rusty

Proposal

Yanlin Duan	Zhuo Kong	Emily Meng	Shiyu Qiu
yd2380	zk2202	ewm2136	sq2156
System Architect	Tester	Language Guru	Manager

Motivation

C is a powerful and widely used language. However, sometimes C can be tricky - we've all had our share of headaches with segmentation faults and data races. When deciding the fundamental goal of our project, we knew that we wanted our language to be easy to use, but still flexible with a high degree of control over the computer. Most notably, we wanted our language to be safe, both in terms of memory and concurrency. Thus, we were inspired by Rust, a systems programming language that prevents segmentation faults and manages data concurrency. We hope to build our own language that captures the essence of Rust, a "rusty" version of Rust, if you will. Furthermore, we wanted to incorporate useful, key ideas from other popular languages such as object-oriented and functional programming concepts. We present Rusty, a safer systems programming language that combines essential features from Rust with some of the conveniences of higher level programming languages.

Description

Rusty aims to be a version of the Rust programming language, keeping its key features of safety in memory and concurrency but shedding some of the other extraneous features to try and retain the speed of C as much as possible. It can be used as a general purpose programming language with a focus on helping the programmer avoid many of the common memory safety-related pitfalls when programming in C, such as segmentation faults or null pointers, as well as providing useful features often found in higher level programming languages, such as automatic memory management and pattern matching.

Rusty will automatically clean up the memory associated with variables on both the stack and heap when the variable goes out of scope. There will only be one binding per variable, so when memory is moved to a new variable, the old variable is no longer associated with that memory. Since passing memory back and forth between variables is a pain, Rusty will use a Borrow feature implemented with references to variables, essentially borrowing ownership of that memory and will not deallocate after going out of scope. There will be two types of references, denoted by & and &mut. The former indicates the borrowed memory is immutable, while the latter allows a user to change the memory. For any given variable, only one of the two types of reference will be allowed at any time. There can be multiple & references or only one &mut reference to keep concurrency and avoid data races. In the case of primitives, they will all have a Copy feature that automatically creates a complete copy of memory for the new variable, allowing the old variable to keep its value. Copy may be implemented by the user for user defined types as well.

C is very powerful, but with great power there is great responsibility - Rusty will help programmers manage some of the intricacies and care needed when using C, so they can spend more effort on the design of their programs rather than worry about accidentally dereferencing a null pointer or accessing invalid memory. The days of segmentation faults are over. Target users include systems programmers and people who have suffered enough while learning the semantics of C and can fully appreciate the features of Rusty.

Features

- Strong statically typed
- No null referencing pointers
- Automatic memory management
- Borrow feature, immutable and mutable references
- Concurrency, avoid data races
- Functional programming pattern matching
- Lenient keyword matching (eg. “True” and “true” are the same)

Syntax

Primitive Data types

Type	**Description**
char	Basic character, same as in C
int, double, float	Basic numeric primitive types, same as in C
bool	Boolean, value of true or false
array	Fixed size list of elements of same type, immutable
str	String literal
tuples	Finite ordered list

Operators

Operator Name	**Syntax**
Logical operators	&&, ||, !
Arithmetic	+, -, *, /, %
Bitwise operators	&, |, !, ^, <<, >>
Comparison	<, <=, ==, !=, >=, >
String concatenation	+, “hello” + “world”
Assignment	=, let y=x;

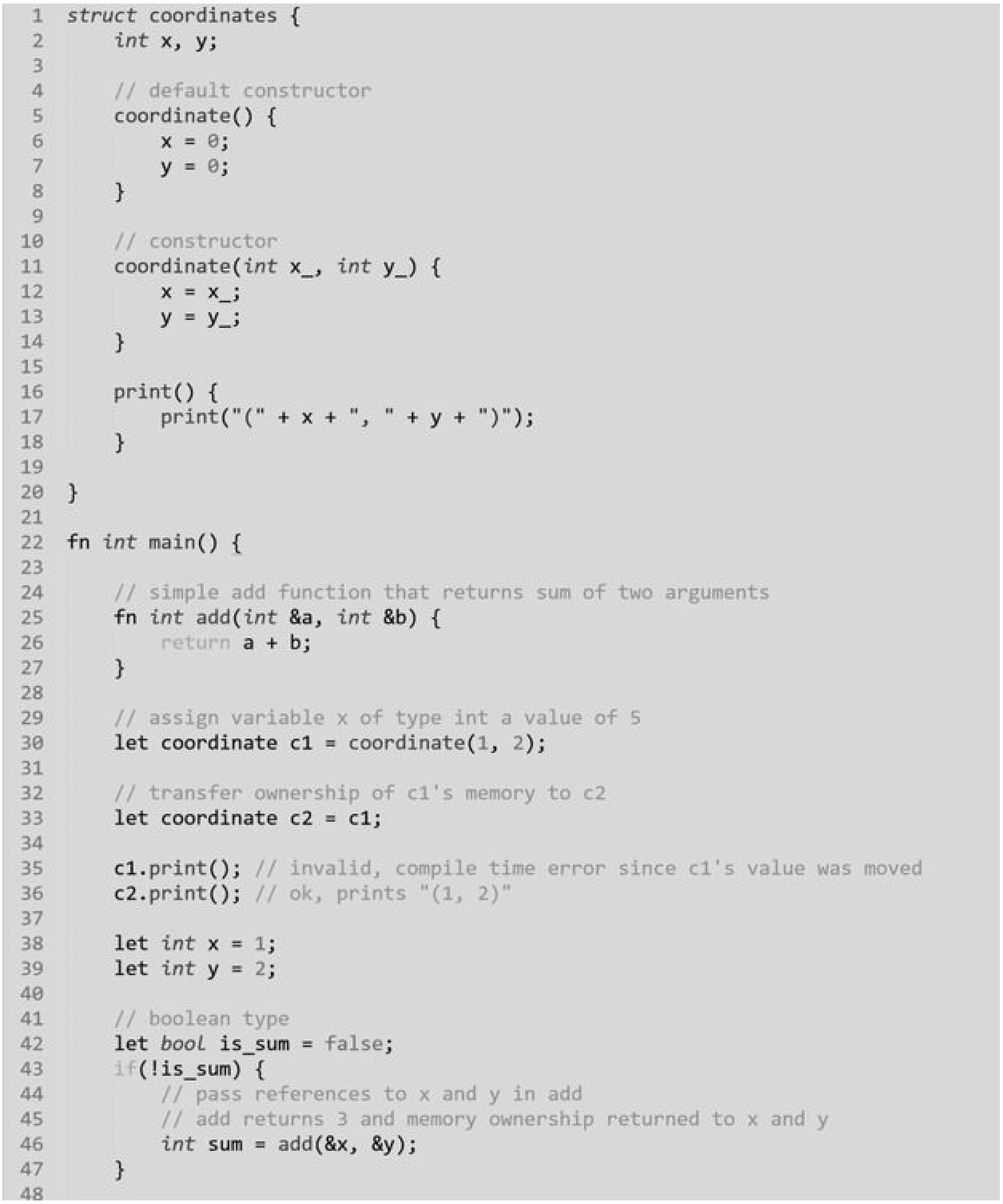
Immutable reference (variable borrowed)	type& x
Mutable reference (variable borrowed)	type mut x when declared type &mut x when borrowed
Reference (variable borrowing)	type* x
Pattern binding	@ (ident @ pat)
Ignored pattern binding	_
Member access	‘.’ , struct.member

Keywords

Keywords	**Description**	**Syntax**
if/else/else if	Conditional expression	if(condition)
while/for	Conditional/iterative loop	while(condition), for(condition)
fn	Function	fn return_type name (type1 name1, ...)
let	Bind value to variable	let type name = value;
return	Return from function	return expr;
struct	Structure definition	struct name

match	Pattern matching	let num = match x{
		1=>”one”,
		2=>”two”,
		3=>”three”
		_=>”something else”,
		};

Code Examples



```
struct coordinates {
    int x, y;

    // default constructor
    coordinate() {
        x = 0;
        y = 0;
    }

    // constructor
    coordinate(int x_, int y_) {
        x = x_;
        y = y_;
    }

    print() {
        print("(" + x + ", " + y + ")");
    }

}

fn int main() {

    // simple add function that returns sum of two arguments
    fn int add(int &a, int &b) {
        return a + b;
    }

    // assign variable x of type int a value of 5
    let coordinate c1 = coordinate(1, 2);

    // transfer ownership of c1's memory to c2
    let coordinate c2 = c1;

    c1.print(); // invalid, compile time error since c1's value was moved
    c2.print(); // ok, prints "(1, 2)"

    let int x = 1;
    let int y = 2;

    // boolean type
    let bool is_sum = false;
    if(!is_sum) {
        // pass references to x and y in add
        // add returns 3 and memory ownership returned to x and y
        int sum = add(&x, &y);
    }

    // x and y retain values since above code did not transfer ownership
    // but only passed references
    print(x + " and " + y); // "1 and 2"

    // c3 is mutable coordinate
    let coordinate mut c3 = coordinate(0, 0);

    // boolean type
    let bool pos_coor = true;

    if(pos_coor) {
        // c4 is borrowing
        let coordinate* c4 = mut& c3;
        // c4.x is pointing to c3.x
        c4.x -= 1; // c3.x is now -1, ok since c3 is mutable
        c4.y -= 1;
    }

    c3.print(); // ok, prints "(-1, -1)"
    c4.print(); // invalid, c4 gone since out of scope
    return 0;
}
```

References

https://doc.rust-lang.org/book/README.html
https://docs.python.org/3/
http://scratch-lang.notimetoplay.org/index.html
http://www1.cs.columbia.edu/~sedwards/classes/2016/4115-spring/proposals/JSJS.pdf
http://www1.cs.columbia.edu/~sedwards/classes/2015/4115-fall/proposals/Dice.pdf
http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/proposals/Stitch.pdf