

a systems language
pursuing the trifecta
safe, concurrent, fast

mozilla

Motivation

- 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**

Systems Programming

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

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

Tobin-Hochstadt 2006,
Wadler 2009

Simple source \Leftrightarrow 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: (irrefutable) 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");
playpen (http://is.gd/FgtM0y)
```

- (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>) -> int {
  match l {
    One(a)      => a,
    Two(a, b)   => a + b,
  }
}
```

playpen (<http://is.gd/gd2tDG>)

Rust: Bounded Polymorphism (No functors)

playpen (<http://is.gd/J7hH2a>)

```
enum Lonely<A> { One(A), Two(A, A) }
trait Plus { fn plus(self, rhs: Self) -> Self; }
fn combined<N:Plus>(l: Lonely<N>) -> N {
    match l {
        One(a)      => a,
        Two(a, b) => a.plus(b),
    }
}
```

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) ))) ;;
```

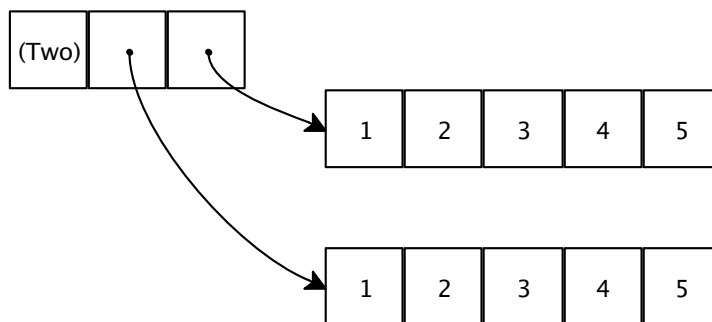
- Answer:

```
- : 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

playpen (<http://is.gd/pzY6p9>)

```
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**



- Here is how Rust represents a **One**



Implications

To Move or To Copy?

can't build: `playpen` (<http://is.gd/kYJQzk>)

```
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);  
      ^
```

What sleight of hand is this?

- "You said 'move semantics'; looks like linear or affine types."
- Did we not see a bunch of examples earlier like:

```
let y = { let x = 2 + 3; x > 5 };  
if y { x + 6 } else { x + 7 }
```
- Obviously **x** is not being used linearly there
- Magic?

Clarke's third law

- It's not magic; it's the type system

Clarke's third law

- It's not magic; ~~it's the type system~~ it's the type + trait system
- The **Copy** bound expresses that a type is freely copyable
 - and it is checked by the compiler
- Many built-in types implement **Copy**...
- ... but a type parameter with no given bounds does not.

To Move or To Copy? (II)

"Works": **playpen** (<http://is.gd/fOEE4C>)

```
fn twice<T: Show+Copy>(x: T, f: fn (T) -> T) -> T {  
    // ^~~~~~ new code here  
    let w = f(x);  
    println!("temp w: {}", w);  
    let y = f(x);  
    println!("temp y: {}", y);  
    let z = f(y); return z;  
}
```

but that's not the point.

- (Cannot generally just add **Copy** bounds)

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

okay there is ***T** too, aka **unsafe** pointers

- (and library smart pointers like **Box<T>** or **Rc<T>**, but those are not core)

&T : shared reference

playpen (<http://is.gd/aJ1TZx>)

```
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;
let (z1, z2) = (y, y); // &T impl's Copy for any T.
assert! (*y == 4);
```

&mut T : mutable unaliased reference

playpen (<http://is.gd/sqVUIa>)

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

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

let y = &mut x;
// let (z1, z2) = (y, y); /* Does not type-check */
```

pattern matching and refs: Why

playpen (<http://is.gd/0050Xc>)

```
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

playpen (<http://is.gd/Npbf17>)

```
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`.
      // (even happens when `p` is Copy!)
      a + f(b) + f(&p.b)
    }
  }
}
```

Why all the fuss about aliasing?

It is for type soundness

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/jPR99P>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      /* WATCH: */ *p2 = B(0xBadC0de);
      println!("{}", (*adder)(14));
    }
  }
}
```

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

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/8WqUji>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      /* was p2 */ *p1 = B(0xBadC0de);
      println!("{}", (*adder)(14));
    }
  }
}
```

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/CA00LG>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
    match p1 {
        &B(..) => fail!("cannot happen"),
        &A(ref adder) => {
            /* was p2 */ *p1 = B(0xBadC0de);
            println!("{}", (*adder)(14));
        }
    }
}
```

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

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/ql9yce>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      unsafe { *(p1 as *mut E)=B(7); }
      println!("{}", (*adder)(14));
    }
  }
}
```

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/TxPLzy>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;               let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      unsafe { *(p1 as *mut E)=B(7); }
      println!("{}", (*adder)(14));
    }
  }
}
```

- Emphasis: **unsafe** means "can crash."

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/Kbyb9z>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m2);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      /* watch? */ *p2 = B(0xBadC0de);
      println!("{}", (*adder)(14));
    }
  }
}
```

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

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/RoHtew>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m1);

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      /* watch? */ *p2 = B(0xBadC0de);
      println!("{}", (*adder)(14));
    }
  }
}
```

- what changed, nothing in `foo...`

mutable aliasing \Rightarrow soundness holes

playpen (<http://is.gd/4MQeii>)

```
enum E { A(fn (int) -> int), B(int) }
fn add3(x:int) -> int { x + 3 }
let mut a = A(add3);           let mut b = B(17);
let m1 = &mut a;              let m2 = &mut b;
foo(m1, m1); // <~~ AHHHHH

fn foo(p1: &mut E, p2: &mut E) {
  match p1 {
    &B(..) => fail!("cannot happen"),
    &A(ref adder) => {
      /* watch? */ *p2 = B(0xBadC0de);
      println!("{}", (*adder)(14));
    }
  }
}
```

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

Why all the fuss about move semantics?

Allows us to reason about aliasing

Lifetimes

- Earlier claim: **&T** is shared reference
- Does this mean you can copy it anywhere you like?
- Of course not; must disallow dangling pointers

Reality of types

- Core types are not really $\&\mathbf{T}$ and $\&\mathbf{mut\ T}$
- They are really $\&'a\ \mathbf{T}$ and $\&'b\ \mathbf{mut\ T}$
 - $'\mathbf{IDENT}$ is distinguished syntax for lifetime parameters
 - (similar to "regions" used by Tofte/Talpin 1994)
- $x: \&'a\ \mathbf{T}$ means x is a reference that will survive at *least* as long as $'a$ and perhaps longer
 - implicitly, \mathbf{T} itself must also live that long

Reality of functions

A function

```
fn foo(x: &int, y: &int)
```

is really sugar for the more explicit form

```
fn foo<'a, 'b>(x: &'a int, y: &'b int)
```

You can also put in bounds:

```
fn baz<'a, 'b: 'a>(x: &'a int, y: &'b int)
```

meaning 'b lives at least as long as 'a (and perhaps longer)

Reality of structs / enums

- Did you notice that none of the examples put references inside structs?
- Rust requires explicit lifetimes on reference-types in fields.
 - which effectively means such structs need to be lifetime-parametric

```
let y : int = 3;  
struct S<'a> { x: &'a int }  
let z : S = S { x: &y };
```

playpen (<http://is.gd/xCt1tX>)

Is every kind of mutability forced into a linearly passed type?

There two kinds of mutability in Rust:

- Inherited
- Interior

(remember: goal is to prevent data-races; not to hamstring developers)

Inherited mutability

playpen (<http://is.gd/ZUj4Jg>)

```
struct S { x: int, y: int }
let a = S { x: 3, y: 4 }; // a.x and a.y *immutable*
let mut b = a;           // b.x and b.y are mutable
let mut c = b;           // c.x and c.y are mutable
b.x = 5; b.y = 6;

struct T<'l> { p: &'l S,
              q: &'l mut S }

let u;
u = T { p: &b, q: &mut c };
u.p.x = 7;
```


Inherited mutability

playpen (<http://is.gd/Yb0Zg2>)

```
struct S { x: int, y: int }
let a = S { x: 3, y: 4 }; // a.x and a.y *immutable*
let mut b = a;           // b.x and b.y are mutable
let mut c = b;           // c.x and c.y are mutable
b.x = 5; b.y = 6;

struct T<'l> { p: &'l S,
              q: &'l mut S }

let u;
u = T { p: &b, q: &mut c };
u.p.x = 7;
```

you can't mutate `u.p.x` this way; `u` is not marked `mut`

Inherited mutability

playpen (<http://is.gd/qq1Cfu>)

```
struct S { x: int, y: int }
let a = S { x: 3, y: 4 }; // a.x and a.y *immutable*
let mut b = a;           // b.x and b.y are mutable
let mut c = b;           // c.x and c.y are mutable
b.x = 5; b.y = 6;

struct T<'l> { p: &'l S,
              q: &'l mut S }

let mut u;
u = T { p: &b, q: &mut c };
u.p.x = 7;
```

Inherited mutability

playpen (<http://is.gd/1CtD1D>)

```
struct S { x: int, y: int }
let a = S { x: 3, y: 4 }; // a.x and a.y *immutable*
let mut b = a;           // b.x and b.y are mutable
let mut c = b;           // c.x and c.y are mutable
b.x = 5; b.y = 6;

struct T<'l> { p: &'l S,
              q: &'l mut S }

let mut u;
u = T { p: &b, q: &mut c };
u.p.x = 7;
```

you can't mutate `u.p.x` this way; `u.p` is not `&mut`

Inherited mutability

playpen (<http://is.gd/BlhZ40>)

```
struct S { x: int, y: int }
let a = S { x: 3, y: 4 }; // a.x and a.y *immutable*
let mut b = a;           // b.x and b.y are mutable
let mut c = b;           // c.x and c.y are mutable
b.x = 5; b.y = 6;

struct T<'l> { p: &'l S,
              q: &'l mut S }

let mut u;
u = T { p: &b, q: &mut c };
u.q.x = 7;
```

(this one works; path all way down `u.q.x` is marked `mut`)

Inherited mutability in summary

Inherited mutability: if you own something or have exclusive access to it, you can modify it.

- You just might need to move it into a local slot marked **mut** first.
- But the inverse does not hold!

Interior mutability

Library types **Cell** and **RefCell** can be modified via shared references

- **Cell<T>**: provides **get** and **set** methods
 - easy access, but only usable for **T** : **Copy**
 - for other types, must use ...
- **RefCell<T>**: you claim temporary exclusive access.
 - Compiler does not check your claim!
 - Dynamically checked; task failure if you broke rules.

Demo of Cell<T>

playpen (<http://is.gd/bfXxT0>)

```
enum E { A(int), B(int) }
let a = Cell::new(A(16)); let b = Cell::new(B(17));
let c1 = &a; let _c2 = &b;
foo(c1, c1);

fn foo(p1: &Cell<E>, p2: &Cell<E>) {
  p2.set(B(0xBaC0de));
  match p1.get() {
    B(ref val) | A(ref val) => {
      println!("{:X}", *val);
    }
  }
}
```

(this is fine!)

Back to inherited vs interior

So what does the **mut** marker actually mean?

- **mut** does not mark all and only "mutable things".
 - It marks the items for which, at some point, someone wants *exclusive access* to do some operation (often mutation).

Topics not covered

- subtype relation of $\&'a \mathbf{T}$ and $\&'b \mathbf{T}$ induced by $'b: 'a$
- traits as existentials (object-oriented dispatch)
- borrow-checking static analysis rules
- Rust and closures

Join the Fun!

`rust-lang.org`



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

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

mozilla