a systems language
pursuing the trifecta
safe, concurrent, fast *-lkuper*

mozilla

“rust is like c++ grew up and went to grad school, shares an office with erlang, and is dating sml”

-rpearl, #rust

stack allocation; memory layout;
monomorphisation of generics

safe task-based concurrency, failure

type safety; destructuring bind; type classes

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**
- **Part IV: Concurrency model**

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 remains as way to “go wrong”)
  - and even safe code can fail (but only in controlled fashion)

Simple source ⇔ compiled code relationship

- This is a reason C persists to this day
- Programmer can build mental model of machine state
- Programmer can also control low-level details (e.g. memory layout)
- Goal: Rust should preserve this relationship …
  - … while **retaining** memory safety …
  - … without runtime cost.

Zero-cost abstractions

- Goal: do not pay at runtime for a feature unused by program
- There is still a non-zero cognitive cost
  - Often must think more about data representation
  - Make choices about memory allocation
- But in safe blocks of code, compiler checks our assumptions

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

  Systems programming under the influence of FP
- **Part III: Ownership and borrowing**
- **Part IV: Concurrency model**

Expression-oriented

- not statement-oriented (unless you want to be)

- An expression: `2 + 3 > 5`

- An expression: `{ let x = 2 + 3; x > 5 }`

- A binding of `y` followed by an expression:

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

- Function definition and invocation

```
fn add3(x:int) -> int { x + 3 }
let y = foo(2) > 5;
```

Expression-oriented

- not statement-oriented (unless you want to be)

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

-
```
fn add3(x:int) -> int { x + 3 }
```

Expression-oriented

- not statement-oriented (unless you want to be)
- ```
  let y = { let x = 2 + 3; x > 5 };
  if y { x + 6 } else { x + 7 }
  ```
- ```
  fn add3(x:int) -> int { x + 3 }
  ```

- But `return` statement is available if you prefer that style

```
fn add3(x:int) -> int { return x + 3; }

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

Syntax extensions

- C has a preprocessor
- Likewise, Rust has syntax extensions
- Macro-invocations in Rust look like `macroname!(...)`
  - Eases lexical analysis (for simple-minded ...)

```
println!("Hello World {:d}", some_int);
assert!(some_int == 17);
fail!("Unexpected: {:?}", structure);
```

- (User-defined macros are out of scope of talk)

Mutability

- Local state is immutable by default

```
let x = 5;
let mut y = 6;
y = x;       // fine
x = x + 1; // static error!
```

Enumerated variants I

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

Rust enum

```
typedef enum
{
    Red,
    Green,
    Blue
} color_t;
```

C enum

Matching enums

```
fn f(c: Color) {
  match c {
    Red   => /* ... */,
    Green => /* ... */,
    Blue  => /* ... */
  }
}
```

Rust match

```
void f(color_t c) {
  switch (c) {
    case Red:   /* ... */
                break;
    case Green: /* ... */
                break;
    case Blue:  /* ... */
                break;
  }
}
```

C switch

Matching nonsense

```
fn f(c: Color) {
  match c {
    Red   => /* ... */,
    Green => /* ... */,
    17    => /* ... */
  }
}
```

Rust type error

```
void f(color_t c) {
  switch (c) {
    case Red:   /* ... */
                break;
    case Green: /* ... */
                break;
    case 17:    /* ... */
                break;
  }
}
```

C switch

- Rust also checks that cases are exhaustive.

Enumerated variants II: Algebraic Data

```
enum Spot {
    One(int)
    Two(int, int)
}
```

Destructuring match

```
fn magnitude(x: Spot) -> int {
    match x {
      One(n)    => n,
      Two(x, y) => (x*x + y*y).sqrt()
    }
}
```

Structured data

- Similar to **`struct`** in C
  - lay out fields in memory in order of declaration
- Liveness analysis ensures initialization

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

let p34 = Pair{ x: 3, y: 4 };

fn zero_x(p: Pair) -> Pair {
  return Pair{ x: 0, ..p };
}
```

Closures

- Rust offers C-style function-pointers that carry no environment
- Also offers closures, for environment capture
- Syntax is inspired by Ruby blocks

```
let p34 = Pair{ x: 3, y: 4 };
let x_adjuster =
  |new_x| { Pair{ x: new_x, ..p34 } };
let p14 = x_adjuster(1);
let p24 = x_adjuster(2);
println!("p34: {:?} p14: {:?}", p34, p14);
```

⇒ `p34: Pair{x: 3, y: 4} p14: Pair{x: 1, y: 4}`

What about OOP?

- Rust has methods too, and interfaces
- They require we first explore Rust's notion of a "pointer"

Pointers

```
let x: int = 3;
let y: &int = &x;
assert!(*y == 3);

// assert!(y == 3); /* Does not type-check */
```

Pointers and Mutability

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

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

Ownership and Borrowing

- Memory allocated by safe Rust code, 3 cases
  - stack-allocated local memory
  - owned memory: “exchange heap”
  - intra-task shared memory: managed heap
- code can “borrow” references to/into owned memory; static analysis for safety (no aliasing)
  - Can also borrow references into "GC" heap
  - in that case sometimes resort to dynamic enforcement of the borrowing rules

Methods

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

impl Pair {
  fn zeroed_x_copy(self) -> Pair {
    return Pair { x: 0, ..self }
  }

  fn replace_x(&mut self) { self.x = 0; }
}

let mut p_tmp = Pair{ x: 5, y: 6 };
let p06 = p_tmp.zeroed_x_copy();
p_tmp.replace_x(17);
println!("p_tmp: {:?} p06: {:?}", p_tmp, p06);
```

Prints

```
p_tmp: Pair{x: 17, y: 6} p06: Pair{x: 0, y: 6}
```

Generics

- aka Type-Parametericity
- Functions and data types can be abstracted over types, not just values

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

```
fn safe_get<T>(opt: Option<T>, dflt: T) -> T {
  match opt {
    Some(contents) => contents,
    None           => dflt
  }
}
```

Bounded Polymorphism

```
struct Dollars { amt: int }
struct Euros { amt: int }
trait Currency {
    fn render(&self) -> ~str;
    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 }
}
```

Trait Impls

```
impl Currency for Dollars {
    fn render(&self) -> ~str {
      format!("${}", self.amt)
    }
    fn to_euros(&self) -> Euros {
      let a = ((self.amt as f64) * 0.73);
      Euros { amt: a as int }
    }
}

impl Currency for Euros {
    fn render(&self) -> ~str {
      format!("€{}", self.amt)
    }
    fn to_euros(&self) -> Euros { *self }
}
```

Static Resolution

```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

    let eu100 = Euros { amt: 100 };
    let eu200 = Euros { amt: 200 };
    println!("{:?}", add_as_euros(&eu100, &eu200));
```

⇒ `Euros{amt: 300}`

Static Resolution

```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

    let us100 = Dollars { amt: 100 };
    let us200 = Dollars { amt: 200 };
    println!("{:?}", add_as_euros(&us100, &us200));
```

⇒ `Euros{amt: 219}`

Static Resolution (!)

```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

    let us100 = Dollars { amt: 100 };
    let eu200 = Euros { amt: 200 };
    println!("{:?}", add_as_euros(&us100, &eu200));
```

⇒

Static Resolution (!)

```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

    let us100 = Dollars { amt: 100 };
    let eu200 = Euros { amt: 200 };
    println!("{:?}", add_as_euros(&us100, &eu200));
```

```
error: mismatched types: expected `&Dollars`
       but found `&Euros` (expected struct Dollars
       but found struct Euros)
     println!("{:?}", add_as_euros(&us100, &eu200));
                                          ^~~~~
```

Dynamic Dispatch

```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

fn accumeuros(a: &Currency, b: &Currency) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

let us100 = Dollars { amt: 100 };
let eu200 = Euros { amt: 200 };
println!("{:?}", accumeuros(&us100 as &Currency,
                            &eu200 as &Currency));
```

⇒ `Euros{amt: 273}`

An example from C/C++

A (contrived, strawman) example from C/C++

```
enum Flavor { chocolate, vanilla };
struct Cake {
    Flavor flavor; int num_slices;
    void eat_slice();
};
```

```
enum Flavor { chocolate, vanilla };
struct Cake {
    Flavor flavor; int num_slices;
    void eat_slice();
};

Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);

void Cake::eat_slice() { this->num_slices -= 1; }
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);

void Cake::eat_slice() { this->num_slices -= 1; }

void eat_at_least(Cake &cake, int const &threshold)
{
    int eaten_so_far = 0;
    while (cake.num_slices > 0
           && eaten_so_far < threshold) {
        cake.eat_slice();
        eaten_so_far += 1;
    }
}
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);

void eat_entire(Cake &cake) {
    eat_at_least(cake, cake.num_slices);
}
```

```
Cake birthday_cake(Flavor f, int num_slices);
void print_status(Cake const &cake, std::string);
void eat_entire(Cake &cake);

// On return, ate >= `count` (or cake is gone).
void eat_at_least(Cake &cake, int const &count);

void eat_entire(Cake &cake) {
    eat_at_least(cake, cake.num_slices);
}

int main () {
    Cake cake = birthday_cake(vanilla, 16);
    print_status(cake, "at outset");
    eat_at_least(cake, 2);
    print_status(cake, "after 2");
    eat_entire(cake);
    print_status(cake, "finally");
}
```

```
int main () {
    Cake cake = birthday_cake(vanilla, 16);
    print_status(cake, "at outset");
    eat_at_least(cake, 2);
    print_status(cake, "after 2");
    eat_entire(cake);
    print_status(cake, "finally");
}
```

```
int main () {
    Cake cake = birthday_cake(vanilla, 16);
    print_status(cake, "at outset");
    eat_at_least(cake, 2);
    print_status(cake, "after 2");
    eat_entire(cake);
    print_status(cake, "finally");
}
```

Transcript of run:

```
cake at outset has 16 slices.
cake after 2 has 14 slices.
cake finally has 7 slices.
```

Oops.

```
void eat_at_least(Cake &cake, int const &threshold)
{
    int eaten_so_far = 0;
    while (cake.num_slices > 0
           && eaten_so_far < threshold) {
        cake.eat_slice();
        eaten_so_far += 1;
    }
}

void eat_entire(Cake &cake) {
    eat_at_least(cake, cake.num_slices);
}
```

Classic aliasing bug

The previous example was contrived, but aliasing bugs are real. Cause crashes, security holes, and other incorrect behavior

We want Rust to make it harder to make silly mistakes.

(but not impossible)

((you need to opt in to write unsafe code))

The previous example was contrived, but aliasing bugs are real. Cause crashes, security holes, and other incorrect behavior

We want Rust to make it harder to make silly mistakes.

(but not impossible)

((you need to opt in to write `unsafe` code))

What does the Cake code look like in Rust?

```
enum Flavor { chocolate, vanilla }
struct Cake { flavor: Flavor, num_slices: int }
```

```
enum Flavor { chocolate, vanilla }
struct Cake { flavor: Flavor, num_slices: int }

fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)

impl Cake {
    fn eat_slice(&mut self) {
        self.num_slices -= 1;
    }
}
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)

impl Cake {
    fn eat_slice(&mut self) {
        self.num_slices -= 1;
    }
}

fn eat_at_least(cake: &mut Cake, threshold: &int) {
    let mut eaten_so_far = 0;
    while (cake.num_slices > 0
           && eaten_so_far < *threshold) {
        cake.eat_slice(); eaten_so_far += 1;
    }
}
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)

fn eat_entire(cake: &mut Cake) {
    eat_at_least(cake, &cake.num_slices);
}
```

```
fn birthday_cake(f:Flavor, num_slices:int) -> Cake;
fn status(cake: &Cake, when: &str);
fn eat_entire(cake: &mut Cake)

// On return, ate >= `count` (or cake is gone).
fn eat_at_least(cake: &mut Cake, count: &int)

fn eat_entire(cake: &mut Cake) {
    eat_at_least(cake, &cake.num_slices);
}

fn main () {
    let mut cake = birthday_cake(vanilla, 16);
    status(&cake, "at outset");
    eat_at_least(&mut cake, &2);
    status(&cake, "after 2");
    eat_entire(&mut cake);
    status(&cake, "finally");
}
```

- So, wait, was the port successful?

```
% rustc cake.rs
error: cannot borrow `(*cake).num_slices` as
       immutable because it is also borrowed
       as mutable
eat_at_least(cake, &cake.num_slices);
                   ^~~~~~~~~~~~~~~~
note: second borrow of `(*cake).num_slices`
      occurs here
eat_at_least(cake, &cake.num_slices);
             ^~~~
error: aborting due to previous error
```

```
fn eat_entire(cake: &mut Cake) {
    eat_at_least(cake, &cake.num_slices);
}
```

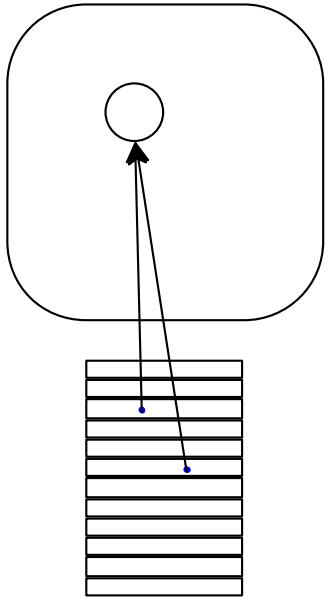
The compiler is complaining about our attempt to alias here!

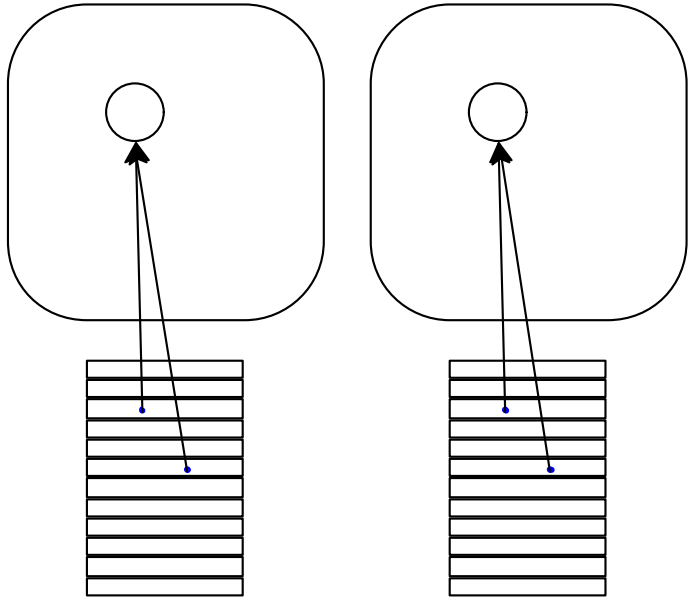
- This fixed version compiles fine.

```
fn eat_entire(cake: &mut Cake) {
    let n = cake.num_slices;
    eat_at_least(cake, &n);
}
```

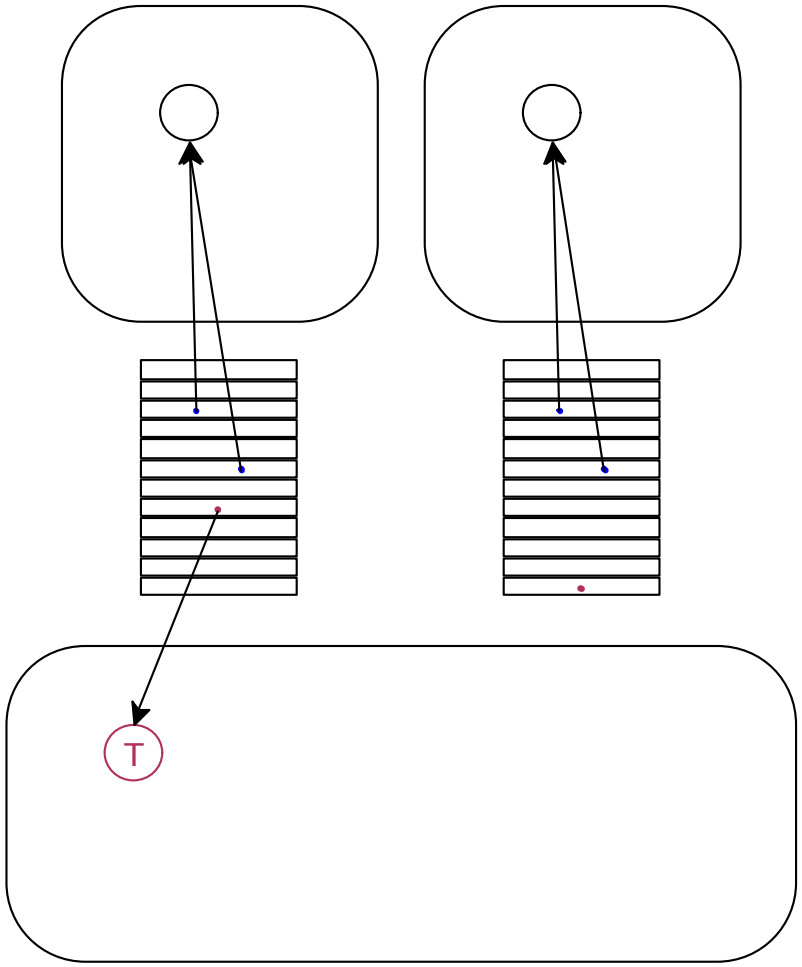
Of course, this fix is applicable to our C++ code too. The point is that Rust enforces these stricter rules outlawing borrows that alias (at least in safe code).

Concurrency

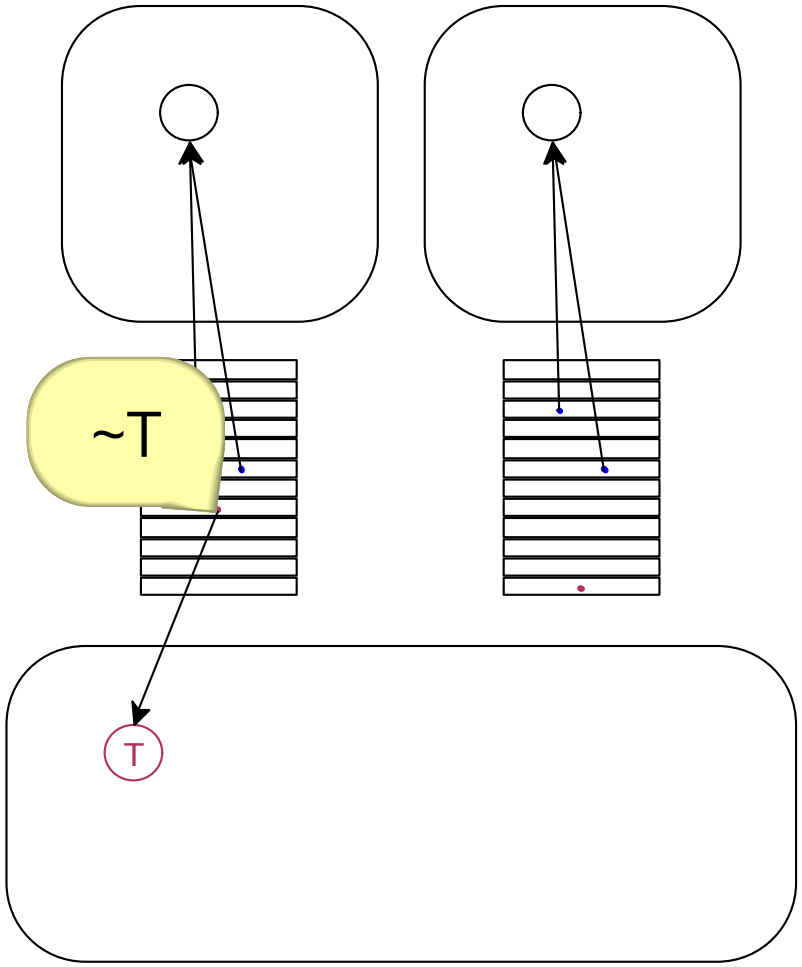




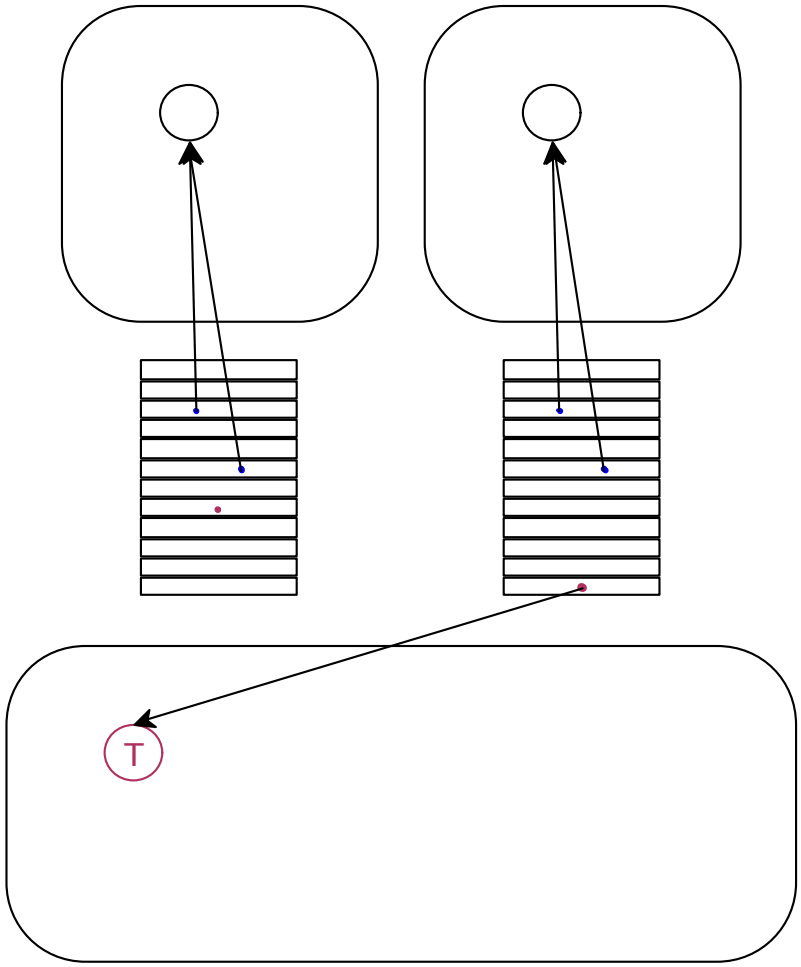
```
let o = ~make_t(); ...
```



```
let o = ~make_t(); ...
```



```
... chan.send(o); /* o is now locally invalid */
```



(telephone demo)

Topics not covered

- regions/lifetimes and their subtyping relationship
- borrow-checking static analysis rules
- freezing/thawing data structures
- one-shot closures: **`proc`**

The Rust team: Brian Anderson, Alex Chrichton, Felix Klock (me), Niko Matsakis, Patrick Walton

Interns/Alumni: Graydon Hoare, Michael Bebenita, Ben Blum, Tim Chevalier, Rafael Espíndola, Roy Frostig, Marijn Haverbeke, Eric Holk, Lindsey Kuper, Elliott Slaughter, Paul Stansifer, Michael Sullivan

(and the many members of the larger Rust community)

http://rust-lang.org/

Join the Fun!

`rust-lang.org`

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

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

mozilla