Go vs. Haskell vs. Rust: Concurrency

Jiaxiao Zhou (周佳孝 Mossaka)

Microsoft

Source Code:

https://github.com/Mossaka/go_rust_haskell_concurrency

About Me (周佳孝 Mossaka)

I'm a software engineer @ Microsoft

Working on tools for data scientists using Azure Machine Learning

Graduated from University of California, San Diego

I am a Programming Language enthusiast

I organize two virtual meetups: PL meetups & Sys meetups

Agenda

1. **History**
2. **A simple resource sharing example**
3. **Safety Haskell 😍**
    a. Monad*
    b. (Bonus) Software Transaction Memory
4. **Safety & Performance ❤️ Rust**
    a. Ownership system
5. **Final Thoughts**

History

2016 - freshman, just heard about Haskell

History

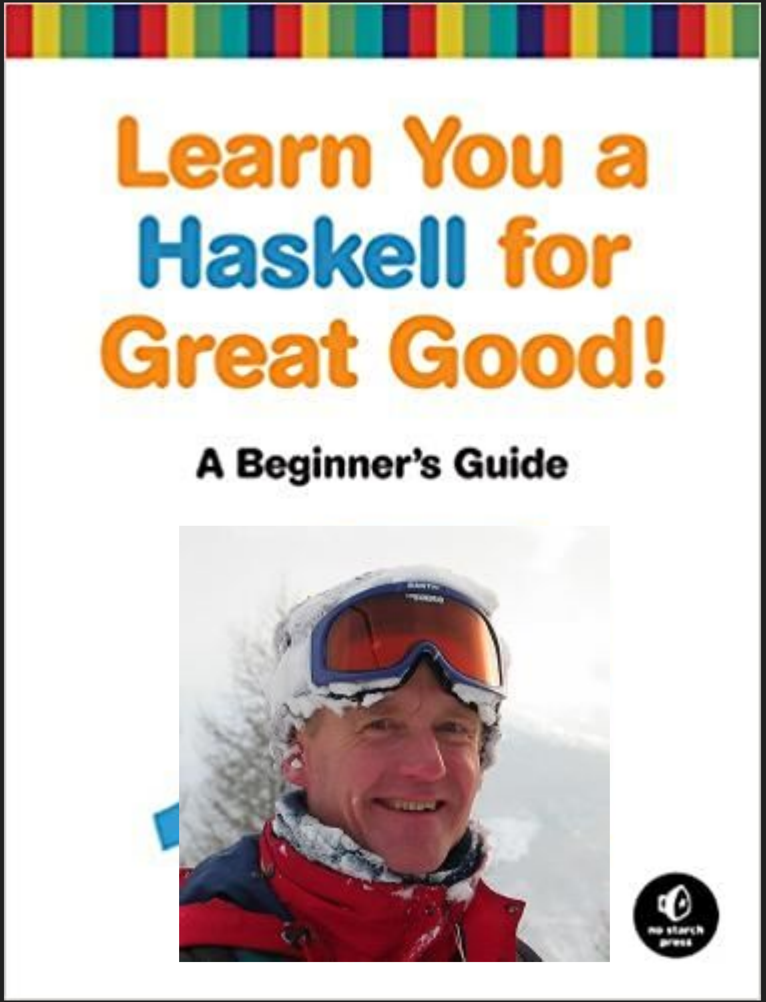
2016 - freshman, just heard about Haskell

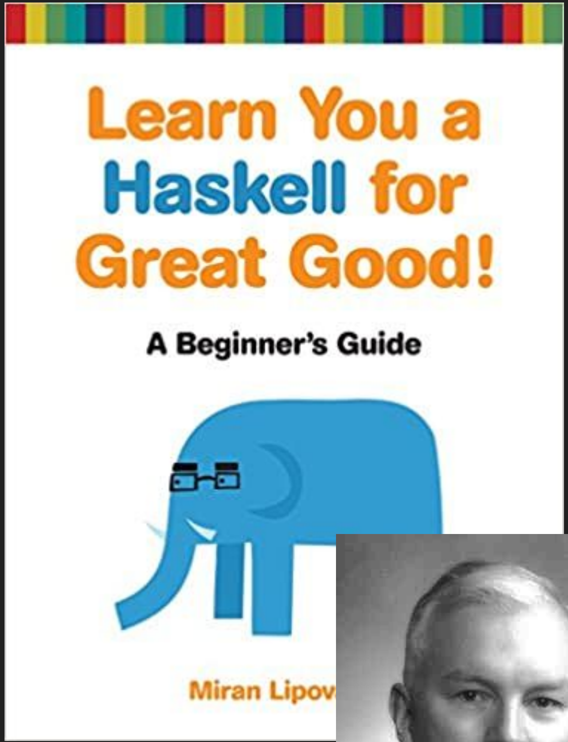
Pure, Statically Typed, Lazy, Functional

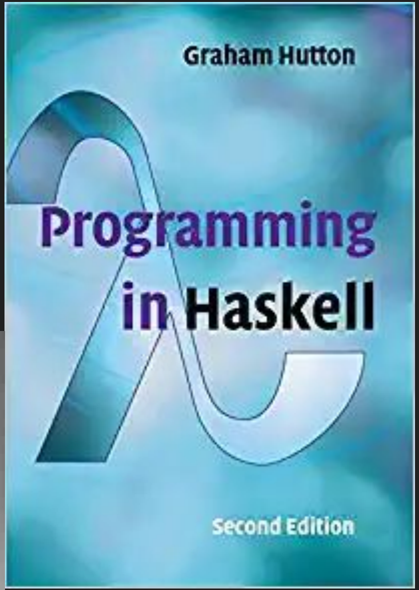
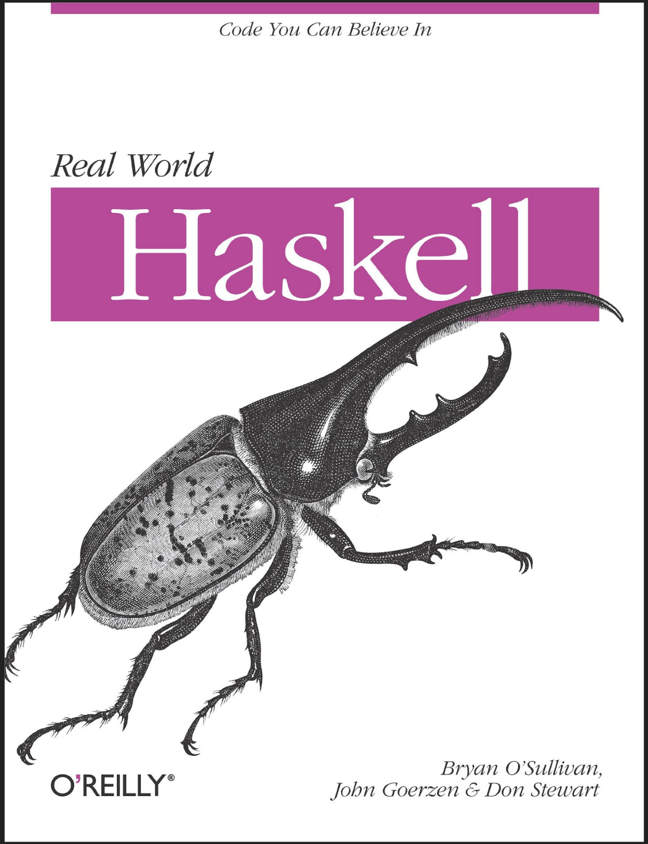
History

2016 - freshman, just heard about Haskell

2018 - learning Haskell seriously




Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipov

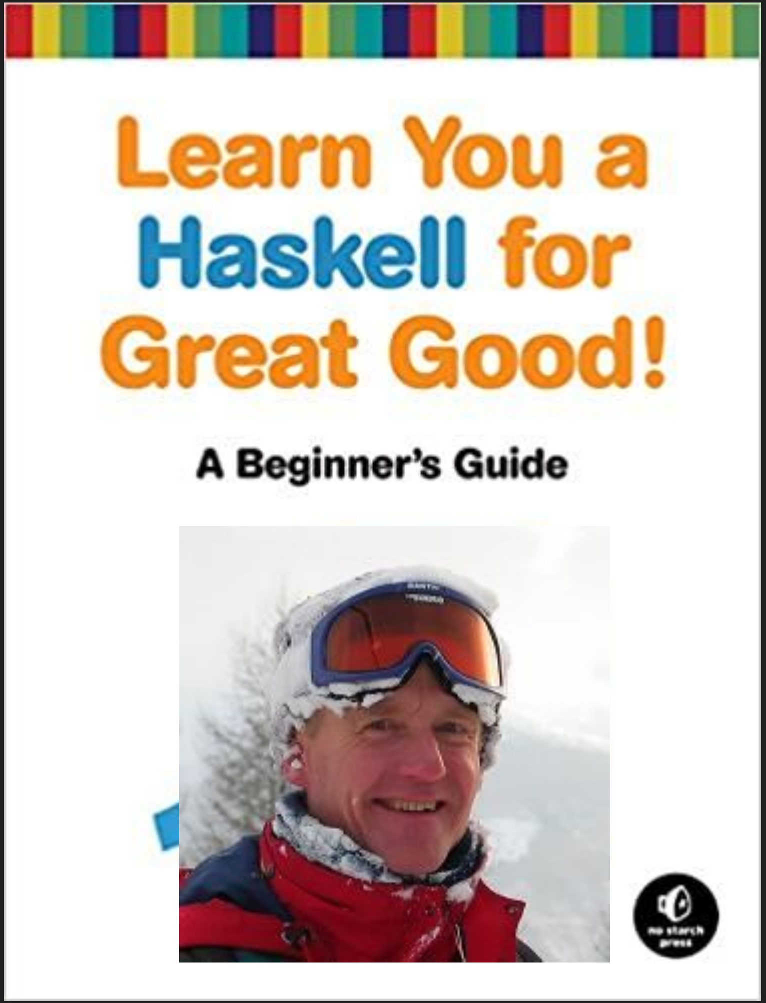

Graham Hutton
Programming in Haskell
Second Edition

Code You Can Believe In
Real World Haskell
O'REILLY
Bryan O'Sullivan, John Goerzen & Don Stewart

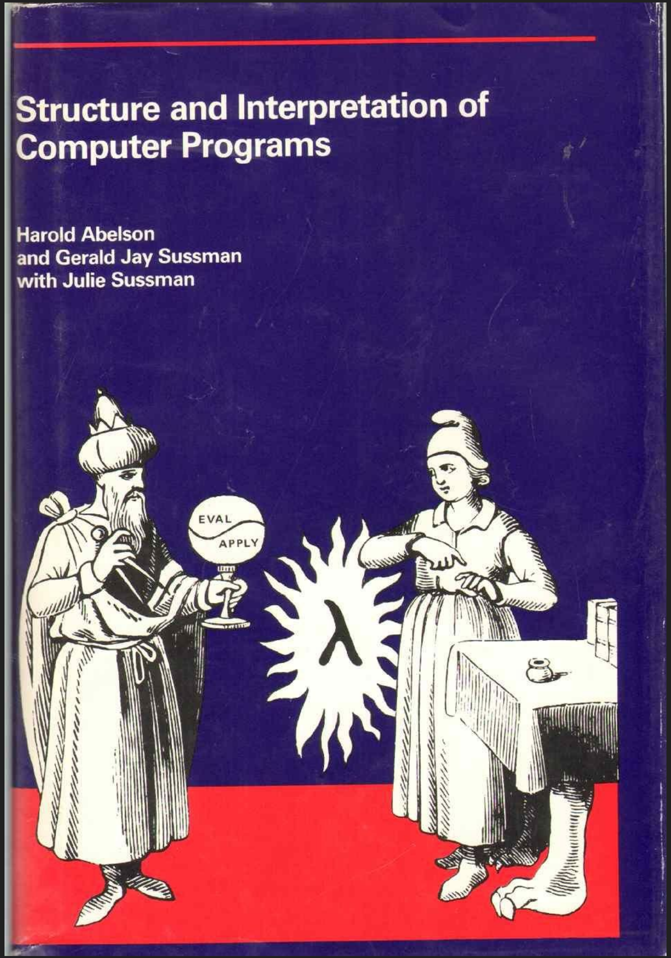
History

2016 - freshman, just heard about Haskell

2018 - learning Haskell seriously

2019 - “zero cost abstraction”





LISP programmers know the value of everything and the cost of nothing

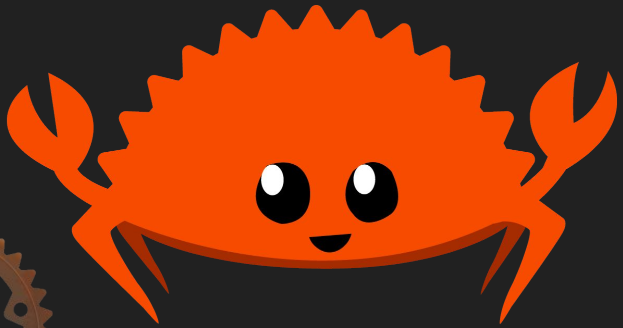
- Alan Perlis

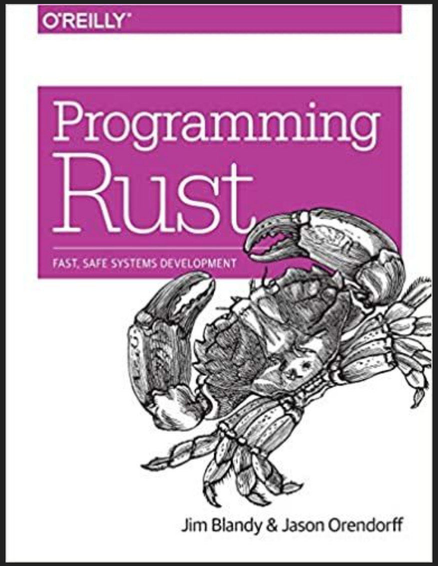
History

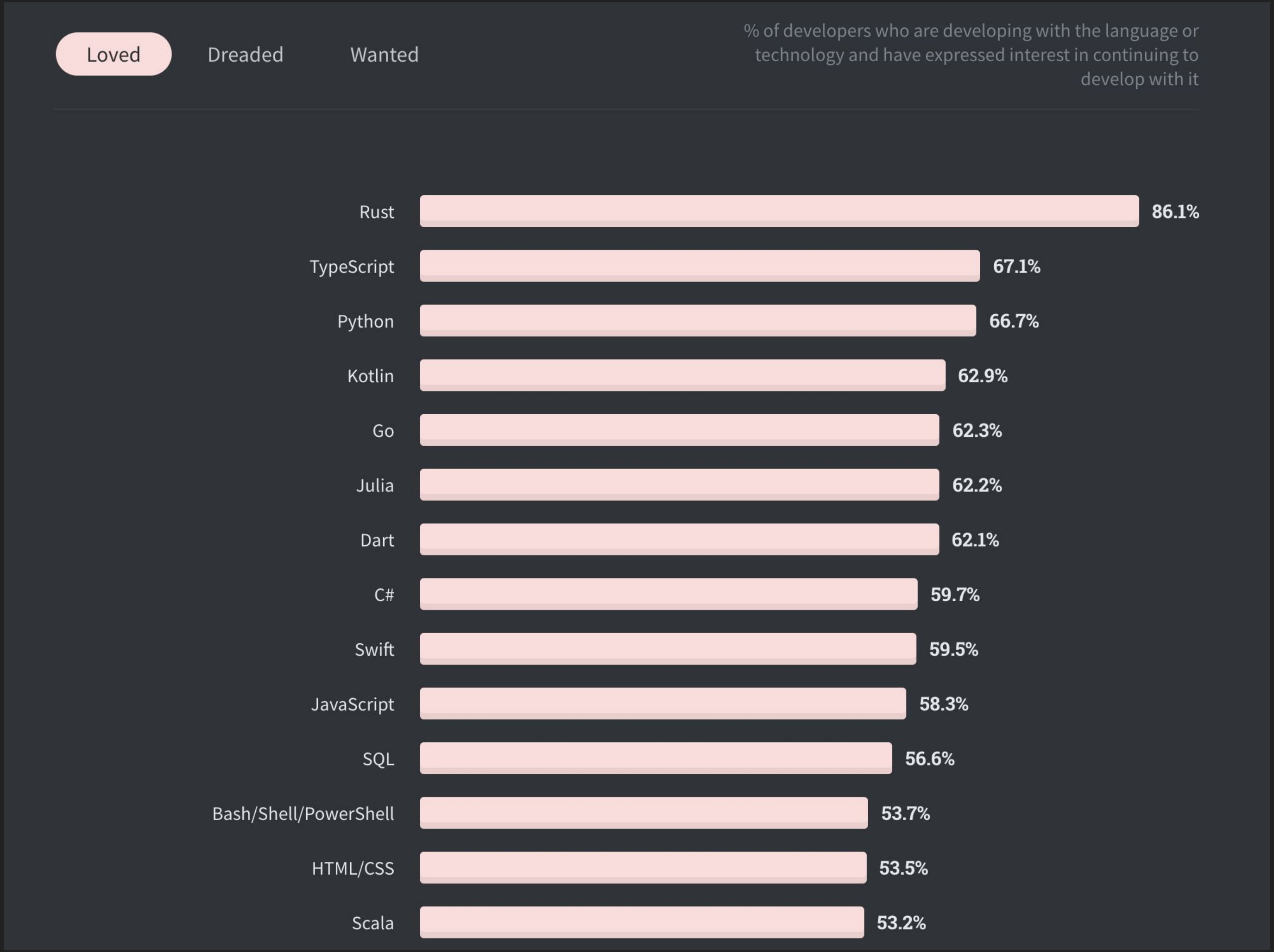
2016 - freshman, just heard about Haskell

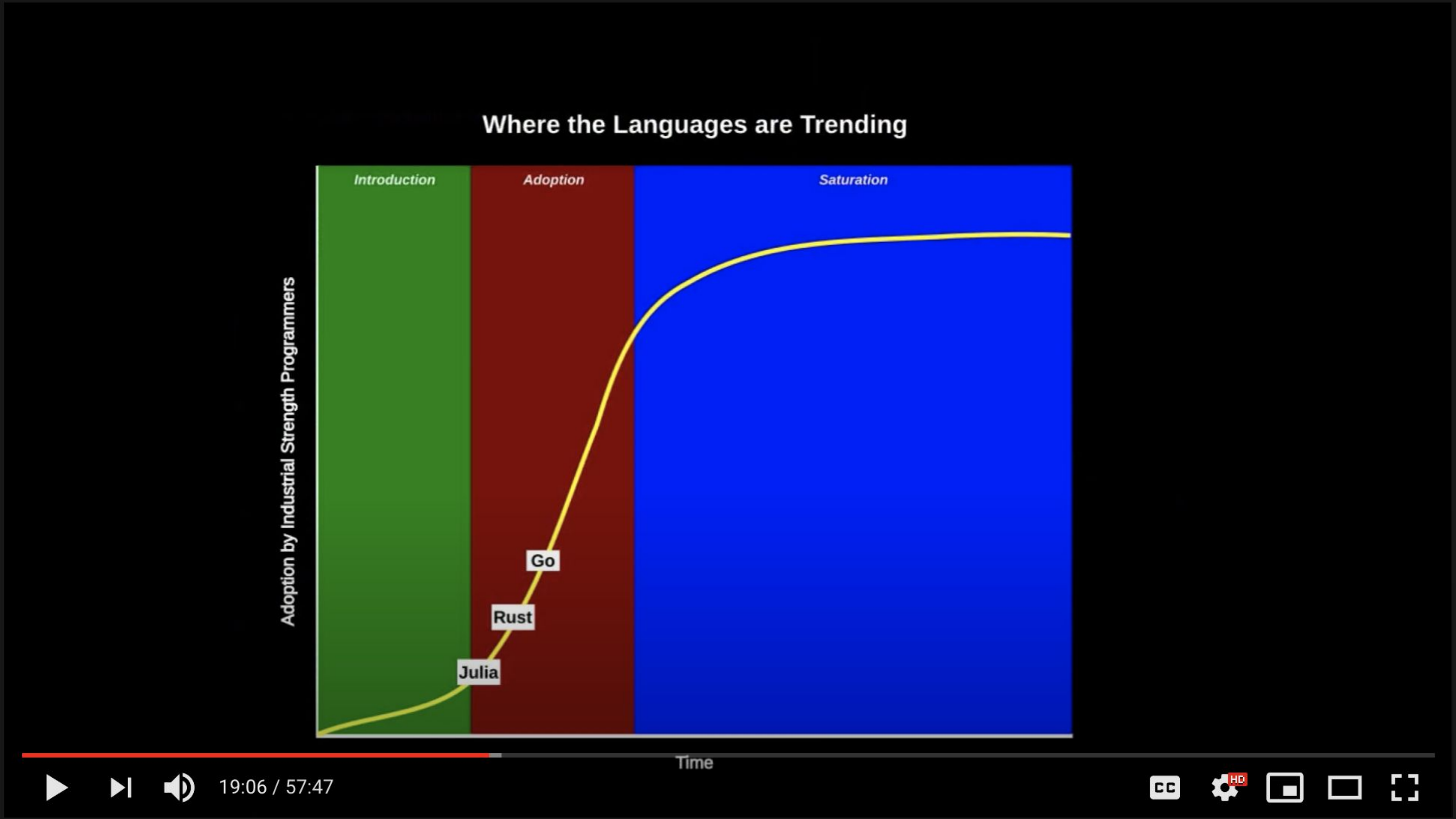
2018 - learning Haskell seriously

2019 - “zero cost abstraction” - Rust






Loved
Dreaded
Wanted
% of developers who are developing with the language or technology and have expressed interest in continuing to develop with it
Rust 86.1%
TypeScript 67.1%
Python 66.7%
Kotlin 62.9%
Go 62.3%
Julia 62.2%
Dart 62.1%
C# 59.7%
Swift 59.5%
JavaScript 58.3%
SQL 56.6%
Bash/Shell/PowerShell 53.7%
HTML/CSS 53.5%
Scala 53.2%



Rust, Julia, and Go: Disruptive New Programming Languages Changing the Face of Computing

33,495 views • Jun 29, 2018

617 29 SHARE SAVE

History

2016 - freshman, just heard about Haskell

2018 - learning Haskell seriously

2019 - “zero cost abstraction” - Rust

2020 - Disruptive languages: Go, Rust, Julia

A common claim

Haskell, Rust, Go all claim easy and painless communications between shared resources.

Concurrency

Share by communicating

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

> Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

Fearless Concurrency with Rust

Apr. 10, 2015 · Aaron Turon

The Rust project was initiated to solve two thorny problems:

- How do you do safe systems programming?
- How do you make concurrency painless?

Initially these problems seemed orthogonal, but to our amazement, the solution turned out to be identical: **the same tools that make Rust safe also help you tackle concurrency head-on**.

Concurrent

Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions.

Click to expand

	Thread-safe	Features	Mutex support	Channel support
Go	No	Go Memory Model	Yes	Yes
Haskell	Yes	Immutability	Yes	Yes
Rust	Yes	Ownership	Yes	Yes

How can we write a simple concurrent program in Go, Rust, and Haskell? 🤔🤔

Let’s start with a basic example

```
{
 n = n + 1;
}
```

```go
for i := 0; i < 1000; i++ {
    go func() {
        mu.Lock()
        n++
        mu.Unlock()
    }()
}
time.Sleep(2 * time.Second)
fmt.Printf("n=[%d]\n", n)
```

```haskell
main = do
    result <- forConcurrently (replicate 1000 1) (\x -> return x);
    putStrLn $ show $ sum result
```

```rust
let safe_data = Arc::new(Mutex::new(0));
let handlers = (0..1000)
    .into_iter()
    .map(|_| {
        let data = safe_data.clone();
        thread::spawn(move || {
            *data.clone().lock().unwrap() += 1;
        })
    })
    .collect::<Vec<std::thread::JoinHandle<_>>>();
for thread in handlers {
    thread.join().unwrap();
}
println!("Data: {:?}", safe_data);
```

How can Haskell be that simple??

Simple answer: I cheated.

It is impossible to express `n = n + 1` in Haskell, because Haskell is pure.

Pure: a program is an expression that evaluates to a value

No side effect - ❌ n++, ❌ print(“hello”) ❌ launchMissile()

Haskell has no side effect

Really!

```haskell
sort []     = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
    ls      = [ l | l <- xs, l <= x ]
    rs      = [ r | r <- xs, x <  r ]
```

It’s a great gain for Parallelism.

Every sub-expression can be evaluated <u>in parallel</u>.

Haskell is functional

Functional = functions are first-class values

1. Functions can be assigned to a variable
2. Functions can be passed to other functions
3. Functions can be returned from other functions

```haskell
map :: (a → b) → [a] → [b]
foldl :: (b → a → b) → b → [a] → b
filter :: (a → Bool) → [a] → [a]
```

Demo Time! ~~🤟~~

Haskell will Blow up your Mind

https://wiki.haskell.org/Blow_your_mind

Wait a minute... 🤔

If Haskell has no side effects, how can I simply print something? How can I do concurrently programming?

Monad comes to rescue

Think monad as a interface (typeclasses) that requires 2 functions.

```haskell
class Monad m where
  bind   :: m a → (a → m b) → m b
  return :: a → m a
```


dont worry about it if you don't
understand

Still confused?

I don’t expect you to learn monad in a few minutes...

I recommend this visualization for understanding Monad

https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

IO Monad

The most common monad (You have seen it)

```haskell
putStrLn :: String → IO ()
getLine :: IO String

do putStrLn "Write your string: "
   string ← getLine
   putStrLn (shout string)
```

```haskell
main = do
    result ← forConcurrently (replicate 1000 1) (\x → return x);
    putStrLn $ show $ sum result
```

After 30 years of research, the most widely-used coordination mechanism for shared-memory task-level concurrency is....

(Bonus) Software Transactional Memory

atomic { … sequential get cod }

Just write the seq , and wrap atomic around it

All-or-noth antics: Atomic commit

Atomic block executes in Isolation

Cannot Deadlock

(Bonus) STM How does it work?

Execute without taking any locks

Each read and write in is logged to a thread-local <u>transaction log</u>

Writes go to the log only, not to memory

At the end, the transaction tries to commit to memory

Commit may fail; then transaction is <u>re-run</u>

To Summarize

Haskell is pure - no side effects!

Haskell is functional - functions are first-class citizens!

Haskell provides advanced type system (monad) to do effectful concurrent programming.

No primitives are used if we are using STM.

We just write high-level Haskell code and be happy 😍!

Our journey won’t stop here...

Enter Rust!

- A system programming language
- Functional
- Gives developers fine control of memory usage
- *“What you don’t use, you don’t pay for”*
- Ownership system: moves, borrows
- Compile time preventing data races!

Promises

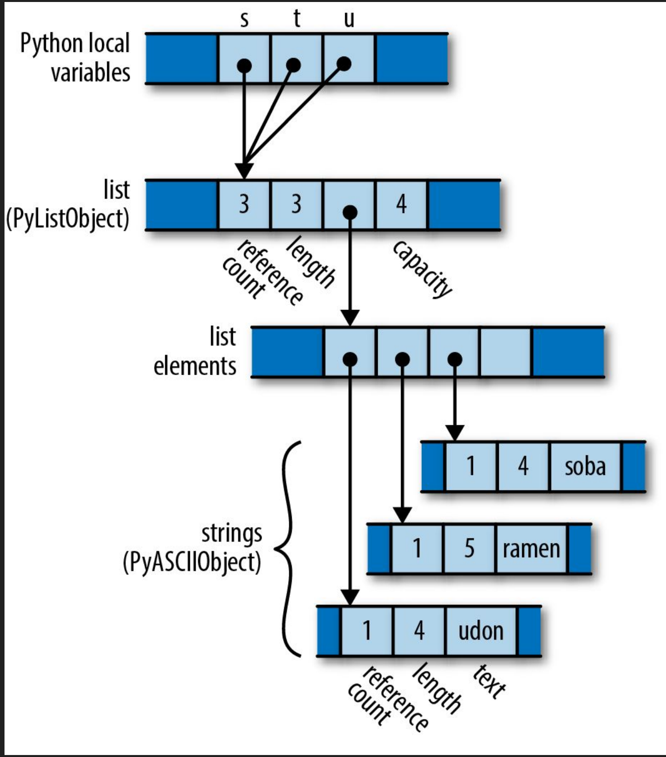
Rust promises:

1. *You decide the lifetime of each value in your program. Rust frees memory at a point under your control*
2. *Your program will never use a pointer to an object after it has been freed*

C, C++ keep the first promise, but the second promise is set aside. Go, Haskell keep the second promise using *garbage collector*, but you won't have control when objects get freed

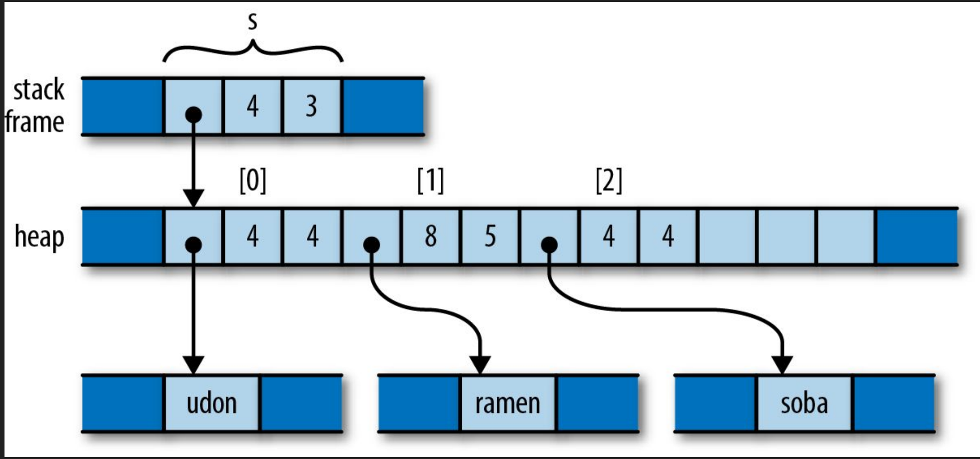
The Memory Model

```python
s = ['udon', 'ramen', 'soba']
t = s
u = s
```



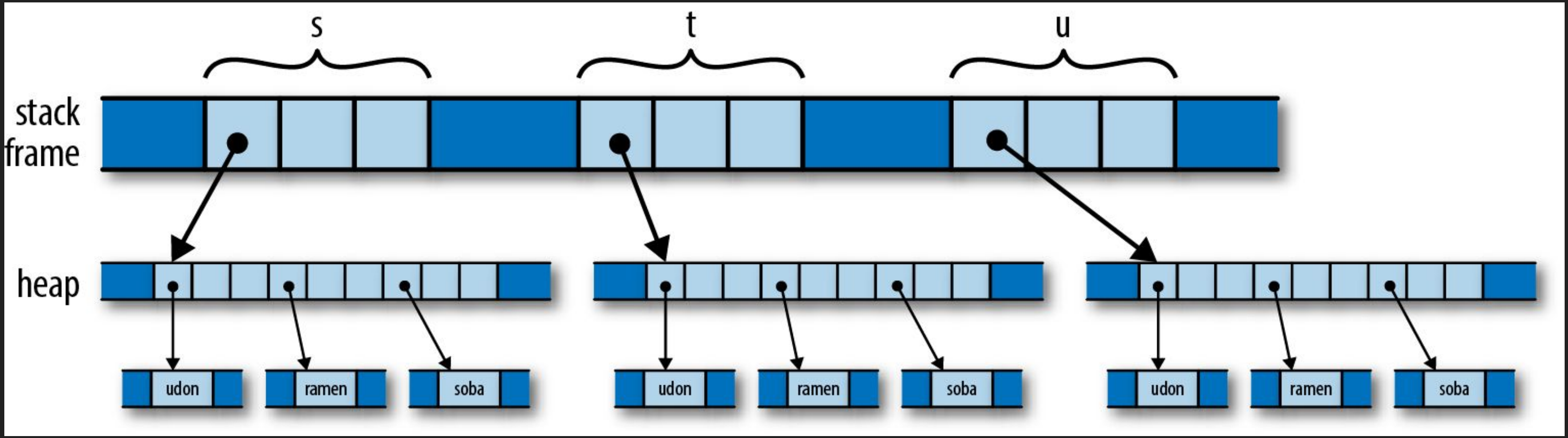
The Memory Model

```cpp
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```



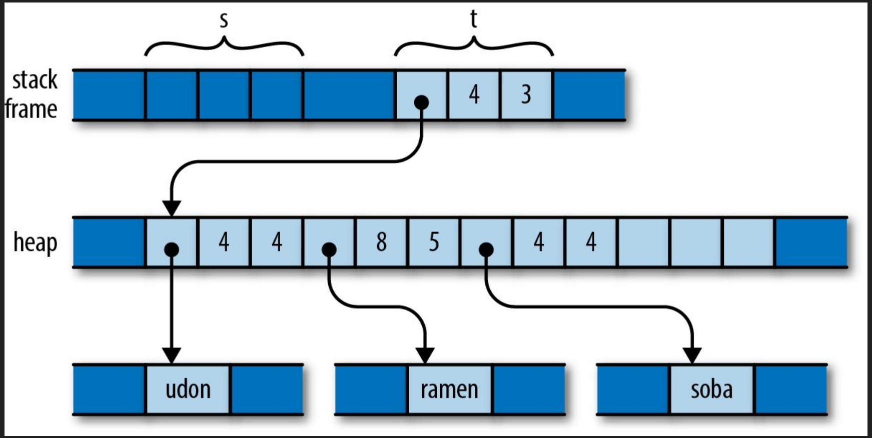
The Memory Model

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```



The Memory Model

```
let s = vec!["udon".to_string(),
"ramen".to_string(), "soba".to_string()];
let t = s;
let u = s;
```



Rust compiler would fail this program!

Variable t moves the ownership from s

s is uninitialized

More about move

```rust
let x = vec![10, 20, 30];
if c {
    f(x); // ok to move from x here
} else {
    g(x); // ok to also move from x here
}
h(x) // bad: x is uninitialized here if
either path uses it
```

```rust
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first
iteration
```

Borrow!

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

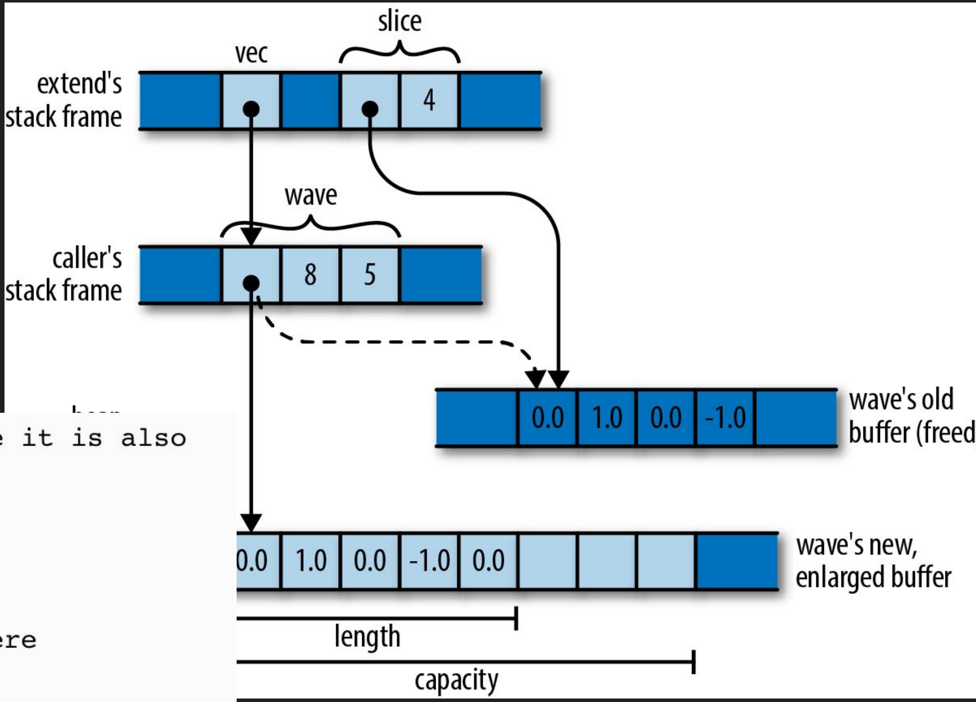
Multiple Shared borrow

Single mutable borrow

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1);  // bad: reads memory
`x` used to occupy
}
```

Why is it powerful?

```
fn extend(vec: &mut Vec<f64>, slice: &[f64])
{
    for elt in slice {
        vec.push(*elt);
    }
}
let mut wave = vec!{0.0, 1.0, 0.0, -1.0}
```




```
error[E0502]: cannot borrow `wave` as immutable because it is also
borrowed as mutable
 --> references_sharing_vs_mutation_2.rs:9:24
  |
9 |     extend(&mut wave, &wave);
  |                 ----   ^^^^- mutable borrow ends here
  |                 |      |
  |                 |      immutable borrow occurs here
  |                 mutable borrow occurs here
```

Why is it powerful?

1. ❌ NO *dangling pointers*
2. ❌ NO double freeing
3. ❌ NO uninitialized memory
4. ❌ NO forgotten to unlock your Mutex
5. Rust compiler is your superpower 🚀

What's about Mutex?

Mutex prevents data race

Data race gives undefined behavior in C++ and Go 🙃

Unlike Go, in Rust the protected data is stored inside the Mutex.

You NEVER need to unlock the Mutex by yourself 😳

What’s about Mutex?

```
let data = Arc::new(Mutex::new(0));
thread::spawn(move || {
        let mut data = data.lock().unwrap();
        *data += 1;
 });
```

Back to the shared resource example

```
let safe_data = Arc::new(Mutex::new(0));
let handlers = (0..1000)
    .into_iter()
    .map(|_| {
        let data = safe_data.clone();
        thread::spawn(move || {
            *data.clone().lock().unwrap() += 1;
        })
    })
    .collect::<Vec<std::thread::JoinHandle<_>>>();
for thread in handlers {
    thread.join().unwrap();
}
println!("Data: {:?}", safe_data);
```

Here's a taste of concurrency in Rust:

- A channel transfers ownership of the messages sent along it, so you can send a pointer from one thread to another without fear of the threads later racing for access through that pointer. **Rust's channels enforce thread isolation.**

- A lock knows what data it protects, and Rust guarantees that the data can only be accessed when the lock is held. State is never accidentally shared. **"Lock data, not code" is enforced in Rust.**

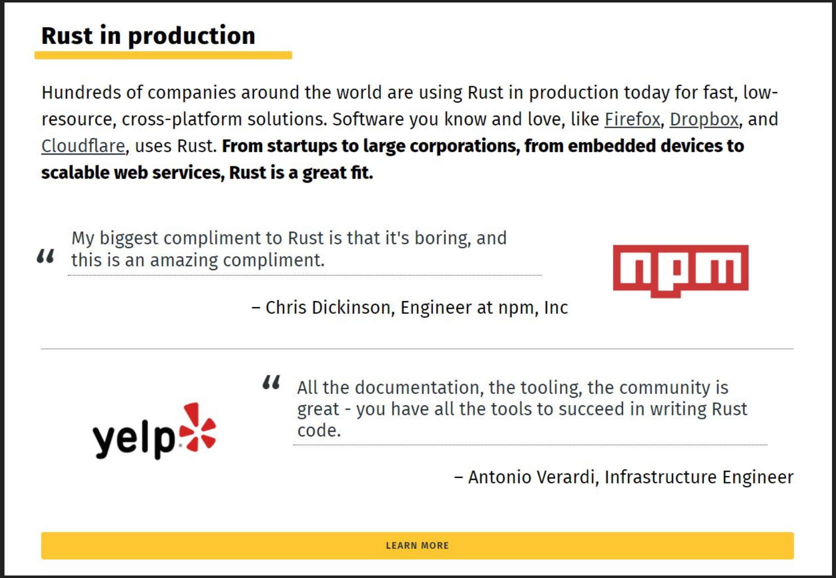
- Every data type knows whether it can safely be sent between or accessed by multiple threads, and Rust enforces this safe usage; there are no data races, even for lock-free data structures. **Thread safety isn't just documentation; it's law.**

- You can even share stack frames between threads, and Rust will statically ensure that the frames remain active while other threads are using them. **Even the most daring forms of sharing are guaranteed safe in Rust**.

Final Thoughts

Who’s using Haskell? [Haskell in industry]

Who’s using Rust?



Final Thoughts

I love Haskell! Where should I get started?

- Learn You a Haskell for Great Good!
- Real world Haskell

I love Rust! Where should I get started?

- Programming Rust
- Fearless Concurrency with Rust