Vector Reform and Static Typeclass Methods

Michael Sullivan

August 15, 2012
Outline

Introduction

Rust

Vectors

Static Trait Methods

Other

Conclusion
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 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

\textit{rustc}

- Self-hosting rust compiler
- Uses LLVM as a backend
- Handles polymorphism and typeclasses by monomorphizing
Status

**rustc**

- Self-hosting rust compiler
- Uses LLVM as a backend
- Handles polymorphism and typeclasses by monomorphizing
- 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
status
the catch

- Not ready for prime time
- Lots of bugs and exposed sharp edges
- Language still changing rapidly
- But getting really close!
Vectors

Rust pointer types (@ and ~)
Vectors
Rust pointer types (@ and ~)

@-pointers

- We want to be able to put objects in the heap
- Want to automatically reclaim memory when all references are dropped
Rust pointer types (@ and ~)

@-pointers

- We want to be able to put objects in the heap
- Want to automatically reclaim memory when all references are dropped
- @-pointers do this; something of type @int is a pointer to a heap allocated int
- When an @-pointer is copied, just the pointer is copied; there can be multiple references to the same object
Rust pointer types (@ and ~)

@-pointers

- We want to be able to put objects in the heap
- Want to automatically reclaim memory when all references are dropped
- @-pointers do this; something of type @int is a pointer to a heap allocated int
- When an @-pointer is copied, just the pointer is copied; there can be multiple references to the same object
- Since we don’t want to have a concurrent GC, these cannot be sent between tasks
Rust pointer types (@ and ~)

~-pointers

- Sometimes we need to be able to send heap values to other tasks, though
Rust pointer types (@ and ~)

~ pointers

- Sometimes we need to be able to send heap values to other tasks, though
- ~-pointers are unique pointers; the object pointed to is owned by exactly one pointer
- When a ~-pointer is copied, the underlying data is copied as well
- ~-pointers can be sent to other tasks by “move”; the sender must relinquish its reference
Vectors

Vector types

- \([T]\) is the type of vectors containing \(T\)
- Vectors are a “second class” type: they can only appear inside some kind of pointer type
- In memory, vectors look like

```rust
def struct vec {
    size_t size;
    size_t allocated;
    char buf[];
}
```
Vectors

Some vector code

```rust
fn seq_range(lo: uint, hi: uint) -> ~[uint] {
    let mut v = ~[];
    for uint::range(lo, hi) |i| {
        vec::push(v, i);
    }
}
```

• v must be the only pointer to the vector, so we can get away with modifying it in place.
Vectors

Some vector code

```rust
def seq_range(lo: uint, hi: uint) -> ~[uint]
{
    let mut v = ~[];
    for uint::range(lo, hi) |i| {
        vec::push(v, i);
    }
}
```

- `v` must be the only pointer to the vector, so we can get away with modifying it in place.
- Unfortunately, this can’t work with an @-vector.
Vectors

*How do we build up @-vectors?*

- We can’t modify or resize an @-vector
- But building a vector by pushing elements on the back seems to be a very natural imperative idiom
Vectors

How do we build up @-vectors?

- We can’t modify or resize an @-vector
- But building a vector by pushing elements on the back seems to be a very natural imperative idiom
- Unless we know for sure that there is only one reference...
Vectors

How do we build up @-vectors?

- We can’t modify or resize an @-vector
- But building a vector by pushing elements on the back seems to be a very natural imperative idiom
- Unless we know for sure that there is only one reference...
- Can build up safe abstractions that wrap a reference to an @-vector; a wrapper object like Java’s ArrayList
Vectors

How do we build up @-vectors?

• We can’t modify or resize an @-vector
• But building a vector by pushing elements on the back seems to be a very natural imperative idiom
• Unless we know for sure that there is only one reference…
• Can build up safe abstractions that wrap a reference to an @-vector; a wrapper object like Java’s ArrayList
• This is somewhat unsatisfying, though; I want a mechanism to construct @-vectors directly
Vectors

An interface for building @-vectors

```rust
fn build<A>(builder: fn(push: fn(+A))) -> @[A];
```

- build allocates a new vector, and then calls builder with an argument that can be used to push onto the array
- build has the only reference to the vector being built until construction is complete
- Implemented with unsafe code, but interface is safe
Vectors
An interface for building @-vectors

```rust
fn build<A>(builder: fn(push: fn(+A))) -> @[A];
```

- build allocates a new vector, and then calls builder with an argument that can be used to push onto the array
- build has the only reference to the vector being built until construction is complete
- Implemented with unsafe code, but interface is safe
- This is a third order function!
Vectors

Using the new interface

```rust
cfn build<A>(builder: fn(push: fn(+A))) -> @[A];
cfn seq_range(lo: uint, hi: uint) -> @[uint] {
    do build |push| {
        for uint::range(lo, hi) |i| { 
            push(i);
        }
    }
}
```

- This seems to be a fairly natural idiom
- Lots of other primitives can be built on it
### 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 Show {
    fn show() -> ~str;
}
impl int : Show {
    fn show() -> ~str { int::to_str(self) }
}

fn exclaim<T: Show>(x: T) -> ~str {
    x.show() + "!";
}
Introduction

Rust

Vectors

Static Trait Methods

Other

Conclusion

Traits

An annoying limitation

- Traits just contain “methods”, which are called with dot notation, and require an element of the trait type
- There are plenty of places where you want to be able to create objects in a type parametric way
Traits

An annoying limitation

- Traits just contain “methods”, which are called with dot notation, and require an element of the trait type
- There are plenty of places where you want to be able to create objects in a type parametric way
- Consider a trait for reading in elements of a type from a string
**Static trait methods**

**A solution**

- I added a static keyword that can be applied to trait methods
- Static methods do not take a `self` parameter and cannot be called with dot notation
- Instead, they are a regular function in the parent namespace of the trait
- This function is parameterized over the trait type
Static trait methods

A solution

- I added a static keyword that can be applied to trait methods
- Static methods do not take a self parameter and cannot be called with dot notation
- Instead, they are a regular function in the parent namespace of the trait
- This function is parameterized over the trait type
- (This is how all typeclass functions work in Haskell)
Static trait methods
Some example code

```rust
trait Read {
    static fn read(~str) -> self;
}

read will have the signature:

fn read<T: Read>(~str) -> T
```
Static trait methods

Bringing it all together

trait Buildable<A> {
    static fn build(builder: fn(push: fn(+A))) -> self;
}

fn seq_range<BT: Buildable<uint>>(lo: uint, hi: uint) -> BT {
    do build() |push| {
        for uint::range(lo, hi) |i| {
            push(i);
        }
    }
}

- buildable is a very powerful interface
Other things

Other projects

- Made major syntax changes to vectors and strings
- Added compiler diagnostics to prevent implicit copying of mutable and heap allocated data
- Explicit self parameters
Other things

Fixed a lot of bugs

#1993, #2189, #2351, #2408, #2417, #2422, #2423, #2426, #2446, #2448, #2450, #2462, #2466, #2468, #2473, #2480, #2503, #2531, #2536, #2547, #2552, #2613, #2629, #2630, #2638, #2652, #2705, #2710, #2725, #2730, #2732, #2746, #2747, #2748, #2759, #2792, #2796, #2863, #2906, #2907, #2908, #2922, #3132, #3191
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.
- Third order functions are apparently useful for constructing arrays imperatively.
- Our traits are now almost as cool as Haskell98’s typeclasses.