Science | DOI:10.1145/1646353.1646361 Alex Wright

Type Theory Comes of Age

Type systems are moving beyond the realm of data structure and into more complex domains like security and networking.

When the philosopher Bertrand Russell invented type theory at the beginning of the 20th century, he could hardly have imagined that his solution to a simple logic paradox—defining the set of all sets not in themselves—would one day shape the trajectory of 21st century computer science.

Once the province of mathematicians and social scientists, type theory has gained momentum in recent years as a powerful tool for ensuring data consistency and error-free program execution in modern commercial programming languages like C#, Java, Ruby, Haskell, and others. And thanks to recent innovations in the field, type systems are now moving beyond the realm of data structure and into more complex domains like security and networking.

First, a quick primer. In programming languages, a type constitutes a definition of a set of values (for example, "all integers"), and the allowable operations on those values (for example, addition and multiplication). A type system ensures the correct behavior of any program routine by enforcing a set of predetermined behaviors. For example, in a multiplication routine, a type system might guarantee that a program will only accept arguments in the form of numerical values. When other values appear—like a date or a text string—the system will return an error. For programmers, type systems help prevent undetected execution errors. For language implementers, they optimize execution and storage efficiency. For example, in Java integers are represented in the form of 32 bits, while doubles are represented as 64 bits. So, when a Java routine multiplies two numbers, the type system guarantees they are either integers or doubles. Without that guarantee, the runtime would need to conduct an expensive check to determine what kinds of numbers were being multiplied before it could complete the routine.

Benjamin C. Pierce, University of Pennsylvania.

What distinguishes a type system from more conventional program-level verification? First, a type system must be "decidable"; that is, the checking should happen mechanically at the earliest opportunity (although this does not have to happen at compilation time; it can also be deferred to runtime). A type system should also be transparent; that is to say, a programmer should be able to tell whether a program is valid or not regardless of the particular checking algorithm being used. Finally, a "sound" type system prevents a program from performing any operation outside its semantics, like manipulating arbitrary memory locations.

Languages without a sound type system are sometimes called unsafe or weakly typed languages. Perhaps the best-known example of a weakly typed system is C. While C does provide types, its type checking system has been intentionally compromised to provide direct access to low-level machine operations using arbitrary pointer arithmetic, casting, and explicit allocation and deallocation. However, these maneuvers are fraught with risk, sometimes resulting in programs riddled with bugs like buffer overflows and dangling pointers that can cause security vulnerabilities.

By contrast, languages like Java, C# , Ruby, Javascript, Python, ML, and Haskell are strongly typed (or "type safe"). Their sound type systems catch any type system violations as early as possible, freeing the programmer to focus debugging efforts solely on valid program operations.

Static and Dynamic Systems

Broadly speaking, type systems come in two flavors: static and dynamic. Statically typed languages catch almost all errors at compile time, while dynamically typed languages check most errors at runtime. The past 20 years have seen the dominance of statically typed languages like Java, C# , Scala, ML, and Haskell. In recent years, however, dynamically typed languages like Scheme, Smalltalk, Ruby, Javascript, Lua, Perl, and Python have gained in popularity for their ease of extending programs at runtime by adding new code, new data, or even manipulating the type system at runtime.

Statically typed languages have restrictions and annotations that make it possible to check most type errors at compile time. The information used by the type checker can also be used by tools that help with program text-editing and refactoring, which is a considerable advantage for large modular programs. Moreover, static type systems enable change. For example, when an important data structure definition is

PHOTOGRAPH BY SALLIE DEAN SHATZ

changed in a larger program, the type system will automatically point to all locations in the program that also need change. In a dynamically typed language it would be extremely difficult to make such changes in larger programs as it would be not known what other parts are affected by the change. On the other hand, some correct programs may be rejected by a static type system when it is not powerful enough to guarantee soundness.

In an effort to make static type systems more flexible, researchers have developed a number of extensions like interface polymorphism, a popular approach introduced by object-oriented languages like Simula, C++, Eiffel, Java, or C#. This method allows for inclusion between types, where types are seen as collections of values. So, an element of a subtype—say, a square—can be considered as an element of its supertype—say, a polygon—thus allowing the elements of different but related types to be used flexibly in different contexts.

Another form of polymorphism, found in almost all programming languages, is ad hoc polymorphism (also called overloading) where code behaves in different ways depending on the type. This approach has found its fullest expression in Haskell, thanks in part to the efforts of Philip Wadler, professor of theoretical computer science at the University of Edinburgh. "When we designed Haskell, it quickly became clear that overloading was important and that there was no good solution," says Wadler. "We needed overloading for equality, comparison, arithmetic, display, and input."

The Haskell system has evolved considerably over the years, thanks to the contributions of a far-flung group of contributors. "Once we'd come up with the initial idea of type classes, it led to a vast body of work, all sorts of clever researchers coming up with neat extensions to the system, or applying it do things that we'd never thought it could do," says Wadler. Today, Haskell ranks as the programming world's premier case study in ad hoc polymorphism.

The dream of unifying static and dynamic type systems has long fascinated researchers. Today, several computer scientists are probing the possibility of merging these approaches. Wadler is pursuing a promising line of research called blame calculus that attempts to incorporate both static and dynamic typing, while Erik Meijer, a language architect at Microsoft Research, proposes to use "static typing when possible, dynamic typing when necessary."

The theory for refinement types has existed for a long time, but recent progress in automatic theorem proving makes refinement types suddenly practical.

Security Type Systems

In recent years, researchers have also been exploring type systems capable of capturing a greater range of programming errors such as the public exposure of private data. These emerging type systems are known as security type systems. Whereas a traditional type system enforces rules by assigning values to data types, a security type system could apply the same principle of semantic checking to determine the owner of a particular piece of information. Those annotations could then help ensure the integrity of data flowing through the system. Two promising security research projects include the AURA programming language, developed by Steve Zdancewic, associate professor of computer science at University of Pennsylvania, and Jif, a Java-based security-typed language developed by Andrew Myers, associate professor of computer science at Cornell University.

Another interesting application of type checking involves hybridizing type systems and theorem provers. "Historically, there have been two parallel tracks in the software engineering world: type systems and theorem provers. The type systems track has always emphasized lightweight methods," says Benjamin C. Pierce, professor of computer science at the University of Pennsylvania, "but the formal methods people aren't interested in that. Today, they're starting to meet in the middle."

Pierce points to refinement types, which are types qualified by a logical constraint; an example is the type of even numbers, that is, the type of integers qualified by the is-an-even-number constraint. While the theory for refinement types has existed for a long time, only recent progress in automatic theorem proving makes refinement types suddenly practical. A promising security project was recently performed by Andrew D. Gordon, principal researcher at Microsoft Research Cambridge, and colleagues. They added a system of refinement types to the F# programming language and were able to verify security properties of F# implementations of cryptographic protocols by type checking.

While type theory has matured considerably over the past 100 years, it still remains an active research arena for computer scientists. As type systems move beyond the realm of data consistency and into headier computational territories, the underlying principles of type theory are beginning to shape the way researchers think about program abstractions at a deep—even philosophical—level. Bertrand Russell would be proud. C

Further Reading

Hindley, J.R.
Basic Simple Type Theory. Cambridge University Press, New York, 2008.

Pierce, B.
Types and Programming Languages. MIT Press, Cambridge, MA, 2002.

Flanagan, C.
Hybrid type checking. ***SIGPLAN Notices 41***, 1, Jan. 2006.

Cardelli, L.
Type systems. ***The Computer Science and Engineering Handbook***, Allen B. Tucker (ed.). CRC Press, Boca Raton, FL, 1996.

Church, A.
A formulation of the simple theory of types. ***Journal of Symbolic Logic 5***, 2, 1940.

Alex Wright is a writer and information architect who lives and works in Brooklyn, NY. Daan Leijen and Wolfram Schulte of Microsoft Research contributed to the development of this article.