# Types Considered Harmful

Leslie Lamport
lamport@src.dec.com

Wed 23 Dec 1992 [10:54]

## 1 Introduction

A mathematician might define $len(s)$ to denote the length of $s$, for any finite sequence $s$. Many computer scientists would find this definition naive. They consider *len* to be a highly polymorphic function, and they believe it can be defined formally only in a language with a sophisticated type system—if it can be defined at all. We show that conventional mathematics can be formalized without the use of types. It is quite easy to define *len* formally.

Types are so widely accepted in programming languages that most computer scientists have forgotten their drawbacks. Even if the benefits outweigh the drawbacks for programming languages, the same balance need not hold for other domains. We claim that types are not just unnecessary, they are harmful in formalizing mathematics. This claim has implications for the design of specification languages and mechanical theorem provers.

## 2 Formalizing Mathematics

### 2.1 Zermelo-Fraenkel Set Theory

The formalism that seems to be most widely regarded as the best foundation for ordinary mathematics is Zermelo-Fraenkel set theory [2]. The particular version of Zermelo-Fraenkel set theory described here will be called ZF. We consider formal definitions, but not formal reasoning, so we will not describe the precise axioms of ZF.

In ZF, one reasons only about sets. Since we want to reason about the number 2, it must be a set. Mathematicians usually don't think of 2 as a set because they don't care what its elements are. To be consistent with ordinary usage, we let *value* be synonymous with the term *set* of

| | |
|---|---|
| $=$ $\neq$ $\in$ $\notin$ $\emptyset$ $\cup$ $\cap$ $\subseteq$ | $\setminus$ [set difference] |
| $\{e_1, \ldots, e_n\}$ | [Set consisting of elements $e_i$] |
| $\{x \in S : p\}$ | [Set of elements $x$ in $S$ satisfying $p$] |
| $\{e : x \in S\}$ | [Set of elements $e$ 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 $i^{\text{th}}$ component is $e_i$] |
| $S_1 \times \ldots \times S_n$ | [The set of all $n$-tuples with $i^{\text{th}}$ component in $S_i$] |

Figure 1: The operators of set theory.

Zermelo-Fraenkel set theory, and we use *set* to refer to those values that mathematicians usually regard as sets, such as the set **R** of real numbers.

Figure 1 describes a collection of operators from set theory that we have found adequate for ordinary mathematics. Some of these operators can be defined in terms of the others. The rest are effectively defined by axioms. For example, the Separation Axiom [2, page 325] asserts that, for any set $S$, the expression $\{x \in S : p\}$ denotes a set consisting of all elements of $S$ satisfying $p$. This axiom is valid only because $S$ is considered to lie outside the scope of the bound variable $x$. Thus, $\{x \in \{x\} : x \notin x\}$ equals $\{y \in \{x\} : y \notin y\}$, which equals either $\{x\}$ or $\emptyset$. Similarly, $S$ lies outside the scope of $x$ in $\{e : x \in S\}$.

For most uses of ZF, it suffices to understand the meanings of the operators of Figure 1, and to know that two sets are equal iff (if and only if) they have the same elements. Certain applications require axioms that guarantee the existence of "enough" distinct sets. In computer science, it seems necessary only to assume that, for every set $S$, there is some value $v$ such that $v \notin S$. More sophisticated mathematical applications may need more powerful assumptions, such as the Axiom of Choice [2, page 335].

Naive informal reasoning about sets can be unsound. A common source of error is reasoning about collections of elements that are "too big" to be sets, where a collection is too big if it has "as many elements as" the collection of all sets. For example, the collection of all sets having a single element is too big because the correspondence $S \leftrightarrow \{S\}$ shows that there are as many single-element sets as there are sets. Such errors are avoided by using the operators of Figure 1, with which one can construct only legal sets. For example, Russell's paradox asserts that the "set" $\mathcal{R}$ of all sets that are not elements of themselves satisfies $\mathcal{R} \in \mathcal{R}$ iff it satisfies $\mathcal{R} \notin \mathcal{R}$. The "set" $\mathcal{R}$ cannot be expressed with the operators of Figure 1.

The formulas of ZF are formed from the operators of set theory, the usual operations of first-order predicate logic, and two additional constructs. The first is **if** $p$ **then** $e_1$ **else** $e_2$, which equals $e_1$ if $p$ is true, and otherwise equals $e_2$. The second is **choose** $x : p$, which denotes some arbitrarily chosen value $x$ for which the predicate $p$ is true, if one exists, and otherwise denotes a completely arbitrary value. For example, **choose** $x : x \notin \mathbf{R}$ denotes some unspecified value that is not a real number. The **choose** operator is known to logicians as Hilbert's $\varepsilon$-symbol [1]. Although this operator appears to introduce the Axiom of Choice to ZF, it actually adds no new properties of sets.

## 2.2 Functions

It is customary to define functions to be sets of ordered pairs. For example, the *reciprocal* function that maps every nonzero real number $r$ to $1/r$ is defined to be the set $\{\langle r, 1/r\rangle : r \in \mathbf{R}\backslash\{0\}\}$. However, it does not matter how functions are defined. What we need are the four operators described in Figure 2. For reasons discussed below, we write $f[x]$ instead of the more conventional $f(x)$ to denote the result of applying the function $f$ to the value $x$. Every function $f$ has a domain, which is a set denoted **dom** $f$. The set $[S \rightarrow T]$ consists of all functions $f$ such that **dom** $f = S$ and $f[x] \in T$ for all $x \in S$. The notation $[x \in S \mapsto e]$ is used to describe a function explicitly; for example, $[r \in \mathbf{R}\backslash\{0\} \mapsto 1/r]$ denotes the *reciprocal* function, whose domain is the set $\mathbf{R}\backslash\{0\}$ of nonzero reals.

We now have the operators we need to describe all the functions of ordinary mathematics. Consider, for example, the factorial function. This is the unique function *fact*, whose domain is the set $\mathbf{N}$ of natural numbers, such that $\mathit{fact}[0] = 1$ and $\mathit{fact}[n] = n \cdot \mathit{fact}[n-1]$, for $n > 0$. Hence, *fact* equals

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

| | |
|---|---|
| $f[e]$ | [Function application] |
| **dom** $f$ | [Domain of the function $f$] |
| $[S \rightarrow T]$ | [Set of functions with domain $S$ and range a subset of $T$] |
| $[x \in S \mapsto e]$ | [Function $f$ such that $f[x] = e$ for $x \in S$] |

Figure 2: Operators for expressing functions

We use the following syntax to mean that *fact* is defined to be this function.

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

### 2.3 Operators

Let us now consider the problem of defining $len(s)$ to denote the length of an arbitrary sequence $s$. We represent a sequence $s$ of length $n$ by a function whose domain is the set $\{1 \ldots n\}$ of natural numbers from 1 through $n$, letting $s[i]$ represent the $i^{\text{th}}$ element of the sequence. (We let $\{1 \ldots 0\}$ denote the empty set, so the empty sequence, which has length zero, is the unique function whose domain is the empty set .)

As a warmup exercise, we define the set $seq(S)$ of all sequences of elements in $S$. An $n$-element sequence in $seq(S)$ is a function with domain $\{1 \ldots n\}$ and range a subset of $S$. The set of all such sequences is denoted by $[\{1 \ldots n\} \rightarrow S]$, and $seq(S)$ is the union of all such sets, for all natural numbers $n$. Hence, $seq(S)$ is the set $\bigcup \{ [\{1 \ldots n\} \rightarrow S] : n \in \mathbf{N}\}$.

Now, let us define $len(s)$. A sequence of length $n$ is a function with domain $\{1 \ldots n\}$. Thus, $n$ is the length of a sequence $s$ iff $n$ is a natural number and **dom** $s$ equals $\{1 \ldots n\}$. Hence, $len(s)$ can be defined to equal **choose** $n : (n \in \mathbf{N}) \wedge (\mathbf{dom}\ s = \{1 \ldots n\})$.

What is *len*? One would like to call it a function and consider $len(s)$ to be an instance of function application. However, the domain of a function is a set. If *len* were a function, its domain would have to be the collection of all sequences, which is "too big" to be a set. We must consider *len* to be an *operator*, just like $\bigcup$. For any value $v$, the expression $\bigcup v$ denotes a value, so $(\bigcup v) \in S$ is a formula of ZF. However, $\bigcup$ by itself does not denote a value, and $\bigcup \in S$ is a meaningless string of symbols, not a formula of ZF. Similarly, $len(v)$ denotes a value, for any value $v$; but *len* by itself does not denote a value, and $len \in S$ is a syntactically incorrect, meaningless string of symbols.

We find it helpful to distinguish functions from operators by using square brackets for function application. However, this is not necessary; ordinary parentheses can be used for both. In a properly designed formal language, one can determine syntactically whether a symbol denotes a value or an operator, so $f(x)$ is unambiguous.

### 2.4 What is 1/0?

For any value $v$, the expression $len(v)$ denotes a value. What value is denoted by $len(\mathbf{R})$? By definition, it equals **choose** $n : (n \in \mathrm{N}) \land (\mathbf{dom}\ \mathbf{R} = \{1 \ldots n\})$. Since $\mathbf{R}$ is not a function, we don't know what value is denoted by the expression **dom R**, so we don't know whether it equals $\{1 \ldots n\}$ for any natural number $n$. Hence, we don't know what value $len(\mathbf{R})$ denotes; perhaps it equals $\sqrt{2}$, perhaps it doesn't. More precisely, the axioms of ZF do not allow us to determine whether the formula $len(\mathbf{R}) = \sqrt{2}$ is valid.

Let us switch to a more familiar example. We might define $x/y$ to equal **choose** $r : (r \in \mathbf{R}) \land (r \cdot y = x)$. Since there is no $r$ satisfying $(r \in \mathbf{R}) \land (r \cdot 0 = 1)$, this definition allows us to say nothing about 1/0 except that it is a value. It might equal $\sqrt{2}$; it might equal $seq(\mathbf{R})$.

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 $(0 = 1) \lor (0 = \perp)$ equals $\perp$, but $(0 = 1) \land (0 = \perp)$ equals $\mathsf{false}$.

Instead of calling 1/0 an error or a nonvalue, it is much simpler to say that 1/0 is some value—we just don't know what value. Even $1/(\bigcup \mathbf{R})$ is a value. This causes no problems. Consider the formula

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

It is valid when any value is substituted for $x$. Substituting 0 for $x$ yields

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

which is valid regardless of what 1/0 equals and whether or not $0 \cdot (1/0)$ equals 1, because $0 \neq 0$ is false. Substituting $\bigcup \mathbf{R}$ for $x$ yields

$$((\bigcup \mathbf{R}) \in \mathbf{R}) \land ((\bigcup \mathbf{R}) \neq 0) \;\Rightarrow\; ((\bigcup \mathbf{R}) \cdot (1/(\bigcup \mathbf{R})) = 1)$$

which is valid regardless of what $1/(\bigcup \mathbf{R})$ equals, because $(\bigcup \mathbf{R}) \in \mathbf{R}$ is false.

### 2.5 An Example: The Riemann Integral

To illustrate how to formalize ordinary mathematics in ZF, we define the Riemann integral of elementary calculus as the limit of trapezoidal approximations to the (signed) area under the graph of the function. Define a *partition* $p$ of the interval from $a$ to $b$ to be a monotonic sequence $p[1], \ldots,$ $p[n]$ of reals such that $p[1] = a$ and $p[n] = b$. Approximating the area under the graph of $f$ between $p[i]$ and $p[i{+}1]$ by a trapezoid, such a partition $p$ yields the following approximation $\mathcal{S}_p f$ to $\int_a^b f$.

$$\sum_{i=1}^{n-1} (p[i{+}1] - p[i]) \cdot (\, f[\, p[i]\,] + f[\, p[i{+}1]\,]\,) \,/\, 2$$

Define a partition $p$ to be a $\delta$-*partition* iff $|p[i{+}1]{-}p[i]| < \delta$, for $i = 1, \ldots, n-1$. The Riemann integral $\int_a^b f$ is defined to be the limit of approximations $\mathcal{S}_p f$ for $\delta$-partitions $p$, as $\delta$ goes to zero. More precisely, $\int_a^b f$ is the unique number satisfying the following property: for every $\epsilon > 0$ there exists a $\delta > 0$ such that $|\int_a^b f - \mathcal{S}_p f| < \epsilon$ for every $\delta$-partition $p$. If no such number exists, then $f$ is not integrable on the interval from $a$ to $b$, and $\int_a^b f$ is not defined.

This definition is formalized in Figure 3, assuming only the set $\mathbf{R}$ of real numbers, the subset $\mathbf{N}$ of naturals, and the usual arithmetic operations and ordering relations on numbers. First, the set $\mathbf{R}^+$ of positive reals, the absolute value, and the set $\{m \ldots n\}$ are defined. Then, $\sum_m^n f$ is defined to equal $\sum_{i=m}^n f[i]$. We consider $\sum_m^n f$ to be syntactic sugar for $\sum(f, m)[n]$, where we have made $\sum(f, m)$ a function with domain $\mathbf{N}$ to permit a recursive definition. Next, we define the set $\mathcal{P}_a^b(\delta)$ of all $\delta$-partitions of the interval from $a$ to $b$, and the approximation $\mathcal{S}_p f$. (It does not matter what $\mathcal{S}_p f$ equals if $p$ is not a partition.) Finally, we define the integral in terms of the operator $\mathcal{S}$. Note that the value of $\int_a^b f$ is unspecified if $f$ is not integrable on the interval from $a$ to $b$.

Figure 3 may not be simple, but its complexity comes from the mathematical concept it is defining. The translation of the ordinary definition into ZF was straightforward. The major departure from conventional mathematical notation is in writing $\sum_m^n f$ instead of $\sum_{i=m}^n f[i]$. To permit such notation, we would need some way of indicating in the definition of $\sum_{i=m}^n f[i]$ that $i$ is a bound variable, and that $f$ lies in its scope, but $m$ and $n$ do not.

### 2.6 Further Remarks

In ordinary, informal mathematics, the meaning of a symbol depends on its context. The symbol $\times$ in a text on set theory denotes the Cartesian

$$
\begin{array}{l}
\mathbf{R}^{+} \;\triangleq\; \{r \in \mathbf{R} : 0 < r\} \\[4pt]
|r| \;\triangleq\; \mathbf{if}\ r < 0\ \mathbf{then}\ -r\ \mathbf{else}\ r \\[4pt]
\{m \ldots n\} \;\triangleq\; \{i \in \mathbf{N} : (m \leq i) \wedge (i \leq n)\} \\[4pt]
\sum_{m}^{n:\mathbf{N}} f \;\triangleq\; \mathbf{if}\ n \leq m\ \mathbf{then}\ 0\ \mathbf{else}\ f[n] + \sum_{m}^{n-1} f \\[4pt]
\mathcal{P}_a^b(\delta) \;\triangleq\; \{\, p \in \mathit{seq}(\mathbf{R}) : \\
\qquad\qquad \wedge\ (p[1] = a)\ \wedge\ (p[\mathit{len}(p)] = b) \\
\qquad\qquad \wedge\ \forall\, i \in \{1 \ldots \mathit{len}(p) - 1\} : \ \wedge\ \mathbf{if}\ a \leq b\ \mathbf{then}\ p[i] \leq p[i+1] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ p[i] \geq p[i+1] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \wedge\ |p[i+1] - p[i]| < \delta \qquad\qquad \} \\[4pt]
\mathcal{S}_p f \;\triangleq\; \sum_{1}^{\mathit{len}(p)-1} [\, i \in \mathbf{N} \mapsto (p[i+1] - p[i]) \cdot (\, f[\,p[i]\,] + f[\,p[i+1]\,]\,)\,/\,2\,] \\[4pt]
\int_a^b f \;\triangleq\; \mathbf{choose}\ r : \wedge\ r \in \mathbf{R} \\
\qquad\qquad\qquad\qquad \wedge\ \forall\, \epsilon \in \mathbf{R}^{+} : \exists\, \delta \in \mathbf{R}^{+} : \forall\, p \in \mathcal{P}_a^b(\delta) : |\, r - \mathcal{S}_p f\,| < \epsilon
\end{array}
$$

Figure 3: The definition of the Riemann integral.

product, while in an advanced calculus text it may denote the product of vectors in three-space. The symbol $x$ may be used to denote different values in different parts of a proof. A complete formal language for mathematics would include hierarchically structured contexts, and definitions that apply only within a specified context. This structuring is independent of whether or not one uses types.

Mathematicians sometimes use the same symbol to denote different things within a single context. An advanced calculus text might let $\times$ denote both Cartesian product and vector product, hoping the reader will figure out that $x \times y \in S \times T$ asserts that the vector product $x \times y$ is in the Cartesian product $S \times T$. When the rules for disambiguating the use of a symbol can be stated precisely, they can be defined formally. For example, $a \times b$ can be defined to equal $\mathbf{if}\ a \in \mathbf{R} \times \mathbf{R} \times \mathbf{R}\ \mathbf{then}\ \ldots\ \mathbf{else}\ \ldots$. However, it is probably better to write $x \boldsymbol{\times} y \in S \times T$ than to explain how to distinguish between two different uses of $\times$.

# 3 Types

We have never found a rigorous, universally accepted definition of a type. However, we can define type correctness. We first define a *class* to be a collection of values that satisfies some property expressible in ZF. For example, the collection of all finite sequences is a class, since a value $s$ is a finite sequence iff it satisfies the ZF formula $\exists S : s \in seq(S)$. In a typed system, every operator has a domain that is a class. The domain of operator *len* would be the class of all finite sequences. The function-application operator, which maps the pair $\langle f, x \rangle$ to the value $f[x]$, has as its domain the class of all $\langle f, x \rangle$ such that $x \in \mathbf{dom}\ f$. A formula is *type correct* iff its validity does not depend on the result of applying an operator to any argument not in its domain. Thus, $(\exists x \in \mathbf{R} : x^2 < 0) \Rightarrow (7 = len(\mathbf{R}))$ is type correct even though $\mathbf{R}$ is not in the domain of *len*, since the formula's validity does not depend on the value of $len(\mathbf{R})$.

Type correctness is a reasonable requirement for a formula. However, type systems generally require a formula to be *type checkable*, meaning that a type-checking algorithm can determine that the formula is type correct. To make type checking easy, most typed languages allow only a restricted form of classes to be used as types, and have conservative rules for type checkability. A type correct formula need not be type checkable, and it may have to be re-expressed in a more complicated fashion to appease the type checker. More complex type systems can provide more expressive power, at the cost of making the language more complicated. But, any language with a decidable type-checking procedure must disallow some type-correct formulas.

Conventional wisdom asserts that type checking catches enough errors in programs to be worth the inconvenience caused by prohibiting some type-correct programs. We do not dispute this assertion. But, mathematics is not programming. Programs operate on simple, finite data structures—not on Riemann-integrable functions on the real line. The complexity introduced by working around the restrictions of a type system is a potential source of error. In the realm of mathematics, type checking could cause more errors than it catches.

Our experience has been that, when writing a rigorous proof of a theorem, one quickly finds any error in its statement that could be caught by a type checker. There are applications in which theorems are written (perhaps implicitly) but not proved—for example, specifying complex systems. Type checking can catch errors in these applications. However, type checking need not imply the constraints of a type system. Type correctness can be expressed as a theorem in ZF. For example, type correctness of the definition of *len* is expressed by the formula $\forall S : \forall s \in seq(S) : len(s) \in \mathbf{N}$. Type checking does not require a typed language. One can assert type correctness with theorems, and rename the type checker to be an automatic theorem prover. A language could even provide a syntax for writing those theorems as if they were type declarations—for example, a "typed" definition of *len* might be written

$$len(s : seq(\mathbf{Any})) : \mathbf{N} \;\triangleq\; \textbf{choose } n : (n \in \mathbf{N}) \wedge (\textbf{dom } s = \{1 \ldots n\})$$

But unlike type declarations, failure of the theorem prover (type checker) to verify the type-correctness theorem asserted by such a definition need not force one to rewrite the definition. The theorem could be proved by other methods, or simply accepted without proof.

## 4 Conclusion

Although types may be good for programming languages, we claim that formal mathematics should employ an untyped language. We also claim that specification should be more like mathematics than like programming. A specification language should encourage an abstract description of what a program or system is supposed to do, not a low-level description of how it behaves. Everything we have said about the unsuitability of types for mathematical formulas applies to specifications.

It is sometimes argued that, because programming languages use types, a language for specifying programs should also be typed. However, an untyped formalism such as ZF is likely to be better than a typed one for describing a different type system. A typed specification language would be convenient only for specifying programs written in a language with the same type system. But, verifying that a program implements a specification that uses the same type system is not likely to catch an error caused by misunderstanding some subtle feature of the type system. We do not find Pascal to be a good language for specifying Pascal programs.

It should be obvious that theorem proving is mathematics, not programming. Yet, most mechanical theorem provers use a typed language, making it harder to express mathematics. These provers are better at reasoning about types than at reasoning about sets, the most fundamental data type of mathematics. Given the type declarations $m$, $n : \mathbf{N}$ and $+ : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$, a theorem prover will immediately deduce that $m + n : \mathbf{N}$. But, it can take considerable effort to convince the prover that $m \in \mathbf{N}$, $n \in \mathbf{N}$, and $+ \in [\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}]$ imply $m + n \in \mathbf{N}$ (where $m + n$ means $+[\langle m, n\rangle]$). Any algorithm for deducing the types of expressions provides a method for deducing facts about set membership. Sets are a more natural choice than types as the basis for a general-purpose theorem prover.

We have found that most computer scientists have an unreasonable attachment to types. They are suspicious of any attempt to eliminate types, and many will refuse to believe that the brief outline of ZF presented here can be made completely formal. ZF serves as the foundation for a complete specification language, with a precise formal semantics, that will be described elsewhere. Here, we can only assure the reader that mathematicians have gotten along quite well for two thousand years without types, and they still can today.

## Acknowledgements

I have profited from comments on an earlier version by Mike Gordon, Peter Hancock, James Horning, Heiko Krumm, and Peter Ladkin.

## References

[1] A. C. Leisenring. *Mathematical Logic and Hilbert's $\varepsilon$-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.