Dynamically Typed Languages

Laurence Tratt, *Bournemouth University*
Roel Wuyts, *IMEC*

The languages discussed in this special issue have a long history, which is perhaps why some have had several different names over the years. One such language is Lisp, the second-oldest programming language. For years, many somewhat dismissively described languages such as Lisp as "scripting languages." Today, we more commonly refer to them as dynamically typed languages, typified by Python and Ruby, and their impact is arguably greater than ever. This issue highlights the practical uses of such languages and shows how they're frequently a vehicle for innovation in the development sphere.

What is a dynamically typed language?

Simply put, a dynamically typed language doesn't require the user to statically specify types—for example, no declarations of `int i`, and so on. However, this doesn't mean that such languages are weakly typed. If, for example, a user tries to add a string to a bool at runtime, an error will immediately be raised. Statically typed languages enforce the declarations of (at least a minimum of) types and notify the user at compile time of type-related errors that must be fixed. Shades of gray also exist between the two extremes: many statically typed languages defer certain checks to runtime, while dynamically typed languages often perform some checks statically to warn users.

Many people familiar with statically typed languages such as Java or C wonder if those who use dynamically typed languages are too lazy to declare types. Furthermore, they assume that not declaring types will decrease the software's quality: "Imagine how many runtime errors will result!" Not surprisingly, they view "prototyping" or "scripting" languages as suitable only for very small or toy programs. However, if these languages are so limited, why do leading companies such as Google publicly proclaim the benefits of large systems written in such languages? There are two answers to this question.

First, current static type systems in mainstream object-oriented languages have little expressive power. For example, although they prevent users from adding a string to a bool, they don't prevent them from accessing the first element of an empty list, creating off-by-one errors, or using null pointers. In fact, static type systems can't detect most common programming errors. Yet for such systems to work, developers must spoon-feed (and sometimes strong-arm) the types at development time so that the type system—which is in essence a separate system from the main language—knows what the code is really doing.

The waters are muddied further because even very simple approaches to testing capture

0740-7459/07/$25.00 © 2007 IEEE

virtually all, if not all, the errors that a static type system would capture. Plus, as many of us unfortunately know, even extensive testing has a hard job identifying off-by-one errors. Don Roberts formulated it perfectly at OOPSLA 2005: "Static types give me the same feeling of safety as the announcement that my seat cushion can be used as a floatation device" (see http://martinfowler.com/bliki/OOPSLA2005.html).

Second, static type systems—as their name suggests—implicitly constrain the programmer at both compile time and runtime. In other words, static type systems tend to ossify programs, making them less flexible, more difficult to work with, and less amenable to change. When a user wants to modify one aspect of a system, that system's types could force numerous rewrites in unrelated parts of the system. (Think about all the problems for subclasses once overloading comes into the picture.) These rewrites can seriously inhibit any system experimentation. In an age where we strive for reusable software, and where software must adapt quickly to external changes, programming languages that reduce agility can be a significant disadvantage.

However, this issue doesn't aim to be one-sided zealotry for dynamically typed languages. Sometimes, users will pay whatever it costs (in time or money) for extra software reliability, such as when requesting software for a nuclear power plant or an airplane's autopilot function. So static type systems have their part to play, but they're by no means the only option. As in most areas of life, the majority of software needs to balance various factors, and in many cases the advantages of dynamically typed languages come to the fore. Rapid application development, with its emphasis on quickly producing incremental results, showcases this in many areas.

A distinguished history

People often say of the Velvet Underground that although they didn't sell many records, every person who bought one of their records started a band. Two dynamically typed languages deserve a similar epithet: Lisp and Smalltalk. While neither ever quite gained mainstream success, both had a huge influence on later languages—including many statically typed languages—and both are still used today. They're an important part of the long and distinguished history of dynamically typed languages.

Lisp was the source of many programming-language firsts, such as automatic memory management. It's the ancestor of all extant functional languages, although, interestingly, most of its descendants are statically typed languages. One notable exception is Scheme, a very clean Lisp dialect with a sophisticated macro mechanism and module system.

Smalltalk is a purely object-oriented language that pioneered the bytecode virtual machine and graphical integrated development environments, long before C++ or Java were thought of. It inspired prototype-based (classless) languages, notably Self, a language whose many advances in VM technology we now take for granted via Java and.NET.

An excellent example of a highly performant dynamically typed language is Lua. This small but powerful language is extremely well suited to being embedded into existing systems, as shown in the article "Traveling Light, the Lua Way."

However, for many years, dynamically typed languages were viewed as slow, unreliable, and suitable only for small throw-away tasks. Although a handful of people used dynamically typed languages for complex systems—such as the banks that embraced Smalltalk in the beginning of the '80s—it was generally felt that "real" programmers used statically typed languages.

As with many other areas of computing, the emergence of the Web changed things. Creating dynamic Web pages in languages such as C was painful, and the dynamically typed language Perl quickly became synonymous with Web systems. However, Perl's origins in systems administration software—where small, inscrutable solutions were often the order of the day—prevented wider scale adoption.

In the late '90s, the object-oriented language Python—in many ways, a more traditional language than Perl—came to the fore as a dynamically typed language suitable for both small and large systems. Python continues to be used in a variety of applications. The article "A Common Medium for Programming Operations-Research Models" shows how to apply Python to domains traditionally dominated by statically typed languages. Today, languages such as Ruby continue to emerge and prove themselves useful in new spheres.

The present

For some time, the scarce resource when developing software hasn't been hardware but

Lisp and Smalltalk are an important part of the long and distinguished history of dynamically typed languages.

About the Authors

Laurence Tratt is a senior lecturer at Bournemouth University and is a software consultant. He's also the chief designer and implementer of the Converge programming language. His research interests include modeling (UML and MDD)—he has contributed to several international standards—and domain specific languages. He received his PhD from King's College London. He's a member of the *IEEE Software* Advisory Board. Contact him at laurie@tratt.net; http://tratt.net/laurie.

Roel Wuyts is a senior scientist at IMEC (Leuven, Belgium) and a part-time professor at KU Leuven. His research interests include declarative metaprogramming, language symbiosis, components and composition, and development environments. He's especially interested in dynamic languages—primarily Smalltalk, Prolog, Self, and Scheme. He received his PhD from the Programming Technology Lab at the Vrije Universteit Brussel. Contact him at IMEC, Kapeldreef 75, B-3001 Leuven, Belgium; roel.wuyts@imec.be.

rather developer time. With hardware development continuing to follow, and sometimes surpass, Moore's law, execution performance or memory consumption are no longer the major bottlenecks for most applications (excluding embedded systems or very large application servers). This removes what many people consider dynamically typed languages' major obstacle—performance. Although this perception has often been at odds with reality (in several applications, CLOS runs nearly as fast as C, and production Smalltalk systems have execution speeds comparable to Java), in the modern world, performance is generally irrelevant. Most applications running on a modern desktop computer are fast enough, no matter what language was used to write them. So, a chief benefit of dynamically typed languages is significantly reduced development time.

A fundamental property of dynamically typed languages is their flexibility, and from the earliest days of Lisp, developers have used this property to turn the languages into domain-specific languages. Of course, you can implement DSLs in statically typed languages, but the complex syntax and semantics that generally accompany static type systems make this a more arduous task. DSLs in such languages often resort to a parser or XML/XSLT-based approach, which defines a custom syntax and translates it into another language. Needless to say, developing such systems isn't for the faint of heart.

Building a DSL in a dynamically typed language is far easier, as shown in "Building Domain-Specific Languages for Model-Driven Development." The article also highlights that statically typed systems (such as modeling software) are often implemented in dynamically typed languages. Because the syntax of most dynamically typed languages is quite small and flexible, we can often forget parser and transformation approaches altogether. The resulting DSL is therefore more maintainable and amenable to evolution, better fitting the domain users' ever-changing needs.

Last, but certainly not least, we return to the development of Web applications. Even though scripting languages unlocked Web development's potential for many users, application development on the Web is still much more challenging than traditional approaches. Simple user features such as the browser back button can wreak havoc on age-old programming models. The article "Seaside: A Flexible Environment for Building Web Applications" shows how to combine the functional programming concept of continuations with pure object-oriented principles to make developing sophisticated Web applications as easy as developing desktop applications.

In addition to solving "old" problems, dynamically typed languages are also playing their trump cards of easy use and deployment to help develop Web 2.0 or the Semantic Web. "A Flexible Integration Framework for Semantic Web 2.0 Applications" discusses the benefits of dynamically typed languages in this domain.

Looking forward

The outlook for dynamically typed languages is better than ever before. Industrial uptake is increasingly rapid—for example, the financial sector is using dynamically typed languages in critical systems such as JPMorgan's Kapital system. Supporting new types of bonds and funds faster than your competitors results in multibillion-dollar gains. Political barriers to use are also diminishing.

Innovation in the dynamically typed languages sphere remains strong—for example, advances in languages for distributed, concurrent, or parallel programming. We also expect new, beneficial features to continue to emerge and to gradually trickle down to the more conservative statically typed languages. We hope that this special issue gives you a flavor of not only how dynamically typed languages can help you and your organization today, but how they might drive future innovation.