Default Methods in Rust

Michael Sullivan

August 14, 2013

Outline

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
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
- A somewhat idiosyncratic typeclass system ("traits")
- Data structures are immutable by default
- Region pointers allow safe pointers into non-heap objects

Design
Other features

- Lightweight tasks with no shared state
- Control over memory allocation
- Move semantics, unique pointers

Design
...What?

"It's like C++ grew up, went to grad school, started dating Haskell, 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 and typeclasses by monomorphizing

Status
The catch

- Not *quite* ready for prime time

Status
The catch

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

Status
The catch

- Not *quite* ready for prime time
- Lots of bugs and exposed sharp edges
- Language still evolving

Status
The catch

- Not *quite* ready for prime time
- Lots of bugs and exposed sharp edges
- Language still evolving
- But getting really close!

Traits
What are traits?

- Traits are interfaces that specify a set of methods for types to implement
- Functions can be parameterized over types that implement a certain trait
- Like typeclasses in Haskell

Traits
Trait example

```
trait ToStr {
    fn to_str(&self) -> ~str;
}
impl ToStr for int {
    fn to_str(&self) -> ~str { int::to_str(*self) }
}

fn exclaim<T: ToStr>(x: T) -> ~str {
    x.to_str() + ~"!"
}
```

Traits
More trait example

```
impl<T: ToStr> ToStr for ~[T] {
    fn to_str(&self) -> ~str {
        let strs = self.map(|x| x.to_str());
        fmt!("[%s]", strs)
    }
}

impl<T: ToStr> ToStr for Option<T> {
    fn to_str(&self) -> ~str {
        match self {
            &None => ~"None",
            &Some(ref t) => fmt!("Some(%s)", t.to_str
        }
    }
}
```

Default methods
A solution

- Sometimes you have a method that has a straightforward "default"
- But want to be able to override it

Default methods
A simple example: equality

```
trait Eq {
    fn eq(&self, other: &Self) -> bool;
    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
```

Default methods
An implementation without overriding

- Implementations can choose to use the default implementation...

```
impl Eq for int {
    fn eq(&self, other: &int) -> bool {
        *self == *other
    }
}
```

Default methods
An implementation with overriding

- ... or to override it

```
impl Eq for int {
    fn eq(&self, other: &int) -> bool {
        *self == *other
    }
    fn ne(&self, other: &Self) -> bool {
        *self != *other
    }
}
```

Default methods
Why override?

- Overriding can be useful for performance

Default methods
Why override?

- Overriding can be useful for performance
- And is sometimes semantically necessary (the default implementation is *not* correct for floating point)

Problems
The state at the start of the summer

- The above examples worked...

Problems
The state at the start of the summer

- The above examples worked...
- But anything much more complicated didn't

Problems
Type parameters...

```
trait A<T> {
    fn g(&self, x: T) -> T { x }
}
impl A<int> for int { }

fn main () {
    assert!(0i.g(2i) == 2i);
}
```

- Triggered an ICE

Problems
Type parameters...

```
trait A<T> {
    fn g(&self, x: T) -> T { x }
}
impl A<int> for int { }

fn main () {
    assert!(0i.g(2i) == 2i);
}
```

- Triggered an ICE
- Need to mediate between the different type parameters...

Problems
Calling a default method from another one

```
trait Cat {
    fn meow(&self) -> bool;
    fn scratch(&self) -> bool { self.purr() }
    fn purr(&self) -> bool { true }
}
```

- Triggered an ICE

Problems
And a bunch of related ones

- Calling a default method through a type parameters
- Packaging up an object with a default method

Problems
And a bunch of related ones

- Calling a default method through a type parameters
- Packaging up an object with a default method
- Originally fixed by searching for default methods in more cases

Problems
And a bunch of related ones

- Calling a default method through a type parameters
- Packaging up an object with a default method
- Originally fixed by searching for default methods in more cases
- Eventually fixed by reworking how method lookup is done in trans

Problems
Cross-crate calls

- Couldn't call default methods on a trait in another library

Problems
Cross-crate calls

- Couldn't call default methods on a trait in another library
- Some false starts here - the library code is *scary*

Problems
Cross-crate calls

- Couldn't call default methods on a trait in another library
- Some false starts here - the library code is *scary*
- Solution is to properly export information about default methods

Problems
Cross-crate calls

- Couldn't call default methods on a trait in another library
- Some false starts here - the library code is *scary*
- Solution is to properly export information about default methods
- Required a major rework of what information we track about impls

Problems
Interacting with trait bounds

- Trait bounds on a trait's type params didn't work

Problems
Interacting with trait bounds

- Trait bounds on a trait's type params didn't work
- Calling a function bounded over the trait didn't work

Problems
Supertraits

- Couldn't call methods on a supertrait if there was any polymorphism

Problems
Supertraits

- Couldn't call methods on a supertrait if there was any polymorphism
- Major rework to how supertrait calls are handled

Problems
Supertraits

- Couldn't call methods on a supertrait if there was any polymorphism
- Major rework to how supertrait calls are handled
- Needed to actually check that an impl implemented supertraits...

Problems
Supertraits

- Couldn't call methods on a supertrait if there was any polymorphism
- Major rework to how supertrait calls are handled
- Needed to actually check that an impl implemented supertraits...
- Which required improving the trait resolution algorithm...

Other things
Other projects

- Improved the trait resolution algorithm, removing the need for a hoky workaround in iterators
- Fixed some pattern matching codegen bugs
- Fixing some problems with objects and supertraits

Other things

Fixed a lot of bugs

#2410, #3121, #4055, #4099, #4102, #4102, #4103, #4350, #4396, #4946, #6554, #6868, #6909, #6959, #6967, #7183, #7266, #7278, #7295, #7301, #7341, #7460, #7481, #7536, #7569, #7571, #7661, #7675, #7862

Conclusion

- Rust is a new systems language out of Mozilla Research that is designed to be fast, concurrent, and safe
- I worked on a bunch of different stuff on it this summer
- Default methods now work!