Closures for Rust

Michael Sullivan

August 18, 2011

Outline

Introduction

Rust

Closures

Disclaimer

Disclaimer

- Rust is under heavy development.

Disclaimer

- Rust is under heavy development.
- The things described in this talk may not be true tomorrow.

Disclaimer

- Rust is under heavy development.
- The things described in this talk may not be true tomorrow.
- What I discuss and how I present issues reflect my personal biases in language design.

Goals
What do we want in a programming language?

Goals
What do we want in a programming language?

- Fast: generates efficient machine code

Goals

What do we want in a programming language?

- Fast: generates efficient machine code
- Safe: type system provides guarantees that prevent certain bugs

Goals
What do we want in a programming language?

- Fast: generates efficient machine code
- Safe: type system provides guarantees that prevent certain bugs
- Concurrent: easy to build concurrent programs and to take advantage of parallelism

Goals
What do we want in a programming language?

- Fast: generates efficient machine code
- Safe: type system provides guarantees that prevent certain bugs
- Concurrent: easy to build concurrent programs and to take advantage of parallelism
- "Systemsy": fine grained control, predictable performance characteristics

Goals
What do have?

- Firefox is in C++, which is Fast and Systemsy

Goals
What do have?

- Firefox is in C++, which is Fast and Systemsy
- ML is (sometimes) fast and (very) safe

Goals
What do have?

- Firefox is in C++, which is Fast and Systemsy
- ML is (sometimes) fast and (very) safe
- Erlang is safe and concurrent

Goals
What do have?

- Firefox is in C++, which is Fast and Systemsy
- ML is (sometimes) fast and (very) safe
- Erlang is safe and concurrent
- Haskell is (sometimes) fast, (very) safe, and concurrent

Goals
What do have?

- Firefox is in C++, which is Fast and Systemsy
- ML is (sometimes) fast and (very) safe
- Erlang is safe and concurrent
- Haskell is (sometimes) fast, (very) safe, and concurrent
- Java and C# are fast and safe

Rust

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

Rust

Design
Status

Design
Goals (straight from the docs)

Design
Goals (straight from the docs)

- Compile-time error detection and prevention

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment
- System building, analysis and maintenance affordances

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment
- System building, analysis and maintenance affordances
- Clarity and precision of expression

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment
- System building, analysis and maintenance affordances
- Clarity and precision of expression
- Implementation simplicity

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment
- System building, analysis and maintenance affordances
- Clarity and precision of expression
- Implementation simplicity
- Run-time efficiency

Design
Goals (straight from the docs)

- Compile-time error detection and prevention
- Run-time fault tolerance and containment
- System building, analysis and maintenance affordances
- Clarity and precision of expression
- Implementation simplicity
- Run-time efficiency
- High concurrency

Design
Type system features

- Algebraic data type and pattern matching (no null pointers!)
- Polymorphism: functions and types can have generic type parameters
- Type inference on local variables
- Lightweight object system
- Data structures are immutable by default

Design
Other features

- Lightweight tasks with no shared state
- Control over memory allocation
- Move semantics, unique pointers
- Function arguments can be passed by alias
- Typestate system tracks predicates that hold at points in the program

Design
...What?

“It’s like C++ grew up, went to grad school, started dating ML, and is sharing an office with Erlang.”

Status
rustc

- Self-hosting rust compiler

Status
rustc

- Self-hosting rust compiler
- Uses LLVM as a backend

Status
rustc

- Self-hosting rust compiler
- Uses LLVM as a backend
- Handles polymorphism through type passing (blech)

Status
rustc

- Self-hosting rust compiler
- Uses LLVM as a backend
- Handles polymorphism through type passing (blech)
- Memory management through automatic reference counting (eww)

Status
The catch

- Not ready for prime time

Status
The catch

- Not ready for prime time
- Lots of bugs and exposed sharp edges

Status
The catch

- Not ready for prime time
- Lots of bugs and exposed sharp edges
- Language still changing rapidly

Closures

What closures are
Closures in rust

What closures are
Definition

- In civilized languages, functions are first-class values and are allowed to reference variables in enclosing scopes
- That is, they close over their environments

What closures are
Example

```
function add(x) {
  return function(y) { return x + y; };
}
var foo = add(42)(1337); // 1379
```

- Produces a function that adds x to its argument
- Note that the inner function outlives the enclosing function. x can't just be stored on the stack.

What closures are
Another Example

```
function scale(x, v) {
  return map(function(y) { return x * y; },
             v);
}
var v = scale(2, [1, 2, 3]); // [2, 4, 6]
```

- Multiplies every element in an array by some amount
- Note that here the lifetime of the inner function is shorter than the lifetime of the enclosing one. x could just be stored on the stack.

What closures are
Traditional implementation

- Represent functions as a code pointer, environment pointer pair

What closures are
Traditional implementation

- Represent functions as a code pointer, environment pointer pair
- Heap allocate stack frames (or at least the parts that are closed over)

Closures in rust
Design constraints

- Want to be explicit about when we are allocating memory
- Don't want to have to heap allocate closures when it isn't necessary

Closures in rust
Solutions

- Have two function types: **block** and **fn**

Closures in rust
Solutions

- Have two function types: **block** and **fn**
- Values of a **block** type may not be copied, but can be passed by alias; this prevents them from escaping

Closures in rust
Solutions

- Have two function types: **block** and **fn**
- Values of a **block** type may not be copied, but can be passed by alias; this prevents them from escaping
- Values of **fn** type can be automatically coerced to **block** type when passed as function arguments; this allows more code reuse

Closures in rust
Solutions

- Have two function types: **block** and **fn**
- Values of a **block** type may not be copied, but can be passed by alias; this prevents them from escaping
- Values of **fn** type can be automatically coerced to **block** type when passed as function arguments; this allows more code reuse
- Explicitly state what sort of function you writing

Closures in rust
Lambda example

```
fn add(x: int) -> fn(int) -> int {
  ret lambda(y: int) -> int { ret x + y; };
}
```

- `lambda` produces a **fn** that closes over its environment by copying upvars into a heap allocated environment
- Since the variables are copied, changes made to the variables in the enclosing scope will not be reflected in the nested function

Closures in rust
Block example

```
fn scale(x: int, v: &[int]) -> [int] {
  map(block(y: &int) -> int { x * y }, v)
}
```

- `block` produces a **block** that closes over its environment by storing pointers to the stack locations of the variables in a stack allocated environment

Closures in rust
Inference example

```
fn scale(x: int, v: &[int]) -> [int] {
  map({|&y| x * y}, v)
}
```

- Provides an abbreviation for `block`; the argument and return types are type inferred
- Only allowed to appear as a function argument, making type inference easy

Closures in rust
Coercion example

```
fn add1(x: &int) -> int { x + 1 }
fn increment(v: &[int]) -> [int] {
  map(add1, v)
}
```

- add1 is coerced to a **block** when passed to map

Conclusion

- Rust is a new systems language out of Mozilla Research that is designed to be fast, concurrent, and safe
- Closures are a tricky design space in languages that want to be explicit about performance
- Rust approaches the issues by separating functions into multiple varieties.