Wigner GPU Lab

Overview of programming languages and paradigms

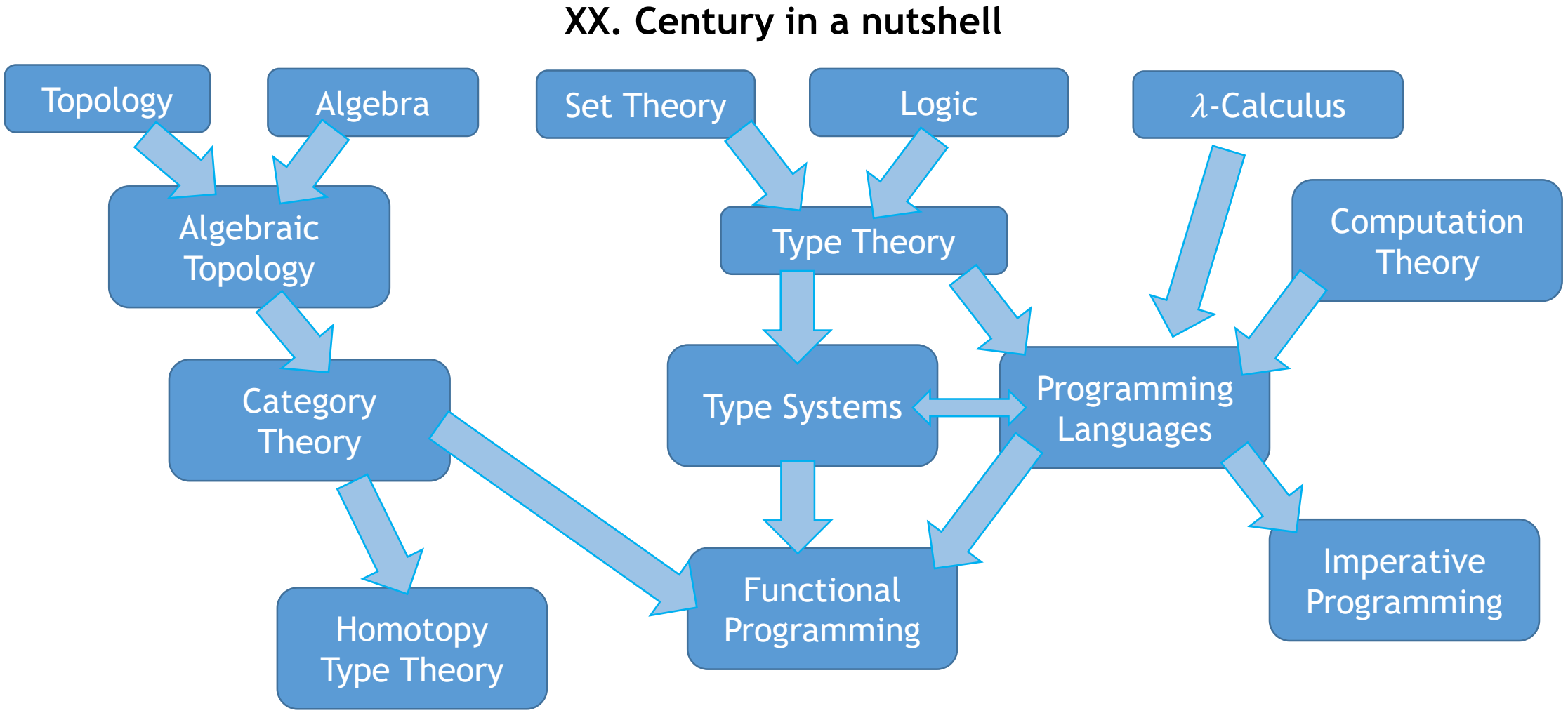
Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

Programming Languages

Programming is just applied logic.

History


XX. Century in a nutshell
Topology
Algebra
Set Theory
Logic
λ-Calculus
Algebraic Topology
Type Theory
Computation Theory
Category Theory
Type Systems
Programming Languages
Homotopy Type Theory
Functional Programming
Imperative Programming

Programming paradigms

- Machine code — Binary, Assembly
- Procedural programming — COBOL, FORTRAN, ALGOL, PL/I, BASIC, C
- Object-oriented programming — Simula, Smalltalk, C++, C#, Java
- Functional programming — Lisp, Scheme, Clojure, Ocaml, Haskell, F#

Programming paradigms

Machine code / Assembly:

- One writes a 1-1 mapping of the instructions that will be executed

Programming paradigms

Procedural programming:

- Step-by-step representation of what to do
- At higher-level, than instructions
- Code is structured by statement blocks, subroutines
  that can do everything the language supports
  these remove code repetitions
- Basic type checking

Programming paradigms

Object-oriented programming:

- Data + data-manipulation routines grouped into „objects”
- Encapsulation: inner working of objects is hidden
- Interfaces, reusable patterns by inheritance of objects
- More advanced type system (subtyping, polymorphism)

Programming paradigms

Functional programming:

- Main structures are expressions and functions
  e.g. recursion instead of loop blocks
- Higher-order functions functions operating on functions
- Move away from global state changes
  emphasis on purity, referential transparency, ...
- Even more advanced type system
  pattern matching, stronger type inference, algebraic data types, higher order and inductive types, higher polymorphism

Programming languages by purpose

Programming languages can be split into two large groups:

- General-purpose Languages
  They let you express many things: file management, mathematics, abstract concepts, data management...

- Domain-specific Languages
  They are designed to be used in a smaller domain, there they are better, more concise, more understandable

Programming languages

Why does the programming language matter?

Programming languages

In a more sophisticated language one can:

- express complex problems with less code
- create more abstract, parametrizable, reusable parts

The language can host a more advanced type system that:

- Can express stronger invariants that the compiler can check
  this results in less bugs, less debugging
- Let the compiler perform more optimizations
  because it knows more about what the program is trying to achieve

Programming languages

Why does the programming language matter?

Programming languages

Sapir-Whorf hypothesis (linguistic relativity):

- The spoken/written language structures influence how we conceptualize the world
- The language constructs may limit the cognitive categories

Programming languages

Paul Graham: The blub paradox (2001)

- „What's so great about Lisp?
  And if Lisp is so great, why doesn't everyone use it?”
- „Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available.”
- „And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower.”

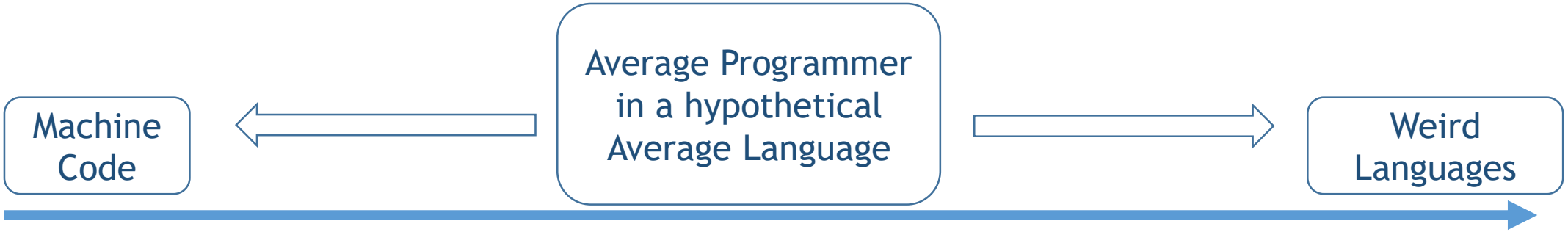
Programming languages

Paul Graham: The blub paradox (2001)

At any point on the spectrum

Looking down:
he/she knows that these languages are less expressive

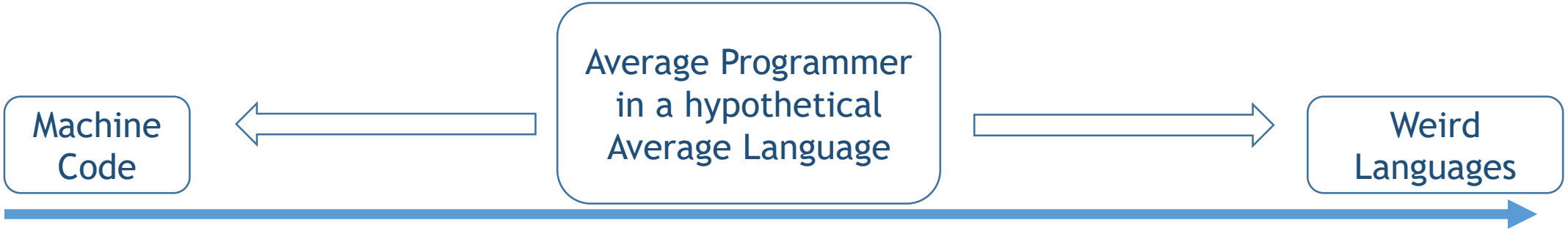
Looking up:
he/she just sees weird stuff



Programming languages

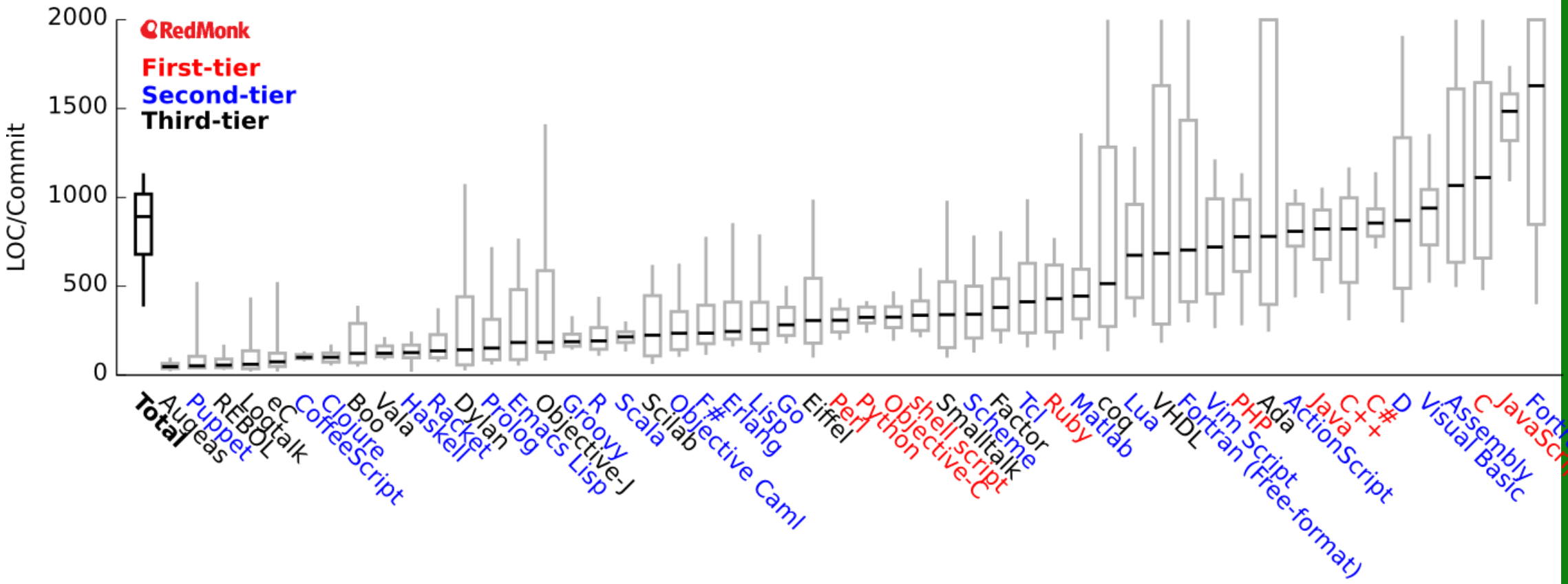
Paul Graham: The blub paradox (2001)

By induction,
the only point, where you see the differences between the languages
is the right most one!



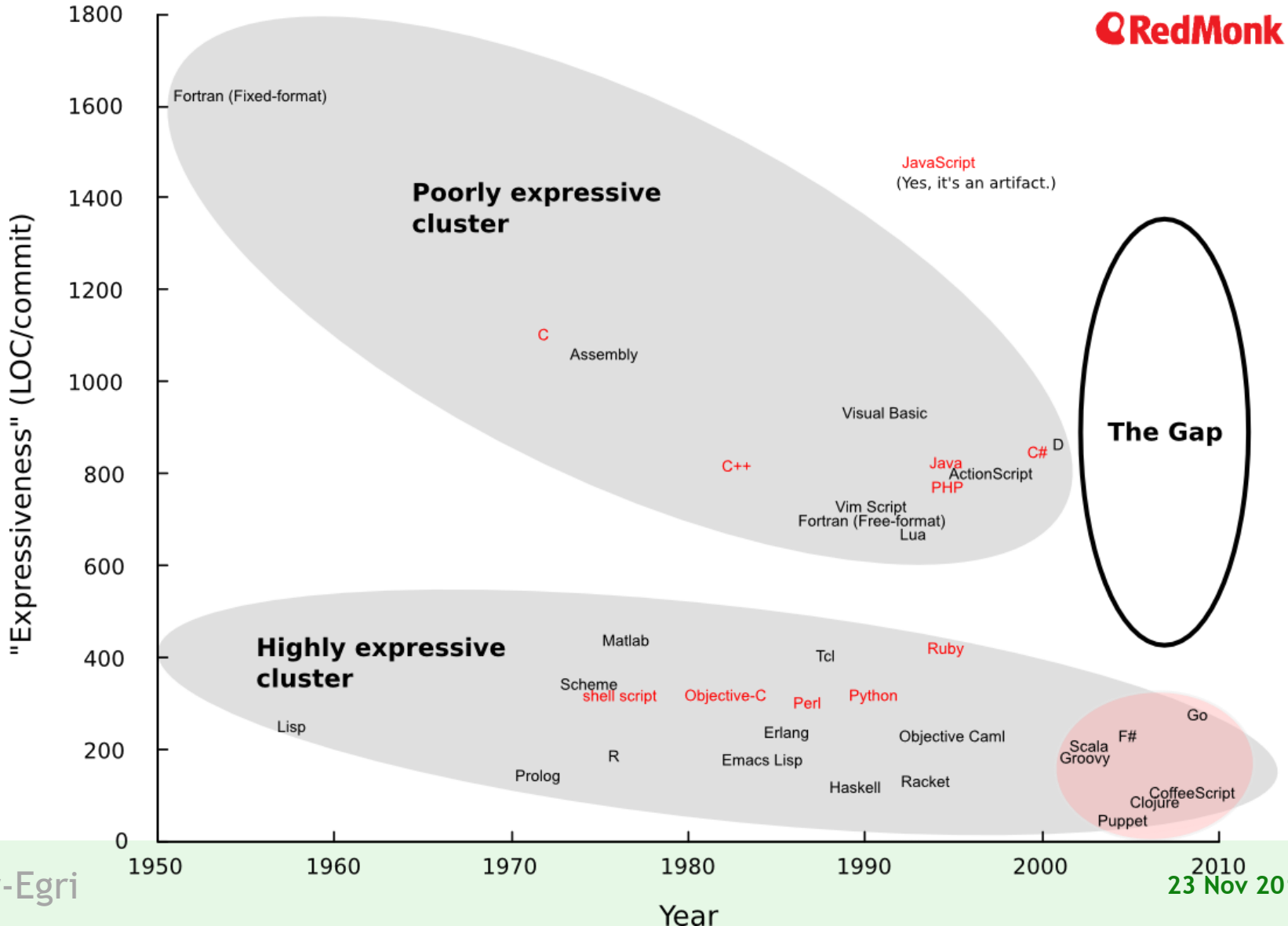
Programming languages

- An attempt to measure language expressivity:



Programming languages

- An attempt to measure language expressivity:



Programming languages

In scientific computing the picture needs more detail: performance!

One needs explicit and deterministic control over memory and execution. Many languages usually can't give that.

Programming languages

In scientific computing the picture needs more detail: performance!

We would like to argue, that using modern C++ in a functional way can be very expressive but still maintain close-to-metal performance!

Type systems

Why does types matter?

Type systems

Types were introduced, so that one may not unintentionally perform meaningless operations on data

Example:

multiplying an integer with a character and writing the result into a place representing a floating point number...

Type systems

But it goes further:

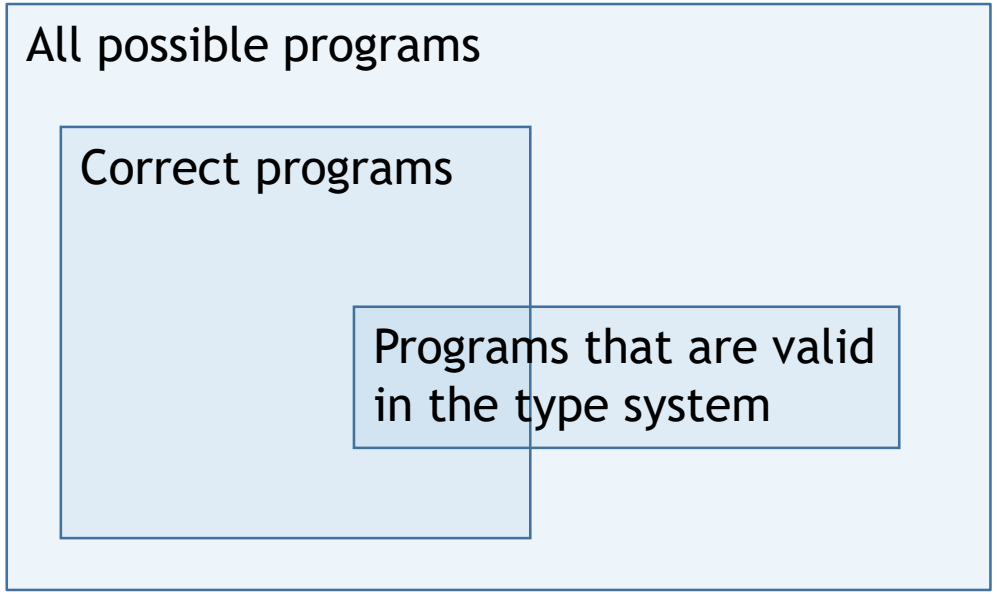
Types group together a set of rules,
that the values of those types must obey

A program, that cannot by typed (type checked)
is invalid!

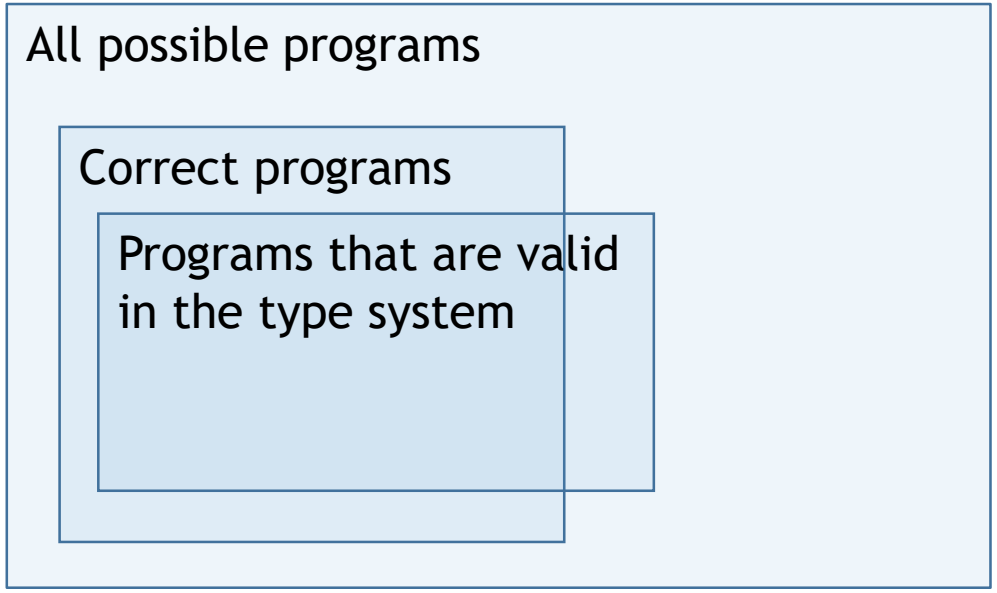
Type systems

- A type system should reject invalid programs… but it shouldn't be too restrictive, to make meaningful programs invalid...

Bad type system:



Good type system:



Type systems

Disasters,
that could have been prevented by strong typing...

Type systems

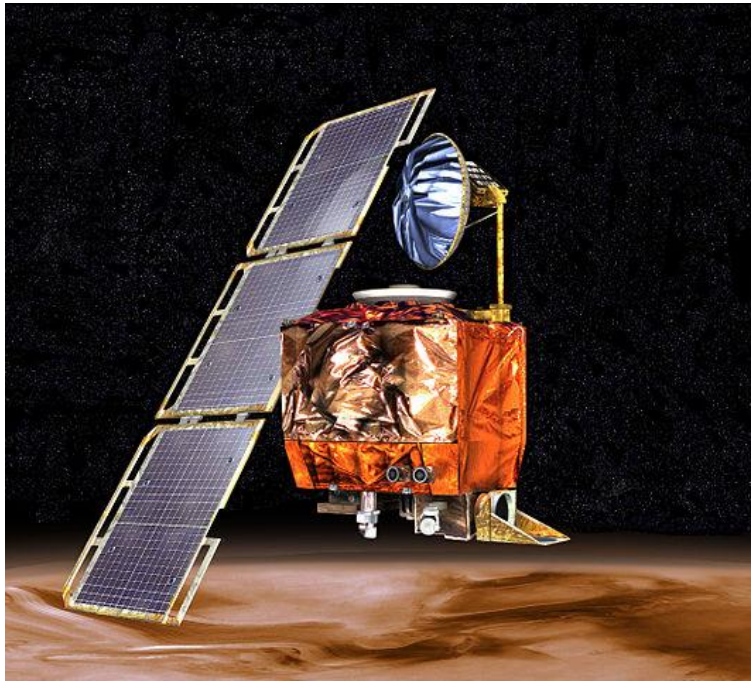
- ESA Ariane 5 (1996):
  64-bit float → 16-bit int conversion overflow
- 370M USD

Type systems

- NASA Mars Climate Orbiter (1999)

  metric-imperial
  difference between
  two software parts:

  pound-seconds vs newton-seconds



- 327M USD

Type systems

Besides preventing bugs, types are good for:

- Representing behaviour
- Providing genericity and parametricity (C++: templates)
- Providing polymorphism
  different meaning in different contexts
- Making the code readable (overloading)
- Making composable building blocks (tuples, inheritance)

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

Where:

$$\sigma_1 = \begin{pmatrix} 0, 1 \\ 1, 0 \end{pmatrix} \quad \sigma_2 = \begin{pmatrix} 0, -i \\ i, 0 \end{pmatrix} \quad \sigma_3 = \begin{pmatrix} 1, 0 \\ 0, -1 \end{pmatrix}$$

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

The outer structure is a 3-vector,
if u, v are such vectors, then this means that the following are valid:

$$v + u, v - u, c \cdot v, v/c$$

And there is a dot product: $u \cdot v$

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

The one inner structure is a 2 x 2 matrix algebra, this means:

$$A + B, A - B, c \cdot A, A/c, A \cdot B$$

are valid.

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

The second inner structure is the complex number algebra, this means:

$$c + d, c - d, c * d, c/d$$

are valid.

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

The inner most structure is the basic number algebra, again

$$a + b, a - b, a * b, a/b$$

are valid.

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

We would like to express something like this:

```
structure Vector3   is defined over elements of type T.
structure Matrix2x2 is defined over elements of type T.
structure Complex   is defined over elements of type T.
structure Integer   is represented by a type T.
```

```
structure PauliVector represented by type T is just a synonym of
Vector3 over Matrix2x2 over Complex over Integer over T.
```


Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

In C++ this becomes:

```
template<typename T> class Vector3;
template<typename T> class Matrix2x2;
template<typename T> class Complex;
template<typename T> class Integer;

template<typename T>
using PauliVector = //just a synonym
Vector3< Matrix2x2< Complex< Integer< T > > > >;
```

The T type parameter represents the underlying structure as a parameter

Generic Types

Motivating example from Physics: the Pauli vector

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3]$$

In C++ operators (like +, -, *, /) can be overloaded to behave differently for each type, still being parametrized by the underlying structure!

```
template<typename T>
Vector3<T> operator+ (Vector3<T> u, Vector3<T> v);
```

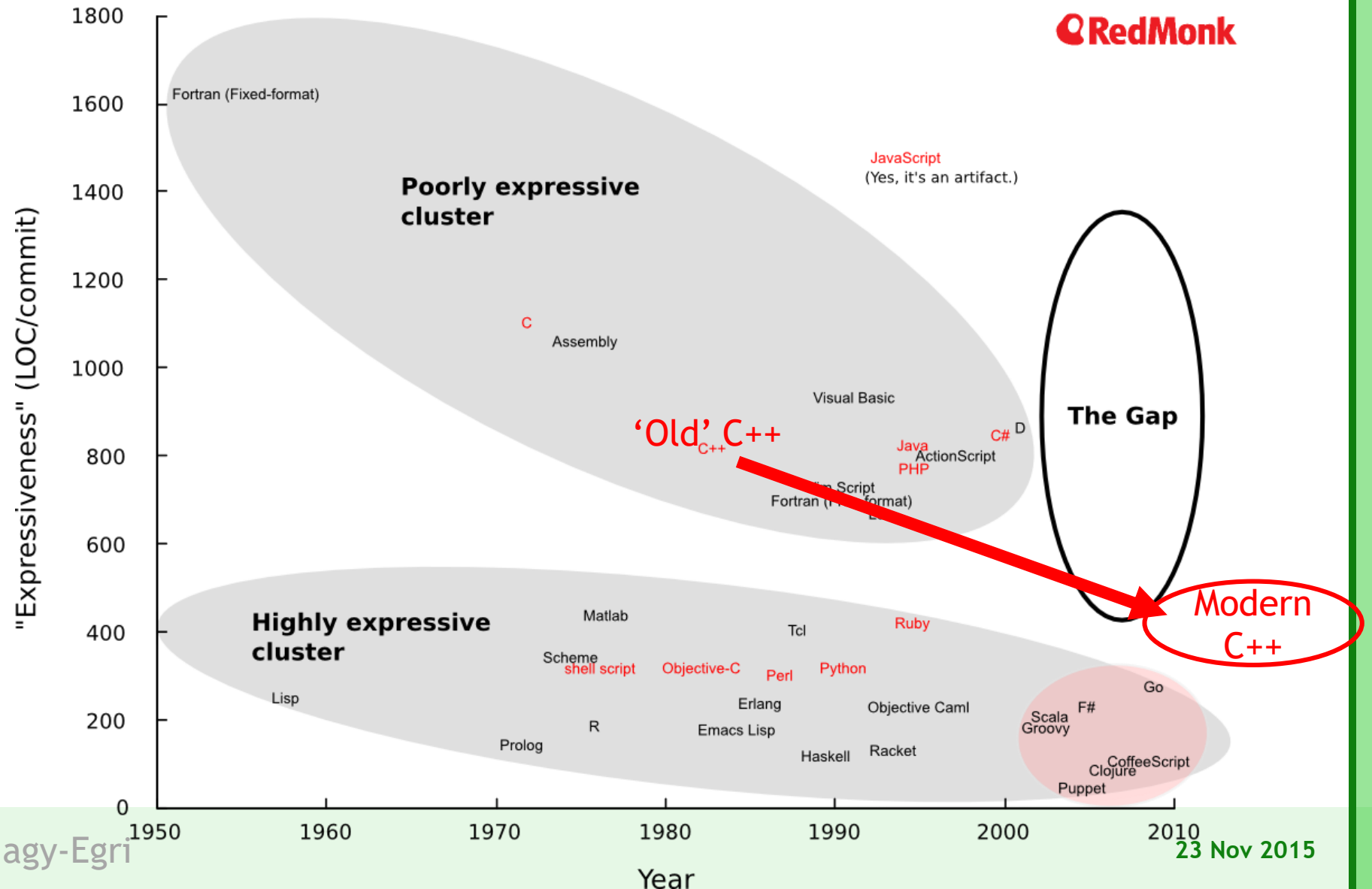
Thus, the algebraic structure can be stated parametrically, and will be checked by the compiler!

Types

- Programming is about structure and behaviour
- If you can recognize the structures in your problem, it is already half solved,
  you just represent those structures as types
- In scientific computing it is even more simpler
  Mathematics already gives the structure

Types

We would like to argue, that proper usage of programming language features make it possible, to bring C++ there:



Compiled vs. Interpreted

Another possible way to categorize languages is based on whether they are:

- Compiled, or
- Interpreted

Compiled vs. Interpreted

In compiled languages, the source code is transformed into machine code at once before any execution take place

In traditional interpreted languages usually no byte code conversion occurs, but the program dynamically loads (jumps to) precompiled parametric parts during evaluation or translate to some executable representation

Compiled vs. Interpreted

In reality the spectrum is continuous

interpreted languages are using many trick to gain performance:

- Compilation of frequently used parts
- Just-in-time compilation (compilation at runtime)
- Use of an intermediate representation, and/or byte code (platform independent instruction set)

Compiled vs. Interpreted

Advantages of interpreted languages:

- Dynamic language features
  - Reflection
    (inspect and/or modify program structure, e.g.: types, objects, functions, etc.)
  - Dynamic typing (apply / modify / check types at runtime)
- Platform independence

Compiled vs. Interpreted

Disadvantages of interpreted languages:

- Overhead...
  - By the interpreter phase
  - Lack of optimizations
  - Dynamic memory / lifetime management
- Security issues
- Reverse engineering issues

Compiled vs. Interpreted

As we said, today the spectra is continuous, especially because many languages gain support in both interpreted and both compiled ways.

However, the performance overhead is prohibitive for scientific applications to use interpreted languages

Compiled vs. Interpreted

From time to time, post appear on the internet and got picked up, that say:

“interpreted / dynamic language X beats all other compiled languages by a large factor N”

When investigating these claims, it always turns out, that the comparison is not meaningful, because:

- The program logic is not the same (lack of expertise in the compiled lang.)
- Allocations / deallocations managed completely differently
- Compiler optimizations settings are completely overlooked or unused
- Etc.

Compiled vs. Interpreted

Also, it should be taken into account, that it is possible to call compiled parts from interpreted languages.

It is especially common to call C/C++/Fortran numerics from python, Matlab etc. to improve floating-point performance.

The problem is, that all language support and type system are advantages lost at the boundary.

Mathematical systems

Further notes on mathematical software, like:

- Mathematica
- Maple
- Matlab

Mathematical systems

The overhead of different routines in these languages are highly dependent on how close the data structures and functions to the metal. This could mean factors of 1000 in runtime.

There are long books that compare different seemingly equivalent solutions for best performance. It is still hard to catch up with native code.

Language Safety

Yet another direction:

How much safety can a language guarantee?

Language Safety

Types are not the only parts of the language that is used to prevent erroneous behaviour at runtime.

A large class of problems arise from lifetime management and multithreading issues.

Language Safety

One of the most critical applications in terms of safety are browsers. Security issues are central.

It may not be a coincidence, that Mozilla started to invest in the development of a new language, rust that has a strong emphasis on safety.

Language Safety

Rust is a general purpose, multi-paradigm, compiled, systems programming language, much like C++

It is like writing C++ from scratch using the best practices from all modern languages.

But why yet another language?

Language Safety

Rust can statically guarantee that
(unless you explicitly opt-out):

- Memory management is safe
  - No array over-indexing, no nullpointer dereference, no indirect access to already freed variables etc (dangling references), no uninitialized variables, no iterator invalidation errors etc.
- Thread safety
  - No undefined behaviour due to:
    dataraces, thread unsafe operations

Language Safety

Rust is prefers safety a little bit over speed, but:

- It has explicit memory management (no garbage collection)
- It is a fair compiled language built over the LLVM infrastructure

According to available sources and our tests it worth considering for scientific purposes in the near future.

Language Safety

We highly advise to check out rust, it is still a language under development, you may run into missing features in the standard library, but the core language is reliable and...

all the errors that you were continually committing in C/C++ are completely avoidable in rust.

You are not going to waste your time on segfaults and crashes.