The Rust Programming Language

G.D. Ritter

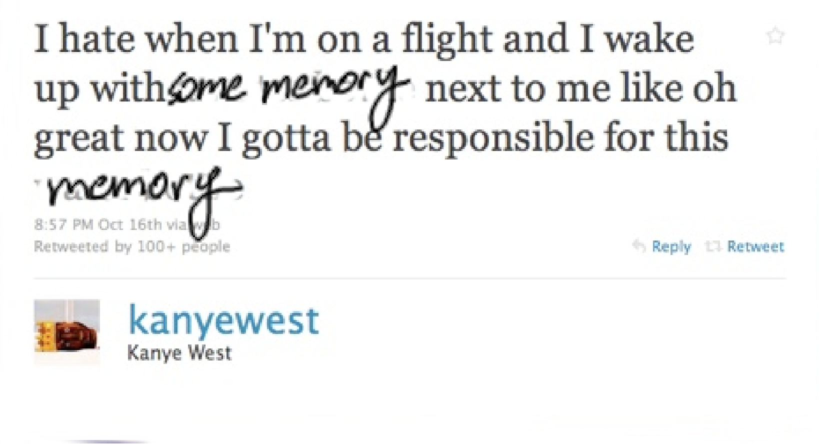
March 2014

The Rust Programming Language

A new systems programming language being developed by Mozilla Research, with an emphasis on correctness while still allowing for very low-level programing by emphasizing *zero-cost abstractions*.

Low-Level Programming

Low-Level Programming


I hate when I'm on a flight and I wake up with some memory next to me like oh great now I gotta be responsible for this memory
8:57 PM Oct 16th via web
Retweeted by 100+ people
Reply
Retweet
kanyewest
Kanye West

Systems Programming Languages

System software is computer software designed to operate and control the computer hardware and to provide a platform for running application software, and includes such things as operating systems, utility software, device drivers, compilers, and linkers.
—Wikipedia

"Systems programs" means "programs where the constant factors are important".
—Comment by `neelk` on Lambda the Ultimate

Systems Programming Languages

Example Program

```
data Point = { x, y : Int }

addPoint : Point -> Point -> Point
addPoint p1 p2 = { x = p1.x + p2.x, y = p1.y + p2.y }

main : ()
main = { let a = { x = 1, y = 2 }
       ; let b = malloc { x = 4, y = 3}
       ; print (addPoint a (deref b))
       ; free(b)
       }
```

Systems Programming Languages

C

```
typedef struct { int x, y; } point;

point add(point a, point b) {
  point result = { a.x + b.x, a.y + b.y };
  return result;
}

void main(int argc, char* argv[]) {
  point a = { 1, 2 };
  point* b = malloc(sizeof(point));
  b->x = 4; b->y = 3;
  point c = add(a, *b);
  printf("{.x = %d, .y = %d}\n", c.x, c.y);
  free(b);
}
```

Systems Programming Languages

C++

```
struct point {
  int x, y;
  point(int _x, int _y) { x = _x; y = _y; }
  point add(point other) {
    return point(x + other.x, y + other.y);
  }
};
int main(int argc, char* argv[]) {
  point a(1, 2);
  point* b = new point(4, 3);
  point c = a.add(*b);
  std::cout << "{ .x = " << c.x;
  std::cout << ", .y = " << c.y << " }" << std::endl;
  delete b;
}
```

Systems Programming Languages

Go

```go
type Point struct { X, Y int }

func (a Point) add(b Point) Point {
    return Point{ a.X + b.X, a.Y + b.Y }
}

func main() {
    a := Point{1, 2}
    b := new(Point)
    b.X, b.Y = 4, 3
    fmt.Println(a.add(*b))
    // No free, because Go is garbage-collected
}
```

Systems Programming Languages

D

```
struct Point {
  int x, y;
  Point add(Point other) {
    return Point(this.x + other.x, this.y + other.y);
  }
}

void main() {
  Point a = Point(1, 2);
  Point* b = cast(Point*)GC.malloc(Point.sizeof);
  b.x = 4; b.y = 3;
  writeln(a.add(*b));
  GC.free(b);
}
```

Systems Programming Languages

Nimrod

```
type Point = tuple[x: int, y: int]

proc add(a: Point, b: Point): Point =
  (x: a.x + b.x, y: a.y + b.y)

var a : Point
var b : ptr Point

a = (x: 1, y: 2)
b = cast[ptr Point](alloc(sizeof(Point)))
b.x = 4
b.y = 3
echo(add(a, b[]))
dealloc(b)
```

Systems Programming Languages

Rust

```
struct Point { x: int, y: int }

impl Point {
    fn add(self, other: Point) -> Point {
        Point { x: self.x + other.x,
                y: self.y + other.y }
    }
}

fn main() {
    let a = Point { x: 1, y: 2 };
    let b = ~Point { x: 4, y: 3 };
    println!("{:?}", a.add(*b));
}
```

Basics of Rust

Basics of Rust

It's like C++ grew up, went to grad school, started dating Haskell, and is sharing an office with Erlang...
—Michael Sullivan

Basics of Rust

Recursive Factorial

```
fn fact1(n: int) -> int {
  if n <= 0 {
    1
  } else {
    n * fact1(n-1)
  }
}
```

Another Recursive Factorial

```
fn fact2(n: int) -> int {
  match n {
    0 => { 1 }
    _ => { n * fact2(n-1) }
  }
}
```

Basics of Rust

An Imperative Factorial

```
fn fact3(mut n: int) -> int {
  let mut res = 1;
  while (n > 0) {
    res *= n;
    n   -= 1;
  }
  res
}
```

One More Imperative Factorial

```
fn fact4(mut n: int) -> int {
  for i in range(1, n) { n *= i; }
  return n;
}
```

Basics of Rust

Tuples

```
{
  let t: (int, int, int) = (1,2,3);
  let (a,b,c)            = t;
  let r = match t { (a,b,c) => a + b + c };
}
```

Tuple Structs (i.e. named tuples)

```
struct T(bool, int);
fn f(t: T) -> int {
  let T(myBool, myInt) = t;
  return if myBool { myInt } else { -myInt };
}
```

Basics of Rust

Structs

```
struct Point { x: f64, y: f64 }

fn isOrigin1 (p: Point) -> bool {
  p.x == 0.0 && p.y == 0.0
}

fn isOrigin2 (p: Point) -> bool {
  match p {
    Point { x: 0.0, y: 0.0 } => true,
    _                        => false
  }
}
```

Enums

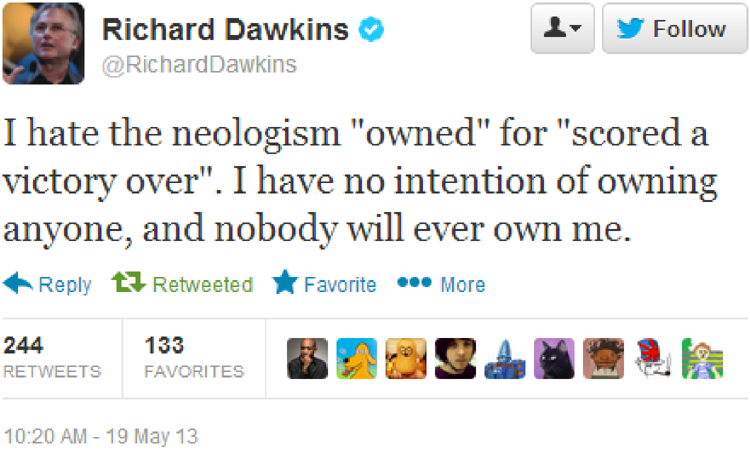
```
enum Color { Red, Green, Blue }

enum Shape {
  Circle(Point, f64),
  Rectangle(Point, Point),
}

fn area(s: Shape) -> f64 {
  match s {
    Circle(_, sz)      => f64::consts::pi * sz * sz,
    Rectangle(p1, p2) => (p2.x - p1.x) * (p2.y - p1.y)
  }
}
```

Pointers and Memory

Pointers and Memory



Pointers and Memory

"Owned" Pointers

```
fn main() {
  let x: ~[int] = ~[1,2,3];
  /* x in scope */
  {
    let y: ~[int] = ~[4,5,6];
    /* x, y in scope */
  }
  /* x in scope */
}
```

Pointers and Memory

"Owned" Pointers

```
fn main() {
  let x: ~[int] = ~[1,2,3];   // malloc |----+
  /* ... */                   //             |
  {                           //             |
    let y: ~[int] = ~[4,5,6]; // malloc |-+  |
    /* ... */                 //          |  |
  }                           // free <---+  |
  /* ... */                   //             |
}                             // free <------+
```

Pointers and Memory

“Owned” Pointers

```
fn f0() -> ~[int] {
  return ~[1,2,3]; // returning ownership
}
fn f1() -> ~[int] {
  let a = ~[1,2,3];
  let b = a;
  return a; // error: use of moved value: `a`
}
fn f2() -> ~[int] {
  let a = ~[1,2,3];
  let b = a.clone();
  return a; // fine now; `a` and `b` both valid
}
```

Pointers and Memory

“Owned” Pointers

```
#[deriving(Clone)]
enum List<T> { Cons(T, ~List<T>), Nil }

fn f3() -> ~List<int> {
  let mut a = ~Cons(1, ~Cons(2, ~Nil))
  /* a is mutable */
  let b = a;
  /* can no longer use a, b is immutable */
  let mut c = b.clone();
  /* can use both b and c */
  return b;
}
```

Pointers and Memory

Dispreferred Style

```
type t8 = (u32,u32,u32,u32,u32,u32,u32,u32);

fn eight_nums() -> ~t8 {
  ~(1,2,3,4,5,6,7,8)
}

fn main() {
  let t: ~t8 = eight_nums();
  /* ... */
}
```

Pointers and Memory

Preferred Style

```
type t8 = (u32,u32,u32,u32,u32,u32,u32,u32);

fn eight_nums() -> t8 {
  (1,2,3,4,5,6,7,8)
}

fn main() {
  let t: ~t8 = ~eight_nums();
  /* ... */
}
```

Pointers and Memory

References

```
{
  let p = Point { x: 1.2, y: 3.4 };
  let q = & p;
  // both p and q usable
}
{
  let q = & Point { x: 1.2, y: 3.4 };
}
{
  let p = Point { x: 1.2, y: 3.4 };
  let r = & p.x;
}
```

Pointers and Memory

References

```
fn eq(xl: ~List<int>, yl: ~List<int>) -> bool {
  /* elided */
}

fn main() {
  let l1 = ~Cons(1, ~Cons (2, ~Nil));
  let l2 = ~Cons(3, ~Cons (4, ~Nil));
  println!("{}", eq(l1, l2));
  println!("{:?}", l1);
}
```

Pointers and Memory

References

```
fn eq(xl: ~List<int>, yl: ~List<int>) -> bool {
  /* elided */
}

fn main() {
  let l1 = ~Cons(1, ~Cons (2, ~Nil));
  let l2 = ~Cons(3, ~Cons (4, ~Nil));
  println!("{}", eq(l1, l2)); // ownership of l1 and l2
                              // moves to eq function
  println!("{:?}", l1); // error: use of moved value!
}
```

Pointers and Memory

References

```
fn eq(xl: ~List<int>, yl: ~List<int>) -> bool {
  /* elided */
}

fn main() {
  let l1 = ~Cons(1, ~Cons (2, ~Nil));
  let l2 = ~Cons(3, ~Cons (4, ~Nil));
  println!("{}", eq(l1.clone(), l2.clone()));
  println!("{:?}", l1);
}
```

Pointers and Memory

References

```
fn eq(xl: &List<int>, yl: &List<int>) -> bool {
  /* elided */
}

fn main() {
  let l1 = ~Cons(1, ~Cons (2, ~Nil));
  let l2 = ~Cons(3, ~Cons (4, ~Nil));
  println!("{}", eq(l1, l2));
  println!("{:?}", l1);
}
```

Pointers and Memory

References

```
fn eq(xl: &List<int>, yl: &List<int>) -> bool {
  match (xl, yl) {
    (&Nil, &Nil) => true,
    (&Cons(x, ~ref xs), &Cons(y, ~ref ys))
      if x == y => eq(xs, ys),
    (_, _) => false
  }
}
```

Pointers and Memory

References

```
fn eq<T: Eq>(xl: &List<T>, yl: &List<T>) -> bool {
  match (xl, yl) {
    (&Nil, &Nil) => true,
    (&Cons(x, ~ref xs), &Cons(y, ~ref ys))
      if x == y => eq(xs, ys),
    (_, _) => false
  }
}
```

Pointers and Memory

References and Lifetimes

```
{
  let     a = ~5;
  let mut p = &a;
  {
    let b = ~8;
    p     = &b;
  }
  println!("{}", **p)
}
```

Pointers and Memory

References and Lifetimes

```
{
  let     a = ~5;      // malloc |---+
  let mut p = &a;      //            |
  {                    //            |
    let b = ~8;        // malloc |-+ |
    p     = &b;        //          | |
  }                    // free <---+ |
  println!("{}", **p) //            |
}                      // free <-----+
```

Pointers and Memory

References and Lifetimes

```
{
  let     a = ~5;
  let mut p = &a;
  {
    let b = ~8;
    p     = &b; // error: borrowed value does
                // not live long enough
  }
  println!("{}", **p)
}
```

Pointers and Memory

References, Pointers, Mutability

```
{
  let mut x = ~5;
  *x = *x + 1;
  {
    let y = &x;
    /* x is not mutable for the rest of this block */
  }
  /* x regains mutability */
}
```

Pointers and Memory

References, Pointers, Mutability

```
enum IntList {
  Cons { head: int, tail: ~IntList },
  Nil,
}
{
  let mut lst = ~Cons { head: 5, tail: ~Nil };
  {
    let y = &(lst.head); // or &((*lst).head)
    lst = ~Nil;
    println!("{}", y);
  }
}
```

Pointers and Memory

References, Pointers, Mutability

```
enum IntList {
  Cons { head: int, tail: ~IntList },
  Nil,
}
{
  let mut lst = ~Cons { head: 5, tail: ~Nil };
  {
    let y = &(lst.head);
    lst = ~Nil;
    println!("{}", y); // BAD
  }
}
```

Pointers and Memory

Named Lifetimes

```
fn tail<T>(lst: &List<T>) -> &List<T> {
  match *lst {
    Nil               => &Nil,
    Cons(_, ~ref xs) => xs
  }
}
```

Pointers and Memory

Named Lifetimes

```
fn tail<'s, T>(lst: &'s List<T>) -> &'s List<T> {
  match *lst {
    Nil             => &Nil,
    Cons(_, ~ref xs) => xs
  }
}
```

Pointers and Memory

Reference Counting

```
use std::rc::Rc;
{
  let x = Rc::new([1,2,3]);
  let y = x.clone(); // two references, one vector
  assert!(x.ptr_eq(y));
  assert!(*y.borrow() == [1,2,3]);
}
```

Garbage Collection

```
use std::gc::Gc;
{
  let x = Gc::new([1,2,3]);
  // etc.
}
```

Pointers and Memory

C Pointers

```
use std::ptr::RawPtr;

#[link(name="foo")]
extern {
  fn unsafe_get() -> *int;
}

fn safe_get() -> Option<int> {
  unsafe {
    let i = unsafe_get();
    i.to_option()
  }
}
```

Closures

Closures

[...] Lambdas are relegated to relative obscurity until Java makes them popular by not having them.
—James Iry, "A Brief, Incomplete, and Mostly Wrong History of Programming Languages"

Closures

Functions

```
fn main() {
  let x = 5;
  fn inner(y: int) -> int {
    return x + y;
  }
  println!("{}", inner(1));
}
```

Closures

Functions Do NOT Close Over Env

```
fn main() {
  let x = 5;
  fn inner(y: int) -> int {
    return x + y; // error: can't capture dynamic env
  }
  println!("{}", inner(1));
}
```

Closures

Stack Closure

```
fn main() {
  let x = 5;
  let inner = |y| x + y;
  println!("{}", inner(1));
}
```

Stack Closure with Type Annotations

```
fn main() {
  let x = 5;
  let inner = |y: int| -> int { x + y };
  println!("{}", inner(1));
}
```

Closures

Stack Closures

```
fn my_map<A,B>(f: |&A|->B, l: &List<A>) -> List<B> {
    match *l {
        Nil => Nil,
        Cons(ref x, ~ref xs) =>
          Cons(f(x)), ~my_map(f, xs))
    }
}

fn main() {
    fn incr(x: &int) -> int { x + 1 }
    let l = ~Cons(1, ~Cons(2, ~Cons(3, ~Nil)));
    println!("{:?}", my_map(|x| x + 1, l));
    println!("{:?}", my_map(incr, l));
}
```

Closures

Owned Closures

```
use std::task::spawn;

fn main() {
    let x = ~5;
    spawn(proc() {
        println!("{}", x);
    });
    // x is now owned by the proc above
}
```

Methods

Methods on a Struct

```
use std::f64::{sqrt,pow};
struct Point { x: f64, y: f64 }
impl Point {
    fn magnitude(&self) -> f64 {
        sqrt(pow(self.x,2.0)+pow(self.y,2.0))
    }
    fn new((my_x, my_y): (f64, f64)) -> Point {
        Point { x: my_x, y: my_y }
    }
}
fn main() {
    let p = Point::new((2.0,4.0));
    println!("{}", p.magnitude());
}
```

Methods

Methods on an Enum

```
impl<T> List<T> {
  fn is_empty(&self) -> bool {
    match self {
      &Nil          => true,
      &Cons(_, _) => false,
    }
  }
}
```

Traits

Head of a List By Reference

```
fn head<'a, T>(lst: &'a List<T>) -> Option<&'a T> {
  match lst {
    &Nil               => None,
    &Cons(ref hd, _) => Some(hd)
  }
}
```

Traits

Head of a List By Value

```
fn head<T>(lst: &List<T>) -> Option<T> {
  match lst {
    &Nil                => None,
    &Cons(ref hd, _) => Some(*hd)
  }
}
```

Traits

Head of a List By Value

```
fn head<T>(lst: &List<T>) -> Option<T> {
  match lst {
    &Nil              => None,
    &Cons(ref hd, _) => Some(*hd)
    // cannot move out of dereference of & pointer
  }
}
```

Traits

Head of a List By Value

```
fn head<T: Clone>(lst: &List<T>) -> Option<T> {
  match lst {
    &Nil              => None,
    &Cons(ref hd, _) => Some(hd.clone())
  }
}
```

Traits

Declaring Traits

```
trait Printable {
    fn print(&self);
}

impl Printable for int {
    fn print(&self) { println!("{}", *self) }
}

impl Printable for bool {
    fn print(&self) { println!("{}", *self) }
}

fn main() {
    5.print(); true.print();
}
```

Traits

Using Multiple Traits

```
fn print_head<T: Clone+Printable>(lst: &List<T>) {
    match lst {
        &Nil               => { println!("Nothing!") }
        &Cons(ref hd, _) => { hd.clone().print() }
    }
}
```

Traits

Static Dispatch

```
fn printAll<T: Printable>(vec: &[T]) {
    for p in vec.iter() { p.print() }
}
fn main() {
    printAll([1, 2, 3]);
}
```

Dynamic Dispatch

```
fn print_all(vec: &[~Printable]) {
    for p in vec.iter() { p.print() }
}
fn main() {
    print_all([~1 as ~Printable, ~true as ~Printable]);
}
```

Tasks and Communication

Tasks

```
fn main() {
    spawn(proc() {
      println!("Hello from another task!");
    });
    println!("Hello from the parent task!");
}
```

Tasks and Communication

Communication

```
fn main() {
    let (port, chan): (Port<int>, Chan<int>) = Chan::new();
    spawn(proc() {
        chan.send(some_computation());
    });
    some_other_computation();
    let result = port.recv();
}
```

Tasks and Communication

Atomic Reference Counting

```
fn main() {
    let parent_copy = Arc::new(something_very_large());
    let (port, chan) = Chan::new();
    chan.send(parent_copy.clone());
    spawn(proc() {
        let task_copy = port.recv();
        task_copy.get().do_something();
    });
    parent_copy.get().do_something_else();
}
```

Tasks and Communication

Failure

```
fn main() {
    let r : Result<int, ()> = try(proc() {
        if some_operation_succeeds() {
            return 5;
        } else {
            fail!("Hark! An error!");
        }
    });
    match r {
        Ok(i)  => println!("Got {}", i),
        Err(_) => println!("Hark!"),
    };
}
```

Crates and Modules

- A "crate" is a compilation unit; `rustc` produces a single crate if it is run (either a library or an executable.)
- A module is a grouping of definitions. Modules can be hierarchical and can be defined in a single file in `mod { ... }` blocks, or in separate files.

Crates and Modules

main.rs

```
mod mylist {
    pub enum List<T> { Cons(T, ~List<T>), Nil }
    pub fn from_vec<T>(mut vec : ~[T]) -> ~List<T> { ... }
    impl<T> List<T> {
        pub fn length(&self) -> int { ... }
    }
}

fn main() {
    let v = ~[1,2,3];
    let l = ::mylist::from_vec(v);
    /* ... */
}
```

Crates and Modules

mylist.rs or mylist/mod.rs

```
mod mylist {
    pub enum List<T> { Cons(T, ~List<T>), Nil }
    pub fn from_vec<T>(mut vec : ~[T]) -> ~List<T> { ... }
    impl<T> List<T> {
        pub fn length(&self) -> int { ... }
    }
}
```

main.rs

```
mod mylist;
main() {
    let v = ~[1,2,3];
    let l = ::mylist::from_vec(v);
    /* ... */
}
```

Crates and Modules

main.rs

```
use mylist::from_vec;
mod mylist;

main() {
    let v = ~[1,2,3];
    let l = from_vec(v);
    /* ... */
}
```

Crates and Modules

Crate Metadata

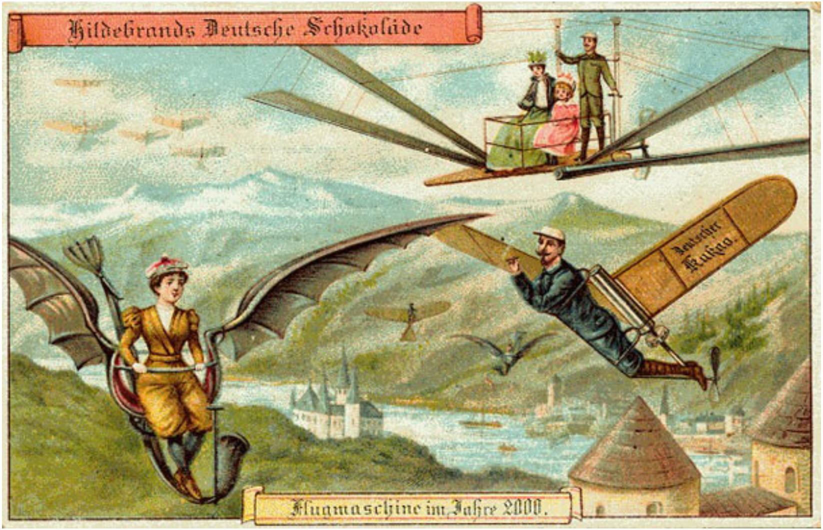
```
#[crate_id = "mycrate#1.2"];
#[crate_type = "lib"];
```

Requesting Crate Metadata

```
extern crate mycrate "mycrate#1.2";
extern crate oldmycrate "mycrate#0.6";
```

The Future

The Future


Hildebrands Deutsche Schokolade
Deutscher Kukuk.
Flugmaschine im Jahre 2000.

The Future

Possible Syntax Changes

- `~foo` might become `box foo`
- `~[T]` might become `Vec<T>`
- Operator overloading

Possible Language Changes

- Speculations about inheritance, subtyping
- Stronger restrictions on `unsafe` code

Standard Library Improvements

Package Manager