CPS506 - Comparative Programming Languages

Safety & Rust

Dr. Dave Mason
Department of Computer Science
Ryerson University

Why Program Safety?

- economic costs
- health and safety
- personal information
- exploits and compromises

How Safety?

- testing can only go so far
- dynamically typed languages are "safe"
- but static typing *can* provide more confidence
- ultimate is proof of correctness

How Safe?

- static language safety dependent on type system
- "C" and "C++" are statically typed, but ...
  - null pointer exceptions
  - memory corruption
  - buffer overflows
- want expressive power too
- sometimes need all the performance you can get

Rust

- need a safe systems-programming language
- minimal, predictable overhead
- strongly, statically typed
- no “undefined behaviour” a la C or C++ specs
- no data races

History

- created at Mozilla - started 2010
- version 1.0 in May 2015
- Servo - browser currently at 100,000 LOC
- Dropbox internals
- Redox OS

Paradigm

- imperative
- safe side-effects - even for multi-threading
- no mutable aliasing
- expressive type system
- no accidental run-time costs

Syntax Rules

1. literals
   - numbers: (un)signed ints, floats -17
     3.141592
     13i8
   - characters: 'a' - Unicode
   - strings: "this isn't \"hard\"!"
     r###"raw string with ' \ " #" ##""###
     r#"useful for html <a href="fofof" & etc.."#
   - arrays: [1,2,3] [0;20]
   - slice: part of an array - &a[..] &a[3..6]
   - tuples: (1,"abc")
   - blocks/closures/lambdas:
     || 3
     |arg| arg-delta
2. names
   - upper/lower case, digits, underscore; case sensitive
   - arguments to methods and blocks
   - default immutable - mut keyword if needed
   - snake-case for variables/functions/parameters
   - camel-case for enum/struct/trait
3. functions

Statements - Conditionals

- if

```
let mut x = 3;
if x == 5 {
    x = 10
}
else {
    x += 1
}
```

- match

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
enum Message {
```

Statements - Loops

- while

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

- loop

```
let mut x = 5; // mut x: i32

loop {
    x += x - 3;

    println!("{}", x);
```

Statements - Iterators

- iterators - inlined (ranges, vectors, etc.)

```
for (index, value) in (5..10).enumerate() {
    println!("index = {} and value = {}", index, value)
}
let lines = "hello\nworld".lines();

for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

```
if option.is_some() {
```

Structs

- data containers

```
struct MyData {

}
impl MyData {

}
```

```
inu
inu
```

Traits

- like Haskell type-classes
- like Java interfaces - except not part of definition of base class
- `trait` *traitname*
- `impl` for any types

```
trait Shape {
    fn draw(&self, Surface);
    fn bounding_box(&self) -> BoundingBox;
}
impl Shape for i32 {

}
```

Trait Objects

```
struct Shape { ... }

impl Shape {
    fn draw(&self, u32) { ... }
    fn bounding_box(&self) -> BoundingBox { ... }
    fn default() -> &Self {
}

let s = Shape{}
s.draw(42)
```

- no hierarchy
- like Haskell type-classes

Memory Safety

- no null pointers
  - way to create null pointers
  - `Option` enumerated type
- no dangling pointers
  - value lifetimes are calculated
  - *Rule 1: Every value has a single owner at any given time.* You can move a value from one owner to another, but when a value's owner goes away, the value is freed along with it.
  - *Rule 2: You can borrow a reference to a value, so long as the reference doesn't outlive the value (or equivalently, its owner).* Borrowed references are temporary pointers; they allow you to operate on values you don't own.
  - *Rule 3: You can only modify a value when you have exclusive access to it.*
- no memory leaks
  - value lifetimes are calculated
  - values freed when leave scope
  - additionally, reference-counted values
- no buffer overruns
  - no pointer arithmetic
  - slices used for partial arrays
  - arrays and slices are bounds checked

Data Lifetimes

- small data implements Copy trait
- all arrays where element implements Copy trait
- everything else is moved
- assignment, parameter, result

Lifetime...

```
fn make_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
   // scope ends, `vec` is destroyed
    vec // transfer ownership to the caller
}
fn print_vec(vec: &Vec<i32>) -> Vec<i32> {
 // the `vec` parameter is part of this scope, hence owned by `print_vec`
 // the `vec` parameter is borrowed for this scope
    for i in vec.iter() {
        println!("{}", i)
    }
        // now, `vec` is deallocated
    vec // now, pass ownership back
        // now, borrow ends
}
fn use_vec() {
    let vec = make_vec(); // take ownership of the vector
    let vec = print_vec(&vec);  // pass ownershiplend access to `print_vec`
         // returned value is destroyed, as not used subsequently
    for i in vec.iter() {  //  Erroneously continue usinguse returned `vec`

        println!("{}", i * 2)
    }
    // scope ends, `vec` is destroyed
}
```

Lifetimes ...

- formally lifetimes of results are functions of parameter lifetimes
- `fn bar<'a>(x: &'a i32) -> &'a i32`
- lifetimes can sometimes be elided

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
```

Pointers

- `Box<T>` - heap allocated, moved
- `&T` and `&mut T` - references
- `*const T` and `*mut T` - C-like references - unsafe
- `Rc` - heap allocated immutable, clonable
- not sendable

Cells

- `Cell<T>` - mutable copy values
- `RefCell<T>` - mutable non-copy values
- usually used inside structs
- removes some of the simultaneous update guarantees
- not sendable

Closures or Lambdas

- capture context

```
fn ten_times<F>(f: F) where F: Fn(i32) {
    for index in 0..10 {
        f(index);
    }
}

let greeting = "hello";
ten_times(|j| println!("{}, {}", greeting, j));
```

Modules

- `mod` *modname* ;
- `mod` *modname* { · · · }

Package Manager

- cargo
- creates for library or executable

Macros

- hygenic, matching
  - zero or more items,
  - zero or more methods,
  - an expression,
  - a statement, or
  - a pattern.

```
let x: Vec<u32> = vec![1, 2, 3];
```

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
```

Synchronous Types

- Arc<T> - heap allocated, clonable, sendable
- Mutex<T> - heap allocated, locked
- RwLock<T> - heap allocated, locked - read lock (multiple)

Multi-processing

- channels
- mutex
- condition variables
- only types implementing Send can be sent or put in a Mutex
- means type system prevents data races

Unsafe blocks

- occasionally need to reach under the covers
- including building the Rust library
- module or block can be declared unsafe to bypass type system

Pragmatics

- predictable, high performance
- almost no run-time system required
- native compilation
- simple heap manager (no tracing or GC)
- array/slice bounds checking

Evaluation

- Simplicity
  - Size of the grammar
  - Type system
  - complexity of navigating modules/classes
- Orthogonality
  - number of special syntax forms
  - number of special datatypes
- Extensibility
  - functional
  - syntactically
  - defining literals
  - overloading

Zig

- another safe systems-programming language (also Odin, D, Nim, Jae)
- minimal, predictable overhead - even more than Rust
- statically typed, including array sizes
- no "undefined behaviour" a la C or C++ specs
- casts without `unsafe`
- 4 compilation models - Debug, ReleaseSafe, ReleaseSmall, ReleaseFast

History

- created by Andrew Kelly
- version 0.10 in March 2022
- Zig Foundation funding development of self-hosting 1.0

Paradigm

- imperative
- first-class types
- compile-time interpreter
- no accidental run-time costs
- no allocation without passing an allocator
- uses LLVM - dozens of targets (including wasm)

Syntax Rules

1. literals
   - numbers: (un)signed ints, floats `-17` `3.141592` (comptime - no default size) `@as(i56,42)`
   - characters: `'a'`
   - UTF-8 strings are u8 arrays: `"this isn't \"hard\"!"`
   - arrays: `[1,2,3]` `[5]u8{'h','e','l','l','o'}` `[_]u8{'w','o','r','l','d'}`
   - slice: part of an array - `a[0..]` `a[3..6]`
   - compile-time tuples (anonymous structs): `.{1,"abc"}`
2. names
   - upper/lower case, digits, underscore; case sensitive `@"any thing!"`
   - arguments to methods and blocks
   - declarations: `const` or `var` - must be initialized (even if `undefined`)
   - all variables must be used (even if _ `= variable`
   - snake_case for variables/parameters
   - camelCase for functions
   - PascalCase enum/struct
3. functions
   - `fn` name `(` p1 `:` t1 `...` `)` tr `{` expr `}`
   - `fn foo(x: i32) i32 { return x }`
   - `var x: fn(i32) i32 = foo;`

Parsing

- `const` is used for types, errors, “normal values”, modules
- modules are `struct`s lazily imported from files/build-environment
  `const expect = @import("std").testing.expect;`
- values designated `pub` are visible to importers
- code is only cursorily parsed unless it is needed - very fast compile; lazy error detection; circular imports
- generics are done with type arguments-to/return-from functions
- no exceptions - errors or error-unions are return types for functions

Statements - Conditionals

- `if`

```
const expect = @import("std").testing.expect;

test "if statement" {
    const a = true;
    var x: u16 = 0;
    if (a) {
        x += 1;
    } else {
        x += 2;
    }
    try expect(x == 1);
}
```

- `switch`

```
test "switch statement" {
    var x: i8 = 10;
    switch (x) {
        -1...1 => {
            x = -x;
        },
        10, 100 => {
            //special considerations must be made
            //when dividing signed integers
```

Statements - Loops

- `while`

```
test "while with continue expression" {
    var sum: u8 = 0;
    var i: u8 = 1;
    while (i <= 10) : (i += 1) {
        sum += i;
    }
    try expect(sum == 55);
}
```

- `while` with payload capture

```
var numbers_left: u32 = 4;
fn eventuallyNullSequence() ?u32 {
    if (numbers_left == 0) return null;
    numbers_left -= 1;
    return numbers_left;
}

test "while null capture" {
    var sum: u32 = 0;
    while (eventuallyNullSequence()) |value| {
        sum += value;
    }
    try expect(sum == 6); // 3 + 2 + 1
```

Optional types and Iterators

- struct type with a next function with an optional in its return type
- returns `null` if no more values

```
const text = "robust, optimal, reusable, maintainable, ";
var iter = std.mem.split(u8, text, ", ");
try expect(eql(u8, iter.next().?, "robust"));
try expect(eql(u8, iter.next().?, "optimal"));
try expect(eql(u8, iter.next().?, "reusable"));
try expect(eql(u8, iter.next().?, "maintainable"));
try expect(eql(u8, iter.next().?, ""));
try expect(iter.next() == null);
```

```
const text = "robust, optimal, reusable, maintainable, ";
var iter = std.mem.split(u8, text, ", ");
var count : usize = 0;
while (iter.next()) |str| {
    count += str.len;
}
try expectEqual(count,33);
```

Structs

- data containers
- created by `const` declaration, or by function
- contain constants, variables, functions

```
const Point = struct {
    x: i32,
    y: i32,
    const Self = @This();
    pub fn new(x: i32, y: i32) Self {
        return Point{.x = x, .y = y};
    }
    pub fn abs(self: Self) Self {
        return new(if (self.x>=0) self.x else -self.x,
                   if (self.y>=0) self.y else -self.y);
    }
};
```

Parametric Types

- functions can have types as parameters and can return types

```
pub fn Point_(comptime T: type) type {
    return struct {
        x: T,
        y: T,
        const Self = @This();
        pub fn new(x: T, y: T) Self {
            return .{.x = x, .y = y,};
        }
        pub fn abs(self: Self) Self {
            return new(if (self.x>=0) self.x else -self.x,
                       if (self.y>=0) self.y else -self.y);
        }
    };
}
test "parametric point" {
    const Point_i32 = Point_(i32);
    const p1 = Point_i32.new(3,-4);
    try expectEqual(p1.abs(),Point_i32.new(3,4));
}
```

Error handling

- no exceptions
- error returns
- must be handled - `catch` or `try`

Memory Safety

- much weaker than Rust
- null pointers
  - but have to be recognized and dealt with
- dangling pointers
  - `defer` statement allows release adjacent to allocation
- buffer overruns
  - careful pointer arithmetic
  - slices used for partial arrays
  - arrays and slices are bounds-checked

Modules

- just constant structs
- `@import("std") @import("heap.zig")`

Macros

- no macros
- achieve similar ends with comptime first-class types

Pragmatics

- predictable, high performance
- almost no run-time system required
- native compilation
- no automatic heap manager (no tracing or GC)
- array/slice bounds checking in safe/debug compilation modes
- undefined behaviour - detectable at compile time or run time

Evaluation

- Simplicity
  - Size of the grammar
  - Type system
  - complexity of navigating modules/classes
- Orthogonality
  - number of special syntax forms
  - number of special datatypes
- Extensibility
  - functional
  - syntactically
  - defining literals
  - overloading