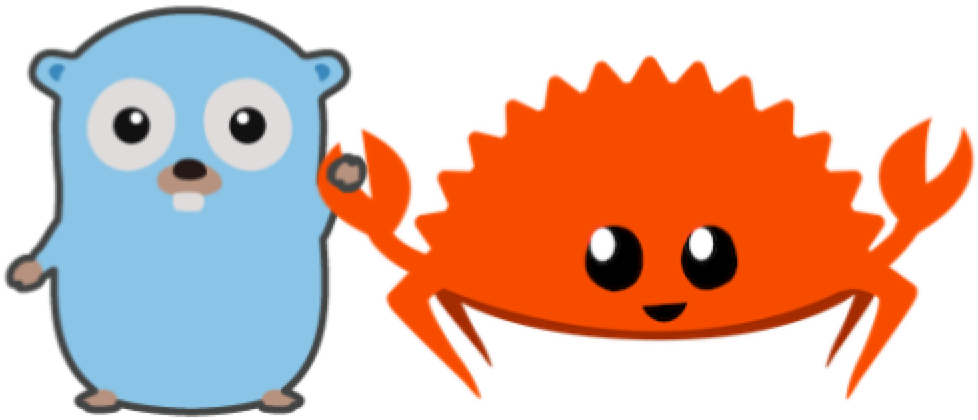
Rust from Go's Perspective

or, Rust for Gophers



@damienstanton

Senior Machine Learning Engineer @ SignalFrame

- I wear many hats, working in several languages (among them Go, Scala, and Rust)
- I've been writing Go professionally, and Rust on/off since ~2014
- I think Rust is the future of systems programming

What's Rust?

> A language empowering everyone to build reliable and efficient software.

- Native cross-platform binaries ✅
- Statically-typed ✅
- Designed for writing concurrency-safe and memory-safe code ✅
- First-class WebAssembly support (which is important)
- No runtime
- Robust type system, first-class macros

Package management & documentation

`crates.io`

`docs.rs`

Act 1: The Rust build system & tools

`rustup` ✓

`cargo` ✓

Under the hood

`rustc` `rustdoc`

Third-party "components"

`rls` `rust-analysis` `rust-src` `clippy`

DEMO 😬

Let's dive in through a series of examples

This example will help us understand:

- Strings
- Mutability
- References
- Closures

```go
package main

import (
	"fmt"
)

func calculate_len(s string) int {
	return len(s)
}

func main() {
	s1 := "Hello"
	fmt.Println(calculate_len(s1))
}
```

DEMO 😬

A tale of two strings

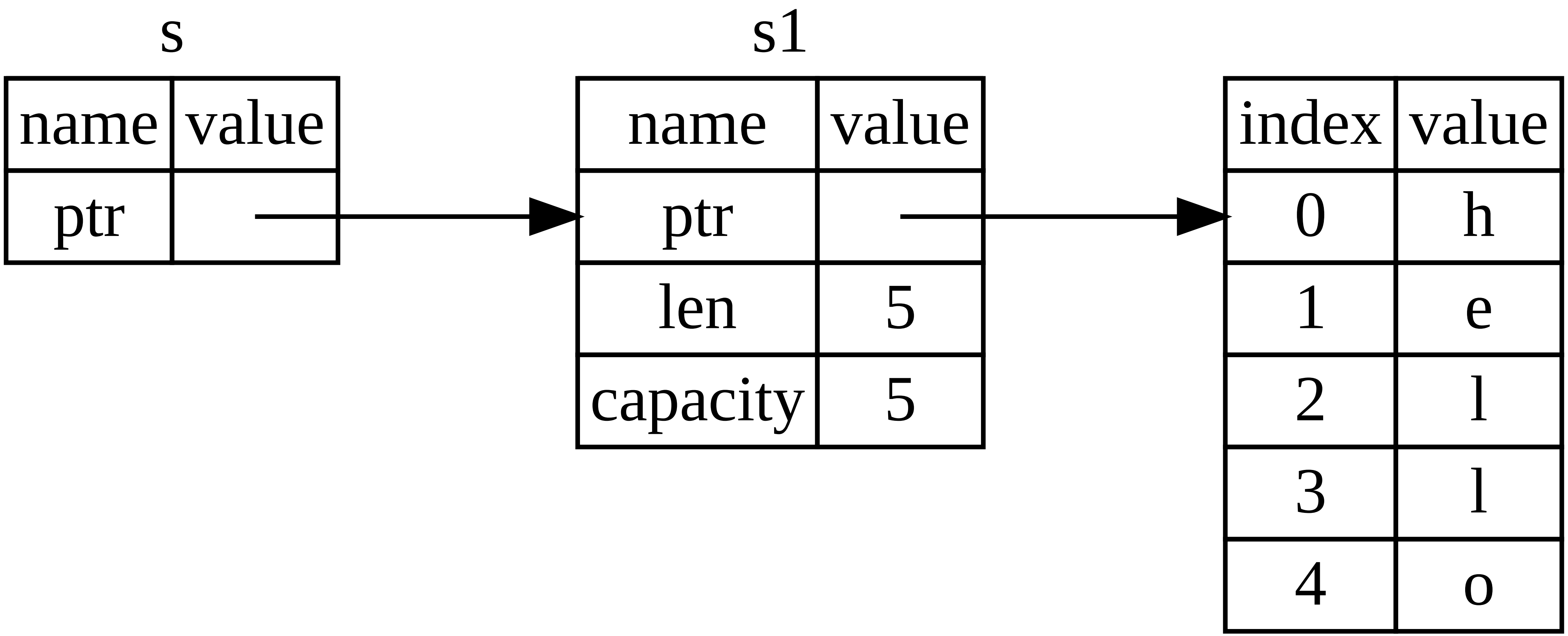
`String` vs `str`

- `String`s are owned.
- `str`s are *string slices*, and slices do not have ownership.
- String literals are a *reference to a string slice*, not a `String`!

Don't worry --if this is unclear-- it will become clearer as we go.

```
fn calculate_len(s: &String) -> isize {
    s.len() as isize
}

let mut s1 = String::from("Hello");
s1 += ", gophers";
let len = calculate_len(&s1);
```



Act 2: The Rust memory model

Ownership

- Each value in Rust has a variable that’s called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

References

- `&T` means take a *reference* to `T`
- `*T` means *dereference* `T`: that is, follow the pointer to the real value

Borrowing

- Using references as function parameters is what is called *borrowing*
  - Say `foo()` is given a parameter (call it `&MyType`)
  - ... When `foo()` returns (and its scope is dropped)
  - ... the *reference* is dropped, not the original value
  - So, `foo()` only ever *borrowed* `&MyType`; it never *owned* `MyType`

Borrowing (continued)

- Strictly allowing values to live in memory within a certain scope (and lifetime) is the heart of Rust's memory model.
- This is how Rust is able to not have a garbage collector, but also not require pointer arithmetic or `malloc` / `free`
- Based in part on `Grossman et al. 2002`

This example will help us understand lifetimes:

```go
package main

import (
	"fmt"
)

func longestString(a, b string) string {
	if len(a) == len(b) {
		return "Neither"
	}
	if len(a) > len(b) {
		return a
	}
	return b
}

func main() {
	s1 := "Java"
	s2 := "C++"

	fmt.Println(longestString(s1, s2))
}
```

DEMO 😬

Smart Pointers

All about flexibly of ownership, "interior" type mutability, and allocation

- `Box<T>`: allocate `T` on the heap instead of the stack
- `Rc<T>` and `Arc<T>`: Reference a heap-allocated `T`
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`
  - Whenever we need *multiple references* to `T` **and** to *mutate* `T`

We won't have time to detail these, but documentation on these modules is 💯

Safer code with `Option` and `Result`

- Handling null and error conditions revolve around these data structures:

```
pub enum Option<T> {
    None,
    Some(T),
}

pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

This example will help us understand:

- Attributes, and how to use `Option` & `Result`

```go
package main

import (
	"fmt"
	"errors"
)

type Point struct {
	name *string
	x int
	y int
}

func (p *Point) getName() (string, error) {
	if p.name == nil {
		return "", errors.New("No name set")
	}
	return *p.name, nil
}

func main() {
	p := Point{}
	name, err := p.getName()
	if err !=nil {
		fmt.Printf("Error: %v", err)
	}
	fmt.Println(name)
}
```

DEMO 😬

`trait` vs `interface`

io.Writer
*gzip.Writer
[]bool
*strings.Reader
io.Reader
int
*os.File
io.ReadWriter

```go
package main

import "fmt"

type Summary interface {
	Summarize() string
	SummarizeAuthor() string
}

type Tweet struct {
	username string
	content string
	reply bool
	retweet bool
}

func (t *Tweet) SummarizeAuthor() string {
	return fmt.Sprintf("@%s", t.username)
}

func (t *Tweet) Summarize() string {
	return fmt.Sprintf("Read more from %s...", t.SummarizeAuthor())
}

func foo(s Summary) {
	fmt.Println(s.Summarize())
}

func main() {
	p := Tweet{
		username: "damienstanton",
	}
	foo(&p)
}
```

- Just like Go interfaces, traits parameterize *behavior* instead of data shape

```
// in parameters
pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

// and in return values
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("damienstanton"),
        content: String::from("Hi, everyone."),
        reply: false,
        retweet: false,
    }
}
```

- Traits are not exactly like interfaces, however

```rust
// default and self-referential methods
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}

// implementations are explicit in Rust, not implicit like in Go
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

```rust
println!("1 new tweet: {}", tweet.summarize());
// 1 new tweet: (Read more from @damienstanton...)
```

- When combined with generics, we can almost get *Haskell*-like typeclass behavior

```rust
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
    // ...
}
```

Act 3: Concurrency

`thread` + `sync::mpsc` vs `chan`

`task` vs `go[routine]`

Threads & channels

```rust
use std::thread;
use std::sync::mpsc::channel;

let (tx, rx) = channel();

thread::spawn(move|| {
    tx.send(10).unwrap();
});

assert_eq!(rx.recv().unwrap(), 10);
```

```rust
use std::thread;
use std::sync::mpsc::channel;

let (tx, rx) = channel();
for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move|| {
        tx.send(i).unwrap();
    });
}

for _ in 0..10 {
    let j = rx.recv().unwrap();
    assert!(0 <= j && j < 10);
}
```

What about coroutines?

This one is tricky, because:

- `futures` are becoming part of the stdlib
- `async` / `await` keywords are being added to the language
- abstract `tasks` (thread-like, similar to goroutines) are being added to the stdlib

Third-party solution: tokio.io

The new standard: futures

See the `futures_examples` code in this repo to get the flavor

- If you come from `JS` / `C++` / `Python` / `C#`, you'll already be familiar with this

`async` / `await`, `task`, `future`

```rust
pub trait Spawn {
    fn spawn_obj(
        &mut self,
        future: FutureObj<'static, ()>
    ) -> Result<(), SpawnError>;

    fn status(&self) -> Result<(), SpawnError> { ... }
}
```

```rust
pub async fn simple_stream() {
    let stream = stream::iter(1..=3);
    let (first, stream) = stream.into_future().await;
    assert_eq!(Some(1), first);
    let (second, _) = stream.into_future().await;
    assert_eq!(Some(2), second);
}
```

`async/await` is an ongoing & **major** change to the Rust concurrency ecosystem

Epilogue: The release cycle / community

- RFC
- The Nursery
- The Rust Forge

Many more links in this talk's repo 👇

https://github.com/damienstanton/rfg

Q/A