Why they care
and why you should too

Raphael 'kena' Poss,
Jan 12th 2017

What this talk is not about

The design of Rust and why it works well — this is better:

https://air.mozilla.org/guaranteeing-memory-safety-in-rust/

A tutorial to teach you how to program in Rust — go there instead:

https://doc.rust-lang.org/book/getting-started.html

A feature-to-feature comparison — check this out instead:

http://kukuruku.co/hub/rust/comparing-rust-and-cpp
http://science.raphael.poss.name/rust-for-functional-programmers.html

Instead:

Does Rust have a chance to replace C?

I won't tell you!

But I'll teach you how to guess.
Suppose you want to learn how to answer the question,
for any new language X

"is X going to be successful? and do I need to care?"

Mental equipment you'll need:

- Abstract machine models
- Conceptual complexity
- Understanding of Relevance and Survival criteria
- (just a little) Language features with qualitative impact

Abstract machine models

Abstract computing model: mental model to predict functional behavior

Abstract machine model: mental model to predict operational behavior
(AMM = computing model + cost function)

Observation 1:
All general-purpose computing models are (functionally) equivalent
(Turing-equivalence) and thus everyone makes their own and nobody cares

Observation 2:
Different AMMs are (usually) not operationally equivalent
some are *strictly better* than others for specific tasks

Abstract machine models — today

Two groups with backward *operational compatibility*:

NB: this has nothing to do with memory

Turing machine → register machine → random-access machine (RAM)

FORTRAN, C, C++, Ada, ML ...

→ Parallel RAM (PRAM)

OpenMP, OpenCL, CUDA

→ PRAM with partitioned memories

MPI, PGAS ... JVM! (Java, Scala...)

Dataflow machines → Spineless, Tagless Machine → MIO

Occam — Haskell — (modern) Haskell

Abstract machine models — value ranking

The “goodness” of a model depends on how accurately it predicts stuff

(Science 101)

Observations:

1. Today’s computers are accurately modelled by RAMs *(albeit barely)*
2. Today’s computers are *less and less well* modelled by PRAMs
3. Today’s computers are *not operationally modelled* by dataflow models and followups — *these models simply don’t inform well about operation*

AMM not a good predictor of operational behavior? Bad for production.

”Haskell programmers know the value of everything but not the cost”

That’s why…

- C and C++ are still successful
  — their base AMM is RAM, and each implementation tweaks that

- “C/C++ with threads” is moderately successful with few threads
  — PRAM on traditional computers is still accurate with few processors

- PRAM with many threads (e.g. CUDA) only successful on accelerators
  — these are the only platforms where PRAM is an accurate model

- Java is hard to “work with” (operationally) with large programs
  — partitioned PRAM is too hard to think about

Abstract machine models — study material

Just one:

Peter van Emde Boas, Handbook of theoretical computer science (vol. A), chapter Machine models and simulations, p. 1-66, MIT Press, 1990,
ISBN 0-444-88071-2

This will teach you how to **quantify AMM adequacy**.

(We have a paper copy at the library!)

Now you're the hero

"Rust keeps the C abstract machine model but innovates on the language interface. "
— someone , 2014

What do *you* think this implies?

Also, "Corrode" https://github.com/jameysharp/corrode . Check it out!

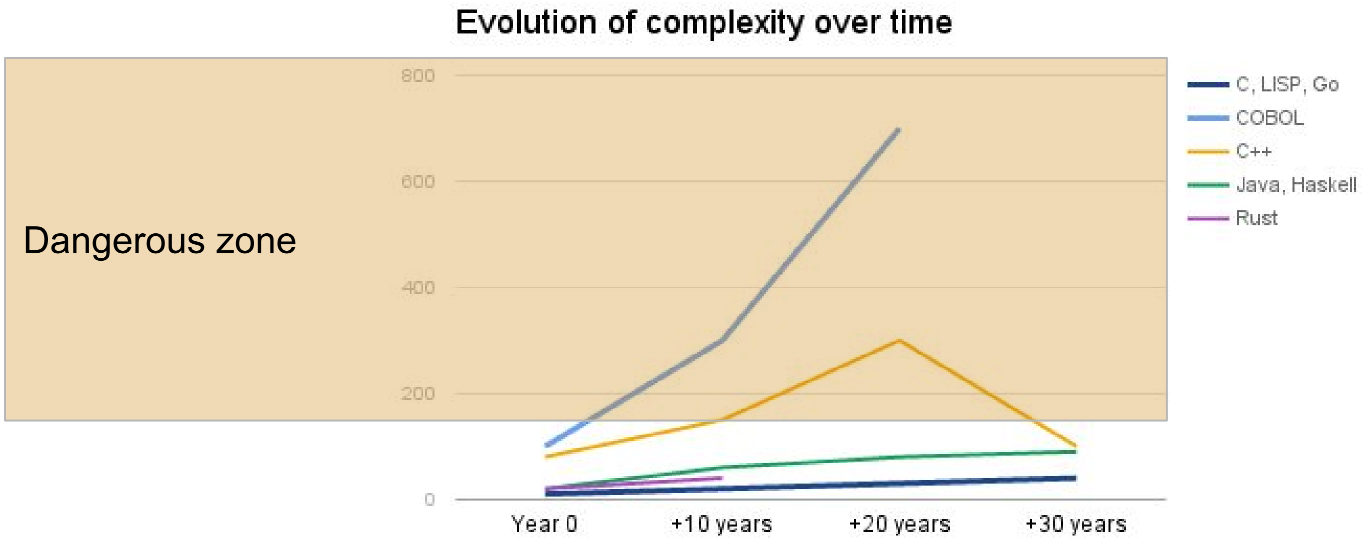
Conceptual models & complexity

Conceptual model: the stuff you need to know before you understand what's going on functionally
Conceptual complexity: **how many pages in the book you need to read(*)**

	Size (book pages)	Examples
Simple — good	Less than 10 pages	LISP, C89, Go, SQL'82 Rust (today)
Moderate — good only if it pays back in *productivity*	Less than 100 pages	ISO C'11 / C++'14 (good) Java (not good) Modern Haskell (good) Rust (probably in 5 years)
Absolutely insane	More than 200 pages	COBOL, SQL'11, C++'03

Conceptual model — as predictors

Method: plot time as X, complexity as Y



Relevance and survival criteria

Relevance:

- Usually phrased as: "is there a need for this?"
- In reality: **"how much are people annoyed with the status quo?"**

Quantify with *"How many man-hours spent to define similar stuff per 10 years"*

- **C, C++, Haskell**: super relevant (*tons* of work in the 70s-80s)
- **Python**: super relevant (*tons* of work in the 90s)
- F#: not very relevant (very little work in 2000-2010)
- Scala, Clojure, Go: moderately relevant
- **Rust**: decide for yourself

Relevance and survival criteria

Survival criteria:

- Usually phrased as “becomes big” (#users, money, literature…)
  That’s only observable in hindsight!
- In reality, *predicted by* **public bus factor + complexity growth + anchors**

Public bus factor (https://en.wikipedia.org/wiki/Bus_factor)
≈ number of public FTEs that need to disappear before the project is dead

Complexity growth: *shape* of the conceptual complexity curve
→ ok under the danger zone; quadratic/exponential: super bad

Anchors: why people keep coming back to it

Survival predictors

Language	Bus factor	Complexity growth	Anchors
Pascal	<50	Near-constant	Approachability
CUDA	0 + NVIDIA	Quadratic, not good	Performance
Python	>1000	Linear, small!	Productivity for fast prototyping
Go	>100 + Google	Linear, small	(I have no idea)
Julia	3	Quadratic, not good	(I have no idea)
Haskell	>100	Inverse quadratic, ok	Purity + expressivity
C	>10000	Linear, moderate	Dark resistance [1]
Rust	>100 + Mozilla	(Maybe too soon to tell)	Modern + Zero overhead link to C

[1] http://science.raphael.poss.name/posts/2014/12/20/dark-resistance/

Features in context

Year 2000 was here!
17 years ago
Also cpus are not faster anymore since then

	FORTRAN	C	C++	Haskell	Java	Go	Rust
Age	64 years	45 years	38 years	30 years	22 years	8 years	7 years
Zero-cost abstractions	✓	✓	✓				✓
Minimal runtime	✓	✓	ಠ_ಠ				✓
Type inference			ಠ_ಠ	✓			✓
Trait-based generics			ಠ_ಠ	✓			✓
Pattern matching / ADTs				✓			✓
Threads without data races	✓	(ノಠ益ಠ)ノ彡┻━┻		✓	ಠ_ಠ	😾	✓
Guaranteed memory safety	✓	(ノಠ益ಠ)ノ彡┻━┻		✓			✓
Design guided by PL experts	✓		ಠ_ಠ	✓	ಠ_ಠ	(ノಠ益ಠ)ノ彡┻━┻	✓
Debuggers & troubleshooting	✓	✓	✓	ಠ_ಠ	✓	ಠ_ಠ	✓

Zero-cost abstractions

C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [Stroustrup, 1994]. And further: What you do use, you couldn't hand code any better. – Bjarne Stroustrup (Rust does this too)

Why: can't really make code the **fastest possible** otherwise

Counter-examples:

- Mandatory **dynamic dispatch** (C++*, Java*, Python, Go*)
- Mandatory **run-time array bounds checking** (Java, Go, Python, Haskell)
- Mandatory **run-time type checking** for conversions (Java, Go, Python)
- Mandatory **garbage collector** (Java, Go, ML, Haskell, Python)

Garbage collection vs. zero overhead

- The programmer's need for conciseness and avoidance of errors
  → demand for **automatic** deallocation ("no explicit free()")
- The **means** by which mem. mgt. is automated:

	Examples	Zero overhead
No management (do dynamic allocation or no deallocation)	FORTRAN77	✓
Eager run-time deallocation via **reference counting**	Python, C++*	
Lazy run-time deallocation via **asynchronous GC** (mark-sweep etc)	Go, Java, Haskell	
Compiler-generated precise deallocation via **linear or affine typing**	Rust, Idris, Clean, C++*	✓

(They call it "borrow checker" in Rust)

Minimal runtime

Run-time system (simplified definition): code+data next to your program without which the program wouldn't run.

"Minimal": count how many bytes in exec + libs: smaller is better

Why: makes **portability** easier, often makes program faster because I-caches

- Simplest "hello world" program in C:
  <100 bytes code+data, runtime optional (in Rust too)
- In **C++:** **100KiB - 1MiB** (also, needs C's entire runtime)
- In **Haskell:** **1MiB - 10MiB** (also, needs C's entire runtime)
- In **Java:** **10MiB - 200MiB** (also, needs C's entire runtime)

“Modern” (40 years old) language features

Pattern matching, Algebraic Data Types, generic functions and data structures:

make code smaller, closer to specifications, easier to read and understand, easier to **maintain and reuse**, easier to formally prove (for correctness)

Type inference:

makes code smaller, easier to read and understand, easier to **maintain and reuse**

(sensing the pattern yet?)

Why: human time is now the most expensive resource in tech.

Safety & Robustness

50 years ago: an error shouldn’t stop the entire computer

40 years ago: an error shouldn’t stop the entire program

30 years ago: an error shouldn’t influence other users sharing the computer

20 years ago: an error shouldn’t kill people or help an adversary to hurt you

10 years ago: an error shouldn’t kill people or help an adversary to hurt you

Now, still after 20 years:

an error shouldn’t kill people or help an adversary to hurt you

Software errors kill people (or nearly do)

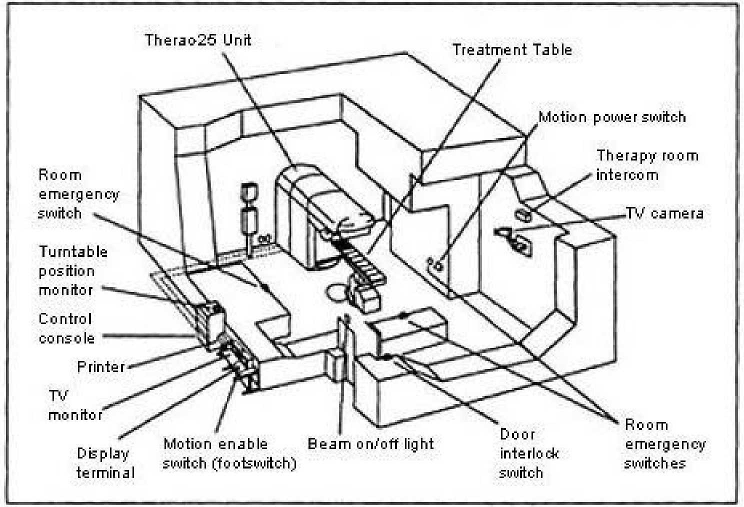


Figure 1. Typical Therac-25 facility

https://en.wikipedia.org/wiki/Therac-25

3 people died because of a race condition in concurrent code

See also:

- Ariane 5 disaster - insufficient data typing
- Toyota brake system
  - improper schedule verification

This can be (oh so easily!) averted with **adequate expressivity** in and **static checks** by programming languages

Software errors are used to hurt people

A.k.a "Malware"

- Fraud
- Impersonation
- Tampering
- Unwanted disclosure
- Blackmail

Enabling technical factors:

- Off-by-one errors
- Buffer overflows
- Stack overflows
- Use-after-free
- Insufficient typing

Malware is a human (non-technical) problem but can be (partly) alleviated by tech solutions

Language-based solutions can achieve (some) **protection by default**

Functional languages got this (mostly) right 40 years ago
But the run-time overhead was a non-starter, until recent innovations

Why you should care - to summarize

If **you create software for work**

And you care about **productivity, performance**, **safety** and **robustness**

Then you'd be seriously irresponsible

unless

you **seriously study 21st century programming languages**

NB: Rust is just an example — other examples: Elixir, Scala

rust-lang.org

Documentation Install Community Contribute

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

See who's using Rust.

Install Rust **1.14.0**

December 22, 2016

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Run

More examples