Topic -How programming Languages will co-evolve with Software Engineering: A Bright Decade Ahead

Team members

Sachin Kumar & Abhishek Kr. Prasad

Main Focus :-

1. *The design of new programming languages will increasingly focus on leveraging supportive IDEs, as well as assuming powerful social networks or more.*
2. *In future, 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. *With the use of interactive proof assistants (AI :- Artificial Intelligence) 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. *The conventional attitude change from static to 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.*

Points Summary:-

1) *Design of programming languages focused on IDE based apps and networks:-*

Co-developing languages and tools will have many 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, for example such as LLVM [10], which makes the back-end of tools easier to build.

2) *Beyond ASCII level for source code:-*

Programming has always been limited to source code ,i.e. ASCII level. But, in future the need for Multiview programming may arise to help developers to design and computation problems efficiently and easily. Suppose, a graphical view for a 3-D problem may be easy to solve than to solve it by textual view of the problem statement.

3) *Data-driven language design:-*

Many programming languages designed today are designed by passionate individuals or teams who create new languages just to solve challenges poorly met by existing languages. These challenges are often defined by personal experience and anecdotes.

In future, it is expected that better language analytic tools will help language designers to ask and answer how developers use existing languages and its features. It will also help developers to design new languages which will be more suitable for language designers, more detailed information about language feature usage etc.

4) *Formal verification by virtual assistants: -*

Proof of correctness for any software in the past has been done by developers and researchers manually by giving arguments, examples or counter-examples etc. But by building real systems with machine-checked proofs of non-trivial correctness properties to co-develop robust programs or software which could help developers to prove more powerful properties of real programs, with the proof-engineering difficulties becoming a primary research focus.

5) *Gradual (Dynamic) Typing will take over Static :-*

One of the challenge that 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.

6) *Scalable programs:-*

Dealing with "Big data Analysis" 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?

7) *Rise of Functional Programming:-*

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 conference3 attracts hundreds of people each year. The Haskell wiki lists dozens of companies using the language.