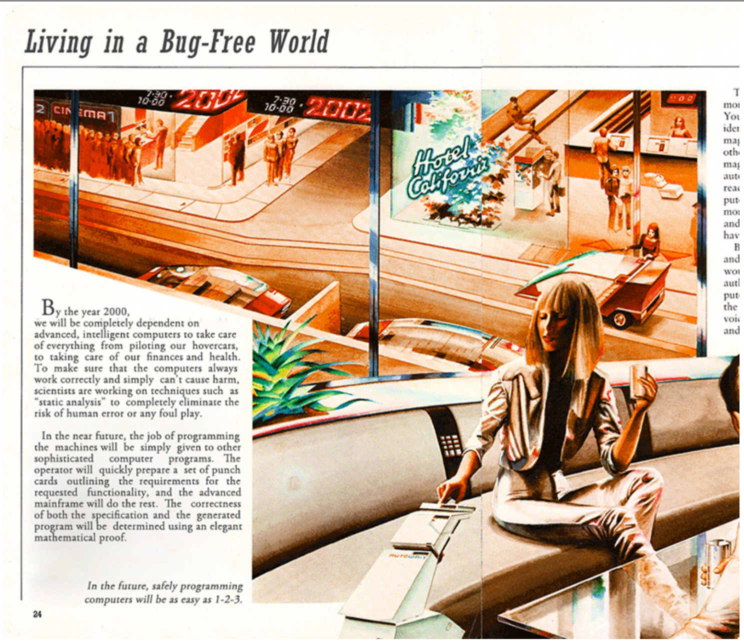
Lecture 10: Types, Type checking, & Type inference

CSC 131
Spring, 2019

Kim Bruce



Living in a Bug-Free World

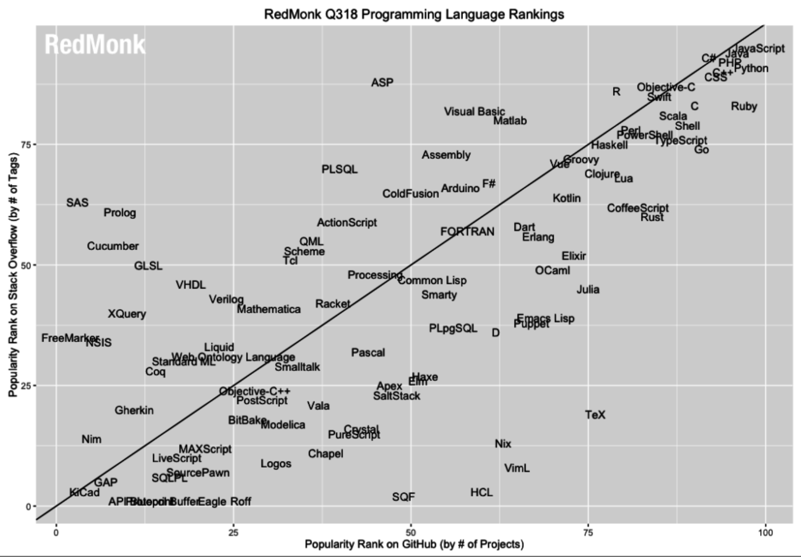
By the year 2000, we will be completely dependent on advanced, intelligent computers to take care of everything from piloting our hovercars, to taking care of our finances and health. To make sure that the computers always work correctly and simply can't cause harm, scientists are working on techniques such as "static analysis" to completely eliminate the risk of human error or any foul play.

In the near future, the job of programming the machines will be simply given to other sophisticated computer programs. The operator will quickly prepare a set of punch cards outlining the requirements for the requested functionality, and the advanced mainframe will do the rest. The correctness of both the specification and the generated program will be determined using an elegant mathematical proof.

In the future, safely programming computers will be as easy as 1-2-3.

24

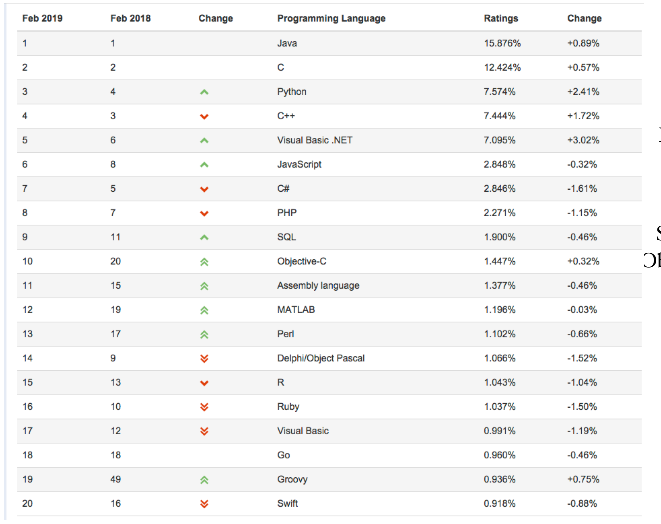
Programming Language Rankings



Top Combined

1 JavaScript	11 Swift
2 Java	12 Scala
3 Python	12 Shell
4 PHP	14 Go
5 C#	14 R
6 C++	16 TypeScript
7 CSS	17 PowerShell
8 Ruby	18 Perl
9 C	19 Haskell
9 Objective-C	20 Lua

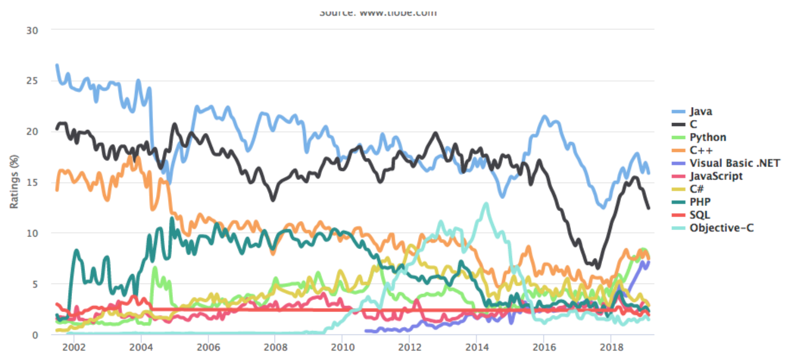
Tiobe Index



Feb 2019	Feb 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.876%	+0.89%
2	2		C	12.424%	+0.57%
3	4	▲	Python	7.574%	+2.41%
4	3	▼	C++	7.444%	+1.72%
5	6	▲	Visual Basic .NET	7.095%	+3.02%
6	8	▲	JavaScript	2.848%	-0.32%
7	5	▼	C#	2.846%	-1.61%
8	7	▼	PHP	2.271%	-1.15%
9	11	▲	SQL	1.900%	-0.46%
10	20	▲▲	Objective-C	1.447%	+0.32%
11	15	▲▲	Assembly language	1.377%	-0.46%
12	19	▲▲	MATLAB	1.196%	-0.03%
13	17	▲▲	Perl	1.102%	-0.66%
14	9	▼▼	Delphi/Object Pascal	1.066%	-1.52%
15	13	▼	R	1.043%	-1.04%
16	10	▼▼	Ruby	1.037%	-1.50%
17	12	▼▼	Visual Basic	0.991%	-1.19%
18	18		Go	0.960%	-0.46%
19	49	▲▲	Groovy	0.936%	+0.75%
20	16	▼▼	Swift	0.918%	-0.88%

2014 -> 2019
Scala #41 -> 29
Haskell #44 -> 39
ML #25 -> 48
Go #38 -> 18
Dart #42 -> 25
Scheme #48 -> 36
Objective C # 3 -> 10
Swift # 18 -> 20

Changes over Time



Static Type Checking

- Static type-checkers for strongly-typed languages (i.e., rule out all "bad" programs) must be conservative:
  - Rule out some programs without errors.
- if (*program-that-could-run-forever*) {
      expression-w-type-error;
  } else {
      expression-w-type-error;
  }

Type checking

- Most statically typed languages also include some dynamic checks.
  - array bounds.
  - Java's instanceof
  - typecase or type casts
- Pascal statically typed, but not strongly typed
  - variant records (*essentially union types*), dangling pointers
- Haskell, ML, Java strongly typed
- C, C++ not strongly typed

Type Compatibility

- When is x := y legal?

```
Type T = Array [1..10] of Integer;
Var A, B : Array [1..10] of Integer;
C : Array [1..10] of Integer;
D : T;
E : T;
```

- Name Equivalence$_A$ (*Ada)*
- Name Equivalence (*Pascal, Modula-2, Java)*
- Structural Equivalence (*Modula-3, Java arrays only)*

Structural Equivalence

- Can be subtle:

```
T1 = record a : integer; b : real end;
T2 = record c : integer; d : real end;
T3 = record b : real; a : integer end;
```

- Which are the same?

```
T = record info : integer; next : ^T end;
U = record info : integer; next : ^V end;
V = record info : integer; next : ^U end;
```

Type Checking & Inference

- Write explicit rules. Let a, b be expressions
  - if a, b:: Integer,
    then a+b, a*b, a div b, a mod b:: Integer
  - if a, b:: Integer then a < b, a = b, a > b : Bool
  - if a, b: Bool then a && b, a || b: Bool
  - ...

Formal Type-Checking Rules

- Can rewrite more formally.
- Expression may involve variables, so type check wrt assignment E of types to variables.
  - E.g., E(x) = Integer, E(b) = Bool, ...

$$\textit{Hypothesis} \longrightarrow \frac{E(x) = t}{E \vdash x : t} \longleftarrow \textit{Conclusion}$$

$$\frac{E \vdash a : \text{int},\ E \vdash b : \text{int}}{E \vdash a{+}b : \text{int}}$$

Can write formally

Function Application:

$$\frac{E \Vdash f: \sigma \to \tau, \quad E \Vdash M : \sigma}{E \Vdash f(M) : \tau}$$

Function Definition:

$$\frac{E \cup \{v{:}\sigma\} \Vdash \text{Block} : \tau}{E \Vdash \text{fun } (v{:}\sigma) \text{ Block} : \sigma \to \tau}$$

Can write for all language constructs.
Based on context free grammar.
Can read off type-checking algorithm.

More later!

Haskell Type Inference

How does Haskell know what you meant?

Haskell Type Inference

1. An identifier should be assigned the same type throughout its scope.
2. In an "if-then-else" expression, the condition must have type Bool and the "then" and "else" portions must have the same type. The type of the expression is the type of the "then" and "else" portions.
3. A user-defined function has type a → b, where a is the type of the function's parameter and b is the type of its result.
4. In a function application of the form f x, there must be types a and b such that f has type a → b, x has type a, and the application itself has type b.

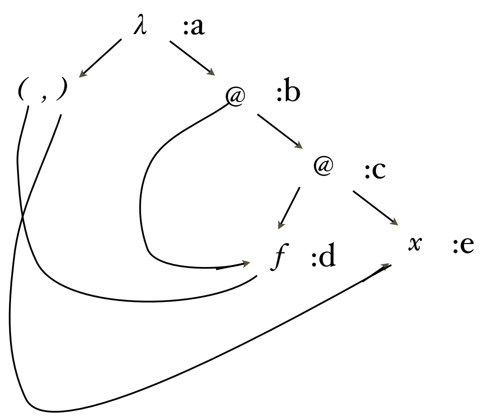
Examples of Type Inference

- Use rules to deduce types:

```
map = \ f -> \ l ->
           if l == [] then []
                else (f (head l)): (map f (tail l))
```

- map:: a → b *because function*
- f :: a, `\ l -> ...` :: b, Thus b = c → d
- l :: c, `if l = [] then ...` :: d
- ...

@ = application



```
double(f,x) = f(f(x))
```
or equivalently
```
double = \ (f,x) -> f(f(x))
```

Outcome of Type Inference

- Overconstrained: no solution

```
Prelude> tail 7

<interactive>:1:5:
    No instance for (Num [a])
      arising from the literal `7' at <interactive>:1:5
    Possible fix: add an instance declaration for (Num [a])
    In the first argument of `tail', namely `7'
    In the expression: tail 7
    In the definition of `it': it = tail 7
```

- Underconstrained: polymorphic
- Uniquely determined

By the way, ...

- Inference due to Hindley-Milner
- SML/Haskell type inference is doubly exponential in the worst case!
- Can write down terms t_n of length n such that the length of the type of t_n is of length 2^{2^n}
- Luckily, it doesn't matter in practice,
  - no one writes terms whose type is exponential in the length of the term!

Restrictions on ML/Haskell Polymorphism

- Type $(a \to b) \to ([a] \to [b])$ stands for:
  - $\forall a.\ \forall b.\ (a \to b) \to ([a] \to [b])$
- Haskell functions may not take polymorphic arguments. E.g., no type:
  - $\forall b.\ ((\forall a.(a \to a)) \to (b \to b))$
  - define: foo f (x,y) = (f x, f y)
  - id z = z
  - foo id (7, True) -- gives type error!
  - Type of foo is only (t -> s) -> (t, t) -> (s, s)

Restrictions on Implicit Polymorphism

Polymorphic types can be defined at top level or in let clauses, but can't be used as arguments of functions

```
id x = x
   in (id "ab", id 17)
```

OK, but can't write

```
test g = (g “ab”, g 17)
```

Can't find type of test w/unification.
More general type inference is undecidable.

Explicit Polymorphism

Easy to type w/ explicit polymorphism:

```
test (g: forall t.t -> t) = (g “ab”, g 17)
   in test (\t => \(x:t) -> x)
```

Languages w/explicit polymorphism:
Clu, Ada, C++, Eiffel, Java 5, C#, Scala, Grace

Explicit Polymorphism

- Clu, Ada, C++, Java
- C++ macro expanded at link time rather than compile time.
- Java compiles away polymorphism, but checks it statically.
- Better implementations keep track of type parameters.

Summary

- Modern tendency: strengthen typing & avoid implicit holes, but leave explicit escapes
- Push errors closer to compile time by:
  - Require over-specification of types
  - Distinguishing between different uses of same type
  - Mandate constructs that eliminate type holes
  - Minimizing or eliminating explicit pointers
- Holy grail: Provide type safety, increase flexibility

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
    - Examples: hd, tail ::[t]->t , map::(a->b)->[a]->[b]
- Overloading
  - A single symbol may refer to more than one algorithm.
  - Each algorithm may have different type.
  - Choice of algorithm determined by type context.
    - (+) has types Int → Int → Int and Float → Float → Float, but not t→t→t for arbitrary t.

Why Overloading?

- Many useful functions not parametric
  - List membership requires equality
    - member: [w] -> w -> Bool (for "good" w)
  - Sorting requires ordering
    - sort: [w] -> [w] (for w supporting <,>,...)
- What are problems in supporting it in a PL?
  - Static type inference makes it hard!
  - Why are Haskell type classes a solution?

Overloading Arithmetic

- First try: allow fcns w/overloaded ops to define multiple functions
  - square x = x * x
    - versions for Int -> Int and Float -> Float
  - But then
    - squares (x,y,z) = (square x, square y, square z)
    - ... has 8 different versions!!
  - Too complex to support!

ML & Overloading

- Functions like +, * can be overloaded, but not functions defined from them!
    - 3 * 3 -- legal
    - 3.14 * 3.14 -- legal
    - square x = x * x -- Int -> Int
    - square 3 -- legal
    - square 3.14 -- illegal
  - To get other functions, must include type:
    - squaref (x:float) = x * x -- float -> float

Equality

- Equality worse!
  - Only defined for types not containing functions, files, or abstract types -- *why?*
  - Again restrict functions using ==
- ML ended up defining eq types, with special mark on type variables.
  - member: “a -> [“a] -> Bool
  - Can’t apply to list of functions

Type Classes

- Proposed for Haskell in 1989.
- Provide concise types to describe overloaded functions -- avoiding exponential blow-up
- Allow users to define functions using overloaded operations: +, *, <, etc.
- Allow users to declare new overloaded functions.
  - Generalize ML’s eqtypes
  - Fit within type inference framework

Recall ...

- Definition of quicksort & partition:

```
partition lThan (pivot, []) = ([],[])
partition lThan (pivot, first : others) =
  let
    (smalls, bigs) = partition lThan (pivot, others)
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)
```

- Allowed partition to be parametric
  - Steal this idea to pass overloaded functions!
  - Implicitly pass argument with any overloaded functions needed!!

Example

- Recall

```
class Order a where
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  ...
```

- Implement w/dictionary:

```
data OrdDict a = MkOrdDict (a -> a -> Bool) (a -> a ->
Bool) ...

getLT (MkOrdDict lt gt ...) = lt

getGT (MkOrdDict lt gt) = gt
...
```

Using Dictionaries

```
partition dict (pivot, []) = ([],[])
partition dict (pivot, first : others) =
  let
    (smalls, bigs) = partition dict (pivot, others)
  in
    if ((getLT dict) first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition:: OrdDict a -> [a] -> ([a].[a])
```

Compiler adds dictionary parameter to all calls of partition.

Reports type of partition (w/out lThan parameter) as
(Ord a) => [a] -> ([a].[a])

Instances

- Declaration
  - instance Show TrafficLight where
      show Red = "Red light"
      show Yellow = "Yellow light"
      show Green = "Green light"
  - Creates dictionary for "show"

Implementation Summary

- Compiler translates each function using an overloaded symbol into function with extra parameter: the *dictionary*.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the *dictionary*.
- The compiler converts each type class declaration into a *dictionary* type declaration and a set of *selector* functions.
- The compiler converts each instance declaration into a *dictionary* of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a *dictionary*. It uses the static, qualified type of the function to select the dictionary.

Multiple Dictionaries

- Example:
  - squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
  - squares(x,y,z) = (square x, square y, square z)
- goes to:
  - squares (da,db,dc) (x, y, z) =
      (square da x, square db y, square dc z)