The Next 7000 Programming Languages

Robert Chatley[1], Alastair Donaldson[1], and Alan Mycroft[2](✉)

[1] Department of Computing, Imperial College, London, UK
{robert.chatley,alastair.donaldson}@imperial.ac.uk
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
alan.mycroft@cl.cam.ac.uk

Abstract. Landin's seminal paper "The next 700 programming languages" considered programming languages prior to 1966 and speculated on the next 700. Half-a-century on, we cast programming languages in a Darwinian 'tree of life' and explore languages, their features (genes) and language evolution from the viewpoint of 'survival of the fittest'.

We investigate this thesis by exploring how various languages fared in the past, and then consider the divergence between the languages *empirically used in 2017* and the language features one might have expected if the languages of the 1960s had evolved optimally to fill programming niches.

This leads us to characterise three divergences, or 'elephants in the room', where actual current language use, or feature provision, differs from that which evolution might suggest. We conclude by speculating on future language evolution.

1 Why Are Programming Languages the Way They Are? and Where Are They Going?

In 1966 the ACM published Peter Landin's landmark paper "The next 700 programming languages" [22]. Seven years later, Springer's "Lecture Notes in Computer Science" (LNCS) was born with Wilfred Brauer as editor of the first volume [5]. Impressively, the contributed chapters of this first volume covered almost every topic of what we now see as core computer science—from computer hardware and operating systems to natural-language processing, and from complexity to programming languages. Fifty years later, on the occasion of LNCS volume 10000, it seems fitting to reflect on where we are and make some predictions—and this essay focuses on *programming languages and their evolution*.

It is worth considering the epigraph of Landin's article, a quote from the July 1965 American Mathematical Association Prospectus: "... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas". Getting an equivalent figure nowadays might be much harder—our title of 'next 7000 languages' is merely rhetorical.

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 250–282, 2019.
https://doi.org/10.1007/978-3-319-91908-9_15

On one hand, Conway and White's 2010 survey[1] (the inspiration behind RedMonk's ongoing surveys) found only 56 languages used in *GitHub* projects or appearing as *StackOverflow* tags. This provides an estimate of the number of languages "in active use", but notably excludes those in large corporate projects (not on GitHub) particularly where there is good local support or other disincentives to raising programming problems in public. One the other hand, programming languages continue to appear at a prodigious rate; if we count every proposed language, perhaps including configuration languages and research-paper calculi, the number of languages must now be in six digits.

The main thrust of Landin's paper was arguing that the next 700 languages after 1966 ought to be based around a language family which he named ISWIM and characterised by: (*i*) nesting by indentation (perhaps to counter the Fortran-based "all statements begin in column 7" tendency of the day), (*ii*) flexible scoping mechanisms based on λ-calculus with the ability to treat functions as first-class values and (*iii*) imperative features including assignment and control-flow operators. Implicit was an expectation that there should be a well-defined understanding of when two program phrases were semantically equivalent and that compound types such as tuples should be available.

While the lightweight lexical scope '{...}' is now often used for nesting instead of adopting point (*i*),[2] it is entertaining to note that scoping and control (*ii*) and (*iii*) have recently been drivers for enhancements in Java 8 and 9 (e.g. lambdas, streams, `CompletableFutures` and reactive programming).

Landin argued that ISWIM should be a family of languages, parameterised by its 'primitives' (presumably to enable it to be used in multiple application-specific domains). Nowadays, domain-specific use tends to be achieved by introducing abstractions or importing libraries rather than via adjustments to the core language itself. Indeed there seems to be a strong correlation between the number and availability of libraries for a language and its popularity.

The aim of this article is threefold: to explore trends in language design (both past, present and future), to argue that Darwinian evolution by fitness holds for languages as well as life-forms (including reasons why some less-fit languages can persist for extended periods of time) and to identify some environmental pressures (and perhaps even under-occupied niches) that language evolution could, and we argue should, explore.

Our study of programming-language niches discourages us from postulating a universal core language corresponding to Landin's ISWIM.

1.1 Darwinian Evolution and Programming Languages

We start by drawing an analogy between the evolution of programming languages and that of plants colonising an ecosystem. Here species of plants correspond to

[1] http://www.dataists.com/2010/12/ranking-the-popularity-of-programming-langauges/ [sic].

[2] Mainstream languages using indentation include Python and Haskell.

programming languages, and a given area of land corresponds to a family of related programming tasks (the word ‘nearby’ is convenient in both cases).

This analogy enables us to think more deeply about language evolution. In the steady-state (think of your favourite bit of land—be it countryside, scrub, or desert) there is little annual change in inhabitation. This is in spite of the various plants, or adherents of programming languages, spreading seeds—either literally, or seeds of dissent—and attempting to colonise nearby niches.

However, things usually are not truly steady state, and invasive species of plants may be more fitted to an ecological niche and supplant current inhabitants. In the programming language context, invasive languages can arise from universities, which turn out graduates who quietly adopt staid programming practices in existing projects until they are senior enough to start a new project—or refactor[3] an old one—using their education. Invasive languages can also come from industry—how many academics would have predicted that, by 2016 according to RedMonk, JavaScript would be the most popular language on GitHub and also be most tagged in StackOverflow? A recent empirical study shows that measuring popularity via volume of code in public GitHub repositories can be misleading due to code duplication, and that JavaScript code exhibits a high rate of duplication [24]. Nevertheless, it remains evident that JavaScript is one of the most widely used languages today.

It is useful here to distinguish between the success of a species of plant (or a programming language) and that of a gene (or programming language concept). For example, while pure functional languages such as Haskell have been successful in certain programming niches the idea (gene) of passing side-effect-free functions to *map*, *reduce*, and similar operators for data processing, has recently been acquired by many mainstream programming languages and systems; we later ascribe this partly to the emergence of multi-core processors.

This last example highlights perhaps the most pervasive form of competition for niches (and for languages, or plants, to evolve in response): climate change. Ecologically, an area becoming warmer or drier might enable previously non-competitive species to get a foothold. Similarly, even though a given programming task has not changed, we can see changes in available hardware and infrastructure as a form of climate change—what might be a great language for solving a programming problem on a single-core processor may be much less suitable for multi-core processors or data-centre solutions.

Amusingly, other factors which encourage language adoption (e.g. libraries, tools, etc.) have a plant analogy as symbiotes—porting (or creating) a wide variety of libraries for a language enhances its prospects.

The academic literature broadly lumps programming languages together into *paradigms*, such as *imperative*, *object-oriented* and *declarative*; we can extend our analogy to view paradigms as being analogous to major characteristics of plants, with languages of particular paradigms being particularly well-adapted to certain niches; for example xerophytes are well-adapted for deserts, and functional

[3] Imagine the discussions which took place at Facebook on how to post-fit types to its one million lines of PHP, and hence to the Hack programming language.

languages are well-suited to processing of inductively defined data structures. Interestingly, the idea of *convergent evolution* appears on both sides of the analogy, in our example this would be where two species had evolved to become xerophytes, despite their most recent common ancestor not being a xerophyte. Similarly language evolution can enable languages to acquire aspects of multiple paradigms (Ada, for example, is principally an imperative language despite having object-oriented capabilities, and C# had a level of functional capabilities from the off, amplified by the more-recent LINQ library for data querying).

Incidentally, the idea of a programming-language ecosystem with many niches provides post-hoc academic justification for why past attempts to create a 'universal programming language' (starting back as far as PL/I) have often proved fruitless: a language capable of expressing multiple programming paradigms risks becoming inherently complex, and thus difficult to learn and to use effectively. A central cause of this complexity is the difficulty of reasoning about *feature interaction*. A modern language that has carefully combined multiple paradigms since its inception is Scala. However, due to the resulting flexibility, there can be many different stylistic approaches to solving a particular programming problem in Scala, using different elements of the language. The language designer, Martin Odersky, describes Scala as "... a bit of a chameleon. ... depending at [sic] what piece of code you look at, Scala might look very simple or very complex."[4]

Finally, there is the issue of *software system* evolution. Just as languages evolve, a given software system (solution to a programming problem) is more likely to survive if it evolves to exploit more powerful concepts offered by later versions of a language. It is noteworthy that tool support often helps here, and we observe the growing importance of tools in supporting working with, adding to and transforming large programs in a given language.

We discuss some of these ideas more concretely in Sect. 3 but to summarise, the main external (climate-change) pressures on language evolution as we currently see them are:

- the change from single-core to multi-core and cloud-like computing;
- support for large programs with components that change over time;
- error resilience, helping programmers to produce reliable software;
- new industrial trends or research developments.

Conceptual Framework. In our setting the principal actors are *programming tasks* which are implemented to produce *software systems* using *programming languages*; the underlying available range of *language concepts* and *hardware and systems models* continue to change, and together with fashion (programmer-perceived and industrial views of fitness) drive the mutual evolution of programming languages and software systems.

We see evolution as 'selection of the fittest' following mutation (introduction of new genes etc.). While the mechanism for mutation (human design in programming languages vs. random mutation in organisms) differs this does not affect

[4] http://www.scala-lang.org/old/node/8610.

the selection aspect. While all living things undergo evolution, we centre on plant analogies as these help us focus on colonies rather than worrying about individual animal conflicts. 'Fitness' extends naturally: it captures the probability that adopting a given programming language in a project will cause programmers to report favourably upon it later—just as botanical fitness includes the probability a seed falling in a niche will germinate and mature to produce viable seeds itself. We discuss aspects of fitness later (e.g. ease of programming).

1.2 Paper Structure

We start by taking a more detailed look at history (Sect. 2), discuss the factors that drive programming language evolution (Sect. 3), and review the most popular programming languages at time of writing, according to the RedMonk, IEEE Spectrum and TIOBE rankings (Sect. 4). Our analysis identifies three 'elephants in the room'—language trends that rather conflict with a 'survival of the fittest' viewpoint—which we discuss in some detail (Sect. 5). We conclude by speculating on future language evolution (Sect. 6).

2 What's New Since 1966?

By 1966, much of the modern gene-pool of programming language concepts was well-established. We had Fortran, Algol 60 and Lisp capturing familiar concepts, e.g. stack-allocated local variables, heap-allocated records, if-then-else and iteration, named procedures and functions (including recursion), both static and dynamic typing, and a peek around the corner to 1967 and 1968 gave us references to structured data with named components (Algol 68) and object-orientation and subtyping (Simula 67).[5] The historic importance of COBOL should not be understated: from 1959 it provided a way to describe, read, print and manipulate fixed-format text and binary records—which fed both into language design (records) and databases.

Some early divergences remain: compare programming paradigms (functional, relational, object-oriented etc.) with the four major groups of plants (flowering plants, conifers, mosses and ferns). Another programming-language example is between static and dynamic typing. Modern fashion in real-world programming appears once again to be embracing dynamic typing for practical programming (witness the use of JavaScript, Python, Ruby and the like), often acting as glue languages to connect libraries written in statically typed languages.

We now briefly outline what we see as some of the main developments in programming languages, and drivers of programming language evolution, since these early days.

[5] Landin emphasised the λ-calculus as a theoretical base for programming languages, and did not mention the richer notion of scoping which object-orientation adds.

What We Do not Cover. Space precludes us from covering all such languages and drivers. Our selection is guided by those languages currently most used for mainstream software development, and their ancestors. As a result, there are certain important categories of language that we do not cover, e.g. logic and probabilistic programming languages.

2.1 Tasks, Tools and Teams

In the early days of computing, computers were mainly used for business processing or for scientific endeavour. Over the years, the range of problems that programming is applied to has exploded, along with the power and cost-effectiveness of computing hardware. As programming languages have evolved to inhabit these niches, the range of people who use them and the situations that they use them in has expanded, and conversely the widening set of applications has encouraged language evolution: there has been an increasing need for languages to provide features that help programmers to manage complexity, and to take better advantage of computing resources.

Another decision point in choosing a language is "get it working" versus "get it right" versus "get it fast/efficient". In different situations, each might be appropriate, and the software-system context, or niche, determines the fitness of individual languages and hence guides the language choice. A quick script to do some data-processing is obviously quite different from an I/O driver, or the control system of a safety-critical device.

Software was initially developed by one person at one computer. Now it is developed by distributed teams, often spanning across continents or organisations. The size of software systems has also increased massively. From systems running on one machine, distributed systems can now execute across hundreds or thousands of machines in data-centres around the world, and comprise tens of millions of lines of code. Language implementations have evolved to help humans manage this complexity, e.g. by providing sophisticated support for packages and modules, including dynamic loading and versioning. Some languages (or runtimes) provide these as core functionality, for example Java and C# provide the ability to dynamically load classes from jar files or assemblies. For other languages, accompanying tools have been developed that help to manage and compose dependencies, e.g. 'npm' (Node Package Manager) for Node.js. Such language features and tools can help to avoid so-called "dependency hell" ("DLL hell" in Windows applications), whereby an application depends on a given shared library that, due to poorly planned sets of inter-component dependencies, in turn depends on an intricate mixture of specific versions of additional library components, some of which conflict with one another, and many of which are in essence irrelevant to the application under construction.

The way that teams work to develop software has also changed. Single-person approaches were succeeded by waterfall-style development processes for managing multi-person software projects, which in turn have been largely superseded (for all but the most safety-critical systems) by more *agile* approaches such as eXtreme Programming [4]. Agile methods require analysis, testing and transformation tools to support frequent and reliable change, and to provide the rapid feedback essential for developer productivity. Languages where these are well supported have a natural advantage in the ecosystem.

Finally, one of the biggest changes since 1967 has been in the tools we use to support programming—enabled, of course, by the vast increase in computing power over this time. While classical editors, compilers and (static) linkers are still present there has been an explosion in new forms of tool support and their use: integrated development environments (IDEs) including support for code analysis and refactoring, version control systems, test generators, along with tools for dynamically linking to code and data outwith the project. These evolve much more quickly and largely independently of the huge code bases in software systems; the latter in general only evolve incrementally to capture software-feature change (and even more slowly in reflecting their underlying programming-language evolution). We claim that appropriate tool support is a strong factor in programming-language fitness, and hence for language choice in a given software project.

2.2 Systems Programming and the Rise of C

The success of the C programming language and that of Unix are inseparably intertwined. Ritchie [35] writes: "C came into being in the years 1969–1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972." In ecosystem terms, C is significant as it successfully out-competed rival languages to become almost the only widely used systems programming language. Gabriel remarks that "Unix and C are the ultimate computer viruses" [11], arguing that their "worse-is-better" design, where implementation simplicity is favoured over other features such as full correctness and interface simplicity, makes Unix and C so easy to port that they virally infected practically all systems. Alluding to natural selection, Gabriel writes: "worse-is-better, even in its straw-man form, has better survival characteristics than the-right-thing".

Designed for relatively low-level systems programming, C provides mechanisms for fine-grained and efficient management of memory—needed to build operating systems and device drivers, or for programming resource-constrained environments such as embedded systems. C is usually the first higher-than-assembly-level language supported on a new architecture. Regarded for a long time (and still to a large degree) as an inevitable price for its high performance, C is an *unsafe* language: run-time errors, such as buffer overflows, are not typically caught by language implementations, so a program may continue to execute after an erroneous operation. This provides wide scope for attacks, e.g. control-flow hijacking and reading supposedly confidential data. A botanical analogy might be the prevalence of cacti in deserts, when some would prefer orchids as prettier and lacking spines; we have to either adjust the niche—requiring beauty (or security), contribute more to fitness by human intervention, or produce a new plant species (or language) that better fits the existing niche while having the desired characteristics.

Certainly C has been one of the most influential programming languages, influencing the syntax, and to some degree semantics, of C++, Java and C#, and forming the basis of parallel programming languages such as CUDA (NVIDIA's proprietary language for graphics-processing unit (GPU) programming) and OpenCL (an industry-standard alternative). Moreover, C continues to be very widely used; arguably much more so than it should be given its unsafe nature. We return to this point in Sect. 5.1.

2.3 Object-Orientation and the Rise of Java

The object-oriented paradigm, where domain entities are represented as objects that communicate by sending messages, became the dominant style of commercial software development in the 1990s. Building on Kay's original ideas, embodied in Smalltalk, came a number of very influential languages, including Delphi, Eiffel, Oberon, Self, and Simula 67. These impacted the design of what are now the most widely used object-oriented languages—Java, C++ and C#—which continue to thrive while many of these earlier languages are now extinct or restricted to communities of enthusiasts. (We acknowledge that some purists question whether languages such as Java, C++ and C# are truly object-oriented—e.g. they fail the so-called "Ingalls Test"[6] [30, Sect. 11.6.4]—but they are widely regarded as belonging to the object-oriented paradigm.)

We also see influence in the growth of tooling, with the modern day Eclipse IDE having its roots in IBM's VisualAge for Java, which in turn evolved from VisualAge for Smalltalk. A significant driver of Java's early spread was its "write once, run anywhere" philosophy, whereby a Java compiler generates code that can run on any conformant implementation of the *Java Virtual Machine* (JVM). Support for the JVM by web browsers, and the popularity of 'applets' embedded in web-pages drove the adoption of Java. Its clean design and safe execution made it a popular teaching language in universities.

Other features also drove adoption: Java, C++ and C# provided support for exceptions—meeting a practical need for mainstream languages to capture this idiom. Java and C# embraced ideas originating in languages such as Lisp and Smalltalk—automatic memory-management (garbage collection) and the idea of *managed run-time environment* abstracting away from the underlying operating system (or browser!). There remains a tension between C/C++, with manual allocation and the associated lack of safety, and Java and C# which, while safe, can be problematic in a resource-constrained or real-time environment.

Note that there are two separate genes, representing class-based inheritance (originating in Simula) and prototype-based inheritance (originating in Self, initially a dialect of Smalltalk) for object-oriented languages. There are arguments in favour of each: the former allows static typing and is used by more main-

[6] https://www.youtube.com/watch?v=Ao9W93OxQ7U, 26 min in.

stream languages; the latter is used in JavaScript, and therefore in some sense more successful!

2.4 Web Programming, the Re-emergence of Dynamic Typing, and the Rise of JavaScript

Although some early programming languages were *dynamically* typed—notably Lisp—the focus of much research into programming languages in the 70s, 80s and 90s was on statically typed languages with stronger and richer type systems. In industry, relatively strongly, statically typed, languages also became increasingly popular, with dynamically typed languages largely restricted to scripting and introductory programming.

The massive change since then has been the importance of programming for the web, through languages such as JavaScript and PHP, which are dynamically typed. Web programming is so pervasive that JavaScript is now one of the most widely used programming languages. From a language-design perspective this is somewhat ironic. Although a lot of high-end research into dynamically typed programming languages had been conducted (e.g. notably working with systems like CLOS [13], and building on those with languages like Self [47]), in fact Brendan Eich designed and implemented JavaScript (then named Mocha) in 10 days, seemingly with no recourse to these research endeavours [42].

Other dynamically typed languages have developed a strong following including Python and Ruby. Both are general-purpose languages, but Python has become particularly popular among scientists [33], and a major driver of Ruby adoption was the *Rails* framework of Heinemeier Hansson [36] designed to facilitate server-side web-application programming and easy database integration. We reflect further on the popularity of dynamically typed languages in Sect. 5.2.

2.5 Functional Programming Languages

Functional languages, where function abstraction and application are the main structuring regime, originated in Lisp. For many years, functional programming languages lived in an "enthusiasts' ghetto", attracting strong supporters for specific programming areas, but their discouraging of mutating existing values was a step too far for many mainstream programmers. Recently however, their ideas appear to be becoming mainstream. Two of the most influential functional languages, Haskell and ML, are currently among the most widely used functional languages (with the OCaml dialect enjoying the most use in the case of ML), and both languages being used in quantitative (equity) trading applications in banks, who argue that these languages allow their *quant* analysts to code more quickly and correctly.[7] One justification for this resurgence is that concurrency and mutation appear hard for programmers to use together in large systems, needing error-prone locks or hard-to-document whole-program assumptions of when and how data structures can be modified.

[7] https://adtmag.com/Ramel0911.

Functional languages have also inspired so-called "multi-paradigm" languages, principally F# and Scala, both of which feature first-class functional concepts; these in turn have been incorporated into mainstream object-oriented languages, most notably the LINQ extensions to C#, and lambdas in Java 8 and C++11.

Even aspects (genes) of functional languages which previously seemed abstruse to the mainstream have been incorporated into modern general-purpose languages. For example Java 8's streams incorporate the ideas of lazy evaluation, deforestation and the functional idiom of transforming one infinite stream into another.

2.6 Flexible Type Systems

At the time of Landin's article, and indeed for most of the decade following it, there was a split between, on one hand, dynamically typed languages such as Lisp, which checked types at run time at the cost of reduced execution efficiency and unexpected errors and, on the other hand, statically typed languages giving increased efficiency and the possibility of eliminating type errors at compile time (even if this had holes, such as in C and Algol 60). However, such static type systems were often inexpressive; a running joke at the time of one author's PhD was that, e.g. in Pascal one had to write functions only differing in types for finding the lengths of integer lists and string lists so that the compiler could generate duplicated code. The late 1970's saw parameterised types, and polymorphically typed (or 'generic') functions for operating over them. The language ML (originally part of the Edinburgh LCF theorem-proving system) was hugely instrumental here. In a sense ML was almost exactly "ISWIM with types", albeit without syntax-by-indentation. Types have continued to grow in expressiveness, with type systems in Java and C# including three forms of polymorphism (generics, overloading and (bounded) sub-type polymorphism) all in the same language.

More heavyweight type disciplines, such as dependently typed languages (e.g. Agda, Coq) remain firmly away from the mainstream, in spite of the high precision that they offer and their potential link to program verification (see Sect. 6.2). There is an argument that very precise types can end up revealing details about a system's internals, and that the desire for a particular type structure can have a more direct influence on program structure than might really be desirable; these have been dubbed the "Visible Plumbing Program" and the "Intersection Problem", respectively [9].

2.7 Parallelism and the Rise of Multi-core

The appearance of multi-core x86 processors from 2005 was, in retrospect, hugely disruptive to programming languages. While parallel processing was not new (e.g. High Performance Fortran offered relatively sophisticated language support for parallel programming [20]), multi-core (and later GPU) technology caused everyday programmers to want their language to support it. This world feels a little like Landin's in 1966—there are many language features and libraries offering support for various aspects of parallelism. Dedicated languages and libraries include Cilk Plus, Threading Building Blocks and OpenMP for multi-core CPUs, and OpenCL, CUDA, Vulkan, Metal and RenderScript for targeting GPUs. Within a single language there can be many mechanisms by which to exploit parallelism: in Java one can use parallel streams, an executor service, create one's own thread objects; in Haskell there are also multiple approaches [27]. More than a decade on, language support for parallelism remains patchy and has converged less than we might have hoped, a point to which we return in Sect. 5.3.

2.8 Domain-Oriented Programming Languages

Just as in Landin's day, many languages have been created for solving problems in particular domains. While Turing completeness means that we should be able to apply any general purpose language to any programming task, domain-specific languages often offer more convenient syntax and library support than would be possible in a mainstream language, with examples including spreadsheets, SQL, MATLAB and R, along with scripting languages for computer-game and graphics-rendering engines.

3 Observed Programming Language Evolution

We now re-cast the changes of the previous section as language evolution pressures, discussing: the factors that keep programming languages alive (Sect. 3.1), the forces that lead to language evolution (Sect. 3.2), and cases where languages have become practically extinct due to not having evolved (Sect. 3.3).

3.1 Factors that Keep Programming Languages Alive

Although the landscape of programming languages evolves, many languages take root and stick around. We observe several forces that keep languages alive. In the evolutionary model, these 'forces' sum to contribute to the overall fitness of a language in a given niche.

Legacy Software. The amount of software in the world increases day by day. New systems (and modules within systems) are created at a much faster rate than they are deleted. Existing code that performs valuable business functions needs to be updated and upgraded to provide additional functionality or to integrate with other systems. Replacing a system wholesale is a major decision, and costly for large systems. The need to tend to existing successful systems, using whatever language they were originally written in, is a strong force in keeping languages alive. Even though COBOL is often perceived as a dead language, as recently as 2009, it was estimated to have billions of lines of code in active use[8] and some of these surely remain, even if they are not widely advertised.

[8] http://skeptics.stackexchange.com/questions/5114/did-cobol-have-250-billion-lines-of-code-and-1-million-programmers-as-late-as-2 [sic].

Community. Enthusiasm and support for a particular programming language is often fostered if there is an active and encouraging community around it. This can be helpful in encouraging beginners, or in supporting programmers in their daily tasks, by providing advice on technical problems as they come up. Language communities, like any social group, tend to reflect the design parameters of the language they discuss; some are seen as formal and academic and others more down-to-earth.. For example, there is a saying in the Ruby community, which has a reputation for being supportive and helpful: "Matz [the language creator] is nice and so we are nice".[9] In contrast, the community that has grown up around Scala is perceived to be much more academically focused, biased towards more mathematical thinking and language, and sometimes perceived as less welcoming to people from a "general developer" background. In his (somewhat polemical) ScalaDays 2013 keynote[10], Rod Johnson gives his impressions of the Scala community as being somewhere where "there do seem to be quite a few people who aren't highly focused on solving real world problems" and where some members appear to have the opinion that "ignorance should be punished".

Ease of Getting Started. In order for a new language to take hold, it helps if it is easy for new programmers (either novices, or just newcomers to the language in question) to get started quickly. The speed with which a new programmer can write and run their first program depends on many things: the simplicity of language design, clarity of tutorials, the amount one needs to learn before getting started, and the support of tooling (including helpful error messages, a subject taken to heart by languages like Elm). This in turn can affect the longevity of a language—as this depends on a continued influx of new programmers.

Habitability. In his book *Patterns of Software* [12], Gabriel describes the characteristic of *habitability*: "Habitability makes a place livable, like home. And this is what we want in software—that developers feel at home, can place their hands on any item without having to think deeply about where it is." If a language has widely adopted idioms, and projects developed in that language are habitually laid out according to familiar structure then it easy for programmers to feel at home. Languages that promote "one way to do things" may help to engender this consistency. Habitability may also come from having a common tool-ecosystem[11]. For example, if we download a Java project and find that it is structured as a Maven[12] project, then it is easy to locate the source, tests, dependencies, build configuration, etc., if we are familiar with other Maven projects. Similarly in modern Ruby projects we might expect a certain structure and the

[9] https://en.wiktionary.org/wiki/MINASWAN .

[10] https://www.youtube.com/watch?v=DBu6zmrZ_50 particularly from 21 min in.

[11] Note that other uses of 'ecosystem' in this paper refer to the ecosystem of languages competing for a programming niche, but the 'tool-ecosystem' refers to the toolset available to support programming in a given language—we earlier noted that this improved the fitness of a given language by behaving as a symbiote.

[12] https://maven.apache.org/.

use of Bundler[13] to perform a similar role. These sort of tools are often only de-facto standards, as a result of wide adoption, but consistent project layout, build process, etc., provided by standard tools, reduces cognitive load for the developer, and in turn may make a programmer feel more at home working in a given language, especially when coming to work on a new project [14].

Libraries. The availability of libraries of reusable code adds to the ease of getting things done in a particular language. Whether we want to read a file, invoke a web service over HTTP, or even recognise car licence plates in a photograph, if we can easily find a library to do so in a particular language, that language is appealing for getting the job done in the short term. Some languages come with rich standard libraries as part of their distribution. In other cases the proliferation of community-contributed libraries on sites like GitHub leads to a plethora of options. When there are a large number of libraries available, it often becomes part of the culture in the community centred around a particular language to contribute. The plentiful supply of libraries begets the plentiful supply of libraries. We note, however, that recent trends to rely on third-party libraries for even the simplest of routines can lead to problems. For example, the removal of the widely used ‘left-pad’ library from the ‘npm’ JavaScript package manager caused a great deal of chaos.[14]

Tools. Although language designers may hope or believe that their new language allows programming tasks to be solved in new, perhaps more elegant ways, it is not the language alone that determines the productivity of a programmer at work. A programmer needs to write and change code, navigate existing code bases, debug, compile, build and test, deploy their code, and often integrate existing components and libraries, in order to get their work done and to deliver working features into the hands of their users. A programming language therefore exists within a tool-ecosystem.

Java, for example, is a very popular language that is highly productive for many development teams. This is not only down to the design of the language itself—some might argue that it is in fact in spite of this, as Java is often seen as relatively unsophisticated in terms of language features. It is also—indeed, perhaps largely—due to the supply of good tools and libraries that are available to Java developers. Anecdotally we have heard stories from many commercial developers who have actively chosen to work in Java rather than working with a more sophisticated language, due to the availability of powerful Java-focused tools such as IntelliJ's IDEA,[15] with their rich support for automated refactoring and program transformation, contrasted with the comparative lack of tool support for richer languages. This gap in tool support can even inhibit the uptake of JVM-targeted languages such as Scala, something addressed with the recent IntelliJ IDEA support for Scala and Java 9.

[13] http://bundler.io/.
[14] http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos.
[15] https://www.jetbrains.com/idea/.

To some degree, tools can compensate for perceived deficiencies in a language, and generally evolve faster than the language itself, because a change to a tool does not have the same impact on existing software as a change to a language specification does.

Support of Change. One might think that agile methods would favour dynamically typed languages, and indeed part of their popularity is the sense that they allow a developer to get something done without having to worry about a lot of the "boiler-plate" associated with stricter and more verbose languages. But there is a counter-pressure for statically typed languages when working on large systems. Agile methods promote embracing change, with code bases evolving through a continuous sequence of updates and refactorings [10]. These activities of transforming a code base are more easily done when supported by tools, and refactoring tools work better when type information is available [40]. However, this is not a one-way street. The increased complexity of Scala's type system over Java makes it harder to create effective automated refactoring tools. Also the additional sophistication of the compiler means that compared to Java the compilation and build times are relatively long. This can lead to frustration when developers want to make frequent small changes.

A workflow that involves making frequent small changes to a working system requires a harness that developers can rely on to mitigate risk and to allow sustainable progress. A practice commonly used by agile teams is test-driven development (TDD) [3]. It is misleading to say that particular languages explicitly support TDD, but in a language like Java or C# with good tooling, we can get a lot done working in a TDD fashion because the tools can *generate* a lot of our implementation code for us based on tests and types—although this is really just a mechanisation of mundane manual work, there is no real intelligent synthesis at play.[16] In dynamic languages we need tests even more, because we have less assurance about the safety and correctness of our program from the type system. Fortunately, some flavours of test—for example tests using mock objects [25]—may be more convenient to write in dynamic languages as we do not have to introduce abstractions to 'convince' the type system that a particular *test double*[17] can be used in a type-compatible manner as a substitute for a real dependency, given the late binding that is characteristic of such languages.

Supply of Talent. When building a new system, the choice of language is not just a technical decision, but also an economic one. Do we have the developers needed to build this system? If we need to hire more, or one day need to replace a current staff member, how much would it cost to hire someone with the relevant skills? Knowledge of some languages is easy to find, whilst others are specialist niches. At the time of writing there is a good supply of programmers in the

[16] This is in contrast to methods such as property-based testing, that synthesise tests in a smart manner by exploiting type-system guarantees and programmer-defined property specifications [7].

[17] https://martinfowler.com/bliki/TestDouble.html.

job market who know Java, C#, JavaScript etc. There are comparatively few Haskell, Rust or Erlang developers. This scarcity of supply relative to demand naturally leads to high prices.

High Performance. Some languages (C and C++ in particular) are not going to die in the immediate future because they are so performant. We return to the longevity of C in Sect. 5.1 and speculate on the future of C and C++ in Sect. 6.1. A design goal of the relatively new Rust language is to achieve C-like performance without sacrificing memory safety, through type system innovations. In order to achieve high performance, Rust in particular aims to provide *static* guarantees of memory safety, to avoid the overhead of run-time checks.

An Important Niche. Some languages just solve an important class of problem particularly well. Ada is barely used if we look globally, but it is very much alive (particularly the SPARK subset of Ada) for building high-assurance software [28]. The same is true for Fortran, but with respect to scientific computing. Both languages have reasonably recent standards (Ada 2012 and Fortran 2008).

3.2 Incentives for Evolution

Technological Advances. Advances in technology make new applications possible in principle, and languages adapt to make them possible—and feasible to build—in practice. An early aim of Java was to be the language of the web, and the mass adoption of the web as a platform for applications has led to sustenance and growth of languages such as JavaScript and PHP. The fact that JavaScript is the only programming language commonly supported by all web browsers has made it a de-facto standard for front-end web developers. The rise of the iPhone and its native apps saw a surge in Objective-C development as programmers created apps, with Apple later creating the Swift language to provide a better developer experience on iOS. Multi-core processor technology has led to parallelism being supported, albeit in a fragmented manner, in many more languages than would otherwise be the case.

Reliability and Security. As discussed in Sect. 2.3, many languages are now *managed*, so that basic correctness properties are checked at run time, and such that the programmer can be less concerned with memory allocation and deallocation. This eliminates large classes of security vulnerabilities associated with invalid memory access. It is common for language syntax and semantics to evolve in support of program reasoning: through keywords for programmer-specified assertions and contracts (particularly notable in the evolution of Ada, thanks to its SPARK subset [28][18]); via more advanced type systems, such as generics (to avoid unsafe casts), types to control memory ownership (in Rust, for example), and dependent types to encode richer properties (increasingly available in

[18] Our point here is that some languages have *evolved* to provide support for contracts. Contracts have also enjoyed first-class support from the *inception* of some languages, e.g. Eiffel.

functional languages); by updating language specifications with more rigorous semantics for operations that were previously only informally specified;[19] and by adding facilities for programmers to specify software engineering intent (an example being the option to annotate an overriding method with `@Override` in Java, or with the `override` specifier in C++, to fault misspelt overrides statically). As well as language evolution leading to improved reliability and security, there is also the notion of language *subsets* that promote more disciplined programming, or that provide more leverage for analysis tools, including a proposed safe subset of C++[20], the ECMAScript "strict" mode for JavaScript, and, again, the SPARK subset of Ada.

In addition to the above points, which centre on enabling programmers to avoid *their own* errors, it is also important to manage situations where external failures occur: power loss, network loss, hardware failure and the like. Classically this was achieved by checkpointing and rollback recovery. But as systems grow, especially in concurrency and distribution, there is increasing trend for more-locally managed failure. The Erlang *fail fast* design style seems to work effectively: tasks can be linked together, so that if one fails (for example an unanticipated programming situation leading to an uncaught exception, but particularly useful for external failures) then all its linked tasks are killed, and the creator can either re-start the tasks if the error is transitory, or clean up gracefully. Another inspiration is the functional-style absence of side effects, exploited for example by Google MapReduce [8]. If a processor doing one part of the 'map' fails, then another processor can just repeat the work. This would be far more complicated with side-effects and distributed rollback. An interesting project along these lines was the Murray et al. [31] CIEL cloud-execution engine (and associated scripting language Skywriting) where computational idempotence was a core design principle.

Competition Between Languages. Some languages evolve via competition. For example, many features of C# were influenced by Java; in turn, support for lambdas in Java 8 seems to have been influenced by similar capabilities in C#, and C++ was augmented with higher-order function support at roughly the same time. In the multi-core world there is clear competition between CUDA and OpenCL, with CUDA leading on advanced features that NVIDIA GPUs can support, and OpenCL gradually adopting the successful features that can also be implemented across GPU vendors. Competition-driven evolution demonstrates the value to users of having multiple languages occupying the same niche.

Company and Community Needs. Several languages have been born, or evolved, to meet company needs. Usually this occurs in scenarios where the companies in question are large enough to benefit from the new or evolved language even if it is only used internally, though many languages have found large external

[19] One example is C++11 adding concurrency semantics for weak memory models; the difficulty of this was illustrated by its unwanted "out of thin air" (OOTA) behaviour.
[20] http://www.stroustrup.com/resource-model.pdf.

communities. Notable examples include Microsoft's extensions to C and C++, the design of Go, Rust and Swift by Google, Mozilla and Apple, respectively, and Hack as an extension of PHP by Facebook.[21] Open source communities have also produced influential language extensions—perhaps most notably the various GNU extensions to C and C++.

3.3 Extinction due to Non-evolution

Languages become extinct when they are no longer used, but we must separate "no longer used for new projects" (e.g. COBOL) from "(probably) no systems using them left in existence" (e.g. Algol 60). (Of course community support for historic languages and systems means that even these definitions are suspect.) What interests us is the question of why a previously influential language might become used less and less? There seem to be two overlapping reasons for this: (*i*) *revolutionary replacement:* the concepts of the language were innovative, but its use in the wider ecosystem was less attractive—other languages which incorporated its features have now supplanted it; and (*ii*) *loss of fitness:* the language was well-used in many large projects, but doubts about its continuing ecological fitness arose. Algol 60 and arguably Smalltalk fits the first of these criteria well, while Fortran, Lisp, C and arguably COBOL fit the second.

Languages in the latter category can avoid extinction by evolving. This is most notably the case for Fortran, but C also fit this scheme, as does C++: new features have been added to the languages in successive versions. Similarly Lisp has evolved with Common Lisp, Racket and Clojure being modern forms. The key issue here is backwards compatibility: old programs must continue to work in newer versions of the language with at most minor textual changes. A popular technique for removing old features felt to be harmful for future versions of the language is to *deprecate* them—marking them for future deletion and allowing tool-chains to warn on their use. This technique is used in the ISO standards for C and Fortran (most recent standards in 2011 and 2008 respectively); Lisp dialects have evolved more disparately, but are still largely backwards compatible.

It is worth noting here that Fortran's evolution fits the plant-ecosystem model quite well. However, the *revolutionary replacement* model is more akin to 'artificial life' or 'genetically modified organisms' in the way the existing genetic features are combined into a new plant or programming language.

In closing this section, we note that Fortran (first standardised in 1958, most recent standard in 2008, and Fortran 2015 [sic] actively in standardisation) seems practically immortal for large-scale numeric code, such as weather forecasting and planetary modelling. Anecdotally, it largely fights off C++ because of the power that C++ provides for writing code that others cannot easily understand or that contains subtle bugs. One example of this is that Fortran largely forbids aliasing, which limits the flexibility of the language for general-purpose programming, but reduces the scope for aliasing-related programmer errors, and aids

[21] http://hacklang.org/.

compilers in generating efficient code. By contrast Algol, in spite of being perhaps the most influential language of all time, has effectively become extinct—the attempt to move from Algol 60 to Algol 68 did not prove effective in holding on to its territory. It's hard to pin down the exact reason: other languages developed while the Algol standard was effectively frozen awaiting Algol 68, the lack of support for (or even community-belief in) separate compilation, the default of call-by-name meaning that any sensible upgrade would fail to be backward-compatible, etc.

There's an interesting comparison here: "would you prefer to be immortal or to be posthumously praised for spreading your genes around the world?" A quote from Joe Armstrong also seems relevant here [41]:

> People keep asking me "What will happen to Erlang? Will it be a popular language?" I don't know. I think it's already been very influential. It might end up like Smalltalk. I think Smalltalk's very, very influential and loved by an enthusiastic band of people but never really very widely adopted.

4 Range of Important Languages in 2017

Posing the question "What are the important languages of 2017?" gives a range of answers from "well it's obviously X, Y and Z" to "what do you mean by important?". While there are important legacy languages with millions lines of existing code (Fortran, and perhaps COBOL and arguably even C), here we pragmatically explore various recent programming language popularity rankings: the RedMonk Programming Language Rankings (June 2016),[22] the 2015 Top 10 Programming Languages according to IEEE Spectrum (July 2015),[23] and the TIOBE Index (February 2017).[24]

These are of course just a subset of available rankings; the rankings differ as to how language importance is measured; data for the amount of code written is hard to come by when much deployed code is not publicly available at the source level (Wired reported in 2015 that "Google is 2 billion lines of code—and it's all in one place"[25]); and it unlikely that code duplication is accounted for when judging how much code is associated with a particular language—a recent study by Lopes et al. shows that accounting for duplication is challenging [24]. We note that these rankings assess language importance based on volume of code, rather than by assessing how influential a language is in terms of ideas (genes) it contains, or introduces, that are used in later languages.

Nevertheless, we believe the data these rankings offer provides a reasonable snapshot capturing at least the core set of languages broadly agreed to be in wide use today. They are summarised in Table 1, with the IEEE Spectrum ranking stopping at 10 and the other rankings at 20 (modulo a 20th-place tie).

[22] http://redmonk.com/sogrady/category/programming-languages/.
[23] http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages.
[24] http://www.tiobe.com/tiobe-index/.
[25] https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.

Table 1. Popular languages according to three sources (mid-2015–early 2017)

Position	RedMonk (2016)	IEEE Spectrum (2015)	TIOBE (2017)
1	JavaScript	Java	Java
2	Java	C	C
3	PHP	C++	C++
4	Python	Python	C#
5	(5=) C#	C#	Python
6	(5=) C++	R	PHP
7	(5=) Ruby	PHP	JavaScript
8	CSS	JavaScript	Visual Basic
9	C	Ruby	Delphi/Object Pascal
10	Objective-C	MATLAB	Perl
11	Shell		Ruby
12	R		Swift
13	Perl		Assembly language
14	Scala		Go
15	Go		R
16	Haskell		Visual Basic
17	Swift		MATLAB
18	MATLAB		PL/SQL
19	Visual Basic		Objective-C
20	(20=) Clojure		Scratch
21	(20=) Groovy		

We broadly partition the languages in these rankings into five categories:

– *Mainstream*: languages whose presence and approximate position in the rankings comes as no surprise, and that we expect to be around for the foreseeable future.
– *Rising*: languages that we perceive to have rapidly growing user communities, with an associated buzz of excitement. The popularity of *rising* language is likely to be recent, and expected to increase either directly, or indirectly through influence on mainstream languages.
– *Declining*: languages that are still in active use, including for new projects, but which we perceive to be on the decline, e.g. because they are less widely used than mainstream languages, yet lack the buzz of excitement surrounding rising languages.
– *Legacy*: languages that are largely used in order to maintain legacy systems, and in which we expect very little new code is developed.
– *Domain-oriented*: languages that are key to important application domains, but that one would not normally use to develop general-purpose software.

This partitioning into categories between them is skewed by our personal biases, perspectives and experience; no doubt our allocations to *rising* and *declining* will be controversial.

Mainstream. Firmly in the top ten of all three rankings, it is widely agreed that the major imperative and object-oriented languages—C, C++, C# and Java—are mainstream. These include the languages used to implement most systems infrastructure (C/C++), a large body of enterprise applications (Java/C#), the majority of desktop computer games (C++), and most Android apps (Java). These languages are all statically typed. We also regard JavaScript, Python and Ruby—all dynamically typed—as mainstream. JavaScript tops the RedMonk ranking, Python is in the top 5 in all rankings, and Ruby is top-ten in all but the TIOBE ranking, where it sits at 11th place.

Rising. We sense a great deal of excitement surrounding functional programming languages, and languages with first class functional-programming features, thus we regard Haskell, Clojure and Scala as rising. These three languages feature in the lower half of the RedMonk chart, but do not appear on the TIOBE list. One of the authors recalls being taught Haskell as an undergraduate student in 2000, at which point there seemed to be little general expectation that the language, though clearly important and influential, would become close to mainstream; the situation is very different 17 years later. We also see the influence of Haskell and functional programming in up-and-coming languages like Elm. Elm is written in Haskell and has a Haskell-like syntax but compiles to HTML, CSS and JavaScript for the browser, making it a candidate for a wide range of web-programming tasks.

Swift is evidently rising as the language of choice for iOS development, in part as a replacement for Objective-C. Swift's syntax is clearly influenced by languages like Ruby, although Swift is statically typed. As Swift has now been open-sourced, we are seeing the community around it growing, and helping to improve its tool-ecosystem.

With the exception of Swift, which enjoys good support in Apple's Xcode, when one compares the tool support for these rising languages to that for the languages we class as mainstream, the contrast is stark. If these languages continue to rise, we expect and hope that better tooling will evolve around them.

Some languages that do not make these rankings, but which we regard as rising, include: Rust, which has certainly generated a lot of academic excitement in relation to its type system; F# (well-supported by Visual Studio), which like Scala is multi-paradigm with strong functional programming support; and Kotlin, which by being built together with its IDE might avoid the tools-shortage risk of a new language. Another language with a syntax influenced by Ruby is Elixir, which targets the Erlang virtual machine and promotes an actor model of concurrency.

Declining. It seems that niches occupied by Objective-C, PHP and Perl are gradually being dominated by Swift, JavaScript, and Python/Ruby, thus we view these languages as declining. Similarly our impression is that Visual Basic is a declining language, its niches perhaps being taken over by C#.

Legacy. We attribute the presence of Object Pascal (and its Delphi dialect) in the top ten of the TIOBE ranking to the significant amount of such code being maintained in legacy software, and speculate that this language is rarely used for new projects.

Domain-Oriented. It is encouraging to see Scratch, an educational language, mentioned in the TIOBE list; we class this as domain-oriented since the goal of Scratch is to teach programming rather than for production development. Among the languages used by people who often do not regard themselves as programmers, R and MATLAB are probably the most widely used, with applications including data science and machine learning. We also class the query language dialect PL/SQL as domain-oriented.

We do not have a feeling for whether Go—ranked top 20 by both RedMonk and TIOBE—is rising or declining. Our impression is that it plays a useful role in implementing server software, a niche that it may continue to capably occupy without becoming mainstream. The same might be said of several languages not ranked in Table 1: Erlang, which occupies an important distributed systems niche for which it is widely respected; Fortran, often the language of choice for scientists; VHDL and Verilog in processor design; and OpenCL and CUDA in the parallel programming world, for example.

Of the remaining languages listed in Table 1, CSS is not a full-blown programming language, and "Shell" and "Assembly language" span a multitude of shell scripting and assembly languages for different platforms, which we do not attempt to categorise. (As an aside, we are doubtful regarding the high rank of "Assembly language" in the TIOBE ranking.)

5 The Elephants in the Room

Given the rich array of programming languages that have evolved over the past half century, and in general the successful manner in which languages have emerged or evolved to cope with technological change, we note three strange facts: that C remains an extremely popular language despite its shortcomings (Sect. 5.1), that dynamically typed languages are among the most popular programming languages despite decades of advances in static type systems (Sect. 5.2), and that despite the rise of multi-core processing, support for parallelism is patchwork across today's languages (Sect. 5.3).

At first sight these conflict with our 'selection of the fittest' thesis, and indeed perhaps we are even slightly embarrassed as computer scientists as to the state of the real world. We return this issue in Sect. 6 where we discuss the role of time, and inertia, in evolution.

5.1 The Popularity of C

Legacy aside, our view is that C has two main strengths as a language: its core features are simple, and it offers high performance. The price for performance is that virtually no reliability or security guarantees are provided. An erroneous C program can behave in unpredictable ways, and can be vulnerable to attack. Some of the most famous software vulnerabilities of recent years, including the Heartbleed bug,[26] arise from basic errors in C programs, and a great deal of effort still goes into writing and deploying "sanitiser" tools to find defects in C code, such as AddressSanitizer and MemorySanitizer from Google.[27] Further, despite having a simple core, the semantics of C are far from simple when one considers the host of undefined and implementation-defined behaviours described in the language specification. Indeed, decades since the language's inception, top programming language conferences are still accepting papers that attempt to rigorously define the semantics of C [15,29], and recent programming models for multi-core and graphics processors, such as OpenCL and Metal, have based their accelerator programming languages on C.

Given that the majority of code written today does not need to perform optimally, and given the advances in techniques such as just-in-time compilation that in many cases allow managed languages to achieve performance comparable to that of C, we ask: why does C remain so popular, and will this trend continue?

A major reason for C's longevity is that it is used to develop all major operating systems and many supporting tools. It is also typically the language of choice for embedded programming, partly due to the language's small memory footprint, and also because C is usually the first language for which a compiler targeting a new instruction set becomes available (the latter motivated by the fact that such a compiler is required to compile an operating system for the new target platform). Beyond compilers, the language is also well supported by tools, such as debuggers, linkers, profilers and sanitisers, which can influence a language's selection for use.

Kell points out various fundamental merits to systems programming that C brings, beyond it simply being a de-facto standard [18]. He argues that "the very 'unsafety' of C is based on an unfortunate conflation of the language itself with how it is implemented", and makes a compelling case that a safe implementation of C, with sufficiently high performance for the needs of systems programming, is possible. He also argues that a key property of C that many higher-level languages sacrifice is its ability to facilitate communication with 'alien' system components (including both hardware devices and code from other programming languages). This flexibility in communication owes to the ability to linking together object files that respect the same application binary interface, in C's use of memory as a uniform abstraction for communication. Kell concludes: "C is far from sacred, and I look forward to its replacements—but they must not forget the importance of communicating with aliens."

[26] http://heartbleed.com/.

[27] https://github.com/google/sanitizers.

In short, no other current language approaches the fitness of C (when measured along with its symbiotic tool-ecosystem) for the systems-programming niche.

5.2 The Rise of Dynamically Typed Languages

Static type systems have the ability to weed out large classes of programmer-induced defects ahead of time. In addition, as discussed in Sect. 3.1, static types facilitate automated refactoring—key to agile development processes—and enable advanced compiler optimisations that can improve performance. Yet many of today's most popular languages, including JavaScript, PHP, Ruby and Python, do not feature static types, or make static types optional (and unusual in practice).

In the case of JavaScript, we argue that its prevalence is driven by the web as a programming platform, and web browsers as a dominant execution environment. As the only language supported by all browsers, in a sense, JavaScript is to browsers what C is to Unix. JavaScript has also seen broad uptake in server-side development of recent times, with the Node.js platform. A major driver for this is developer mindshare. Many developers already know JavaScript because of previous work in the browser, so having a server environment where they can code in the language they already know lets them transfer many of their existing skills without having to learn Python, Ruby or PHP. In evolutionary terms, while no language per se is favoured in the server-side world, the additional fitness of JavaScript with its symbiote "programmer experience" has enabled it to colonise this niche.

One reason for the popularity of dynamic languages in general is that they tend to come with excellent library support, providing just the right methods to offer a desired service. Representing structured data without a schema in statically typed languages is generally more challenging (e.g. what types to use), but recent work by Petricek et al. on inferring shapes for F# type providers is promising [34].

A more fundamental reason may be "beginner-friendliness". It is easy to get some code up and running to power a web page using JavaScript; writing a simple utility in Python is usually more straightforward than would be the case in C (where one would need to battle with string manipulation), or in Java (where one would need to decide which classes to create).

The importance of beginner-friendliness should not be underestimated. Many people writing software these days are not trained computer scientists or software engineers. As coding becomes a skill required for a large number of different jobs, we have more and more programmers, but they may not have the time or inclination to learn the intricacies of a complex languages—they just want to get something done. Currently Python (supported by technologies like iPython Notebooks) is popular with scientists and others just wanting to get going quickly on some fairly simple computational task. This category of programmers seems likely only to grow in the future, and as such the world will accumulate growing amounts of fairly simple software, in languages that are comparatively easy for non-specialists to work with.

The danger, historically exemplified by Facebook's use of PHP, is that a system that starts as a simple program in a dynamically typed language may grow beyond practical maintainability. We question the extent to which dynamically typed languages are suitable for building large-scale infrastructure that needs to be maintained and refactored over many years by a changing team. Facebook's Hack language, which extends PHP with types in a manner that permits an untyped code base to be *incrementally* typed, is one example where a valuable code base without static types is being migrated to a typed form to enable faster defect detection and better readability and maintenance.[28] We also see similar trends in the JavaScript world, for example in the TypeScript[29] and Flow[30] initiatives from Microsoft and Facebook respectively.

We can summarise that various features of dynamic languages—rapid prototyping, beginner-friendliness, avoidance of intellectually sound but challenging type systems—adds to the fitness of such languages in niches that appreciate these properties over others.

5.3 The Patchwork Support for Parallelism

As discussed in Sect. 2.7 under "Parallelism and the rise of multi-core", the last decade has seen a wealth of research papers and industrial programming models to aid in writing parallel code, itself building on a long history of focused work by the (previously niche) parallel programming community. Yet it seems that, from a general purpose programmer's perspective, progress has been limited. There are a wide range of language features to support parallelism at different levels of abstraction (see Sect. 2.7 for examples), but even a single abstraction level there are many competing choices, both between and within languages. But despite, and perhaps *because of* all this choice, it is far from clear to a programmer without parallelism expertise, which language and mechanism to choose when wishing to accelerate a particular application.

A seemingly reasonable programmer strategy might be to invoke parallel versions of operations wherever it is safe to do so, e.g. defaulting to a `ParallelStream` over a `Stream` in Java unless the parallel version is unsafe, and leave it to the run-time system to decide when to actually employ parallelism.[31] This strategy is analogous to other strategies that are often followed during software development, such as favouring the most general type for a function argument, making data immutable when possible, and limiting the visibility

[28] https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/.

[29] https://github.com/Microsoft/TypeScript.

[30] https://github.com/facebook/flow.

[31] Determining whether parallel execution is safe might be left to the programmer, as in the case of parallel streams in Java, or might be facilitated by tool support or guaranteed by language semantics (e.g. due to language purity).

of module internals unless higher visibility is required. However, this "use parallelism wherever it would be safe" approach is, at present, naïve, and usually leads to *reduced* performance for reasons of task granularity and memory hierarchy effects when running multiple small threads. This demonstrates that we have a long way to go before parallel programming becomes truly mainstream.

Currently it often seems that the many language concurrency primitives are each fittest for a given niche, with no unifying model.

6 The Next 7000 Programming Languages

Is the Evolutionary Theory Wrong? The previous section observed three situations where languages have counter-fitness aspects (genes). We observe that this is not unexpected; time and inertia are also important in translating evolutionary fitness to niche occupancy. Even if a species, or language, becomes less fit than a competitor, its present dominance may still cause it to produce more seeds in total than the fitter, but less dominant, competitor—even if these seeds individually are less likely to thrive. Thus changes in fitness (e.g. an upgrading of the important of security affecting our perceived fitness of C) are likely only to change the second derivative of percentage niche-occupancy. Incidentally, recent evidence seems to suggest that dinosaurs were in relative decline to mammals for around 50 million years before the Chicxulub asteroid impact which completed the task [39]. We simultaneously argue that Sect. 5 *does* correctly reflect niche occupancy today, and at the same time, in this section propose predictions of future niche occupancy based on current notions of fitness.

Emboldened by Landin's success, we now make some predictions—starting with the 'elephants in the room' of the previous section.

6.1 A Replacement for C/C++?

The short answer to "What will replace C/C++ in the light of its unsafe features?" appears to be "Nothing in the short term". One explanation for this is that the C family of languages is so intimately bound to all major operating systems that its replacement is unthinkable in the short term. One pragmatic reason beyond simple inertia is that much of the system software tool chain (compilers, debuggers, linkers, etc.) either directly targets C or is designed to be most easily used in conjunction with a C software stack. The investment required to re-target this tool chain is so vast that there is a strong economic argument to continue using C.

Taking the idea of evolutionary inertia above, the nearest botanical analogue is perhaps "How long would it take for giant redwoods (sequoia) to be supplanted by a locally fitter competitor species?", answered by "rather longer than it takes moss to colonise a damp lawn".

That having been said, there are innovations quietly chipping away at the current "we'll just have to put up with the insecurities of C/C++".

One direction is languages such as Rust, that offer better safety guarantees through types, as well the safe subset of C++ mentioned in Sect. 3.2.

Another direction is the concept of Unikernel (or 'library OS') exemplified by MirageOS.[32] MirageOS is coded in the managed functional-style language OCaml and exploits its powerful module system. Experiments show that the additional cost of a managed language (5–10% for low-level drivers) is mitigated by the reduced need for context-switching to protect against pointer-arithmetic in unsafe languages [26]. Could the Linux kernel be similarly re-coded? Can the immediate performance penalty be compensated by more flexible and high-level structuring mechanisms?

A second thrust is "let's make C secure". At first this seems impossible because of pointer arithmetic. But using additional storage to represent pointer metadata (e.g. base, limit and offset) at run time can give a pointer-safe C implementation; this can be achieved via *fat pointers*, which change the pointer representation itself and thus break compatibility with code that does not use fat pointers [19,44],[33] or via compile-time instrumentation methods that do not change the representation of pointers and thus facilitate integration with uninstrumented code [16,32,37]. Either way, dynamic bounds checking reduces the attack surface in that buffer overflows and the like can no longer be exploited to enable arbitrary code execution—in short the result is no worse (but no better than) `NullPointerException` in Java. Recent work abstracts fat pointers to *capabilities* and does this checking in hardware: the Cheri project, e.g. [6], has a whole C tool-chain replacement (from FPGA-based hardware to compilers and debuggers) for a hardware-agnostic capability extension instantiated in a MIPS-like instruction set. The run-time cost of this appears in the order of 1%. Intel's MPX[34] (Memory Protection Extensions) has a similar aim, but relative performance data for whole operating systems on MPX is not yet available. As discussed in Sect. 5.1, Kell also makes a compelling argument that a safe and relatively performant implementation of C may be feasible [18].

6.2 From Dynamic to Static Types, to Verified Software

We envisage that *gradual typing* [1,43,46] will become increasingly prominent in future languages, whereby static types can be incrementally added to an otherwise dynamically typed program, so that full static typing is enabled once enough types are present, with type inference algorithms still allowing types to be omitted to some degree. This captures the "beginner-friendliness" and flexibility of dynamic types, but facilitates a gradual transition towards the guarantees that static types provide. Ideas along these lines are explored by Facebook's Hack language, as discussed in Sect. 5.2. Takikawa et al. recently studied the performance

[32] https://github.com/mirage.

[33] A subtlety is that the C standard currently requires pointers to occupy no more space than the integer type `intptr_t`. So often fat pointers need to use indirection or bit-level packing techniques which are both expensive in software.

[34] https://en.wikipedia.org/wiki/Intel_MPX.

of gradual typing in the context of Racket, with current results suggesting that work is needed to reduce the overhead of crossing a typed/untyped boundary at run time, and that performance tended to dip as static types were introduced to programs, unless they were introduced everywhere [45]. Still, it seems that such performance issues can be solved via a combination of advanced type inference and different implementation choices, and furthermore types confer many advantages besides performance, especially in relation to refactoring.

A gradual-typing spectrum provides increased benefits as developers invest in writing more type annotations. As developers devote more time to writing richer specifications they reap the benefits of stronger static and dynamic analysis and verification. We can see this approach as part of a bigger picture of confidence of correctness, given the many pressures to create more reliable software. Types are merely one example of techniques to perform checks on correctness. In addition to such static analysis, we see the rise of dynamic checks and executable specifications, from the automated tests that support test-driven development and agile methods to formally verified software.

We see a unified whole, where types, tests and specifications are complementary, and can be developed before, during or after the software system (a form of '*gradual verification*'). As the system evolves, the degree of correctness checking demanded can evolve too, and we foresee developments that make transitioning along this spectrum a natural progression over the life of a system. A documented example of formal correctness requirements being added to a system post-hoc is the Pentium 4 floating-point multiplier [17] (40,000 lines of HDL); co-development of software and specifications are visible in the CompCert [23] and CakeML [21] verified compiler projects.

6.3 Increased Fragmentation of Parallelism Support

We would like to optimistically predict that revolutions in programming language, compiler and run-time system technology will resolve the situation of patchwork support for parallelism discussed in Sect. 5.3. However, we fear that this is wishful thinking, and that balkanisation may actually increase.

If a programmer is working in a very limited domain and does not require flexibility for their application to evolve, they may benefit from a domain-oriented language, such as MATLAB or Julia, for which there is good potential for automatic parallelism—exploiting domain properties and the lack of hard-to-parallelise language features—even if this has not been achieved yet. So by using programming at higher levels of abstraction, thereby trading against language flexibility, good and predictable parallel performance is possible. But if a programmer *does* require the flexibility offered by more general languages then the situation becomes more difficult and other trade-offs emerge. A functional language may eliminate the problems of aliasing that make imperative and object-oriented languages hard to parallelise, but performance is at the mercy of decisions made by the compiler and run-time system, which can be hard to control in a declarative setting and yet may need to be controlled to avoid brittle performance changes as software evolves. A high-level object-oriented language like Java provides features unrelated to parallelism for high-productivity programming, and various libraries catering for parallelism at a relatively high level of abstraction. But it can still be difficult to achieve high and predictable performance without breaking this abstraction layer, or resorting to otherwise poor software engineering practices that sacrifice modularity and compositionality. Of course, there are lower-level languages such as OpenCL that enable fine-grained performance tuning, but lead to software that is hard to maintain and evolve.

Our prediction is that in domains where exploiting parallelism is essential—e.g. machine learning, computer vision, graph processing— communities will settle on de-facto standard domain-specific languages and libraries that are a good fit for the task at hand. These approaches will in turn be implemented on top of lower-level languages and libraries providing access to underlying hardware and implemented in languages such as OpenCL. We see the greatest potential for parallelism support in higher-level general-purpose languages to be for distributed processing, where work can be partitioned at a sufficiently coarse level of granularity to allow run-time systems to make decisions on parallel deployment dynamically, without sacrificing parallel speedup in the process.

6.4 Error Resilience

As systems grow, and become more distributed, then structured approaches to error resilience (coping with transient and permanent errors) will become more important. Section 3.2 explored the Erlang fail-fast model (fail on unexpected situations and expect your owner to fix things up) along with the suggestion that functional-style idempotence is more likely to be fruitful in highly distributed systems than classical imperative-style checkpoint and rollback.

6.5 Supporting Better Software Engineering Practices

We foresee a closer integration between languages and the tools that support them. Rather than being developed separately, languages and tools will be developed as one in symbiosis. At the time of writing we are beginning to see a move (back) towards tools being created together with languages. With languages like Kotlin, we see an example of the requirement to provide excellent tool support for working with a language being a primary driver for decisions in the design of the language itself.

Programs and software systems will continue to grow in size, and with this we foresee a change in how developers work with programs, a move away from treating programs as text to be edited, and towards graphs to be transformed. We may no longer edit individual lines of code, but encode and apply transformations, either in the form of refactorings (building on those currently supported by sophisticated IDEs for Java and C#) or by writing code that writes (or edits) code, such as those being developed by Atomist.[35] This brings to mind the possibility of code being synthesised from a specification using AI techniques, which we discuss further in Sect. 6.6.

[35] https://www.atomist.com/.

Software systems will continue to grow and so we foresee more explicit language features for describing modular structures and componentisation. Java 9's *module system* is a good start in this direction, but it does not yet address the issues of versioned, and independently developed, software components. Modularity helps humans—whose brains sadly have not increased in computational power at the same rate as the machines that we manufacture—to comprehend, manage and change these large systems by thinking at appropriate levels of abstraction. Alternatively, instead of enriching languages with more powerful features, we can imagine a state where more developers get their work done with simple languages supported by good tools. Although a code base written in a more sophisticated language may be smaller, it may also be more difficult to work with if the developer tools are less powerful. We foresee the speed of evolution of tooling exceeding that of the languages they support, with the result that developers may well be able to get more done because they have better tools, rather than because they have better building blocks.

6.6 Program Synthesis and AI

We predict strong potential for advances in artificial intelligence methods to reduce the human effort associated with programming. In the near future one can imagine routine learning being regularly deployed to provide smart autocompletion and refactoring, learning from data on programmer habits collected from users of a particular IDE. This might go further: automating those laborious program overhauls that often take hours or days to complete, but during which one can feel to be in auto-pilot mode. Nevertheless, making such semantics-aware transformations would appear to go well beyond the pattern-recognition tasks at which machine learning has been shown to excel. It seems apparent that there is scope for effective program synthesis in suitably restricted domains, such as the development of standard device drivers [38]. The prospect of synthesising a correct implementation of a non-trivial class, given a precise definition of its interfaces and a set of unit tests that it should pass, seems within reach in the foreseeable future: recent work on synthesising programs from input-output examples shows promise [2] and has gained widespread media attention.[36] However, we regard software development in general as a creative process that stretches human ingenuity to its limits, and thus do not predict general breakthroughs here unless human-level AI is achieved.

6.7 A Non-prediction

In discussing the three elephants in the room of Sect. 5 we might hope for, although we do not explicitly predict it, general purpose languages with a unified concurrency model, which are type-safe, offer a flexible balance between static and dynamic typing, along with suitability for low-level programming. Indeed

[36] https://www.newscientist.com/article/mg23331144-500-ai-learns-to-write-its-own-code-by-stealing-from-other-programs/.

we might hope for a single universal language which is suitable for all niches, as has been a recurring hope since Landin's time.

However, the evolutionary model does not predict this. It does predict that *if* a language is fitter for multiple niches then it will eventually colonise both. It says nothing about the existence of such a language, and past attempts to create universal languages do not add encouragement.

Some Final Words

While we hope to have provided some updated discussion and predictions following Landin's work half a century ago, these can only reflect the present structure of languages and evolution is ongoing. We wonder what a follow-up article in another 50 years would say.

Acknowledgements. We are grateful to Sophia Drossopoulou, Stephen Kell, Tom Stuart, Joost-Pieter Katoen, Flemming Nielson and Bernhard Steffen for their useful feedback on an earlier draft of this work.

Alastair Donaldson was supported by an EPSRC Early Career Fellowship (EP/N026314/1).

References

1. Anderson, C., Drossopoulou, S.: BabyJ: from object based to class based programming via types. Electr. Notes Theor. Comput. Sci. **82**(7), 53–81 (2003). https://doi.org/10.1016/S1571-0661(04)80802-8
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to write programs. CoRR abs/1611.01989 (2016). http://arxiv.org/abs/1611.01989
3. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
4. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional, Boston (2004)
5. Brauer, W. (ed.): Gesellschaft für Informatik e.V. LNCS, vol. 1. Springer, Heidelberg (1973). https://doi.org/10.1007/978-3-642-80732-9. 3 Jahrestagung, Hamburg, Deutschland, 8–10 Oktober 1973
6. Chisnall, D., Rothwell, C., Watson, R.N., Woodruff, J., Vadera, M., Moore, S.W., Roe, M., Davis, B., Neumann, P.G.: Beyond the PDP-11: architectural support for a memory-safe C abstract machine. SIGARCH Comput. Archit. News **43**(1), 117–130 (2015). https://doi.org/10.1145/2786763.2694367
7. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 18–21 September 2000, Montreal, Canada, pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
8. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010). https://doi.org/10.1145/1629175.1629198
9. Foster, N., Greenberg, M., Pierce, B.C.: Types considered harmful. Invited talk at Mathematical Foundations of Programming Semantics (MFPS) (2008). http://www.cis.upenn.edu/~bcpierce/papers/harmful-mfps.pdf

10. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
11. Gabriel, R.P.: Lisp: good news, bad news, how to win big (1991). https://www.dreamsongs.com/WIB.html
12. Gabriel, R.P.: Patterns of Software: Tales from the Software Community. Oxford University Press Inc., New York (1996)
13. Gabriel, R.P., White, J.L., Bobrow, D.G.: CLOS: integrating object-oriented and functional programming. Commun. ACM **34**(9), 29–38 (1991). https://doi.org/10.1145/114669.114671
14. Garner, S.: Reducing the cognitive load on novice programmers. In: Barker, P., Rebelsky, S. (eds.) Proceedings of EdMedia: World Conference on Educational Media and Technology, pp. 578–583. ERIC (2002)
15. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 15–17 June 2015, Portland, OR, USA, pp. 336–345. ACM (2015). https://doi.org/10.1145/2737924.2737979
16. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: AADEBUG, pp. 13–26 (1997). http://www.ep.liu.se/ecp/article.asp?issue=001&article=002
17. Kaivola, R., Narasimhan, N.: Formal verification of the pentium®4 floating-point multiplier. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2002, pp. 20–27. IEEE Computer Society, Washington, DC (2002). http://dl.acm.org/citation.cfm?id=882452.874523
18. Kell, S.: Some were meant for C: the endurance of an unmanageable language. In: Torlak, E., van der Storm, T., Biddle, R. (eds.) Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, 23–27 October 2017, Vancouver, BC, Canada, pp. 229–245. ACM (2017). https://doi.org/10.1145/3133850.3133867
19. Kendall, A.S.C.: Bcc: runtime checking for C programs. In: USENIX Summer Conference, pp. 5–16. USENIX (1983)
20. Kennedy, K., Koelbel, C., Zima, H.P.: The rise and fall of High Performance Fortran: an historical object lesson. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), 9–10 June 2007, San Diego, California, USA, pp. 1–22. ACM (2007). https://doi.org/10.1145/1238844.1238851
21. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, 20–21 January 2014, San Diego, CA, USA, pp. 179–192. ACM (2014). https://doi.org/10.1145/2535838.2535841
22. Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3), 157–166 (1966). https://doi.org/10.1145/365230.365257
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
24. Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., Vitek, J.: Déjàvu: a map of code duplicates on github. In: PACMPL, vol. 1, no. OOPSLA, pp. 84:1–84:28 (2017). https://doi.org/10.1145/3133908
25. Mackinnon, T., Freeman, S., Craig, P.: Endo-testing: unit testing with mock objects. In: Succi, G., Marchesi, M. (eds.) Extreme Programming Examined, pp. 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston (2001). http://dl.acm.org/citation.cfm?id=377517.377534

26. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: library operating systems for the cloud. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, pp. 461–472. ACM, New York(2013). https://doi.org/10.1145/2451116.2451167
27. Marlow, S.: Parallel and Concurrent Programming in Haskell. O'Reilly, Sebastopol (2013)
28. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
29. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of C: elaborating the de facto standards. In: Krintz, C., Berger, E. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, 13–17 June 2016, Santa Barbara, CA, USA, pp. 1–15. ACM (2016). https://doi.org/10.1145/2908080.2908081
30. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, Cambridge, October 2002
31. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S.: CIEL: a universal execution engine for distributed data-flow computing. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011, pp. 113–126. USENIX Association, Berkeley (2011). http://dl.acm.org/citation.cfm?id=1972457.1972470
32. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: SoftBound: highly compatible and complete spatial memory safety for C. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, 15–21 June 2009, Dublin, Ireland, pp. 245–258. ACM (2009). https://doi.org/10.1145/1542476.1542504
33. Perez, F., Granger, B.E., Hunter, J.D.: Python: an ecosystem for scientific computing. Comput. Sci. Eng. **13**(2), 13–21 (2011). https://doi.org/10.1109/MCSE.2010.119
34. Petricek, T., Guerra, G., Syme, D.: Types from data: making structured data first-class citizens in F#. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 477–490. ACM, New York (2016). https://doi.org/10.1145/2908080.2908115
35. Ritchie, D.: The development of the C language. In: Lee, J.A.N., Sammet, J.E. (eds.) History of Programming Languages Conference (HOPL-II), Preprints, 20–23 April 1993, Cambridge, Massachusetts, USA, pp. 201–208. ACM (1993). https://doi.org/10.1145/154766.155580
36. Ruby, S., Copeland, D.B., Thomas, D.: Agile Web Development with Rails 5.1. Pragmatic Bookshelf, Raleigh (2017)
37. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA. The Internet Society (2004). http://www.isoc.org/isoc/conferences/ndss/04/proceedings/Papers/Ruwase.pdf
38. Ryzhyk, L., Walker, A., Keys, J., Legg, A., Raghunath, A., Stumm, M., Vij, M.: User-guided device driver synthesis. In: Flinn, J., Levy, H. (eds.) 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, 6–8 October 2014, Broomfield, CO, USA, pp. 661–676. USENIX Association (2014). https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ryzhyk
39. Sakamoto, M., Benton, M., Venditti, C.: Dinosaurs in decline tens of millions of years before their final extinction. Proc. Natl. Acad. Sci. **113**(18), 5036–5040 (2016)

40. Schäfer, M.: Refactoring tools for dynamic languages. In: Proceedings of the Fifth Workshop on Refactoring Tools, WRT 2012, pp. 59–62. ACM, New York (2012). https://doi.org/10.1145/2328876.2328885
41. Seibel, P.: Coders at Work, 1st edn. Apress, Berkeley (2009)
42. Severance, C.: Javascript: designing a language in 10 days. Computer **45**, 7–8 (2012)
43. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proceedings of the Scheme and Functional Programming Workshop, pp. 81–92 (2006)
44. Steffen, J.L.: Adding run-time checking to the portable C compiler. Softw. Pract. Exp. **22**(4), 305–348 (1992). https://doi.org/10.1002/spe.4380220403
45. Takikawa, A., Feltey, D., Greenman, B., New, M.S., Vitek, J., Felleisen, M.: Is sound gradual typing dead? In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 20–22 January 2016, POPL 2016, St. Petersburg, FL, USA, pp. 456–468. ACM (2016). https://doi.org/10.1145/2837614.2837630
46. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: Tarr, P.L., Cook, W.R. (eds.) Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, 22–26 October 2006, Portland, Oregon, USA, pp. 964–974. ACM (2006). https://doi.org/10.1145/1176617.1176755
47. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. Lisp Symb. Comput. **4**(3), 223–242 (1991). https://doi.org/10.1007/BF01806107