The Review of Modern Logic
Volume 10 Numbers 3 & 4 (March 2005–May 2007) [Issue 32], pp. 219–224.

Fairouz Kamareddine, Twan Laan, and Rob Nederpelt
A Modern Perspective on Type Theory: From its Origins until Today
Applied Logic Series, Vol. 29
Dordrecht/Boston: Kluwer Academic Publishers, 2004
xiv + 357 pp. ISBN 1402023340

REVIEW

VLADIK KREINOVICH

In 1902, Bertrand Russell discovered a famous paradox of naive set theory—that for the set $S = \{x : x \notin x\}$, $S \in S$ if and only if $S \notin S$. The paradox comes from the fact that traditional set theory allows self-reference: once we have defined a new set (such as Russell's set S), we can then ask whether the new set is an element of itself. To avoid this (and similar) self-reference paradoxes and still allow mathematically useful constructions of sets of sets *etc.*, Russell introduced the following idea.

We start with known consistent sets of the standard type. Based on these sets, we can form new sets, *e.g.*, by considering all standard sets that satisfy a given property P: $\{x \text{ is of standard type} : P(x)\}$. By definition, these sets only contain sets of the standard type; so while we can legitimately consider a new set $S = \{x \text{ is of standard type} : x \notin x\}$, this set will contain only standard sets and not new sets (like the set S itself). These new sets form the first level in the set hierarchy.

We can also consider sets of such first level sets, *i.e.*, sets of the form $\{x \text{ is of first level} : P(x)\}$; these sets form the next (2nd) level, *etc.* Formally, we must explicitly assign to each variable x an integer value called its *type*, with the understanding that possible values of this variable x are sets of this type. The set $\{x^a : P(x^a)\}$ is then of type $a+1$, the set $\{x^a : \forall y^b \exists z^c P(x^a, y^b, z^c)\}$ is of type $\max(a, b, c) + 1$, and the formula $x^a \in y^b$ only makes sense if $a < b$.

In this theory, if elements of the set A are of type a, then subsets of A are of type $a+1$. If elements of the set B are of type b, then pairs—*i.e.*, elements of the Cartesian product $A \times B$—have a type $\max(a, b)$. A function $f : A \to B$ is usually defined as a set of pairs $f \subseteq A \times B$, so a function is an object of type $\max(a, b) + 1$. The resulting Type Theory became the first successful formalization of set theory.

The idea that, *e.g.*, a set of real numbers or a function from real numbers to real numbers are objects of higher type than real numbers themselves is very natural and in good accordance with mathematical practice. Similarly, a real number—naturally defined as a function from natural numbers to natural numbers—is an object of higher type than a natural number itself.

However, the fact that we must distinguish between, *e.g.*, real numbers of type 1, of type 2, *etc.*—depending on the complexity of their definitions—is troubling and mathematically counter-intuitive. For example, in mathematics, it is natural to ask whether a certain property holds for all real numbers. However, in the original Russell's type theory, we cannot even ask this question—we can only ask whether some property holds for all real numbers of type 0, or type 1, *etc.* To resolve this problem, Russell introduced a special Axiom of Reducibility according to which, crudely speaking, the truth of a statement should not change if we simply add a constant to the level of all the variables. Thus, to check that a certain property holds for real numbers, it is sufficient to check this property on some level – although we must still make sure that within the formulas, there is a proper relation between levels (types) of different variables. This axiom is very helpful mathematically, but it is no longer as intuitively clear as the original type theory idea.

From the purely logical viewpoint, the resulting type theory was very similar to the λ-calculus, a formalism introduced by Alonzo Church in the 1930s. One of the motivations behind λ-calculus is the desire to bring precision to mathematics and mathematical logic. In traditional mathematical practice, the notation $f(x)$ may mean a real number that is obtained by applying a function f to the value x, or it can mean a function that transforms a real number x into a new real number $f(x)$. This particular ambiguity is usually resolved by using f as a notation of a function, but in more complex cases such as $f(x, y)$, this simple idea does not work well: $f(x, y)$ can mean a real number, it can mean a function of one variable x, it can mean a function of one variable y, and it can also mean a function of two variables x and y. To avoid this ambiguity, Church proposed to keep $f(x)$ as a notation for a real number, and to denote a function f by $\lambda x.f(x)$. In Church's notations, the four possible interpretations of $f(x, y)$ are clearly distinguished as $f(x, y)$, $\lambda x.f(x, y)$, $\lambda y.f(x, y)$, and $\lambda x, y.f(x, y)$.

In Church's approach, these different objects have a different type: $f(x, y)$ is one of the original objects (type ι); a function $\lambda x.f(x, y)$ is a function from original objects to original objects, *i.e.*, an object of type $\iota \to \iota$, and a function $\lambda x, y.f(x, y)$ transforms pairs into objects, *i.e.*, is of type $\langle \iota, \iota \rangle \to \iota$. A function of two variables can be viewed, equivalently, as a function that maps x into a function $\lambda y.f(x, y)$ of one variable, *i.e.*, as an object of type $\iota \to (\iota \to \iota)$.

Correct interpretation of logical formulas requires an even more complex description of types. For example, a closed formula P is an object whose values can be "true" or "false" (type o, in Church's notations); a proposition $P(x)$ with a free variable x can be viewed as a function that maps objects x into truth values, *i.e.*, as an object of type $\iota \to o$. A logical connective such as $\vee$ is an object of type $\langle o, o \rangle \to o$, and a quantifier $\forall x$ is a function that takes a predicate $\lambda x.P(x)$ and returns a truth value, *i.e.*, a quantifier is an object of type $(\iota \to o) \to o$.

Because of this complexity, Church's λ-notations were not adapted by mainstream mathematicians. At present, when we are all familiar with programming and algorithms, Church's description—which may have sounded counterintuitive to the 1930s mathematicians—is perfectly natural. For example, in the 1930s, a logical operation could sound like something completely different from a normal function (such as addition), but inside the computer, "and" is simply an operation that transforms real values into real values—and moreover, in a natural representation where "true" is 1 and "false" is 0, "and" is simply multiplication. Church's notations did lead to a notion of a recursive function—one of the first general descriptions of algorithms, and eventually, to the appearance of modern programming languages, in which types are explicitly defined, and a type of a function is usually presented in its header. Since Church interprets all objects as functions, Church's notations led to *functional languages*, starting with the famous (and very successful) AI language LISP—which is, in a nutshell, nothing else but an effectively implemented λ-calculus.

In late 1920s and 1930s, the relation between types and efficient computations also appeared in the analysis of intuitionistic (and what we would now call constructive) mathematics. While both Church's approach and constructive mathematics serve similar goals—analyzing and designing efficient algorithms—their approach was drastically different: Church wanted to introduce *new* notations that will be perfectly suited to describe computability, while constructive mathematics tried to retain as much as possible from the *traditional* mathematical notations, trying to achieve efficiency by changing the rule of the game. In other words, Church started from logic and tried to translate mathematics into logical terms, while constructive mathematics started with mathematical notions and ideas and tried to match logic as needed.

Surprisingly, both approaches led to similar results—including the efficient use of types. We have already mentioned how types appeared in Church's approach; let us briefly describe how types appeared in constructive mathematics.

In constructive mathematics, the truth of a statement like $\exists x P(x)$ means that we not only know that $P(x)$ holds for some value x, we actually know a value x for which $P(x)$ is true. Thus, a constructive proof of $\exists x P(x)$ must include a description of this object x (or, at least, an algorithm for computing x) and a proof that $P(x)$ is true. Similarly, an implication $A \to B$ ("A implies B") is constructively true if, whenever we know that A is true, we must be able to constructively imply B. Thus, we must have an algorithm that, given a constructive proof of A, returns a constructive proof of B. So, a constructive proof of an implication $A \to B$ is a (constructively defined) function that transforms each proof of A into a proof of B.

In this approach, first proposed by Heyting (*Mathematische Grundlagenforschung: Intuitionismus – Beweistheorie*, Springer-Verlag, Berlin, 1934; revised English translation *Intuitionism*, North Holland, 1971), and Kolmogorov ("Zur Deutung der intuitionistischen Logik", *Mathematische Zeitschrift*, 35:58–65, 1932), different proofs of $A \to B$ are functions from the set of all proofs of A to the set of all proofs of B. This relation can be naturally described in type theory terms: indeed, a type of an object is, crudely speaking, a class to which this object belongs. So, we can view proofs as objects ("terms"), and the corresponding propositions as types. Thus, a proof of the implication $A \to B$ is an object of type $A \to B$—in the sense that it is a type of functions that transform objects of type A (= proofs of A) into objects of type B (= proofs of B). This "propositions as types"–"proofs as terms" (PAT) principle turned out to be very useful in automatic proof verification. It was first successfully used in the late 1960s in the Automath project that enabled computers to check serious mathematical proofs—*e.g.*, all the proofs from *Grundlagen der Analysis*, Edmund Landau's classical book on foundations of mathematical analysis. The ideas developed in Automath made their way into modern proof checkers.

This brief description—largely borrowed from the book under review—shows an exciting evolution and convergence of ideas related to type theory. While many logicians seem to be aware of this story, many researchers are still only familiar with one side of it. For example, specialists in foundations of set theory are quite familiar with Russell's type theory, and may be aware of intuitionistic mathematics, but they may not know much about the relation between intutionistic (constructive) logic and type theory. Specialists in theoretical computer science are usually quite familiar with recursive functions and λ-calculus, but not with set-theoretical types and the use of PAT in proof checking. Specialists in programming languages may be familiar with types and type checking, but are usually not aware of the relation between these (practical) types and more theoretical work that they may have studied in Theory of Computations and AI classes. Specialists in AI are usually familiar with LISP—and maybe with proof checking software, but they are usually not aware of the deep relation between the two and of the relation between these subjects and foundations of set theory.

One reason why the relationship is not well known is that up to know, different aspects of type theory were presented in different forms, sometimes informally. The authors meticulously go over the past formalisms and approaches and describe them in a similar way, so that modern readers will be able to understand all the details of each formalism—and the relation between different type theories and between their various applications.

The result is a very useful book—but it is not easy reading. The main audience is computer science and applied logic folks who are already familiar with λ-calculus. For these folks, the authors provide a very clear and in-depth introduction to other important areas related to type theory such as set theory paradoxes and constructive mathematics. While doing this, the authors had to bring perfect clarity and unambiguity to formalisms like Russell's type theory which were never presented in a completely satisfactorily formal way. This required a lot of work, and the result is definitely of great value to historians of set theory.

The authors also spend a lot of time explaining how the ideas and algorithms of the Automath proof checker are, in effect, the same as in the general type theory approach—and this explanation helps a lot by clarifying much of the heuristics behind proof checkers.

I would therefore strongly recommend this book to all interested folks who have at least some idea of Church's λ-calculus, whether from a logic course or from the recursive functions notion from a Theory of Computation course. Moreover, for those who are interested in one of the aspects of types—in set theory, in AI, in programming languages—but are not yet familiar with λ-calculus, I would still recommend that they read this book—the need to understand this book will be a good motivation to become better acquainted with Church's techniques.

Department of Computer Science, University of Texas at El Paso, El Paso, TX 79968, USA
E-mail address: vladik@utep.edu