How Programming Languages Will Co-evolve with Software Engineering: A Bright Decade Ahead

Emerson Murphy-Hill
North Carolina State University
Raleigh, North Carolina
emerson@csc.ncsu.edu

Dan Grossman
University of Washington
Seattle, Washington
djg@cs.washington.edu

ABSTRACT
Programming languages are an indispensable foundation of software engineering, so it is essential that innovations in software engineering anticipate and influence innovations in programming languages and vice-versa. We discuss seven emerging trends in the design, adoption, and use of programming languages that have clear and valuable overlap with software engineering. These themes include language design that assumes modern development ecosystems; languages that support multiple views; data-driven language design; formal and machine-checked verification that works for real systems; gradual typing; languages that embrace a distributed, asynchronous world of large external data sources; and the increasing influence of functional-programming concepts. We discuss how the time is now for software-engineering research to influence and improve these significant changes to the field.

Categories and Subject Descriptors
D.2.0 [**Software Engineering**]: General.
D.3.0 [**Programming Languages**]: General.

General Terms
Design, Human Factors, Languages

Keywords
Programming languages, software engineering, future

1. INTRODUCTION
Software engineering is a disciplined approach to creating and maintaining software in a systematic and predictable fashion. While there are many variables that influence how systematic and predictable a software project is, one common factor is the programming language[s] the software engineers use. Software engineers can decide whether to use or not use any number of prescribed software engineering practices, from whether to do code reviews to whether to write acceptance tests, but engineers cannot make software without a programming language.

In the past, innovations in programming languages have typically occurred separately from innovations in software engineering. For example, in 1995 the Java language and runtime environment was first released without any accompanying software engineering tools that would ease the development of large Java systems. These would emerge years later as their necessity became apparent. Even today, the separation between programming languages and software engineering tools is often treated as a virtue, with many environments touting their language independence (e.g. [1]) and many programming languages touting their tool independence (e.g. [2]).

We believe that the software engineering and programming languages fields will become more deeply intertwined in the decade ahead. As such, there are likely innovations in programming languages that the software-engineering research community should anticipate and influence. In particular, we make the following predictions:

1. Over the next decade, the design of new programming languages will increasingly focus on leveraging supportive IDEs, as well as assuming powerful social networks.
2. Over the next decade, the source-code ecosystem will evolve away from an ASCII-as-ground-truth mindset, treating code as rich, structured data supporting many views.
3. Over the next decade, language designers will increasingly use data to drive the design of new languages and language features.
4. Over the next decade, the use of interactive proof assistants for co-developing robust programs and proofs of correctness will allow developers to prove more powerful properties of real programs, with the proof-engineering difficulties becoming a primary research focus.
5. Over the next decade, the passé argument over static versus dynamic typing will give way to languages supporting a continuum and a gradual-typing methodology that can be adapted to application needs.
6. Over the next decade, language innovations will shift from focusing on batch-oriented or single-user programs to distributed, concurrent, and parallel programming; large workflows of asynchronous computations; accessing massive amounts of rapidly changing data; and other modern-computing challenges that will change the boundaries of a well-defined “program” or “code base.”
7. Over the next decade, functional programming will continue to see increased industry adoption, both in terms of developers adopting functional languages (Clojure, Erlang, F#, Haskell, OCaml, Racket, Scala, etc.) and in terms of language designers adopting functional features into other languages. The term "functional language" will continue to lose precise meaning, replaced by a split focus on immutable data and first-class functions.

We expand on each of these predictions in the following sections.

FOSE'14, May 31 – June 7, 2014, Hyderabad, India

2. LANGUAGES ASSUME THE IDE

Over the next decade, the design of new programming languages will increasingly focus on leveraging supportive IDEs, as well as assuming powerful social networks.

In the past, some programming languages and integrated development environments (IDEs) have been tightly coupled, perhaps most notably environments for Smalltalk. More commonly, the language and environment are decoupled, but modern powerful IDEs are now the norm: While more traditional text editors provide simplicity for small programs and retain some popularity, developers expect powerful and feature-rich IDEs. Some IDEs are more feature-rich for a particular programming language (e.g., Eclipse and IntelliJ for Java), but the clear separation is useful for the designers of both the language and the environment.

We argue that to date programming languages have had more influence over IDE features than vice-versa. For example, IDEs often compensate for cumbersome language features. Consider the complaint that Java is unnecessarily verbose. IDEs can compensate by providing auto-completion, such as expanding `Foo x = new` into `Foo x = new Foo(…)`, completing the type signature for a well-known method like `main`, or filling in generic type instantiations since the Java language has no type inference in such contexts.

Programming environments have often "played catch up" in this way, providing features to make programming in particular languages easier only after community experience has identified the most common frustrations. Many refactoring tools, while extremely useful once available, meet this description and are arduous to build. This delay in IDE feature availability can hinder the adoption and efficient use of new languages and language features.

Some programming features can be supported either by introduction into the language or into the IDE. In both cases, post-release evolutionary changes can be disorienting to the programmer. For example, new programming language features may increase expressivity to a language, but often require the programmers to learn those features. Maintaining backward compatibility in the language and IDE do not avoid this problem when the programmer works in a team environment where other programmers are injecting new language features into the programmer's code. Our view is that in the next ten years, such feature changes will be increasingly made to the IDE rather than the language, because IDE changes are generally viewed as less risky (lower commitment) than evolving a code base to use new features. Time will tell if this trend is good or bad: avoiding risk can also avoid progress.

Beyond the IDE, programming is no longer a solitary or even team-based exercise where developers sit alone with their source code and documentation. In either open-source settings or closed-source settings in large organizations, we see large, rich, searchable corpora of code, as well as technical discussions and informal third-party documentation that are consulted often. Sites like GitHub allow developers to observe easily how other developers work, enabling them to learn from "coding rockstars" [3]. Sites like StackOverflow enable developers to ask questions about new languages and language features, where most questions receive acceptable answers within about a half an hour [4] and, anecdotally, many questions are already answered, reducing confusion-resolution to effective web-search. Arguably, this trend has reduced the need for programming language designers to provide comprehensive documentation, or has at least reduced developers' reliance on official documentation.

Evolution of existing programming languages and creation of new programming languages both show no signs of slowing. Indeed, language heterogeneity strikes us as greater than it was a decade or two ago. (We avoid listing [all] modern popular languages for fear of omission.) Only recently, however, have we noticed particular language-design decisions motivated by IDE interaction. In this way, we start to see a virtuous feedback loop where IDEs start to influence languages in a way that complements languages driving IDE features.

We cite three particular examples, though there are surely others. First, consider dot-notation in F#. Like all ML dialects, F# allows defining functions over a type such as `length : string -> int`, which are used like `length x`. But F# also allows *members*, which are used with the more object-oriented syntax `x.Length`. In addition to supporting a more object-oriented perspective, dot-notation's *primary* benefit is IDE code-completion: given that `x` has type `string`, typing "`x.`" can bring up a list of completions including `Length`. This advantage is meaningless without the IDE feature.

The type systems of emerging languages Dart and TypeScript also have IDE-focused benefits. These type systems are, by design, unsound by the conventional definition of actually preventing using a value of the wrong type at run-time. For example, they allow covariant subtyping in places that can lead to run-time type errors, thus requiring the run-time checks normally associated with dynamically typed languages. However, the unsound type system can still provide enough type information to enable key IDE features, like hovering over a variable to learn its type or providing continuous type-checking to identify errors. In purely dynamically typed languages, supporting such IDE features requires static type inference, which remains a difficult research question in practice for such languages [5] [6] [7] [8] [9].

Our third example where language evolution relies on IDEs is so ubiquitous we barely notice it: Modern languages can include *enormous* standard libraries and popular frameworks with thousands of classes or modules and hundreds of thousands of methods or procedures. We argue these frameworks are simply unusable without modern programming-environment (and Internet) features for searching for features and identifying the necessary arguments, rather than manually consulting off-line documentation as in bygone eras.

We anticipate that ongoing and future language design can take for granted IDE features beyond just code completion and API search, including refactoring tools, continuous testing, version control, and much more. While this could lead to more syntactic convergence across languages (e.g., dot-notation), it could more interestingly lead to languages with new features for modularity and programming in the large. After all, if tools and environments can do more to take care of finding the code that is

relevant and hiding the code that is not, then languages may be able to focus on other needs.

We also believe the impact of sites like GitHub and StackOverflow is huge on programming practice, but they have not yet had their full impact on the design of both languages and programming environments. These sites do nothing less than make programming in many ways a global social experience, which can fundamentally change the notion of software team and available programming resources. What, we wonder, could a programming language design do to make StackOverflow searches easier and more effective as developers learn the language and the language evolves? Conversely, would it help to have better support in languages for provenance of code as developers copy-and-modify from online templates and version-control repositories?

Crowd documentation will likely have some new implications as well. For one, the dispersal of documentation will mean that it may be harder for the end programmer to find information that she is looking for. For example, a programmer may find an answer to a very similar question to the one she wants answered, but adapting the answer to her precise needs will remain a challenge. Language-related information may also be difficult to find whenever it is located in places that are not easy to search, such as in screencasts or closed-source codebases. Another challenge will be in bootstrapping documentation – if documentation for mature languages is extensive and crowd-maintained (where, in fact, crowd-maintenance is what drives the extensiveness), then new and emerging languages will need to have at least some minimum amount of documentation before the crowd can use them in earnest. What that documentation should contain remains unclear. Similarly, it seems likely that languages designers will continue to have to produce and maintain at least some core documentation, yet it remains to be seen what types of core documentation is best for the designers to produce and which are best for the crowd to produce.

Will crowd-managed "documentation" become so good that it can replace carefully curated language reference manuals, easing the documentation burden on language providers? We imagine that the breadth and depth of language documentation will increase, but in non-traditional ways. Rather than having a single location where a developer can find documentation, pieces of documentation will be dispersed around the web, in the form of questions and answers, examples, and screencasts.

Finally, we expect the tool-support "barrier to entry" for new programming languages will continue to increase since programmers have rightly come to expect rich tooling. Specifically, we predict language designers will co-develop tools along with languages so that both are released simultaneously. This has the advantage that a developer's transition from an existing, rich language-tool ecosystem will be smoother.

Co-developing languages and tools will have other effects. First, when a language is designed independent of tooling, the design space is constrained because the language itself must support all identified needs. When tools are co-developed with the language, the design space expands; a tradeoff made in a language feature can be ameliorated in an accompanying tool. This expansion of the design space may have a significant impact on new programming languages. Second, development of tools alongside languages will spur innovation in making it easier to develop tools. Some headway has been made in this space already, such as extensible IDEs that make the user interfaces of these tools easier to build, as well as extensible program analysis infrastructure, such as LLVM [10], which makes the back-end of tools easier to build.

3. BEYOND ASCII FOR SOURCE CODE

Over the next decade, the source-code ecosystem will evolve away from an ASCII-as-ground-truth mindset, treating code as rich, structured data supporting many views.

Software source code has traditionally been defined in terms of a single, canonical view: A set of files, each of which is a linear sequence of characters. It is surprising for two reasons that this historical view of ASCII (or Unicode) as ground truth remains the norm (e.g., it is plain-text that is recorded in version-control repositories). First, code is richly structured and there are advantages to viewing source code in terms of syntax trees, as almost every program-analysis tool does. So why not view the syntax tree as the ground truth with the text-editor aspect of an IDE serving just as a tool for modifying the tree?

Second, given the rich multidimensional structure of software, why do we restrict ourselves to a *single* view of source code, choosing one single structure as a fundamental aspect of design, possibly later refactoring to change this central choice?

The idea of *multivew* is that software should instead have multiple views that a developer can work with [11]. Multiple views on software is useful because different views are useful for different purposes. As an example of why multiple views are useful for programming, consider a laboratory experiment performed by Green and Petre [12]. The authors asked programmers to compare several simple programs displayed in four different ways: two textual representations and two graphical representations. Two of the textual representations are reproduced in Figure 1. When answering, "Given this input, what is the output?" programmers could answer the question faster using one view (Figure 1, left) than the other view, but when answering "What input produced a given output?", programmers answered the question faster using the other view (Figure 1, right). Thus, neither view was across-the-board superior to the other – it depends on the programmer's task (and quite possibly the programmer). Much of the theory of multiview, as well as prototype implementations, have been worked out for the semantics of maintaining multiple consistent views of data [13] [14]; a tantalizing next step would be to more thoroughly support these approaches for large bodies of source code in full-fledged languages.

While the notion of multiple views is not new, the technical challenge in implementing it in programming languages or development environments has been the ability to switch back and forth quickly and with strong consistency guarantees. Several trends have enabled such switching, however.

One is refactoring, which enables programmers to switch views, even if they work in one programming language because there may be multiple ways to represent the same computation. With

```
if high:
    if wide:
        if deep: weep
        not deep:
            if tall: weep
            not tall: cluck
            end tall
        end deep
    not wide:
        if long:
            if thick: gasp
            not thick: roar
            end thick
        not long:
            if thick: sigh
            not thick: gasp
            end thick
        end long
    end wide
not high:
    if tall: burp
    not tall: hiccup
    end tall
end high
```

```
weep :   if high & wide & (deep | tall)
cluck :  if high & wide & -deep & -tall
gasp :   if high & -wide & - (long & thick | -long & -thick)
roar :   if high & -wide & long & -thick
sigh :   if high & -wide & -long & -thick
burp :   if -high & tall
hiccup : if -high & -tall
```

Figure 1. Two views of the same program, reproduced and adapted from Green and Petre [12].

refactoring, a programmer can switch among these ways by restructuring his or her program. However, the process is sometimes slow and error prone. Tools speed up the process, yet developers still often do not use them [15].

Other recent approaches have enabled multiple views in a single language, but without modifying the source code. Davis and Kiczales' registration-based abstractions enables programmers to switch between different views of their program at the press of a button [16]. Empirical results evaluating registration-based abstractions suggest that novice programmers can understand new views of code after being exposed to those views four times or fewer, without the need for additional training or documentation [17]. The Intentional Software Workbench is another realization of the multiview idea, with a slightly different focus; different views are intended to help different stakeholders view the software from their preferred perspective [18].

In the next 10 years, we expect the emergence of multiview programming in a mainstream programming environment. Moreover, we feel that the multiview concept will likely expand beyond programming languages into new other programming domains, such as compiler error messages. In the same sense that software is typically viewed as a linear sequence of characters, so too are compiler messages (and, in general, from all types of program analysis tools). And the benefits of multiple views for programming likely extends to such program analysis messages as well.

We believe the next decade is particularly ripe for the mainstreaming of multiview programming for three reasons. First, as shown by the research described in this section, we have enough experimentation with multiview to know what the pain points are, and hopefully, how to deal with them. Second, our feeling is that the programming public is less dogmatic about programming languages than they were in the past, taking more of a "best tool for the job" mentality. The next logical step, then, is to enable them to switch between "tools" seamlessly. Finally, we believe that programmers' increasing comfort with storing and running their programs in the cloud has helped them break free from the traditional file-based mindset, and in turn break into the multiview mindset.

4. DATA-DRIVEN LANGUAGE DESIGN

Many programming languages designed today are designed by passionate individuals or teams who create new languages to solve challenges poorly met by existing languages. These challenges are often defined by personal experience and anecdotes. In the past, some claims made about the benefits of these languages have not been supported by the evidence [19].

At the same time, recent research has been studying how languages and language features are really being used. This research has come roughly in two forms, as controlled experiments and as field studies. In controlled experiments, such as the family of experiments about the effects of types [20] [21] and annotations [22], researchers have begun to establish cause and effect relationships between programming languages and programming outcomes, like productivity and defects. In field studies, such as those that investigate how developers adopt and use generics [23] and languages in open source [24], have investigated how real developers in the wild have used languages and language features.

In the past, this research rarely has influenced language design. As one positive example, research about Java generics influenced whether a new language at Microsoft would include generics (it didn't) [25].

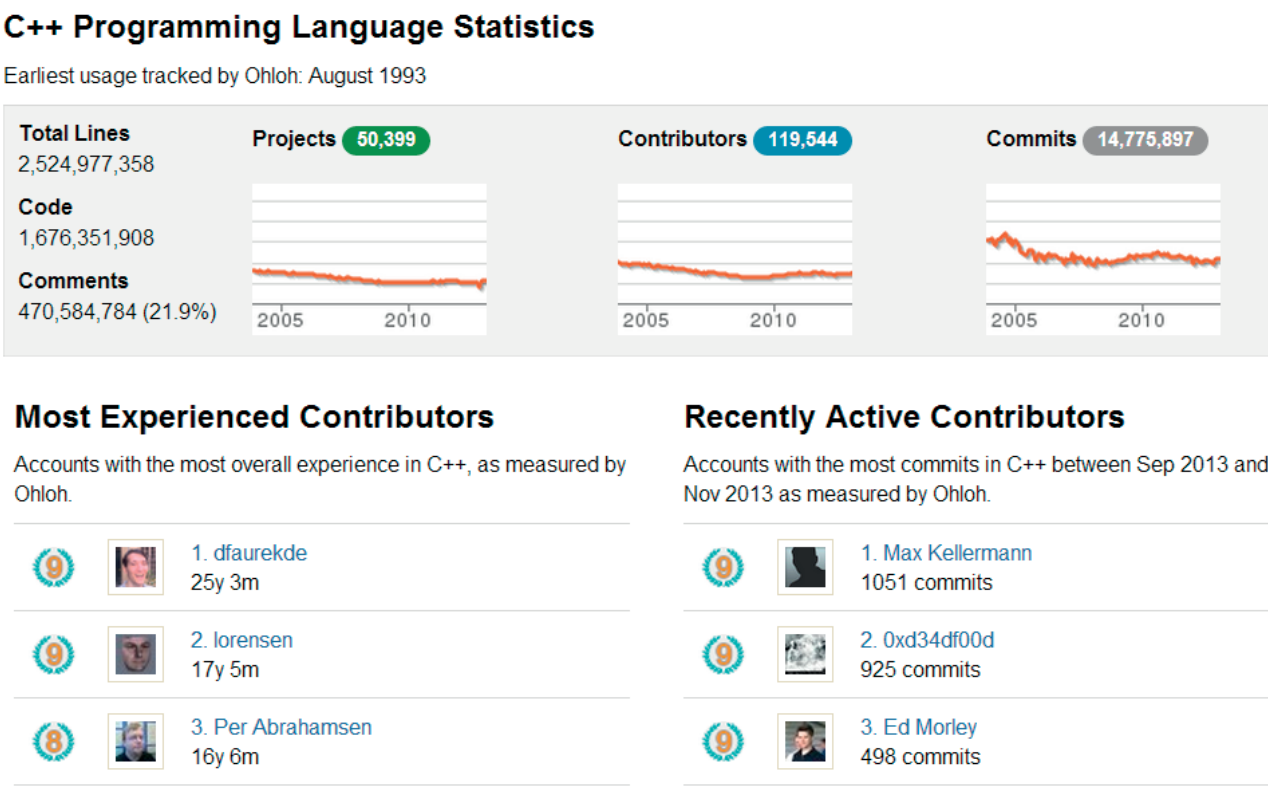

Figure 2. Current programming language analytics on ohloh.net.

This is not to say that designers do not use feedback from their programming community. Eric Lippert, previously a principal developer on the C# compiler team, wrote a blog for many years, explaining the concepts behind C# language features and soliciting feedback from an eager community of C# developers. For instance, in a post about asynchronous programming in C# 5.0, Lippert reflects on confusing concurrent operators and modifiers:

> *It is unfortunate that people's intuition upon first exposure regarding what the "async" and "await" contextual keywords mean is frequently the opposite of their actual meanings. Many attempts to come up with better keywords failed to find anything better. If you have ideas for a keyword or combination of keywords that is short, snappy, and gets across the correct ideas, I am happy to hear them.*[1]

Although Lippert has left the C# compiler team, Microsoft continues to solicit feedback about the language through a formal customer feedback platform.[2]

While such social media has surely encouraged more interaction between programming language designers and programming language users, this interaction does not include the silent majority of programmers who have neither the time nor the interest in interacting with their language's designers.

We believe that the research community will produce increasingly useful findings that will help inform the design of new languages. One way will be to characterize how existing language features have been used, which will inform how language designers can expect the same language features to be adopted and used in different languages. For example, we have good existing data about how generics have been used in multiple languages [25], which we expect forms a sufficient grounding to make predictions about how generics would be accepted by developers in new languages. We also expect that such findings can generalize to completely new language features as well. For instance, comparing the past use of the generics in C# and Java, our past results [25] suggest that forsaking backwards compatibility and offering developers a "carrot" (in the form of new APIs, in the case of C# generics) is a more successful strategy than simply retrofitting an existing language and not offering any new functionality (in the form of Java generics). This principle should generalize to other language features as well, and we expect the future will hold more such findings, and they will have an increasing impact on how language designers evolve existing languages. Moreover, if research can begin to look at languages from a more holistic perspective (as Meyerovich and Rabkin have done [24]), it may also be able to predict and influence how the developer community will react to entirely new langauges.

Beyond research findings, we expect in the next decade that better language analytic tools will help language designers ask and

[1] http://blogs.msdn.com/b/ericlippert/archive/2010/10/29/asynchronous-programming-in-c-5-0-part-two-whence-await.aspx

[2] http://visualstudio.uservoice.com/forums/121579-visual-studio/category/30931-languages-c-

answer their own questions about how developers use existing languages and language features. On one hand, researchers' toolsets are often open source, but our experience has been that these toolsets are often brittle because research does not reward robustness in language analysis tools (some researchers have bucked this trend, such as those creating the Boa framework [26]). On the other hand, we anticipate that industry-created analytics tools will be increasingly helpful to software developers. To take a current example, Figure 2 displays some publically-visible programming language analytics displayed on Ohloh.net. We anticipate such dashboard systems will contain more detailed data, such as about language feature usage, suitable for consumption by language designers.

5. FORMAL VERIFICATION FOR REAL

Over the next decade, the use of interactive proof assistants for co-developing robust programs and proofs of correctness will allow developers to prove more powerful properties of real programs, with the proof-engineering difficulties becoming a primary research focus.

For software that is critical infrastructure affecting our safety, health, financial system, etc., correctness is paramount. For decades, this truism has led to the call for formal specifications and proofs that software meets those specifications. For just as long, skeptics have pointed to the classic arguments of De Millo, Lipton, and Perlis [27] that, "ease of formal verification should not dominate programming-language design" because specifications are too complex to be correct anyway, software requirements change too rapidly, and manual proofs are often riddled with infelicities that undermine our confidence in them.

In the last several years, researchers pursuing formally verified software infrastructure have made substantial progress in winning the argument in the most direct way possible: By actually building real systems with machine-checked proofs of non-trivial correctness properties. As leading examples, we now have the CompCert C-to-x86 compiler guaranteed to preserve semantics [28], an operating system kernel with many guaranteed properties including correct access control and robustness to malicious system-call arguments [29], a web browser guaranteed to isolate distinct browser tabs securely and preserve web-cookie integrity [30], and a relational database with query optimizations proven correct with respect to relational algebra [31]. Some independent studies have confirmed some of these systems are "more correct" and robust. For example, random-testing exposed dozens of bugs in common C compilers, but none in CompCert [32]. In a world that depends on correct and secure software infrastructure, particularly the middleware and compilers on which all other software depends, we can expect advances in formal verification to continue to make progress.

How has this success circumvented De Millo, Lipton, and Perlis' arguments? First, formal specifications need not encompass all requirements. We can prove browser security without formalizing everything a web browser must do, which is essential since even specifying how to render HTML is surely intractable. Second, interactive proof assistants mechanically check all proofs, so there is no longer a need to trust the proof. Of course, the proof checker must now be trusted, but the proof checkers used today are independent of the particular software system being verified, have been refined over many years, and rely on only the most indisputable logical axioms. The modest size and clarity of such proof checkers make them amenable to thorough manual inspection, and once we believe a checker is correct, we can safely trust any proof it checks. Third, the need for changing requirements and verification that can evolve with sophisticated systems has been solved by… well, in fact, this challenge largely remains unsolved!

Indeed, the large formal-verification efforts to date, while impressive engineering achievements, have taken heroic difficult work by leading researchers. These systems can involve person-decades and the need to maintain proofs that can be larger than the code itself makes adding new features or refactoring a system far, far too difficult. The authors of the systems listed above report proofs that were 6-20× larger than the code-bases proven correct, but here "lines of code [and proof]" still make the proofs seem easier than they actually are: they take world experts in formal theorem proving orders of magnitude longer than writing the code. (Of course, these mechanically verified proofs reduce the need for testing and debugging time, but the imbalance in effort remains striking.) It is not uncommon in this domain today to find proof-debugging and proof-evaluation so difficult that one just throws away a proof that takes days to develop and starts over.

So, the challenge for the next decade is to design better [proof] languages, tools, and methodologies to meet the needs of these efforts, particularly for proof evolution and maintenance. We argue this challenge is truly open – the methodologies of interactive theorem proving for real software are sufficiently different from conventional software engineering that we need fresh ideas. On the other hand, we see no fundamental impossibility, i.e., no reason why proof engineering cannot succeed even if, in the next decade, developing formally verified systems remains an important specialized subdiscipline. What remains to be seen is how to combine ideas from the formal methods community with ideas from the software engineering community to bring the same sort of productivity gains that we have seen in other areas of software engineering.

6. GRADUAL TYPING WILL SUCCEED

Over the next decade, the passé argument over static versus dynamic typing will give way to languages supporting a continuum and a gradual-typing methodology that can be adapted to application needs.

Are professional programmers more productive with or without a static type system? Do novices learn better with or without the structure imposed by types? Does static typing reduce the need for testing and the number of defects in deployed products? These questions represent a classic "holy war" that, in their generality and yes/no nature, obscure and oversimplify a large amount of agreed-upon conventional wisdom. In the next decade, we believe the field will move beyond a dichotomy to appreciate the complementary roles of sound static analysis and run-time checking.

Let us try to quell the argument with as little controversy as possible: Programmers make errors. Tools, such as type systems, that can detect some of those errors or prove their absence without

needing test coverage are valuable. As the theory and practice of type systems grows and computational resources are brought to bear on static analysis, the range of properties that can be verified in practice grows as well. However, undecidability renders static analysis either unsound or (more often) incomplete in theory, and we see this in practice as well, requiring false alarms, programmer annotations, or both. Type systems impose structure on code that can limit code reuse and require commitment to design decisions prematurely. They can prevent testing incomplete systems. Thus, while traditional type systems enable developers to find and fix errors early, they also force developers to deal with those errors immediately, before doing anything else.

Where the challenge lies in the next decade is taking two ideas that are widely acknowledged as good ones and making them commonplace and effective. First, *gradual typing* is the idea that development can transition smoothly between dynamic typing and static typing without switching languages or having to rewrite an entire codebase. The typical proposed methodology is to start with little or no static typing and to add types as design decisions harden and invariants become more difficult to maintain. Second, surely many applications, and even many abstractions within an application, have their own internal invariants that would benefit from the rigor and soundness of type systems, so making type systems *extensible* and *application-specific* is valuable. In the extreme, a fully malleable type system would let application developers (re-)implement gradual typing, but we believe it is valuable to keep the former concept distinct.

Mixing static typing and dynamic typing is a very old idea that has received considerable recent attention. A full survey of recent work is not our focus here, but some representative work shows the range of complementary perspectives:

- We can start with a dynamically typed language and infer types to detect likely bugs or perform compile-time optimizations, a now commonplace idea that goes back at least twenty years in programming languages [33].
- We can start with a statically typed language and make it possible to run (incomplete) programs by converting compile-time errors into run-time errors [34] [35].
- We can have a language with typed and untyped modules that can interact while maintaining appropriate *blame* for when an error occurs [36].
- We can enrich a dynamically typed language with optional types that still support common programming patterns like structural conformance [37].
- We can build extensible type systems for encoding application-specific properties, building on well-known semantic foundations like type qualifiers [38].

Much of this work to date, however, has focused on the core programming language design and implementation aspects. Programming environments, development methodologies, and rigorous field studies have been secondary considerations at best. While the success of gradual typing is by no means inevitable (true partisans of type systems or their absence will surely continue to promote their respective endpoints on the type-system continuum), we believe it will succeed, and we challenge the community to prepare software engineering for a gradual-typing world.

7. MODERN, SCALABLE APPLICATIONS

Over the next decade, language innovations will shift from focusing on batch-oriented or single-user programs to distributed, concurrent, and parallel programming; large workflows of asynchronous computations; accessing massive amounts of rapidly changing data; and other modern-computing challenges that will change the boundaries of a well-defined "program" or "code base."

Innovation in programming languages and software engineering is not always spurred internally from those communities. There is no shame in identifying that innovations are often a response to changing needs and priorities for software, and the next decade will be no exception. The world of programs that operate without communicating with other programs, networks, and external data sources is an ever-smaller portion of the software ecosystem and has plenty of sufficient programming environments. Software-system challenges are now often related to asynchrony, distribution, concurrency, and data management. Either programming languages and software engineering tools will help make developing such systems easier or our fields will diminish in importance, where programming will become craft work, which depends "on special skills [which is marked by] the lack of standardization of the product" [39].

Fortunately, there is much work already to build on. To pick just a couple examples, Erlang's success is largely due to its primary consideration of failure in distributed systems, and language or library support for asynchrony (such as Scala's framework[s] for actors or .Net's async) are practical successes built on solid foundations. However, it is far from clear that we have yet achieved conventional wisdom and agreed-upon best practices for language features and software-engineering approaches for this kind of programming. Will the next decade achieve clarity, separating winners from losers, or will new ideas lead to an even wider range of approaches?

Dealing with "big data" also deserves increased attention even if the phrase itself may be a buzzword with unclear boundaries. We know how to approach using, say, Java to build a GUI application for a laptop or smartphone. But is it as clear how to use Java to build an application that processes 1TB of data each day? Such applications are written regularly with a collection of tools that are not yet an integrated part of the conventional languages and toolsets. What synergies lie ahead by treating big-data as the norm in planning and executing software development, and how should these synergies influence software tools and language design?

8. MAINSTREAM FUNCTIONAL PROGRAMMING

Over the next decade, functional programming will continue to see increased industry adoption, both in terms of developers adopting functional languages (Clojure, Erlang, F#, Haskell, OCaml, Racket, Scala,etc.) and in terms of language designers adopting functional features into other languages. The term "functional language" will continue to lose precise meaning, replaced by a split focus on immutable data and first-class functions.

For decades, functional languages have had the reputation of being outside the mainstream, favored more by the programming-languages research community than by industry. As such, functional languages have often been the proving ground for new language features, type systems, etc. and have been used in computer-science education as ways to focus on compositionality and clear, simple semantics.

But this view of the world is already outmoded. First, "functional languages" (an ambiguous term, as discussed shortly) are used for real software all the time. Before Java's success, it was common to state without evidence that any language relying on garbage collection was impractical. Nowadays, dismissing functional programming as impractical is similarly antiquated. The Commercial Users of Functional Programming conference[3] attracts hundreds of people each year. The Haskell wiki lists dozens of companies using the language.[4] In alphabetical order, Clojure, Erlang, F#, Haskell, OCaml, Racket, and Scala have dedicated user communities building real systems and lauding the functional nature of these languages – as well as their convenient facilities for interfacing with other languages and external libraries as needed.

But more importantly and conclusively, the two primary features of functional languages – first-class function closures and immutable data[5] – are increasingly common in languages not deemed "functional." In a world where C++, C#, and Java all "have lambdas," not to mention JavaScript and Ruby, functional programming has already gone mainstream. We challenge the software-engineering research community to focus on functions with the same vigor that objects have received in the past – perhaps there are processes, tools, and best practices yet to be discovered and distilled.

We argue further that the terms "functional programming" and "functional language" will start to lose meaning. The traditional marriage of closures and immutability will remain a valuable pair, but the ideas are orthogonal. For example, the rise of mainstream concurrency has made immutable objects a common design choice (there is arguably no easier way to prevent race conditions and other concurrency bugs), irrespective of programming with closures. Combining the loss of "functional" as a distinct term with the blending of static and dynamic typing discussed previously, we will struggle to categorize emerging programming languages with a convenient simplistic set of adjectives. Already the designers of F#, Racket, Scala, etc. chafe at having their languages pigeonholed into a single "paradigm," and even Haskell has been described by one of its lead designers as, without sarcasm, "the world's finest imperative programming language" [40].

How should this fundamental breakdown of separable paradigms affect research into effective software-engineering practices? What taxonomy should we create to better describe actual practice? Is JavaScript more like C++ or more like Scheme – or does the question even make sense? We challenge the software-engineering research community to understand these questions and search for answers.

[3] http://cufp.org/conference

[4] http://www.haskell.org/haskellwiki/Haskell_in_industry

[5] By immutability, we mean the use of that data structures cannot be modified by a program, but instead can only be modified when copied as a second data structure. Immutable objects are advantageous in that they are naturally thread-safe.

9. CONCLUSION

The next decade will certainly bring changes to the way software engineers use programming languages. In this paper, we have made seven predictions about what some of those changes will be. While these changes will entail challenges to software engineering practice and research, they also present great opportunity. We are of the opinion that one of the best ways to meet these challenges is by encouraging the software engineering and programming language research communities to work together.

ACKNOWLEDGEMENTS

Thanks to Václav Rajlich for providing comments on a prior version of this paper. Portions of this paper benefited from feedback from Kathleen Fisher and Zach Tatlock, but any mistakes and controversies are the fault of the authors alone.

10. REFERENCES

[1] G. Canfora, L. Cerulo and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach," *IEEE Software,* vol. 26, no. 1, pp. 50-57, 2009.

[2] M. Fernández, K. Fisher, J. N. Foster, M. Greenberg and Y. Mandelbaum, "A Generic Programming Toolkit for PADS/ML: First-Class Upgrades for Third-Party Developers," *Lecture Notes in Computer Science: Practical Aspects of Declarative Languages,* vol. 4902, pp. 133-149, 2008.

[3] L. Dabbish, C. Stuart, J. Tsay and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *ACM 2012 Conference on Computer Supported Cooperative Work*, 2012.

[4] S. M. Nasehi, J. Sillito, F. Maurer and C. Burns, "What Makes a Good Code Example? A Study of Proramming Q&A in StackOverflwo," in *28th IEEE International Conference on Software Maintenance*, 2012.

[5] M. Furr, J.-h. An, J. S. Foster and M. Hicks, "Static Type Inference for Ruby," *Proceedings of the Object-Oriented Program Languages and Systems (OOPS) Track at ACM Symposium on Applied Computing (SAC),* pp. 1859-1866, 2009.

[6] A. Guha, C. Saftoiu and S. Krishnamurthi, "Typing Local Control and State Using Flow Analysis," 2011.

[7] P. Heidegger and P. Thiemann, "Recency Types for Analyzing Scripting Languages," *Proceedings of the European Conference on Object-Oriented Programming,* pp. 200-224, 2010.

[8] T. Zhao, "Type Inference for Scripting languages with Implicit Extension," *Proceedings of the International Workshop on Foundations of Object-Oriented Languages,* pp. 37-50 , 2010.

[9] S. Tobin-Hochstadt and M. Felleisen, "Logical Types for Untyped Languages," *Proceedings of the ACM International Conference on Functional Programming,* pp. 117-128 , 2010.

[10] C. Lattner and A. Vikram, "LLVM: a compilation framework for lifelong program analysis & transformation," *Proceedings of the International Symposium on Code Generation and Optimization,* pp. 75-86, 2004.

[11] A. P. Black and M. P. Jones, "The Case for Multiple Views," *Proceedings of the Workshop on Directions in Software Engineering Environments (WoDSEE),* pp. 1-8, 2004.

[12] T. R. G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs," *Proceedings of the 6th European Conference on Cognitive Ergonomics,* 1992.

[13] N. Foster, K. Matsuda and J. Voigtländer, "Three Complementary Approaches to Bidirectional Programming," *Proceedings of the International Spring School on Generic and Indexed Programming,* pp. 1-46, 2012.

[14] N. Foster J., M. B. Greenwald, J. T. Moore, B. C. Pierce and A. Schmit, "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem," *ACM Transactions on Programming Languages and Systems,* vol. 29, no. 3, 2007.

[15] E. Murphy-Hill, C. Parnin and A. P. Black, "How We Refactor, and How we know it," *IEEE Transactions on Software Engineering,* pp. 5-18 , 2012.

[16] S. Davis and G. Kiczales, "Registration-based Language Abstractions," *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications,* pp. 754-773, 2010.

[17] J.-J. Nunez and G. Kiczales, "Understanding Registration-Based Abstractions: A Quantitative User Study," *Proceedings of the IEEE 20th International Conference on Program Comprehension,* pp. 93-102, 2012.

[18] C. Simonyi, M. Christerson and S. Clifford, "Intentional Software," *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* pp. 451-464 , 2006.

[19] S. Markstrum, "Staking Claims: a History of Programming Language Design Claims and Evidence: a Positional Work in progress," *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU),* pp. 1-5, 2010.

[20] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software.," *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications,* pp. 683-702 , 2013.

[21] S. Hanenberg, "An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time," *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications,* pp. 22-35, 2010.

[22] S. Schulze, J. Liebig, J. Siegmund and S. Apel, "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment," *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences,* pp. 65-74, 2013.

[23] C. Parnin, C. Bird and E. Murphy-Hill, "Adoption and Use of Java Generics," *Journal of Empirical Software Engineering,* vol. 18, no. 6, pp. 1047-1089, 2013.

[24] L. A. Meyerovich and A. S. Rabkin, "Empirical Analysis of Programming Language Adoption," *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 1-18, 2013.

[25] D. Kim, E. Murphy-Hill, C. Parnin, C. Bird and R. Garcia, "The Reaction of Open-Source Projects to New Language Features," *Journal of Object Technology,* 2013.

[26] R. Dyer, H. A. Nguyen, H. Rajan and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," *Proceedings of the International Conference on Software Engineering,* pp. 422-431, 2013.

[27] R. A. De Millo, R. J. Lipton and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM,* vol. 22, no. 5, pp. 271-280, May 1979.

[28] X. Leroy, "Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant," *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages,* pp. 42 - 54 , 2006.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, "seL4: Formal Verification of an OS Kernel," *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles,* pp. 207-220, 2009.

[30] D. Jang, Z. Tatlock and S. Lerner, "Establishing Browser Security Guarantees Through Formal Shim Verification," *Proceedings of the 21st USENIX Conference on Security Symposium,* pp. 8-8 , 2012.

[31] G. Malecha, G. Morrisett, A. Shinnar and R. Wisnesky, "Toward a Verified Relational Database Management System," *Proceedings of the 37th ACM Symposium on Principles of Programming Languages,* vol. 45, no. 1, pp. 237-248 , 2010.

[32] X. Yang, Y. Chen, E. Eide and J. Regehr, "Finding and Understanding Bugs in C Compilers," *Proceedings of the ACM Conference on Programming Language Design and Implementation,* pp. 283-294, 2011.

[33] R. Cartwright and M. Fagan, "Soft Typing," *Proceedings of the ACM Conference on Programming Language Design and Implementation,* pp. 278-292 , 1991.

[34] M. Bayne, R. Cook and M. D. Ernst, "Always-Available Static and Dynamic Feedback," *Proceedings of the 33rd*

International Conference on Software Engineering, 2011.

[35] D. Vytiniotis, S. Peyton Jones and J. P. Magalhães, "Equality Proofs and Deferred Type Errors: A Compiler Pearl," *Proceedings of the ACM International Conference on Functional Programming,* pp. 341-352 , 2012.

[36] S. Tobin-Hochstadt and M. Felleisen, "The Design and Implementation of Typed Scheme," *Proceedings of the 35th ACM Symposium on Principles of Programming Languages,* pp. 395-406 , 2008.

[37] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund and J. Vitek, "Integrating Typed and Untyped Code in a Scripting Language," *Proceedings of the 37th ACM Symposium on Principles of Programming Languages,* pp. 377-388, 2010.

[38] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu and T. Schiller, "Building and Using Pluggable Type-Checkers," *Proceedings of the 33rd International Conference on Software Engineering,* pp. 681-690, 2011.

[39] R. Blauner, Alienation and Freedom, University of Chicago Press, 1964.

[40] S. Peyton Jones, "Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell," *Engineering Theories of Software Construction,* pp. 47-96, 2001.