Rust

Vitaly “_Vi" Shukela

URI of this presentation:

`http://vi-server.org/pub/rust.pdf`

source:

`http://vi-server.org/pub/rust.md`

Intro

Rust is a general purpose system programming language.

```
fn main() {
    let mut n = 1; // A counter variable
    while n < 101 {
        let msg : String;
        if n % 15 == 0     {  msg = "fzbz".to_string(); }
        else if n % 3 == 0 {  msg = "fz".to_string(); }
        else if n % 5 == 0 {  msg = "bz".to_string(); }
        else {  msg = format!("{}", n);
        }
        println!("{}", msg);
        n += 1;
    }
}
```

Competitor to

C++

*At Mozilla, there is a sign on the wall behind one of our engineer's desks. The sign has a dark horizontal line, below which is the text, '**You must be this tall to write multi-threaded code.**' The line is roughly nine feet off the ground. We created Rust to allow us to lower that sign.*

Quotes (1/2)

Go feels like a bunch of older C programmers listing their issues with writing concurrent C code.

Rust feels like a bunch of Haskell programmers listing their issues with C++.

Quotes (2/2)

*Something I think Rust gets right, the community, the book, even the compiler, is that it **does** believe in you,*

but it also doesn't expect you to just "get it" overnight.

*There is a **tone** to the literature, to the compiler messages, to the IRC channel, which is this:*

"we understand."

Why?

- Backed by a relatively big thing (Mozilla) and also with a great community
- Designed in a distributed manner, but does not give “design by committee” feeling
- Relatively few new languages target systems programming
- Interesting core ideas:

Core ideas

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead
- Stability without stagnation

Don't invent things, just borrow ideas from other languages and implement them correclty.

Inspirations (1/2)

- SML, OCaml: algebraic data types, pattern matching, type inference, semicolon statement separation
- C++: references, RAII, smart pointers, move semantics, monomorphization, memory model
- ML Kit, Cyclone: region based memory management
- Haskell (GHC): typeclasses, type families
- Newsqueak, Alef, Limbo: channels, concurrency

Inspirations (2/2)

- Erlang: message passing, thread failure, ~~linked thread failure~~, ~~lightweight concurrency~~
- Swift: optional bindings
- Scheme: hygienic macros
- C#: attributes
- Ruby: ~~block syntax~~
- NIL, Hermes: ~~typestate~~
- Unicode Annex #31: identifier and pattern syntax

Projects

- rustc - Rust compiler and associated tools
- servo - Layout engine for Mozilla

Other things like OpenDNS, and a lot of early things like Piston game engine.

Some features

- `&'borrow checker`
- `Generics<T>`
- `macros!()`, compiler plugins
- Sum types
- `match` with guards and destructuring, `if let`
- Haskell-style derivations for types, limited type inferece
- Iterators
- Doctests
- Powerful traits (associated types, default implementations, etc.)
- Error handling: Result and panic

Rust uses LLVM.

Borrow checker

There are three types of values:

- Real, owned thing.

```
fn qqq(self) {}
```

- Reference with write access.

```
fn qqq(&mut self) {}
```

- Reference with read-only access.

```
fn qqq(&self) {}
```

`g.qqq();`

- owned - `qqq` takes responsibility of deallocating `g`
- `&mut` - `qqq` can change `g` knowing that no other references exist
- `&` - `qqq` can only read `g`.

Borrow checker example

```
fn eat_box(boxed_int: Box<i32>) {
    println!("destroying box containing {}", boxed_int);
}
fn peep_inside_box(borrowed_int: &i32) {
    println!("This int is: {}", borrowed_int);
}
fn main() {
    let boxed_int = Box::new(5);
    peep_inside_box(&boxed_int);
    peep_inside_box(&boxed_int);
    {   let _ref_to_int: &i32 = &boxed_int;
        eat_box(boxed_int); /* FAILS */
    } // reference goes out of scope;
    eat_box(boxed_int);
}
```

Types and derivations

```
#[derive(Debug, Eq, PartialEq, Ord, PartialOrd)]
struct SomeEntry {
    pub q : String,
    w : i32,
};

#[derive(Copy)]
enum Q {
    Variant1,
    Variant2(usize),
}
```

Generics

```
struct Qqq<T> {
    w: Vec<T>,
    r: Vec<f64>,
}

fn qqq<T> (q: &Qqq<T>) -> f64 {q.r[0]}

qqq::<bool>(&someval);
```

Rust generics are not like C++ templates in that they are fully type-checked before monomorphization.

Lifetimes (1/4)

When pointers are involved, Rust prefers to be complicated rather than unsafe.

```
struct Qqq<'a, T> {
    w: Vec<T>,
    r: Vec<&'a f64>,
}

fn qqq<'a, T> (q: &Qqq<'a, T>) -> &'a f64 { q.r[0] }
```

Lifetimes (2/4)

```
fn choose(j:&i32, k:&i32) -> &i32 { j }

fn main() {
    let x = 420;
    let refff : &i32;
    {
        let y = 31337;
        refff = choose(&x, &y);
    } //////// y 's (therefore &y's) lifetime ends here
    *refff;
}
```

```
error: missing lifetime specifier [E0106]
```

```
fn choose<'a,'b>(j:&'a i32, k:&'b i32) -> &'a i32 { j }
```

Lifetime hierarchy

Lifetimes form hierarchy.

```
'a: {
    let j = f1();
    'b: {
        let k = f2();
    }
}
```

`'a` is a “subtype” of `'b`
This means that `'a` may be used everywhere where `'b` can be used.

'static lifetime

`'static` lifetime means:

- For a type: no `<'a>` involved in definition
- For a reference: it will live forever (example: string literal)

Macros

```
macro_rules! o_O {
    (  $(
            $x:expr; [ $( $y:expr ),* ]
        );*   ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {   let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]); }
```

Iterators (1/2)

```
for x in 0..10 {
    println!("{}", x);
}
```

std::iter::Iterator functions:

next size_hint count last nth chain zip map filter filter_map enumerate peekable skip_while take_while skip take scan flat_map fuse inspect by_ref collect partition fold all any find position rposition max min max_by max_by_key min_by min_by_key rev unzip cloned cycle sum product cmp partial_cmp eq ne lt le gt ge

Iterators (2/2)

```
let a = [1, 4, 2, 3, 8, 9, 6];
let sum: i32 = a.iter()
                .map(|x| *x)
                .inspect(|&x| println!("filtering {}", x))
                .filter(|&x| x % 2 == 0)
                .inspect(|&x| println!("{} seen", x))
                .fold(0, |sum, i| sum + i);
println!("{}", sum);
```

Doctests

```
/// Clears the map, removing all values.
///
/// # Examples
///
/// ```
/// use std::collections::BTreeMap;
///
/// let mut a = BTreeMap::new();
/// a.insert(1, "a");
/// a.clear();
/// assert!(a.is_empty());
/// ```
```

There is also //!

Traits (1/5)

```
trait Qqq {
    fn a(&self) -> i32;
}

struct Www {
    g: isize;
}

impl Qqq for Www {
    fn a(&self) -> i32 { self.g }
}
```

Traits (2/5)

```
pub trait PartialOrd<Rhs: ?Sized=Self> : PartialEq<Rhs> {

    fn partial_cmp(&self, other: &Rhs)->Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool {
        match self.partial_cmp(other) {
            Some(Less) => true,
            _ => false,
        }
    }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Traits (3/5)

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}

// snippet from a macros:
impl Add for $t {
    type Output = $t;

    #[inline]
    fn add(self, other: $t) -> $t { self + other }
}
```

Traits (4/5)

Future feature: associated consts.

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}
```

Traits (5/5)

Overlapping instances/specialisation (future feature?)

Orhpanage rules - Can't implement foreign trait for foreign type => can combine modules freely

```
impl     MyTrait for u32 {...} // OK
impl<T> MyTrait for T {...}   // OK
impl     IntoIterator for MyType {...} // OK
impl     IntoIterator for u32 {...}    // FAIL
impl<T> IntoIterator for T {...}      // FAIL
```

&Trait objects - existentional datatypes

Unsafe

Sometimes protections must be turned off. One of examples is implementing data structures.

There are actual, C-like pointers, including arithmetic, which are usable only in `unsafe { }` blocks.

All FFI is unsafe and needs wrappers.

There is special book about pecularities of Unsafe Rust: Rustonomicon.

Out of scope for “safe” (1/2)

- Rc Leaks (cycles)
- Missed destructors (RAII failure). There is even `mem::forget`
- Deadlocks
- Integer overflows in `--release`
- Stack overflow (SEGV, but not unsafe)
- Silently ignored errors returned by `close` syscall (“linear types” feature)

Out of scope for “safe” (2/2)

```
use std::process::Command;
fn getpid() -> i32 {
 let so = Command::new("sh").arg("-c")
 .arg("cat /proc/$$/status|grep PPid:|awk '{print $2}'")
 .output().unwrap().stdout;
 String::from_utf8(so).unwrap().trim().parse().unwrap() }
fn hacky_ptr_write<T>(ptr: &T, value: u32) {
 Command::new("gdb").arg("-batch").arg("-quiet")
  .arg("-pid").arg(format!("{}", getpid())).arg("-ex")
  .arg(format!("set {{unsigned long}}{:p}={}",ptr,value))
  .output() .unwrap(); }
fn main() {
    let mut q = &mut Box::new(55);
    hacky_ptr_write(&q, 0);
    *q = Box::new(44); }
```

Concerns

"I had a problem so I thought I'd use Rust. Now I have `&'a &'b mut Problem<T>` *which I can't move out of borrowed context."*

- Complexity reaching C++ levels
- &' lifetimes and borrowck
  Sometimes you need to find just right combination of <&'mut> stuff so it compiles. Sometimes you need to redesign.
- Young IDE support
- Not fast compilation
- Missing language features
- Not stabilized ABI => packaging far from good

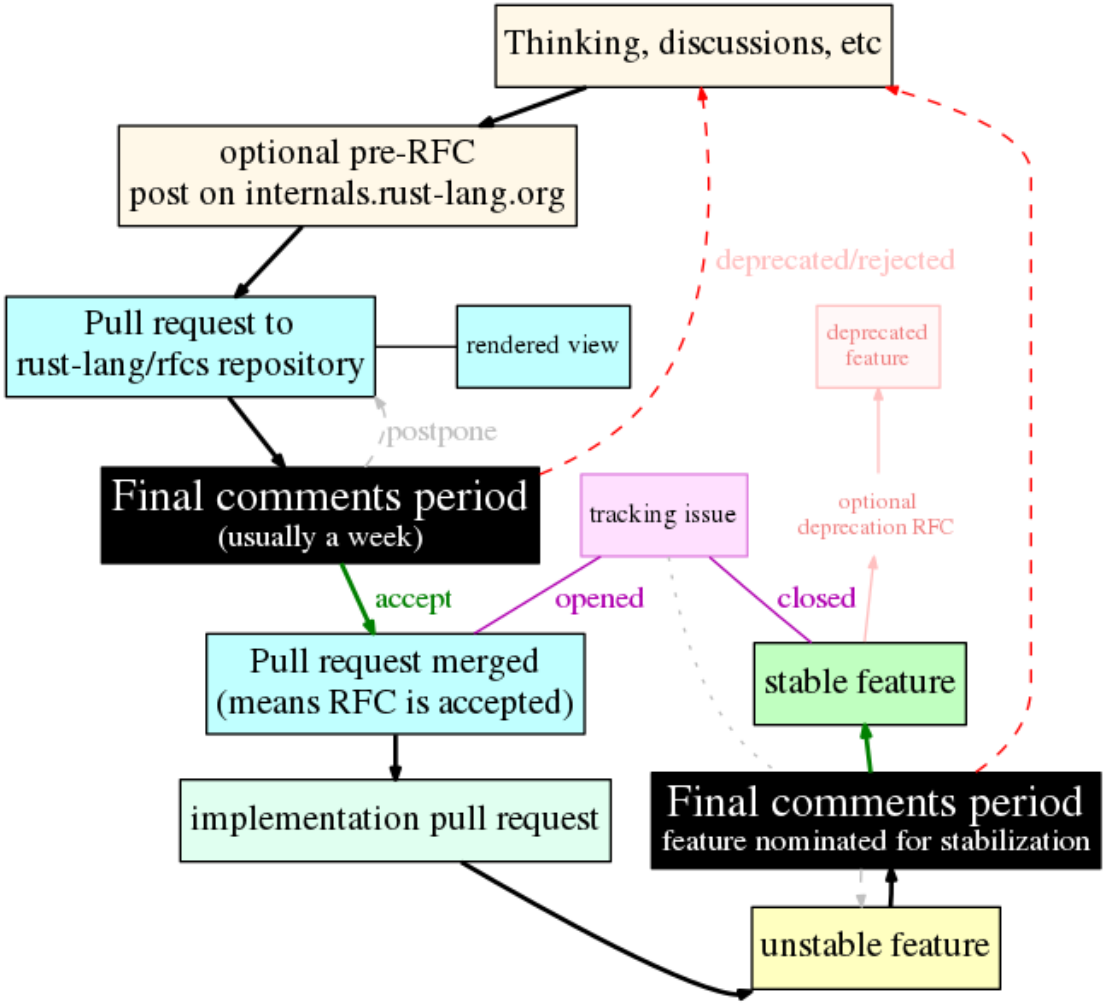
Iterators bloat the binary and ask LLVM back-end to remove tons of abstractions.

Rust improvement process It's not RIP, it's RFC

Making changes to Rust requires some buraeucracy.

1. Discussions on "mailing list" forum internals.rust-lang.org
2. Pull request to "RFC" repository
3. Debates
4. Accepted
5. Pull request with implementation
   - Build server automatically tests everything
   - Bot assigns reviewer
   - **Bot tells you when your PR stops being mergeable** (linking to conflicting PR)
6. Accepted
7. Debates about stabilizing
8. Finally the new feature is available to all users

Diagram


Thinking, discussions, etc
optional pre-RFC
post on internals.rust-lang.org
deprecated/rejected
Pull request to
rust-lang/rfcs repository
rendered view
deprecated
feature
/postpone
Final comments period
(usually a week)
tracking issue
optional
deprecation RFC
accept
opened
closed
Pull request merged
(means RFC is accepted)
stable feature
implementation pull request
Final comments period
feature nominated for stabilization
unstable feature

Modularity: modules

- `mod.rs` is a bit like `__init__.py`
- Three types of modules:
  - *Embedded* `mod mymodule { ... }`
  - *File* `mod mymodule;` and there is `mymodule.rs` in current directory
  - *Directory* `mod mymodule;` and there is `mymodule/mod.rs`
- Can be public or not
- Using conditional compilation, can be opted out entirely `#[cfg(test)] mod test;`

Modularity: crates

A compilation unit.
Can refer other crates.
Corresponds to a library or runnable program.

Cargo and crates.io

- optional features
- conditional dependencies
- `cargo publish` rejects bad `*` deps
- crater: Sometimes entire crates.io being considered when changing the compiler
- `cargo install`

Error handling

- Two worlds: Result/Option and panic/unwinding
- `Result` for functions that may fail, for most I/O
- `panic` is like “assertion failed”, for internal errors.
- There are a lot of functions to convert between Option, Result and T
- Converting Result/Option to panic is easy, the reverse (`catch_unwind`) is not and is not a recommended way
- Unwinding may be turned off and `panic` would just `abort`
- Unwind safety, poisoned Mutex
- For Result there is convenient `?` that means `try!`

Other topics

- Integer overflow, “wrapocalypse”
- nostd, libcore, lang items, intrinsics
- custom error messages, “lints”, clippy
- jemalloc
- From (libs) => Into (users)
- Former green threads, now mioco
- Reexports: `pub use`
- More libs, less bins?
- std::prelude::v1
- Opt-out library-provided OIBITs

Roadmap for 2016

- Infrastructore
- Specialisation
- Non-lexical borrows
- Stabilisation of plugins
- Incremental compilation
- Cross-compilation
- Integration with other languages