Rust: a safe and efficient high-level systems programming language

Paulo Sérgio Almeida

JOIN 2015

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY

INESCTEC
TECHNOLOGY & SCIENCE
ASSOCIATE LABORATORY
COORDINATED BY INESCPORTO PORTUGAL

Universidade do Minho

If you could take only one programming language to a desert island

Many languages

Ada
Basic
C
C++
C#
Clojure
Common Lisp
Elixir
Erlang
Fortran
F#
Haskell
Java
JavaScript
Julia
Go
Groovy
Lua
Nim
OCaml
Objective C
Pascal
Perl
PHP
Python
Ruby
Rust
Scala
Scheme
Smalltalk
Swift

Some options (I)

- Mostly functional languages
  - Haskell, Erlang, Clojure
  - elegant and powerful
  - slow in some domains
  - memory consuming
  - unpredictable performance

Some options (II)

- Classic imperative languages
  - C, C++
  - control
  - speed
  - good memory consumption
  - the segmentation fault hell

Some options (III)

- Dynamic languages
  - Python, Ruby, Perl
  - nice for prototyping
  - too slow
  - too memory consuming

Some options (IV)

- Managed OO languages
  - Java, C#
  - fast
  - can be memory consuming
  - GC pauses
  - spaghetti mutable state hell

Memory consumption in Java

- Why are most games written in C++ and not Java?
- Java:

```
class Point { int x; int y; }
class Rectangle { Point p1; Point p2; }
```

- How much memory for an array of 1M rectangles?
  - around 2*2*4*1M = 16M bytes?
  - more likely at least:
    - (8+2*4 + 2*(8+2*4))*1M = 48M bytes (32 bit JVM)
    - (16+2*8 + 2*(16+2*4))*1M = 80M bytes (64 bit JVM)
  - And making it twice that for efficient GC ...
  - An order of magnitude more memory than in C/C++

Accidental mutable state sharing in OO languages

```
class Rectangle {
    private Point p1;
    private Point p2;
    public void stretchToCorner(Rectangle other) {
        if (...) { p2 = other.p1; } else ...
    }
}
```

- Binary method to stretch `Rectangle` to touch `other`'s corner
- A point object becomes accidentally shared by two rectangles
- The `Point` should have been cloned, but easy to forget
- Class based encapsulation does not prevent this
- Can be subtle bug, with effects much after the invocation

One solution: immutable objects

- Apply lessons from functional languages
- Points being objects are too fragile and error-prone
- Such "objects" as `Point` should really be values
  - and named vectors
  - and used as values, like the mathematical concept
- Immutable object idiom, e.g., Java strings
  - make all instance variables `final`
  - do not let `this` escape in constructor
  - do not allow mutation of reachable objects after construction

Problem with immutable objects: memory locality

- For fast execution, memory locality important
  - cache
  - TLB
- RAM access two orders of magnitude slower than L1 cache
- Array of rectangles traversal in C/C++:
  - 4 rectangles per cache line
- Array of rectangles traversal in Java:
  - it depends
  - if rectangles have been updated at different moments ...
  - potentially 1/2 rectangle per cache line (or worse)

Reasoning about imperative programs (I)

- Functional decomposition
  - Divide and conquer
  - Divide task in sub-tasks: do this, do that
- Scope-based reasoning
  - repeat for each sub-task
  - each implementation declares temporary variables
  - when scope ends, no lasting side-effects should remain
    - i.e., to other things than parameters or result
    - global variables
    - state reachable from other vars from calling scope

Reasoning about imperative programs (II)

- Avoid state interference
- In each context / scope
  - each variable should contain an independent (rep of) value
  - assigning / updating `x` should not impact `y`

```
x = ...
y = ...
print(x)
y.update()
print(x)
```

- Second print should give same result

Garbage collection considered harmful

(Exaggerating a bit . . .)

- For functional languages: essential, transparent, wonderful
- For imperative languages: useful, . . . and dangerous
- WAT? But it is just a useful tool . . .
- Made language designers facilitate widespread sharing of mutable state
  - after all GC makes it easy
  - easy $\neq$ simple (see Rich Hickey talks)
  - mutable state sharing is amazing source of complexity
- Made programmers lower their guards
  - against the problems of mutable state sharing
  - false sense of security: GC makes segfaults disappear
  - so, we feel relaxed and just pass references around

Speaking of making things available because it is easy

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [. . .] But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*
>
> *C.A.R. Hoare*

- Fifty years later there is no excuse for null references
- "Variable may point to something" mindset must end
- Variables should always hold/refer something

Can we have it all?

- High-level functional idioms
- Immutability by default
- Less bugs than in Java (caused by mutable state sharing)
- Control of memory allocation, good memory consumption
- No segmentation faults or uninitialized memory access
- No need for runtime, or GC (and no GC pauses)
- Static typing for programming in the large
- Local type inference for “scripting flavor”
- Shared memory concurrency with no data-races

Can we have efficiency and safety?

Immutable bindings and mutable variables

- Rust adopts immutability by default
- Let binding binds immutable value to identifier

```
let x = 5;
x += 1;      // error: re-assignment of immutable variable `x`
```

- Variables are obtained using `mut` qualifier

```
let mut x = 5;
x += 1;      // ok
```

Variables own representation of value

- Memory layout is dense:
  - variables contain the object, not a reference to the object
  - whether primitive types or composite types
  - whether local variables or members of a `struct`

```
struct Point { x: i32, y: i32 }
struct Rectangle { p1: Point, p2: Point }
```

- `Rectangle` will be 16 bytes as in C/C++

Arrays and Vectors

- Arrays are const sized, inlined, on stack or object

```
let a1 = [1, 2, 3];                            // 3 i32 on stack
let a2 = [0; 5];                               // 5 zeroes on stack
let ap = [Point{x:1,y:2}, Point{x:2,y:3}];  // 2 Points on stack

// struct containing array of 5 i32
struct S { i: i32, a: [i32; 5] }
```

- Vectors: have header inlined, data on heap, grow dynamically

```
let mut v = vec![1, 2, 3];  // type Vec<i32>
let i = v.pop();            // remove last; i = 3; v = [1,2]
let l = v.len();            // l = 2;
v.push(4);                  // append 4; v = [1,2,4]
```

(Im)mutability is transitive

- Local variables / parameters can be declared `mut`
- Fields of structs do not take this qualifier
- They inherit transitively the mutability of owner variable

```
let mut r1 = Rectangle { p1: Point {x: 23, y: 34},
                         p2: Point {x: 14, y: 18} };
let r2 = Rectangle { ... }
r1.p1.x = 45;  // ok;
r2.p1.x = 45;  // error: cannot assign to immutable field `r2.p1.x
```

- We can assign to `x` field of `p1` field of `r1` because `r1` is `mut`

Resource lifetime is scope based

```
fn f() {
  let mut s = HashSet::new();
  s.insert(7);
  ...
}
```

- Struct `HashSet` is kept on the stack
- It may point to other objects on heap
- When scope exits, destructor will be called
- Owned objects on heap will be freed

Variables can be returned

```
fn f() -> HashSet<i32> {
  let mut s = HashSet::new();
  s.insert(7);
  ...
  s
}
```

- Function `f` returns an `HashSet<i32>`
- Last expression implicitly returned from function
- When scope exits, destructor is not called
- Ownership of HashSet is transferred to caller
- Including objects on heap owned by the HashSet
- Physically:
  - at most just a memcpy of the stack allocated struct
  - or nothing, under return value optimization

Variables can be passed to functions

```
fn f() {
  let mut s = HashSet::new();
  s.insert(7);
  g(s);
  s.insert(12);  // error: use of moved value: ‘s‘
}

fn g(s: HashSet<i32>) { ... }
```

- Function `g` takes an `HashSet<i32>` as parameter
- Function `f` invokes it passing `s`
- Ownership of `s` has been transferred into `g`
- Function `f` can no longer use `s` afterwards
- Function `g` is responsible for freeing it

Move semantics in assignment and parameter passing

- Variables own values
- Like parameter passing, assignment also moves ownership

```
let s1 = HashSet::new();
s1.insert(7);
let s2 = s1;
s1.insert(12);  // error: use of moved value: ‘s1‘
```

- In each scope we have a single owner of each resource
- There cannot be two variables aliasing shared mutable state

Single owner per resource

- Improves reasoning:
  - can use composite values like primitive ones
  - no side effects to other variables

```
let mut x = ...
let mut y = ...
println!("{}", x);
y.update();
println!("{}", x);
```

- We know that `x` remains unchanged after update on `y`

Opt-in copy semantics for POD types

- Isn't move too restrictive and artificial sometimes?
- What about primitive types? We are used to copying them
- Plain-Old-Data types can be declared to be `Copy`
  - any type that can be copied by a simple memcpy
  - primitive types are Copy
  - can be copy if all components are copy
  - (types which manage resources or have destructor cannot)
- Useful for small structs; e.g., points, complex numbers

```rust
#[derive(Copy, Clone)]
struct C { re: f32, im: f32 }

let mut c1 = C { re: 4.5, im: 7.8 };
let c2 = c1;
c1.re += 1.0;
```

Opt-in to Copy does not change semantics

- Making type `Copy` does not change semantics
  - Only allows more programs to be compiled
  - If program already compiled, produces same result
- Improves local reasoning
  - not need to review code if type definition changed
- Contrast with value types (e.g., C#, F#, Swift)
  - reference vs. value semantics
  - changing it forces careful code review

Implicit copies only involve memcpy

- Both move and copy can only involve memcpy
- Move in rust makes source of move compile-time inaccessible
  - source can be left alone
  - no need to update source, to make it "empty" but usable
- Comparing with C++
  - No copy constructors
  - No move constructors
  - No hidden arbitrary implicit code being run
  - No hidden effects depending on optimization of temporaries

Borrowing: references that grant temporary access

- What if we want to let a function use or update a resource?
  - that we want to keep owning after the invocation
- The function can borrow the resource

```
fn f(p: &Point) -> i32 {
  p.x + p.y
}

let mut p1 = Point { x: 2, y: 4 }
let i = f(&p1);
p1.x += 1;
```

- `f` cannot store `p` in an arbitrary place
- Here we have an immutable borrowing
  - the point can be only read; not updated

Mutable borrowing: to update object

- We can have mutable references through `&mut`

```
fn f(p: &mut Point) {
  p.x += 1;
  p.y -= 1;
}

let mut p1 = Point { x: 2, y: 4 }
f(&mut p1);
p1.x += 1;
```

- `f` can update `p`
- as before, `f` cannot store `p` in an arbitrary place

Explicit references in Rust improve local reasoning

- C++ references are obtained implicitly

  ```
  int i = 2;
  f(i);
  std::cout << i;  // i = ?
  ```

  Cannot know if `i` changed without looking at `f`'s declaration

- Rust:

  ```
  let mut p = Point { x: 2, y: 4 }
  f(&p);
  ```

  `p` cannot have been updated

- Rust:

  ```
  let mut p = Point { x: 2, y: 4 }
  f(&mut p);
  ```

  `p` may have been updated

Dereferencing a reference

- Either implicit (auto-dereferencing), e.g, for fields or methods

```
fn area(r: &Rectangle) -> i32 {
    ((r.p2.x - r.p1.x) * (r.p2.y - r.p1.y)).abs()
}
```

- Or explicit, C-like, e.g., for primitive types

```
fn inc(ir: &mut i32) {
    *ir += 1;
}
```

- Reference itself can be updated, if `mut`

```
let mut x = 5;
let mut y = 7;
let mut r = &mut x;
*r += 1;    // increments x
r = &mut y;
*r += 1;    // increments y
```

Borrowed references can be returned only if it is safe

- Only if the object lifetime is long enough
- Otherwise, compile time error
- Error if trying to return reference to local var

```
fn return_var() -> &i32 {
    let x = 5;
    &x
}
```

Returning reference to object from caller scope

```
fn largest_coord(p: &mut Point) -> &mut i32 {
    if p.x > p.y { &mut p.x } else { &mut p.y }
}

let mut p = Point { x: 5, y: 7 };
inc(largest_coord(&mut p));
```

- In this case a mutable reference to the interior of the `Point`
- Which the caller uses to operate on the largest coordinate
- Compiler relates lifetimes of result and parameter

Explicit lifetime parameters

- If several references involved, explicit lifetimes can be used
- Function can be generic over lifetime parameter(s)

```
fn greater<'a>(r1: &'a i32, r2: &'a i32) -> &'a i32 {
    if *r1 > *r2 { r1 } else { r2 }
}
```

- Here 'a is a lifetime parameter
- Incorrect usage is compile-time flagged by the borrow checker

```
let x = 5;
let r;
{
    let y = 7;
    // error: `y` does not live long enough
    r = greater(&x, &y);
}
println!("{}", r);
```

Several readers or one writer

- Remember readers-writers from concurrent programming?
- If one is mutating, no one else should be reading or mutating
- Rust enforces similar guarantees for single-threaded scopes
- In each scope, for each resource
  - either there are several references (`&T`)
  - or exactly one mutable reference (`&mut T`)
- A variable
  - cannot be updated while borrowed
  - cannot be accessed while mutably borrowed

Several readers or one writer

```
fn largest_coord(p: &mut Point) -> &mut i32 {
    if p.x > p.y { &mut p.x } else { &mut p.y }
}

let mut p = Point { x: 5, y: 7 };
{
    let r = largest_coord(&mut p);
    *r += 1;
    //error: cannot assign to `p.x` because it is borrowed
    p.x += 1;
}
p.x += 1;    // ok
```

Single mutable reference prevents memory unsafety

- Consider iterating a vector and appending to other

```
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
    for i in from {
        to.push(*i);
    }
}
```

- If both parameters could refer to same `Vec`
  - iterator would traverse a range of memory
  - appending to destination `Vec` could reallocate it
  - iterator would traverse freed memory
- Cannot happen: two refs cannot alias mutable state

```
// error: cannot borrow `vec` as mutable because
// it is also borrowed as immutable
push_all(&vec, &mut vec);
```

Interior mutability

- Some types allow non `mut` variables to refer to mutable state
- `Cell<T>` allows update through explicit copies
  - for `Copy` types
- `RefCell<T>` allows temporary mutable borrows
  - checked at runtime
- `Mutex<T>` allows controlled mutation under concurrency
  - locking the resource
- These are for advanced usages
  - to be used rarely
  - noticeable when used
  - access to mutable state is controlled
- Analogous to explicit references in functional languages
  - ML, Clojure

Other kinds of references

- For advanced uses, Rust exposes other reference types
  - whole programs can be written without them
- `Box<T>`
  - for exclusive mutable ownership of heap data
- `Rc<T>` – reference-counted pointer type
  - for shared referencing of heap data
  - to be used within each thread
- `Arc<T>` – atomically reference-counted pointer type
  - for shared referencing of heap data
  - when sharing data among threads; e.g., `Arc<Mutex<T>>`
- Rust philosophy:
  - only pay performance cost when needed
  - unlike Swift which has a single Arc-like reference

Rust emphasizes generic abstractions

- Not object-orientation
- Exposes many concepts
  - `struct`s, tuples, `enum`s, functions, `trait`s, `impl`s
- These can be generic
  - parameterized over types
  - possibly bounded

Module based encapsulation

- The unit of structuring is the module
  - with possibly nested modules
- Anything not `pub` is not visible outside module
- `pub` items are visible to client module that `use`s them
- `pub` can be applied to many concepts:
  - fields
  - structs
  - enums
  - functions
  - traits

Module based encapsulation

```
pub struct Graph<N,E> { nodes: Vec<Node<N,E>> }

pub struct Node<N,E> {
    neighbors: Vec<usize>,
    edges: Vec<E>,
    pub data: N
}

pub fn add_node<N, E>(g: &mut Graph<N,E>, data: N) {
    g.nodes.push(Node {
        neighbors: Vec::new(),
        edges: Vec::new(),
        data: data });
}
```

- Generic `Graph` type, parameterized over node and edge types
- `Graph` can be used outside module, `nodes` field cannot
- Both `Node` type and its `data` field visible outside module
- function `add_node` can access all fields

Methods

- Methods are functions that take object as first parameter
- Defined in `impl` blocks
- Special syntax with `self`
- As normal parameter passing, three ways to pass object:
  - By reference, borrowing, with `&self`
  - By mutable reference, mutably borrowing, with `&mut self`
  - By move, transferring ownership, with `self` or `mut self`
- At calling side, object is auto-borrowed, if necessary

Taking &self

- The first choice
- Methods that merely perform computations

```
pub struct Circle {
    pos: Point,
    radius: f64,
}

impl Circle {
    pub fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

let c = Circle { pos: Point{x:4.5, y:6.7}, radius: 2.3 };
let a = c.area();
```

Taking &mut self

- For mutator methods

```
pub struct Graph<N,E> {
    nodes: Vec<Node<N,E>>,
}

impl<N,E> Graph<N,E> {
    pub fn add_node(&mut self, data: N) -> usize {
        let id = self.nodes.len();
        self.nodes.push(Node {
            neighbors: Vec::new(),
            edges: Vec::new(),
            data: data });
        id
    }
}
```

Taking `self` / `mut self`

- Takes ownership, allowing returning object (not reference)
- Efficient implementations exposing functional interface
- Example: `String`; from string.rs

```
impl<'a> Add<&'a str> for String {
    type Output = String;

    #[inline]
    fn add(mut self, other: &str) -> String {
        self.push_str(other);
        self
    }
}
```

- Strings can be added, functional style, no wasteful cloning

```
let s1 = "Hello ".to_string();
let s2 = s1 + "big";
let s3 = s2 + " world";
println!("{}", s3);
```

Associated functions

- There are no constructors; no special `new`
- Associated functions do not take self ("static methods")
- By convention, function `new` commonly provided

```
impl<N,E> Graph<N,E> {

    pub fn new() -> Graph<N,E> { Graph { nodes: Vec::new() } }

    pub fn with_capacity(n: usize) -> Graph<N,E> {
        Graph { nodes: Vec::with_capacity(n) }
    }

    pub fn empty(nodes: Vec<N>) -> Graph<N,E> {
        let mut g = Graph { nodes: Vec::with_capacity(nodes.len()) };
        for x in nodes { g.add_node(x); }
        g
    }
}

let mut g: Graph<&str,()> = Graph::empty(vec!["Alice", "Bob"]);
```

Traits

- Notion of interface / protocol
  - which can be implemented for several types
  - even a posteriori for existing types
  - allowing extension methods
- Serve as bounds for parametric polymorphism
  - with impls checked at definition against bounds;
  - not at instantiation (C++ templates nightmare)
  - generic impls monomorphized and statically dispatched
- Allow subtype polymorphism via trait objects
  - for heterogeneous containers
  - for dynamic dispatching

A posteriori implementation for existing types

- Not necessarily `structs`

```
trait Measure {
    fn norm(&self) -> f64;
}

impl Measure for f64 {
    fn norm(&self) -> f64 { self.abs() }
}

impl Measure for (f64, f64) {
    fn norm(&self) -> f64 {
        (self.0 * self.0 + self.1 * self.1).sqrt()
    }
}

5.6.norm()
(23.2, 45.4).norm()
```

Traits as bounds for parametric polymorphism

```
impl<T: Measure> Measure for [T] {
    fn norm(&self) -> f64 {
        let mut sum = 0.0;
        for x in self.iter() {
            let n = x.norm();
            sum += n * n;
        }
        sum.sqrt()
    }
}

[3.4, 4.5].norm()
[(23.2, 45.4), (34.2, 56.1)].norm()
```

Operator overloading

```rust
use std::ops::{Add, Mul};

#[derive(Copy, Clone)]
pub struct C(f64, f64);

impl Add for C {
    type Output = Self;
    fn add(mut self, other: Self) -> Self {
        self.0 += other.0;
        self.1 += other.1;
        self
    }
}

impl Mul for C {
    type Output = Self;
    fn mul(self, other: Self) -> Self {
        let (a, b) = (self, other);
        C(a.0*b.0 - a.1*b.1, a.0*b.1 + a.1*b.0)
    }
}
```

Operator overloading

```
impl Mul<f64> for C {
    type Output = Self;
    fn mul(mut self, other: f64) -> Self {
        self.0 *= other;
        self.1 *= other;
        self
    }
}

impl Mul<C> for f64 {
    type Output = C;
    fn mul(self, other: C) -> C { other * self }
}

fn main() {
    let c1 = C(3.4, 2.3);
    let c2 = C(5.2, 6.4);
    let c3 = c1 + c2;
    let c4 = c1 * c2;
    let mut c5 = 0.2 * c3 + 0.8 * c4;
    c5 = c5 * 1.3;
}
```

Subtype polymorphism and dynamic dispatch

```
trait Shape { fn area(&self) -> f64; }

struct Circle { pos: Point, radius: f64 }

struct Rectangle { p1: Point, p2: Point }

impl Shape for Circle { fn area(&self) -> f64 { ... } }

impl Shape for Rectangle { fn area(&self) -> f64 { ... } }

let c = Circle { pos: Point{x:3.4,y:6.7}, radius: 2.3 };
let r = Rectangle { p1: Point{x:2.3,y:4.5}, p2: Point{x:5.6,y:7.8} };
let a1 = c.area();  // static dispatch
let a2 = r.area();  // static dispatch
let shapes: [&Shape; 2] = [&c, &r];  // array of trait objects
for s in shapes.iter() {
    println!("{}", s.area());  // dynamic dispatch
}
```

Closures

- Anonymous functions
- Capture variables from enclosing scope into an environment
- Not restricted to using values: may update variables
- Possible to be statically dispatched and with no allocation
- Variants according to how environment is passed to call
  - `Fn` – call borrows the environment `&self`
  - `FnMut` – call borrows mutably the environment `&mut self`
  - `FnOnce` – call moves the environment `self`
- And according to how variables are captured
  - by reference
  - by mutable reference
  - by move

Fn closures

- Call takes environment as `&self`
- Do not have side effects on environment
- Environment variables can be read with closure in scope

```
let (min, max) = (5,8);
let between = &|x| x >= min && x < max;
println!("{}", between(9));  // false
use_closure(between);

fn use_closure<F>(f: &F) where F: Fn(i32) -> bool {
    println!("{}", f(6));  // true
    println!("{}", f(4));  // false
}
```

FnMut closures

- Call takes environment as `&mut self`
- Can have side effects on environment
- Updated environment variables cannot be accessed with closure in scope

```
let (min, max) = (5,8);
let mut tot = 0;
{
    let check = &mut |x| if x >= min && x < max { tot += 1; };
    check(7);
    use_closure(check);
}
println!("{}", tot);  // 2

fn use_closure<F>(f: &mut F) where F: FnMut(i32) {
    f(6);
    f(4);
}
```

FnOnce closures

- Call takes environment as self
- Allow environment variables to be moved out of closure
- Can only be called once

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3];
    thread::spawn(|| {
        let v = data;
        thread::sleep_ms(300);
        println!("{:?}", v);
    });
    thread::sleep_ms(600);
}
```

Moving environment into closure

- When closure should survive creation scope
  - e.g., function which returns adder closure
  - e.g., to spawn threads
- move keyword forces environment move
- Closures are unsized types
  - must be put into Box to be returned

```
fn make_adder(x: i32) -> Box<Fn(i32) -> i32> {
    Box::new(move |y| x + y)
}

let a = make_adder(5);
println!("{}", a(7));    // 12
```

Introduction

State: Ownership, Moving, Borrowing, Lifetimes

Abstraction

Control Flow and Iteration

Conclusion

Decision making

- Cover all cases elegantly
  - language makes impossible to forget case
  - extract relevant data for each case
- Ingredients
  - `enum`: discriminated unions of rich data
  - pattern matching
  - exaustiveness of `match` construct
- Blends with imperative idioms
  - does not force expression-orientation
  - allowing `break` and `return`
- Blends with ownership system
  - allowing borrowing of matched substructure

Enums

- Sum types, which represent one of several variants
- Each may: have no data, be tuple-like, or be struct-like
- Can declare struct-like alternatives without pre-existing types
- Space reserved for largest variant, like C unions
- Can represent alternatives inlined, without allocation
  - Unlike OO-idiom of using subclassing
- Mutable var or `&mut` allows changing variant in-place

Message type with enum

```
enum Msg<K,V> {
    Insert(V),
    Get(K),
    Put(K, V),
    Delete(K)
}
use Msg::*;

fn handle<K,V>(msg: Msg<K,V>) {
    match msg {
        Insert(v) =>  { /* code here */ }
        Get(k) =>  { /* code here */ }
        Put(k, v) =>  { /* code here */ }
        Delete(k) => { /* code here */ }
    }
}
```

Binary tree with enum

```
enum BinaryTree<T> {
    Leaf(T),
    Node(Box<BinaryTree<T>>, T, Box<BinaryTree<T>>)
}
use BinaryTree::*;

impl<T> BinaryTree<T> {
    fn depth(&self) -> u32 {
        match *self {
            Leaf(_) => 0,
            Node(ref l, _, ref r) => 1 + max(l.depth(), r.depth())
        }
    }
}
```

- Needs explicit `Box` to avoid infinite size
- `ref` allows borrowing matched substructure

Error handling

- No exceptions (checked or unchecked)
- Panic to unwind stack and abort thread
  - assert-like, for irrecoverable errors; e.g., bugs
- Failures reported through `Option` and `Result` enums
- Macros to help in achieving safety and elegance
  - e.g., `try!()`

Early return within match under error

```
enum MyError { IO(io::Error), Parse{line: String, number: u32} }
use MyError::*;
fn sum_lines(file_name: &str) -> Result<u64, MyError> {
    let mut file = match File::open(file_name) {
        Ok(file) => file,
        Err(err) => return Err(IO(err))
    };
    let (mut sum, mut cnt) = (0, 0);
    for line in BufReader::new(&mut file).lines() {
        cnt += 1;
        let line = match line {
            Ok(line) => line,
            Err(err) => return Err(IO(err))
        };
        let num: u64 = match line.parse() {
            Ok(num) => num,
            Err(_) => return Err(Parse{line: line, number: cnt})
        };
        sum += num;
    }
    Ok(sum)
}
```

Elegant error handling with try!

```
impl From<io::Error> for MyError {
    fn from(err: io::Error) -> MyError {
        IO(err)
    }
}

fn sum_lines(file_name: &str) -> Result<u64, MyError> {
    let mut file = try!(File::open(file_name));
    let (mut sum, mut cnt) = (0, 0);
    for line in BufReader::new(&mut file).lines() {
        cnt += 1;
        let line = try!(line);
        let num: u64 = try!(
            line.parse().map_err(|_| Parse{line: line, number: cnt}));
        sum += num;
    }
    Ok(sum)
}
```

Iterators

- `Iterator`: base trait for external iteration
- Iterator state in object
- Implementations must define `next` method returning `Option`
- Syntactic sugar for `Range` iterators

  ```
  0..n  // iterator over numbers 0 <= i < n
  ```
- `for` loop uses iterator

Containers from library provide iterators

- Vec

```
for x in [1, 5, 7].iter() {
    println!("{}", x);
}
```

- HashSet

```
let mut s = HashSet::new();
s.insert("alice");
s.insert("bob");
for e in s {
    println!("{}", e);
}
```

- HashMap

```
let mut h = HashMap::new();
h.insert("alice", 3);
h.insert("bob", 5);
for (k,v) in h {
    println!("key:{}, value:{}", k, v);
}
```

Iterators can borrow, mutably borrow, or own

- Given vector of P's

```
let mut v = vec![P {x:2, y:4}, P {x:6, y:7}];
```

- Iterate borrowing immutably

```
let mut sum = 0;
for p in &v {
    sum += p.x;
}
```

- Iterate borrowing mutably

```
for p in &mut v {
    p.x += 1;
}
```

- Iterate over moved container

```
let mut s = HashSet::new();
for p in v {
    s.insert(p);
}
v[0].x += 1;  // error: use of moved value: `v`
```

User defined iterators; example: fibonacci numbers

```
pub struct Fibonacci { curr: u64, next: u64, stop: u64 }

impl Iterator for Fibonacci {
    type Item = u64;
    fn next(&mut self) -> Option<u64> {
        let res = self.curr;
        if res >= self.stop {
            None
        } else {
            self.curr = self.next;
            self.next += res;
            Some(res)
        }
    }
}

pub fn fibonacci(stop: u64) -> Fibonacci {
    Fibonacci { curr: 0, next: 1, stop: stop }
}

for i in fibonacci(1000) { println!("{}", i); }
```

Consuming an iterator

- Collecting items

```
let s: HashSet<_> = fibonacci(400).collect();
let v: Vec<_> = fibonacci(200).collect();
```

  - collect generic on result type
  - type inference based on destination allows same code

- Folding

```
fn sum_squares(n: u32) -> u32 {
    (1..n+1).fold(0, |sum, x| sum + x*x)
}
```

- Partitioning according to predicate

```
let (even, odd): (Vec<_>, _) =
    fibonacci(100).partition(|&n| n % 2 == 0);
```

Iterator adapters

- Take iterator, return transformed iterator
- Classic ones: `map`, `filter`, `zip`, `take`, `skip`, `cycle`, ...
- Are lazy: do nothing until consumed

```
warning: unused result which must be used: iterator adaptors are
lazy and do nothing unless consumed, #[warn(unused_must_use)] on by
fibonacci(100).filter(|&i| i % 3 == 0);
^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- "Sum of the first 10 odd numbers that are not multiples of 3"

```
let sum = (0..)
    .filter(|&n| n % 2 != 0 && n % 3 != 0)
    .take(10)
    .fold(0, |s, n|  s + n);
```

Introduction

State: Ownership, Moving, Borrowing, Lifetimes

Abstraction

Control Flow and Iteration

Conclusion

Mutable state taken seriously

- Functional idioms are already widespread
- But purely functional languages can be limiting
- Current OO languages create mutability spaghetti
  - makes some people think functional is the only way-out
- Much research on these problems for ages; starting in the 70s
  - ends up mostly in experimental/research languages
- If successful, Rust can be the first widespread language
  - taking mutable state seriously
  - while allowing safe and “bare metal” efficient programs
  - with high-level abstractions

For another day

- Concurrency
  - leverages ownership, borrowing and immutability
  - concurrency related traits: `Send`, `Sync`
  - guarantee of no data races checked at compile-time
  - possible to create threads that operate safely on creator stack
- Macros
  - syntactic abstraction, at compile time
  - hygienic
- Unsafe blocks and raw pointers
  - to be used very exceptionally
  - e.g., used in the implementation of standard library
  - if one never uses `unsafe`, one never gets segfaults