a systems language
pursuing the trifecta
safe, concurrent, fast

mozilla

Motivation

- Why invest in a new programming language
- Web browsers are complex programs
- Expensive to innovate and compete while implementing atop standard systems languages
- So to implement next-gen browser, Servo ...

  ⇒ **`http://github.com/mozilla/servo`**
- ... Mozilla is using (& implementing) Rust

  ⇒ **`http://rust-lang.org`**

- **Part I: Motivation**

  Why Mozilla is investing in Rust

- **Part II: Rust syntax and semantics**
- **Part III: Ownership and borrowing**

Language Design

- Goal: bridge performance gap between safe and unsafe languages
- Design choices largely fell out of that requirement
- Rust compiler, stdlib, and tools are all MIT/Apache dual license.

Systems Programming

- Resource-constrained enviroments, direct control over hardware
- C and C++ dominate this space
- Systems programmers care about the last 10-15% of potential performance

Unsafe aspects of C

- Dangling pointers
- Null pointer dereferences
- Buffer overflows, array bounds errors
- Format string and argument mismatch
- Double frees

Tool: Sound Type Checking

Milner, 1978

- "Well-typed programs can't go wrong."
- More generally: identify classes of errors ...
  - ... then use type system to remove them
  - (or at least isolate them)
- Eases reasoning; adds confidence

Tobin-Hochstadt 2006, Wadler 2009

- Well-typed programs help assign blame.
  - (**`unsafe`** code can still “go wrong”)
  - and even safe code can **`fail`**

Simple source ⇔ compiled code relationship

- A reason C persists to this day
- Programmer can mentally model machine state
  - can also control low-level details (e.g. memory layout)
- Goal for Rust: preserve this relationship ...
  - ... while **retaining** memory safety ...
  - ... without runtime cost.
  - Do not box everything; do not GC-manage everything.

- **Part I: Motivation**
- **Part II: Rust syntax and semantics**

  Systems programming under the influence of FP

- **Part III: Ownership and borrowing**

OCaml / Rust: basic syntax

OCaml:

```
let y = let x = 2 + 3 in x > 5 in
if y then x + 6 else x + 7
```

Rust:

```
let y = { let x = 2 + 3; x > 5 };
if y { x + 6 } else { x + 7 }
```

OCaml / Rust: functions

OCaml:

```
let add3 x = x + 3 in
let y = add3 7 > 5 in
...
```

Rust:

```
fn add3(x:int) -> int { x + 3 }
let y = add3(3) > 5;
...
```

OCaml / Rust: pattern binding

OCaml:

```
let add3_left (x, y) = (x + 3, y) in
let y = add3_left (7,"hi") > (10,"lo") in
...
```

Rust:

```
fn add3_left<A>((x,y):(int, A)) -> (int, A) {
   (x + 3, y)
}
let y = add3_left((7,"hi")) > (10,"lo")
...
```

- (A generic type parameter snuck in above)

OCaml / Rust: pattern matching

OCaml:

```
type 'a lonely = One of 'a | Two of 'a * 'a;;
let combined l =
    match l with
      One a        -> a
    | Two (a, b) -> a + b
in ...
```

Rust:

```
enum Lonely<A> { One(A), Two(A, A) }
fn combined(l: Lonely<int>) {
    match l {
        One(a)      => a,
        Two(a, b) => a + b,
    }
}
...
```

Rust: Bounded Polymorphism (No functors)

```
// "struct" is Rust's record syntax.
struct Dollars { amt: int }
struct Euros { amt: int }
trait Currency {
    fn render(&self) -> String;
    fn to_euros(&self) -> Euros;
}
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}
```

OCaml / Rust: value model (move semantics)

- In OCaml, under the hood, large values are (tagged) references.
- Passing one parameter == copy one word
  - (a word-sized literal, or a tagged pointer to block on heap)
- Things are different in Rust.

A mini-puzzle

- What does this print?

OCaml:

```
# type 'a lonely = One of 'a | Two of 'a * 'a;;
# Obj.size(Obj.repr(1,2,3,4,5));;
- : int = 5
# Obj.size(Obj.repr(Two((1,2,3,4,5),
                        (1,2,3,4,5))));;
```

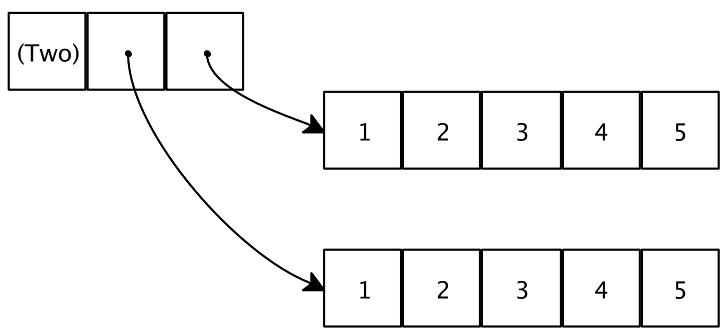
- Answer:

```
- : int = 2
```

Why only 2 words?

```
# type 'a lonely = One of 'a | Two of 'a * 'a
# Obj.size(Obj.repr(1,2,3,4,5));;
- : int = 5
# Obj.size(Obj.repr(Two((1,2,3,4,5),
                        (1,2,3,4,5))));;
- : int = 2
```

- Here is how OCaml represents a **`Two`**



The same puzzle in Rust

Rust:

```
use std::mem::size_of;
enum Lonely<A> { One(A), Two(A, A) }
let size =
    size_of::<Lonely<(int,int,int,int,int)>>();
let word_size = size_of::<int>();
println!("words: {}", size / word_size);
```

- Prints `words: 11`
- Here is how Rust represents a `Two`

(Two)	1	2	3	4	5	1	2	3	4	5

- Here is how Rust represents a `One`

(One)	1	2	3	4	5					

Implications

To Move or To Copy?

- This does not compile

```
fn twice<T:Show>(x: T, f: fn (T) -> T) -> T {
    let w = f(x);
    println!("temp w: {}", w);
    let y = f(x);
    println!("temp y: {}", y);
    let z = f(y); return z;
}
```

```
error: use of moved value: `x`
let y = f(x);
          ^
note: `x` moved here because it has non-copyable
      type `T` (perhaps use clone()?)
let w = f(x);
          ^
```

Why all the fuss about move semantics?

- **Part I: Motivation**
- **Part II: Rust syntax and semantics**
- **Part III: Ownership and borrowing**

  How Rust handles pointers

Rust: Values and References

- Life outside of ref-cells
- There are three core types **`T`** to think about.
- **`T`** non-reference
- **`&T`** shared reference
- **`&mut T`** mutable unaliased reference
- **`*T`** too (unsafe pointers); not this talk

&T : shared reference

```
let x: int = 3;
let y: &int = &x;
assert!(*y == 3);
// assert!(y == 3); /* Does not type-check */

struct Pair<A,B> { a: A, b: B }
let p = Pair { a: 4, b: "hi" };
let y: &int = &p.a;
assert!(*y == 4);
```

&mut T : mutable unaliased reference

```
let mut x: int = 5;
increment(&mut x);
assert!(x == 6);

fn increment(r: &mut int) {
    *r = *r + 1;
}
```

pattern matching and refs: Why

```
struct Pair<A,B> { a: A, b: B }
fn add_b_twice<T>(p: Pair<int,T>,
                  f: fn (&T) -> int) -> int {
  match p {
    Pair{ a, b } => {
        //   ^ `p.b` is moved into `b` here, so
        // cannot compile: use of moved value: `p.b`
        a + f(&b) + f(&p.b)
    }
  }
}
```

pattern matching and refs: How

```
struct Pair<A,B> { a: A, b: B }
fn add_b_twice<T>(p: Pair<int,T>,
                  f: fn (&T) -> int) -> int {
  match p {
    Pair{ a, ref b } => {
        //   ^ now `p.b` is left in place, and
        // `b` is bound to a `&T` instead of a `T`.
        a + f(b) + f(&p.b)
    }
  }
}
```

Why all the fuss about aliasing?

It is for type soundness

mutable aliasing ⇒ soundness holes

```
fn add3(x:int) -> int { x + 3 }
enum E { A(fn (int) -> int), B(int) }
let mut a = A(add3); let mut b = B(17);
let p1 = &mut a;     let p2 = &mut b;
foo(p1, p2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
        &B(..) => fail!("cannot happen"),
        &A(ref adder) => {
            *p2 = B(0xdeadc0de);
            println!("{}", (*adder)(14));
        }
  }
}
```

- (punchline: above is fine; **`rustc`** accepts it)

mutable aliasing ⇒ soundness holes

```
fn add3(x:int) -> int { x + 3 }
enum E { A(fn (int) -> int), B(int) }
let mut a = A(add3); let mut b = B(17);
let p1 = &mut a;     let p2 = &mut b;
foo(p1, p2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
      &B(..) => fail!("cannot happen"),
      &A(ref adder) => {
          *p1 = B(0xdeadc0de);
          println!("{}", (*adder)(14));
      }
  }
}
```

- (punchline: above is badness; **`rustc`** rejects it)

mutable aliasing ⇒ soundness holes

```
fn add3(x:int) -> int { x + 3 }
enum E { A(fn (int) -> int), B(int) }
let mut a = A(add3); let mut b = B(17);
let p1 = &mut a;     let p2 = &mut b;
foo(p1, p1);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
      &B(..) => fail!("cannot happen"),
      &A(ref adder) => {
          *p2 = B(0xdeadc0de);
          println!("{}", (*adder)(14));
      }
  }
}
```

- (punchline: above is badness; **`rustc`** rejects it)

Why all the fuss about move semantics?

Allows us to reason about aliasing

Topics not covered

- regions/lifetimes and their subtyping relationship
- traits as existentials (object-oriented dispatch)
- borrow-checking static analysis rules
- task-local storage
- Rust and closures
- syntax extensions

Join the Fun!

`rust-lang.org`

mailing-list: **`rust-dev@mozilla.org`**

community chat: **`irc.mozilla.org :: #rust`**

mozilla