is seldom a problem in practice, since we are usually content to prove theorems that are valid for any model satisfying Peano's axioms.

We then define

$$
\begin{array}{lcl}
\mathbf{N} & \triangleq & (\textbf{choose } \Psi : \mathit{Peano}(\Psi[1], \Psi[2], \Psi[3]))\,[1] \\
\mathit{succ} & \triangleq & (\textbf{choose } \Psi : \mathit{Peano}(\Psi[1], \Psi[2], \Psi[3]))\,[2] \\
0 & \triangleq & (\textbf{choose } \Psi : \mathit{Peano}(\Psi[1], \Psi[2], \Psi[3]))\,[3]
\end{array}
$$

"Higher-order" operators pose no problem. For example, consider the operator Γ defined by

$$\Gamma(F)(x, y) \;\triangleq\; F(x) = y$$

It has signature $(\mathit{term} \rightarrow \mathit{term}) \rightarrow (\mathit{term} \times \mathit{term} \rightarrow \mathit{formula})$. If G has signature $\mathit{term} \rightarrow \mathit{term}$, then $\Gamma(G)$ has signature $\mathit{term} \times \mathit{term} \rightarrow \mathit{formula}$, and $\Gamma(G)(e, f)$ equals the formula $G(e) = f$, for any terms e and f. We are still in the domain of first-order logic, because we can quantify only over terms, not over arbitrary operators.

2.5 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. More sophisticated logicians say that 1/0 equals the nonvalue $\perp$, which acts like a virus, turning infected expressions to $\perp$. They devise complicated rules to describe how it spreads—for example, declaring that $(0 = 1) \vee (0 = \perp)$ equals $\perp$, but $(0 = 1) \wedge (0 = \perp)$ equals **false**. ZFM provides a simpler answer to the question of what 1/0 is: we don't know and we don't care.

Let us take / to be a function with domain $\mathbf{R} \times (\mathbf{R} \backslash \{0\})$, the set of pairs $\langle r, s \rangle$ of real numbers with s nonzero. Then 1/0 is an abbreviation for $/[\langle 1, 0 \rangle]$. Since $\langle 1, 0 \rangle$ is not in the domain of /, 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

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

This formula is valid, meaning that it is satisfied by all values of x. Substituting 0 for x yields the formula **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 (1) may or may not be valid; we don't know what $0 \cdot (1/0)$ or $\mathbf{R} \cdot (1/\mathbf{R})$ equal, so we don't know whether or not they equal 1.

2.6 Semantic Types

We can define what it means for a ZFM formula to be semantically type correct. For each operator F, we must define an operator $\mathcal{T}_F$ that is true if the arguments of F are "sensible". For example, $\mathcal{T}_{Apply}(f, x)$ equals **true** iff f is a function with x in its domain, in which case we say that the pair f, x is in the *semantic type* of *Apply*. A formula is *semantically type correct* iff its validity does not depend on the value of any operator applied to values not in its semantic type. Formula (1) is semantically type correct, but its subformula $x \cdot (1/x) = 1$ is not.

Theorems should be semantically type correct, but arbitrary formulas that appear within them need not be. If we know nothing about a value, then we can't prove a theorem whose validity depends on that value. Thus, proving a theorem shows that it is semantically type correct. ZFM has rules for proving theorems; there is no need to introduce semantic types and semantic type checking.

3 Types

There is no consensus on what constitutes a type. Although everyone agrees that the prime natural numbers form a set, computer scientists disagree on whether they should form a type. There are many different type systems; they differ in what kinds of types can be declared and what those type declarations mean. However, most type systems share some features, which I now describe.

Type systems allow some sets as types. Although one might consider ZFM to be a typed logic with only two basic types, *term* and *formula*, we expect a type system to include such types as NAT (natural numbers) and REAL (real numbers). Type systems can be classified according to how rich a collection of sets can be expressed as types. A *fine* type system allows the set $\mathbf{R} \setminus \{0\}$ of nonzero reals to be type; a *coarse* one allows only "simpler" sets like $\mathbf{R}$ as types.

Type systems generally replace the separate concepts of functions and operators with the single concept of a typed function. For example, the cube-root function $\sqrt[3]{\ }$ might have type REAL $\rightarrow$ REAL. Replacing an operator like *len* with a typed function requires the concept of polymorphism, discussed in Section 3.1.

Types are used for type checking. Typed formalisms prevent errors by allowing one to write only formulas that are accepted by the type checker.

Other uses of types are discussed in Section 3.2.

3.1 Polymorphism

Advocates of type systems might claim that eliminating the distinction between functions and operators produces a simpler formalism. However, this simplification would be of no interest if it were achieved by eliminating useful operators like *len*. In most type systems, *len* is a polymorphic function, often thought of as a collection of simple functions of type $\textsc{Seq}(T) \rightarrow \textsc{Nat}$, for certain types T. What those "certain types" T are depends on the type system; the type of *len* itself may or may not be one of them. Just replacing functions and operators with simple functions and polymorphic functions does not constitute a simplification. Devising a type system in which there is no essential difference between $\sqrt[3]{}$ and *len* is a nontrivial task. Such a type system is unlikely to be simpler than just distinguishing between functions and operators.

A form of polymorphism that is often called *overloading* is used to formalize some informal notation. For example, a mathematician might write $\times$ for both the Cartesian product and the vector product on $\mathbf{R}^3$, so $x \times y \in S \times T$ asserts that the vector product $x \times y$ is an element of the Cartesian product $S \times T$. This can be formalized by declaring $\times$ to be a polymorphic function. In a typeless system, one could do the same thing by defining the operator $\times$ as

$$a \times b \;\triangleq\; \textbf{if } a \in \mathbf{R}^3 \textbf{ then } \ldots \textbf{ else } \ldots$$

Both approaches seem complicated and potentially confusing. If a rigorous treatment is required, it is probably best to use two different symbols and write $x \boldsymbol{\times} y \in S \times T$.

In some formalisms, $+$ is regarded as polymorphic because $\textsc{Nat}$ and $\textsc{Real}$ are disjoint types, with separate definitions of addition. In such a formalism, the real number 2.0 does not equal the natural number 2. Most mathematicians would regard this as bizarre, but some computer scientists believe that the naturals should not be a subset of the reals. Typed formalisms usually permit either approach. However, with many type systems, it is impossible first to define the naturals, and then to define the reals as a superset. This is easy to do in ZFM by defining $\mathbf{R}$ axiomatically, with $\mathbf{N} \subseteq \mathbf{R}$ as one of the axioms.

3.2 Types as Abbreviations

The type declaration $r : \textsc{Real}$ often means that, within its scope, $\forall r$ is an abbreviation for $\forall r \in \mathbf{R}$. Such an abbreviation can be convenient, and one could add a similar convention to ZFM. However, I have found that the small advantage of eliminating the "$\in \mathbf{R}$" is outweighed by the disadvantage of having to declare bound variables. Many potentially confusing expressions[8] can be ruled out by the syntactic restriction that the same variable cannot appear free and bound in the same expression. Since variables may appear implicitly through the use of previously defined operators, an easy way to enforce this restriction is by requiring that free variables be declared and bound variables be undeclared.

The type declaration $r : \textsc{Real}$ can serve as an assertion that, within its scope, r is assumed to be an element of $\mathbf{R}$. Such an assertion is often used in proofs; it is easily expressed without types by writing "assume $r \in \mathbf{R}$". Moreover, one may want to introduce a variable r satisfying certain properties and then prove $r \in \mathbf{R}$. Requiring a type declaration for r simply to introduce it could make writing a formal proof difficult.

4 The Trouble with Types

Formal reasoning is especially important when proving properties of programs. The social process by which mathematicians check each others' proofs does not exist in the realm of program verification, so formal reasoning is crucial for eliminating errors. I will therefore discuss the problems introduced by types when reasoning about programs.

Formalisms for reasoning about programs usually provide some extension to conventional mathematics—for example, Hoare triples or weakest precondition operators. For my examples, I assume the kind of formalism typically used to reason about concurrency, in which an execution of a program is represented by a sequence of states or events, and the formula $\Box P$ asserts that P is true throughout an execution. However, nothing I say is peculiar to this formalism; types cause exactly the same problems in other formalisms for reasoning about programs.

Consider the program of Figure 4. It uses Dijkstra's **do** construct

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

[8]For example, $\{x \in \{x\} : x \notin x\}$, in which the second x is not the same variable as the other three x's.

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

Figure 4: A simple algorithm.

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 *Append* are the usual operations on sequences, and ϕ is the empty sequence. The program of Figure 4 loops forever, nondeterministically appending and removing 42's from the sequence s, while keeping n equal to the length of s. The property I want to prove of this program is that it never sets s to $\mathit{Tail}(\phi)$.

Figure 4 is meant to describe an abstract program, not a real one. The symbols $+$, $-$, *Append*, and *Tail* represent mathematical operations on natural numbers and sequences, not procedures for manipulating bits in a computer. The program permits executions in which the value of n becomes arbitrarily large; a real program would produce an error if n became too large. We want to write and reason about abstract descriptions of programs, without having to introduce the complexity of a real programming language.

In a logic based on ZFM, one can prove that the program satisfies the property $\Box(s \in \mathit{seq}(\mathbf{N}))$, which asserts that the value of s is always a finite sequence of natural numbers. Since $\mathit{Tail}(\phi)$ is unspecified, one cannot prove that it is in $\mathit{seq}(\mathbf{N})$. Thus, proving $\Box(s \in \mathit{seq}(\mathbf{N}))$ shows that s is never set to $\mathit{Tail}(\phi)$. We prove this result by using induction to show that the program satisfies $\Box((s \in \mathit{seq}(\mathbf{N})) \wedge (n = \mathit{len}(s)))$. This is the same method used to prove any property of the form $\Box P$, such as the partial correctness property $\Box(\mathit{terminated} \Rightarrow Q)$, which asserts that Q holds if and when the program terminates.

Semantic type correctness of a program means that the value of each variable is always an element of the appropriate set. It is a property of the form $\Box((x_1 \in S_1) \wedge \ldots \wedge (x_n \in S_n))$. As the example indicates, proving such a property can be hard. In fact, semantic type correctness is undecidable.

With a typed formalism, one must declare types for the variables n and s and the "functions" $+$, $-$, *Append* and *Tail*. The representation of the program is then type checked. Different type systems will require different type declarations, and will differ in the meanings of those declarations. In almost all systems, one would declare n to be of type Nat and s to be of type $\text{Seq}(\text{Nat})$. Fine and coarse type systems would use different declarations for *Tail*, which cause different kinds of problems.

4.1 Fine Type Systems

In a fine type system, *Tail* will be of type $\text{Seq}_{\neq\phi}(\text{Nat}) \rightarrow \text{Seq}(\text{Nat})$, where $\text{Seq}_{\neq\phi}(\text{Nat})$ is the type of nonempty sequences of naturals. Type checking the program means proving that the program satisfies $\Box(s \in seq(\mathbf{N}))$. Type correctness means semantic type correctness, and type checking requires the same kind of proofs needed to prove other program properties, such as partial correctness. Type checking is undecidable for a fine type system.

In a fine type system, there seems to be no logical justification for giving type declarations a different status than other correctness properties. Still, if a type system did nothing more than introduce a redundant way of expressing properties, it would be relatively harmless. However, fine type systems introduce a worse problem.

In reasoning about programs, as in any kind of mathematics, complexity is handled by combining simple results about small expressions to obtain more difficult results about big expressions. We therefore want to write and reason about parts of programs and then compose those parts. For example, in proving semantic type correctness for the program of Figure 4, we prove that executing the statement

$$n > 0 \;\rightarrow\; n := n - 1;\; s := \mathit{Tail}(s) \tag{2}$$

leaves the assertion $(s \in seq(\mathbf{N})) \wedge (n = len(s))$ true. However, if s is of type $\text{Seq}(\text{Nat})$ and *Tail* is of type $\text{Seq}_{\neq\phi}(\text{Nat}) \rightarrow \text{Seq}(\text{Nat})$, then (2) is not type correct. A logic based on a fine type system would not let us define and reason about this statement.

One can solve the problem of handling statement (2) by introducing the notion of dependent types and declaring the type of s to be all sequences of naturals whose length equals n. However, the substatement $s := \mathit{Tail}(s)$ of (2) still does not type check.

This example illustrates a fundamental problem with type systems that try to capture semantic type correctness. Type systems type check definitions. But, just as semantic correctness in ZFM holds only for complete theorems, semantic correctness in a programming logic holds only for complete programs. Type checking definitions can make it impossible in practice to build up complicated theorems or programs by defining their components separately.

4.2 Coarse Type Systems

In a coarse type system, *Tail* has type $\textsc{Seq}(\textsc{Nat}) \rightarrow \textsc{Seq}(\textsc{Nat})$. With such a system, there is no problem separately type checking statement (2) or its substatements. However, we are left with the question of what it means for *Tail* to have this type. There seem to be three possibilities: *Tail* is a partial function that is defined on a subset of $seq(\mathbf{N})$, it really is a function from $seq(\mathbf{N})$ to $seq(\mathbf{N})$, or the declaration is a piece of syntax that doesn't really mean anything.

4.2.1 Partial Functions

If *Tail* is a partial function, and $Tail(\phi)$ is undefined, then what is the meaning of a program that assigns $Tail(\phi)$ to s—for example, the program obtained by changing the initial value of n to 1 in Figure 4? Simply declaring a program to be illegal if it might set s to $Tail(\phi)$ would leave us unable to reason about substatement (2) by itself, since it would be illegal.

One possibility is that $Tail(\phi)$ is the infectious nonvalue $\perp$. In this case, type checking does not guarantee semantic type correctness. We still have to prove that the original program satisfies $\Box(s \in seq(\mathbf{N}))$. However, as noted earlier, we would have to do this in a more complicated formalism than ZFM. For example, in some methods of coping with $\perp$, the formula $a \neq b$ is not equivalent to $\neg(a = b)$.

Another possibility is that declaring s to be of type $\textsc{Seq}(\textsc{Nat})$ means that the program is guaranteed never to set s to $Tail(\phi)$. The type declaration implies that substatement (2) really means

$$n > 0 \wedge s \neq \phi \;\rightarrow\; n := n - 1;\; s := Tail(s) \tag{3}$$

This possibility is unsatisfactory because it provides no way to distinguish the incorrect program obtained from Figure 4 by initializing n to 1, and the correct program obtained by then replacing statement (2) with (3).

4.2.2 Total Functions

If *Tail* is a function from $seq(\mathbf{N})$ to $seq(\mathbf{N})$, then $Tail(\phi)$ is some sequence of naturals. It might be defined to equal ϕ, or it might be some unspecified sequence of naturals. Semantic type correctness is trivially satisfied. This is unsatisfactory because we want a program to be incorrect if it sets s to $Tail(\phi)$. We don't want to be able to prove reasonable properties about such a program.

4.2.3 Pure Syntax

One can interpret the declaration that *Tail* has type $\textsc{Seq}(\textsc{Nat}) \rightarrow \textsc{Seq}(\textsc{Nat})$ to be just an informal comment meaning that *Tail* should be used only in a context in which its argument and its result are both expected to be sequences of naturals. Type checking provides a sanity check for detecting errors, but type declarations have no semantic significance. Type declarations might also indicate implicit ranges of quantification and of index sets, but these are just syntactic abbreviations.

Syntactic type checking is harmless. It is probably used unconsciously by mathematicians. If s is supposed to be a sequence of naturals, a mathematician notices that the expression $s \in \mathbf{N}$ must be wrong because it doesn't type check. As observed above, there is no reason for mathematicians to formalize this kind of type checking because proving a theorem demonstrates that the theorem is semantically type correct.

Syntactic type checking is redundant when proving theorems, but mathematics is not used only to prove theorems. A formal specification of a program is a mathematical formula, and large specifications are often written without proving anything about them. Type checking is a good way to catch errors in these specifications, and a purely syntactic type system seems like a good choice for a specification language. Its type checker would never declare a specification to be illegal; it would just issue warnings about formulas that it suspects to be incorrect. Since its type declarations have no semantic content, a syntactic type system would be freed from many of the constraints that hinder conventional type systems. In principle, it could be better at detecting errors. Unfortunately, I know of no such type system. Designing one seems to be a good topic for research.

5 Do Types Help?

The usual justification for types is that type checking catches errors. But, is type checking better than other methods for catching those errors?

For programming languages, the answer seems to be yes. Types allow the compiler to catch errors that are more difficult to find at run time, when debugging the program. They also allow more efficient code to be generated. Although types can be inconvenient—mainly by making it difficult or impossible to write some correct programs—it seems clear that the advantages of types can outweigh their inconvenience.

For mathematics, the answer is not so clear. Type checking allows errors to be caught when making definitions. But those errors should be caught when writing a proof. Mathematicians who are so careless in writing proofs that they miss the obvious errors found by type checking are unlikely to detect the subtle errors that lead to incorrect theorems. Most computer scientists believe that type checking helps because it catches errors sooner. But, this belief does not seem to be based on any evidence. I have written many rigorous proofs using an untyped formalism; a few of them were mechanically checked. I have never felt that a type checker could have saved me any significant amount of effort—even if it had required me to do no extra work.

Another justification for types is that type checking isolates the part of theorem proving that can be done automatically. When reasoning in an untyped logic, we must prove type correctness as part of a proof. To prove anything about the expression $m+n$, we must first prove that it is a number. This is what type checking proves. In general, when semantic types coincide with coarse types, as they do for $+$, semantic type correctness can be proved by the type checker.

This argument for type checking can be expressed as follows. A type checker can deduce automatically that, if n and m have type $\textsc{Nat}$ and $+$ has type $\textsc{Nat} \times \textsc{Nat} \rightarrow \textsc{Nat}$, then $n+m$ has type $\textsc{Nat}$. A theorem prover needs human interaction to deduce that, if n and m are elements of $\mathbf{N}$ and $+$ is a function from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$, then $n+m$ is an element of $\mathbf{N}$. The absurdity of this argument is evident. Yet, it has become a self-fulfilling prophecy. Some theorem provers based on a typed formalism will automatically type check the expression $m+n$, but will require a great deal of human effort to prove

$$(m \in \mathbf{N}) \wedge (n \in \mathbf{N}) \wedge (+ \in [\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}]) \Rightarrow (m+n \in \mathbf{N}) \tag{4}$$

This is a problem in the design of theorem provers. When the two are equivalent, there is no reason why type checking should be any easier than proving semantic type correctness. When the two are not equivalent, type checking does not prove any useful mathematical result. If $Tail(\phi)$ need not be a number, then certifying that the expression $Tail(s) + n$ is type correct does not prove that it is a number.

It can be argued that type declarations provide needed guidance to a theorem prover. There are many theorems a prover could conceivably try to prove automatically; how can it tell that it should try to prove (4)? This suggests adding type declarations as hints to the theorem prover, with no semantic content. It is another possible argument in favor of syntactic types. However, it is not clear that theorem provers really need this help.

6 Conclusion

Types are very useful in programming languages. They are useful in mathematics too. Type declarations can help the reader understand a formula, and type checking can catch errors. But types have their cost. The only form of type system that seems to have no significant drawbacks is a purely syntactic one in which type declarations are just comments, and type checking serves only to ensure that the formulas appear to be consistent with those comments. I know of no formal system that uses such a type system.

Computer scientists seem to have embraced types without giving them much thought. There is a general feeling that everyone uses types, so they must be good. But types are not harmless. Their benefits must be weighed against the problems they introduce. For programming languages, there is a great deal of evidence to demonstrate the benefits of types. I know of no similar evidence for mathematical formalisms. Mathematicians reason informally without using types, and their style of reasoning can easily be formalized without types. Mathematics is not programming, and the use of types in mathematics is not justified by the success of typed programming languages. Typed formalisms may turn out to be good for mathematics, but this has yet to be demonstrated.

Acknowledgements

Robert Boyer, Luca Cardelli, Peter Hancock, Peter Ladkin, Denis Roegel, Fred Schneider, and Andrzej Trybulec suggested improvements to previous versions.

References

[1] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol.* Gordon and Breach, New York, 1969.

[2] J. R. Shoenfield. The axioms of set theory. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter B1, pages 317–344. North-Holland, Amsterdam, 1977.