Type Theory and Its Applications to Computer Science*

César Muñoz
munoz@nianet.org
National Institute of Aerospace
100 Exploration Way, Hampton, VA 23666.

April 10, 2007

Abstract

Type theory is a mathematical technique widely used in computer science. In the formal methods community, type theory is at the basis of expressive specification languages and powerful proof assistant tools based on higher-order logic. From a practical point of view, type theory has been used to improve the quality of software systems by enabling the detection of errors before they become run-time problems. This article presents a general overview of type theory and its role in the foundation of programming and specification languages.

Introduction

In elementary school we are taught that a *set* is a collection of elements having some characteristics in common. Later, teachers ask whether or not an element belongs to a given set. For example, they state that the *collection of stars in the universe* is a set, and then they ask if the element *moon* belongs to that set or not. The very simple theory of sets behind these two concepts:

1. every predicate is a set, and
2. it is always possible to ask if an element satisfies a predicate,

is known as the *naîve set theory*. In 1902, Russell has shown that the naîve set theory is inconsistent, i.e., it leads to paradoxes [23]. A well-known paradox due to Russell is the following. Let A be the set of all the sets that are not elements of themselves. Is A an element of itself? Both positive and negative answers to this question raise a contradiction.

To solve Russell's paradox, two theories were proposed: the Set Theory of Zermelo-Fraenkel [6] (a.k.a. *set theory*) and the Type Theory of Whitehead-Russell [23] (later simplified by Ramsey and Church [4, 19]) (a.k.a. *type theory*).

In set theory the postulate that every predicate is a set has been replaced by very precise rules of construction of sets. For instance, the comprehension rule that defines a set by a predicate is only allowed over previously constructed sets. In set theory, the "set" of all the sets that are not elements of themselves is simply not a set since it does not respect the comprehension rule.

On the other hand, in type theory the postulate that allows us to ask if an element belongs to a set has been constrained. In this approach, mathematical objects are stratified in several categories, namely *types*. In particular, the type of a set is different from the type of its elements. Since all the elements of a set must have the same type, the question "Is A an element of itself ?" is not a valid question. From the

*This original version of this article appeared in the Quarterly News Letter of the Institute for Computer Applications in Science and Engineering (ICASE), Vol. 8, No. 4, December 1999. The ICASE institute closed down in December 2002 and its Quarterly News Letter is no longer available. The version provided here is an update of the original one. This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 and Cooperative Agreement NCC-1-02043.

type theory point of view, teachers are not always allowed to ask about arbitrary elements on arbitrary sets.

Set theory (together with classical logic) is the standard foundation of modern mathematics. However, type theory, and all its variants, is widely popular in the computer science community. In the rest of this article we present a general overview of type theory and its role in computer science, i.e., the foundation of programming and specification languages.

Type Theory and Its Applications

A minimal type system, known as the *Simple Type Theory*, was proposed by Church in [4]. In that theory, mathematical objects are of two kinds: *terms* and *types*. The terms of the Simple Type Theory are the terms of the λ-calculus, itself being a formalization of partial recursive functions proposed by Church [5]. Types can be basic types or functional types $A \to B$ where A and B are types.

In λ-calculus, terms can be variables, functions, or applications. Only terms that follow a type discipline are considered to be valid. The type discipline is enforced by a set of *typing rules*. A typing rule says, for example, that a function f can be applied to a term x if and only if f has the type $A \to B$ and x has the type A. In that case, the application $f(x)$ has the type B. Thanks to the typing rules, Russell's paradox cannot be expressed in the simple type theory.

Type checking is decidable in the simply typed λ-calculus. That is, it is decidable whether or not a term has a type according to the typing rules.

The Simple Type Theory can be extended straightforwardly with simple data types such as Cartesian products, records, and disjoint unions [3]. For this reason, simple types have been largely used by designers of programming languages. Indeed, most of the modern programming languages support, to some extent, a simply-typed system. In these languages, programs violating the type discipline are considered harmful, and therefore, they are rejected by the compiler. Thus, type checking is a powerful tool to eliminate run-time problems. For instance, in Pascal [10], boolean functions cannot be applied to integers. This restriction happens to be a simply typed rule enforced by the compiler. On the other hand, C [11] supports a more liberal typing discipline; the compiler warns of some violations, but it seldom rejects a program. Almost every C-programmer knows the danger of this flexibility in the language.

When writing formal specifications, the choice of a *typed* language, in opposition to a language based on set theory, is not always evident [12]. In contrast to programs, specifications are not supposed to be executable. Thus, a too restrictive type theory, as for example the simply typed λ-calculus, quickly becomes cumbersome to write enough abstract specifications. Several variants of type theories have been proposed in the literature, most of them are still convenient to write formal specifications and powerful enough to reject specifications that are undesirable [21]. Let us review some extensions to the Simple Type Theory.

Polymorphism and Data Types

A major improvement to the Simple Type Theory, is the System F proposed by Girard [7]. System F extends λ-calculus with quantification over types; that is, it introduces the notion of *type parameters* which is at the basis of the concept of *polymorphism*. In this system, generic data types as list, trees, etc. can be encoded. Type checking is still decidable in a type system that supports polymorphism and abstract data type declarations.

In programming languages, polymorphism allows for the reuse of code defined over structures parameterized by a type. For instance, a sort function is essentially the same whether it works over a list of integers or a list of strings. Polymorphic-typed languages exploit this uniformity without losing the ability to catch errors by enforcing a type discipline.

Although most of the specification languages based on higher-order logic support polymorphism [2, 13, 17, 18], just a few modern programming languages implement it correctly.

The functional programming languages of the ML family [16] are strongly typed languages that support algebraic data types and polymorphism. They use

a type inference mechanism based on Milner's algorithm [15]. Therefore, although these languages are strongly typed, the types of the expressions occurring in programs are automatically inferred by the compiler. Hence, ML programs are almost free of type declarations.

Object-oriented imperative languages such as Eiffel [14], C++ [22], and Java [1] support parametric classes, which is a form of polymorphism. However, object-oriented features, side effects, and polymorphism result in very complex type systems. Eiffel uses a rather complicated and ad-hoc type system, C++ follows the same liberal discipline of C with respect to type checking, and Java only supports single inheritance.

Dependent Types and Constructive Type Theories

Dependent types is the ability to define types that depend on expressions. For instance, in Pascal the type declaration `array[1..10] of integers` is a dependent-type declaration since this type depends on expressions `1` and `10`. A general theory of dependent types, called LP, was proposed by Harper et al. [9].

Dependent types, polymorphism, and inductive data types are supported by a very expressive extension to the λ-calculus called the *Calculus of Inductive Constructions* (CIC). This calculus is the logical framework of the proof assistant system Coq [2]. Type checking is decidable in CIC and the calculus satisfies the strong normalization property, i.e., functions defined in the CIC formalism always terminate. The Calculus of Inductive Constructions also supports the *propositions-as-types* principle of the higher-order intuitionistic logic. According to this principle, a proof of a logical proposition A is the same as a term of type A. This isomorphism between proofs and terms is also known as the *Curry-Howard* isomorphism [8]. In practice, the Curry-Howard isomorphism is used to extract a program from the constructive proof of the correctness of an algorithm. Programs extracted this way satisfy the termination property.

Although very simple dependent types such as arrays are used in most programming languages, general dependent types and constructive types are still hard to handle in programming languages. One notably exception is DML [24], a the conservative extension of ML with a restricted, but practical, form of dependent types.

Subtyping and Other Mysteries

In type theory every object has at most one type. A drawback of this postulate is that an object as the natural number 1 has to be different from the real number 1. A way to handle this problem is to introduce *predicate subtyping* [21], i.e., the ability to define new types by a predicate on previously defined types. For instance, the type *nat* can be defined as a subtype of *real* such that it contains only the numbers generated from 0 and +1. Via predicate subtyping the natural number 1 is also a real number.

The type theory of the general verification system PVS [20] supports predicate subtyping. Unfortunately, general predicate subtyping rends type checking undecidable. In PVS, the type-checker returns a set of type correctness conditions (TCCs) that should be discharged by the user; these TCCs guarantee the type correction of the formal development. In practice, TCCs are not a problem since PVS provides a powerful theorem prover that implements several decision procedures and proof automation tools.

Due to the undecidability problem, general predicate subtyping is not used in programming languages. However, object-oriented programming languages opened the door to an interesting kind of structural subtyping: inheritance. Via inheritance, two structurally different types may share some elements. Related concepts to inheritance are those of *overloading*, that is, the ability to use the same name for different functions, and *dynamic typing*, that is, the ability for objects to change their types during the execution of a program. The formal semantics of all these concepts in a typed framework is still subject of research and controversy.

Summary

Type theory offers a convenient formalism to write specifications and the ability to reject undesirable specifications long before they are refined into actual implementations. In programming languages, type checking allows for the elimination of run-time errors at the compilation phase. Type systems used in specification languages and in programming languages differ in complexity and in expressiveness. Current research in the area includes bringing the benefits of very expressive type systems to programming languages used by practitioners. In order to do that, it is necessary to adapt and simplify the highly mathematical notations of the complex type systems into easily handled programming language features.

About the Author

César Muñoz received his Ph.D. in Computer Science from the University of Paris 7 in November 1997. During his graduate studies he worked as a Research Assistant in the Coq Project at INRIA Rocquencourt. After completing his Ph.D., he spent one and a half years as an International Fellow in the Computer Science Laboratory at SRI International in Menlo Park. He joined ICASE as a Staff Scientist in May 1999. Since January 2003, Dr. Munoz is a senior staff scientist at the National Institute of Aerospace where he leads the Formal Methods Group. His research focuses on the development of formal methods technology for the verification of critical aerospace applications.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[3] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997. Available at `http://www.research.digital.com/SRC`.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[6] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, second printing, second edition, 1984.

[7] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symp., Oslo, Norway, 18–20 June 1970*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, Amsterdam, 1971.

[8] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proof and Types*. Cambridge University Press, 1989.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[10] K. Jensen and N. Wirth. *The Programming Language Pascal*. Springer-Verlag, 1975.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1988.

[12] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? SRC Research Report 147, Digital Systems Research Center, Palo Alto, CA, May 1997. Available at `http://www.research.digital.com/SRC`.

[13] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[14] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[15] R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.

[16] Robin Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1991.

[17] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[19] F. P. Ramsey. The foundations of mathematics. In D. H. Mellor, editor, *Philosophical Papers of F. P. Ramsey*, chapter 8, pages 164–224. Cambridge University Press, Cambridge, UK, 1990. Originally published in *Proceedings of the London Mathematical Society*, 25, pp. 338–384, 1925.

[20] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *Proc. CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, number 656 in Lecture Notes in Computer Science, pages 36–50, Pont-a-Mousson, France, July 1992. Springer-Verlag.

[21] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[22] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[23] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, revised edition, 1925–1927. Three volumes. The first edition was published 1910–1913.

[24] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.