http://rust-lang.org/

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”

-various, #rust

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

-various, #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

Motivation

- Why invest in a new programming language
- Web browsers are complex programs
- Expensive to innovate and compete while implementing atop standard systems languages

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`**

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**

The Rust Project

- 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.
- (also, very active community)

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

Rust Objectives

- Sound Type checking
  - Eschew runtime overhead in safe abstractions

Rust Objectives

- Sound Type checking
  - Eschew runtime overhead in safe abstractions
- Can opt-in to unsafe code
  - ”Well-typed programs help assign blame.”
  - plus, even safe code can fail (but in controlled fashion)

Rust Objectives

- Sound Type checking
  - Eschew runtime overhead in safe abstractions
- Can opt-in to unsafe code
  - ”Well-typed programs help assign blame.”
  - plus, even safe code can fail (but in controlled fashion)
- Simple source ⇔ compiled code relationship

- **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 }`

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 }
```

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 }
  ```

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

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);
```

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

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

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

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

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 };
}
```

- **`Pair{ fld: value, ..p}`** makes copy of **`p`** with changes

Closures

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

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);
```

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: **`T`**
  - owned memory: “exchange heap”: **`~T`**
  - intra-task sharing: managed library types: **`Gc<T>`**, **`Rc<T>`**

Ownership and Borrowing

- Memory allocated by safe Rust code, 3 cases
  - stack-allocated local memory: **`T`**
  - owned memory: “exchange heap”: **`~T`**
  - intra-task sharing: managed library types: **`Gc<T>`**, **`Rc<T>`**
- code can “borrow” references to/into owned memory; static analysis for safety (no aliasing)
  - **`&T`** or **`&'a T`**

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; }
}
```

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);
```

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

Generics

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

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

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
  }
}
```

(Trait-)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 }
}
```

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));
```

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 }; // (=  € 73)
    let us200 = Dollars { amt: 200 }; // (= € 146)
    println!("{:?}", add_as_euros(&us100, &us200));
```

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 }; // (=  € 73)
    let us200 = Dollars { amt: 200 }; // (= € 146)
    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 }
}
```

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 }; // (= € 73)
    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 }; // (= € 73)
    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 }; // (= € 73)
    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

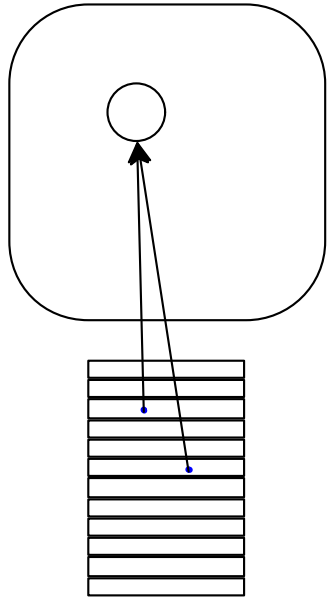
```
fn add_as_euros<C:Currency>(a: &C, b: &C) -> Euros {
    let sum = a.to_euros().amt + b.to_euros().amt;
    Euros{ amt: sum }
}

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));
```

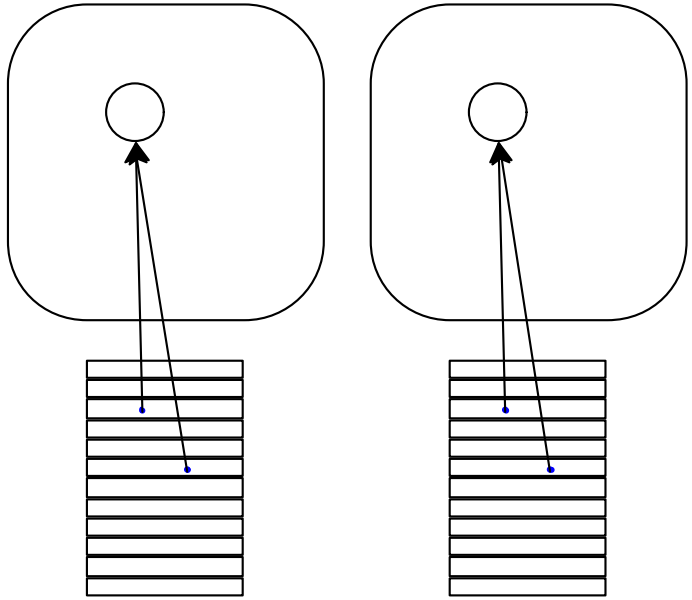
Dynamic Dispatch

```
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));
```

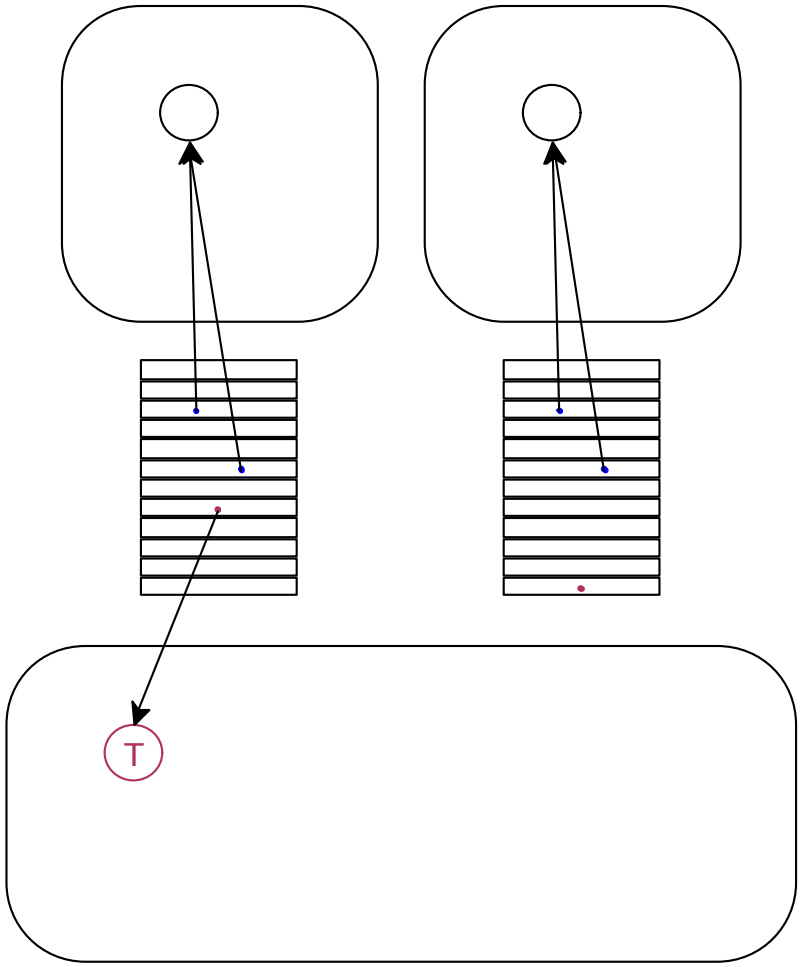
⇒ `Euros{amt: 273}`

Concurrency

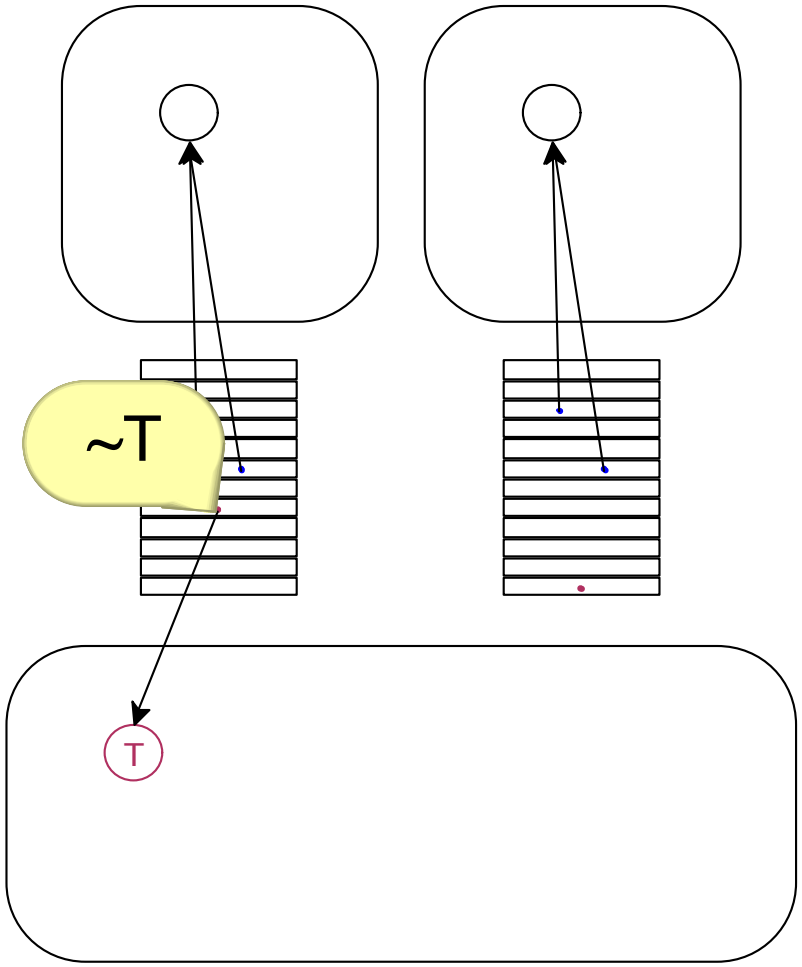




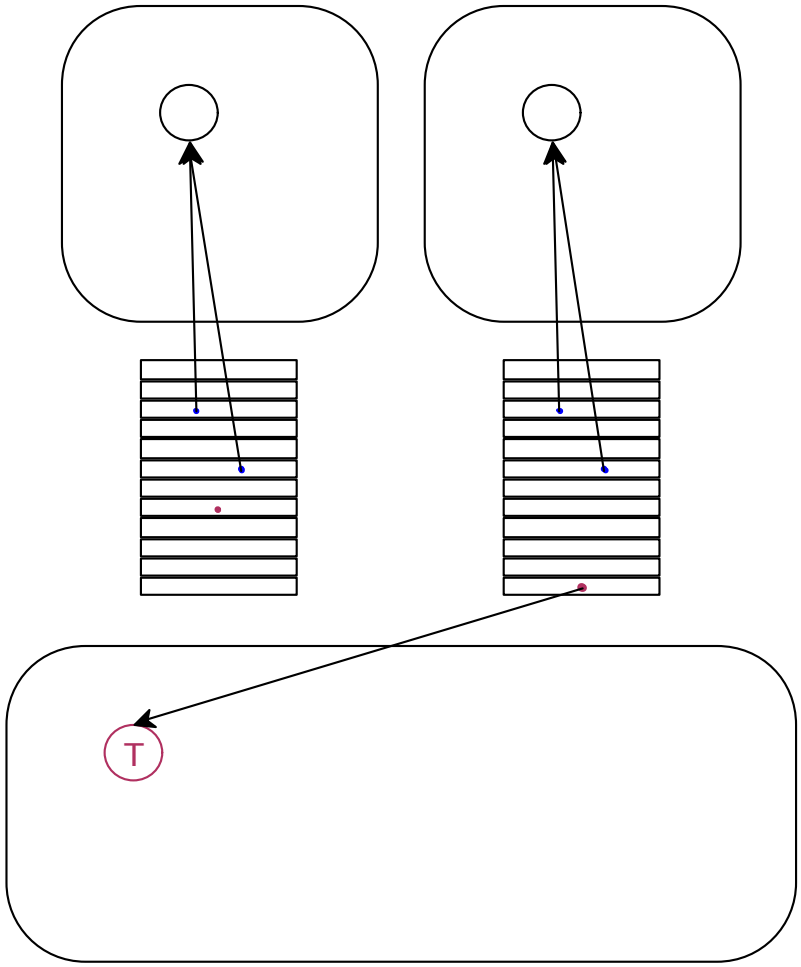
```
let o = ~make_t(); ...
```



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



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



(totally different: circles demo)