

Near Future of Programming Languages

Stephen Diehl

January 15, 2017

Adoption and Industry

Meyerovich, Leo A., and Ariel S. Rabkin. "Socio-PLT: Principles for programming language adoption." Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software. ACM, 2012.

APA

$$\text{Change Function} = F\left(\frac{\text{Perceived Crisis}}{\text{Perceived Pain of Adoption}}\right)$$

Meijer, Erik. "Confessions of a used programming language salesman." ACM SIGPLAN Notices. Vol. 42. No. 10. ACM, 2007.

Landscape

Industrial (1000+ users)

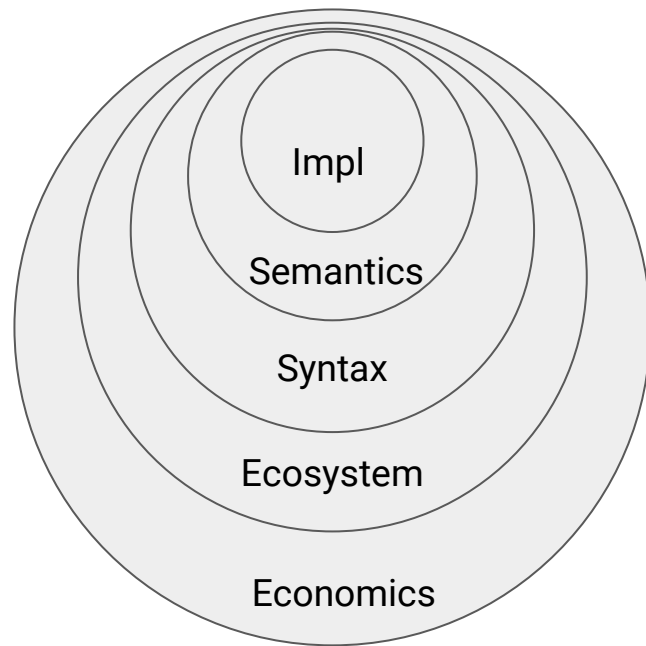
- Java
- Scala
- Python / Ruby
- Javascript
- PHP
- R
- C++
- C
- Go
- C#
- Haskell
- Swift
- Kotlin

Academic (1-100 users)

- Coq
- Agda
- Idris
- OCaml
- Racket

What We Talk About When We Talk about Languages

- The economic bubble supporting a language.
- The social bubble supporting a language.
- The front end surface language.
- The implementation details of the compiler.



What We Actually Talk About

- “It just **feels** readable”.
- “It’s like X but more **practical**”.
- “It’s designed for **humans**”.
- “It’s a **modern** language”.
- “It’s **lightweight**”.



- *“It looks like this other language I know.”*
- *“There’s a library for my domain”.*
- *“The example code masks vast complexity”*
- *“I saw it on HN last week”*
- *“I was able to install the compiler”.*

What is said.

What is actually said.



We talk about languages as a bag of feelings and fuzzy weasel words that amount to “It works for my project”.

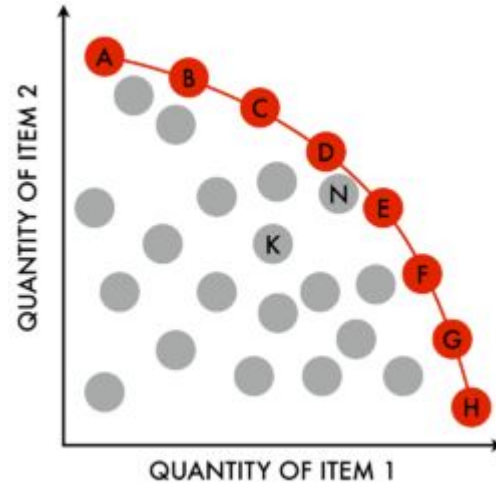
Software is a product of humans.

Confounding Factors

- Trend-following
- Hype
- Novelty seeking behavior
- Employment and career planning
- We don't teach PL theory in the western curriculum
- We confuse surface language with implementation
- Investment by large firms

Coping Mechanisms

- “Everything is shit. Embrace the anger and submit to suffering.”
- “Everything is on the Pareto frontier for all things”.



Dumbest cliché in software.

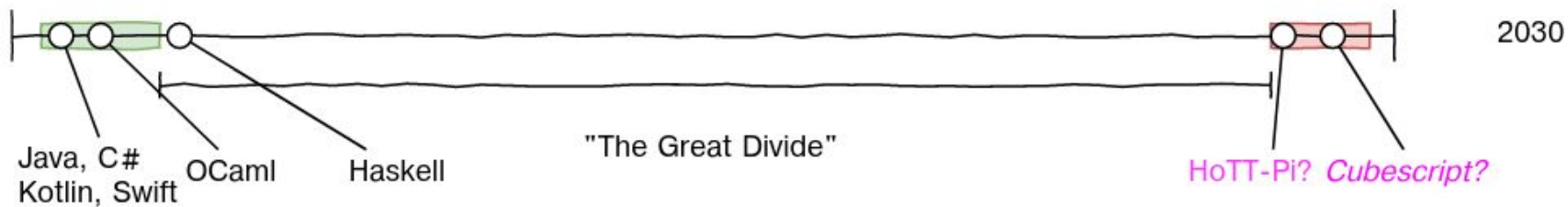
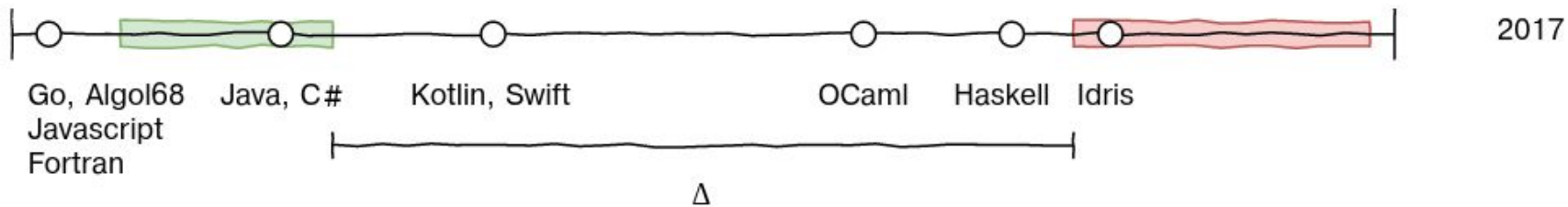
“Use the right tool for the job”

Zero information statement.

You’ll be expected to repeat this useless mantra in meetings and interviews.

Glosses over notion of what “fitness for a job” actually would imply.

Where will the next great programming language come from?



Language Gap

Where will the next great programming language come from?

Academia? **NO.**

No incentive to do engineering.

Those that do are committing career suicide.

Funding is drying up for fundamental research.

Industry? **NO.**

Can't fund anything that doesn't have a return beyond a fiscal quarter.

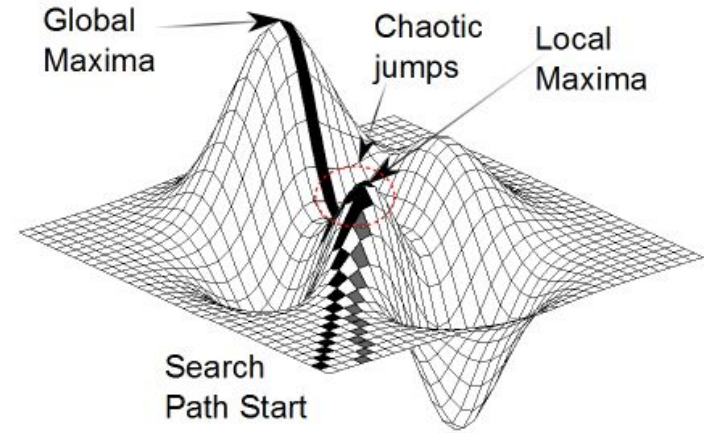
Incrementalism doesn't move things forward.

Hobbyists? **NO.**

No economic means.

Modern implementations require multiple FTE and decades.

Will we just be stuck in a local maxima of Java for next 50 years?



Economic factors

- Lack of software talent.
- Programmers entering the market through coding bootcamps.
- Massive inability to price software talent.
- Strange cargo-cults around hiring and interviewing.
- Enterprise software is a “market for lemons”.
 - People buying the software have no idea what they’re buying.
 - People selling the software have no idea what they’re selling.
 - Adverse selection drives high-quality from the market.
- Lack of standard bodies or engineering credential authorities.
- High failure rate of software projects.
- Vast mismatch between industrial software engineering and standard CS curriculum.

Module Systems

Applicative / Generative Modules

Functors

1ML

```
type MAP =
{
  type key;
  type map a;
  empty 'a : map a;
  lookup 'a : key -> map a -> opt a;
  add 'a : key -> a -> map a -> map a;
};

Map (Key : EQ) :> MAP with (type key = Key.t) =
{
  type key = Key.t;
  type map a = key -> opt a;
  empty = fun x => none;
  lookup x m = m x;
  add x y m = fun z => if Key.eq z x then some y else m z;
};
```

Algebraic Datatypes

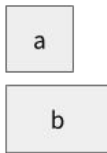
Finally starting to see algebraic datatypes appear in languages after 30 years.

- Swift
- Kotlin
- Rust
- ...

Pattern matching is still not a common language feature for some reason?



```
struct { int a; float b; }
```



```
union { int a; float b; }
```

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```


Functional Language Runtimes

- Shockingly few people in the world who work on this topic.
- No, LLVM is not the solution to everything.
- In fact, LLVM is generally awful for functional languages.
- The JVM is as well.
- Compiling everything to JS / C++ is also a nightmare scenario.
- Many degrees of freedom in design space
 - Separate compilation
 - Closure representation
 - Stack layout
 - Value heap representation and unboxing
 - Autovectorization and SIMD
 - Loop recognition, strength-reduction, and unrolling
 - Deloopification

“Oh but Rust has solved this”.



Effect Systems

- Commutative Effects vs Non-commutative Effects
- The Haskell approaches reveals how complex the nature of the problem is.
- Languages that have dabbled with modeling effects with row types have backtracked on it in favor of IO.
- Fine grained effect tracking becomes too much book-keeping for very little return.
- It's not clear if there is any more complex proposed effect system offers much benefit.

See the work of Daan Leijen.

```
function combineEffects()  
{  
    i = randomInt()    // non-deterministic  
    error("hi")        // exception raising  
    combineEffects()    // and non-terminating  
}
```

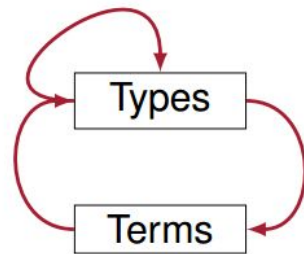
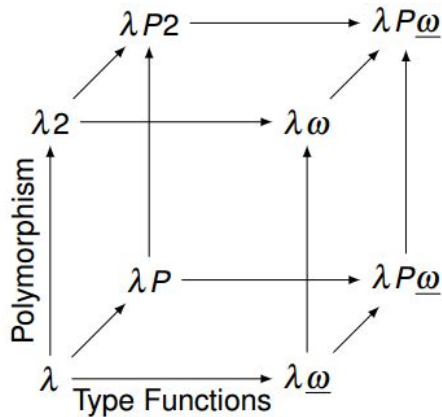
<https://www.rise4fun.com/koka/tutorial>

Dependent Types

After 20 years we're still demoing balancing red-black trees and printf at every conference.

There is no dependently typed language compiler mature enough to compile itself.

The UPenn dependently typed Haskell program shows a great deal of promise and is likely to manifest a decade before other DT languages generate practical backends.



Non-Standard ASTs

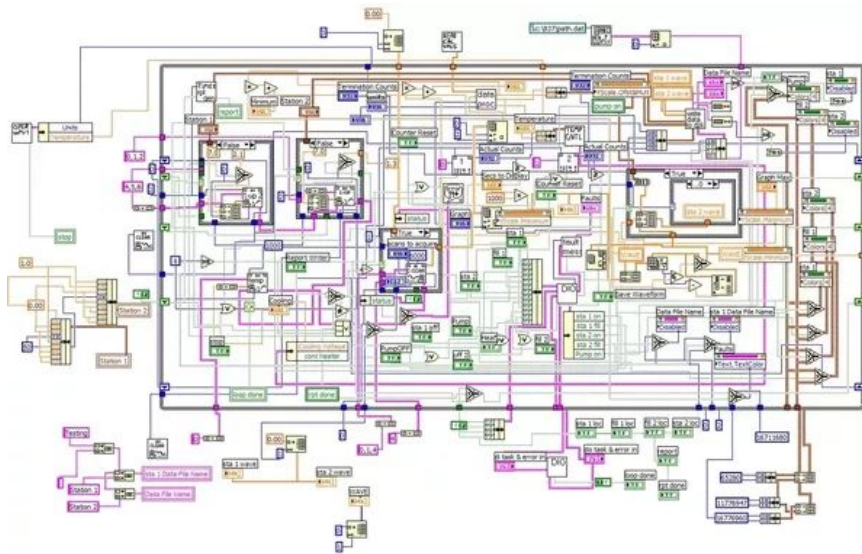
I'm not bullish on non-standard or graphical AST representations of programs.

Text is already a highly-structured graphical notation.

Spent time at a US National Laboratory writing LabView. Still have the scars to show.

Lots of people are reinventing Smalltalk on a Mac.
(See Bret Victor and Eve).

Good conversation for cocktail parties in Silicon Valley, not so good in practice.



Formal Methods

Options in 2016:

1. Rewrite your code in a theorem (Isabelle, Agda, Coq) and prove stuff there.
2. Write your code in these theorem provers extract to Haskell.
3. Prove some invariants in LiquidHaskell and discharge the proof to external solvers.
4. Wait for full dependent types in Haskell.

Interesting ideas out of Microsoft Research on SMT solver directed programming editors that enforce invariants and can generate code during development.

<https://github.com/Microsoft/dafny>

```
proof
  let " ?S = {x.x ∉ fx} "
  show "?S ∉ range f"
  proof
    assume "?S ∈ range f"
    then obtain y where fy: "?S = fy" ..
    show False
  proof cases
    assume "y ∈ ?S "
    hence "y ∉ fy" by simp
```

Typechecker Ergonomics


Debugging the internals of a modern type-checker is extraordinarily complicated.

Lots of non-local reporting problems associated with using unification during type-checker.

There may be 15 places that induce the same constraints on type-variables. Which one is most informative to humans is hard to write a procedure for and may be ambiguous.

Do we demarcate failures at module boundaries?

Example: a type checker



```
tcExpr :: Expr -> Tc Type
tcExpr (App fun arg)
  = do { fun_ty <- tcExpr fun
        ; arg_ty <- tcExpr arg
        ; res_ty <- newTyVar
        ; unify fun_ty (arg_ty --> res_ty)
        ; return res_ty }
```

Tc monad hides all the plumbing:

- Exceptions and failure
- Current substitution (unification)
- Type environment
- Current source location
- Manufacturing fresh type variables

Robust to changes in plumbing



SPJ: Escape From the Ivory Tower, Haskell From 1990 to 2011

Network Computing

Type-safe OTP.

Closure serialization is hard.

Languages with a universal bytecode format or are interpreted have an easier time. These languages also tend to be simpler by design.

Versioned network protocols based on the serialization of data sent over the network.

Distributed task schedulers are immature.

