GRAMMATECH

Putting a Roof over your Head

Object-Oriented Programming in Rust

Nathaniel Berch, Paul Rodriguez, Thomas Wahl

May 8, 2024

This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. HR001122C0025. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Rust: Safe and Efficient System-Level Programming

- **Safe:** memory-access interface defined via *ownership*
- **Efficient:** auto-deallocation (no garbage collector)
- **Modern and "in vogue":**
  - trying-to-be-helpful compiler & build system
  - active user community

GRAMMATECH

Rust: Beloved Features and Non-Features

Object-Oriented Programming

- **Objects:** struct instances, with encapsulated data and methods
- **Information hiding:** private data fields in structs

- **Inheritance:** one struct refines (specializes) data & methods of another
- **Exception handling:** errors/edge cases handled away from mainstream code

Absence of these features steepens the *already steep* learning curve specifically for C++/Java programmers.

GRAMMATECH

Goal of This Work

Roof: Rust with Object-Oriented Features

- Rust-like language with 1st-class support for exceptions and inheritance
- Transpilable into genuine Rust. (No extra runtime support!)

GRAMMATECH

SOME DETAILS

Exception Handling Primer: throw, try, catch!

A Roof program

```
fn f(x: u32) {
    if x > 100 { throw!("Too big!"); }
    ...;
}

fn g(x: u32) {
    f(x);
}

fn main() {
    try! {
        println!("Potentially throwing call");
        g(42);
    }
    catch! { e => { println!("{}", e); } }
}
```

Intended meaning:

- **`throw!`** : generate exception, to be passed up the call stack in search for handler
- **`try!`** + **`catch!`** :
  1. Execute `try` code.
  2. a) If exception is encountered, pass control to `catch` block.
     b) Otherwise skip `catch` block.

(C++ exception model)

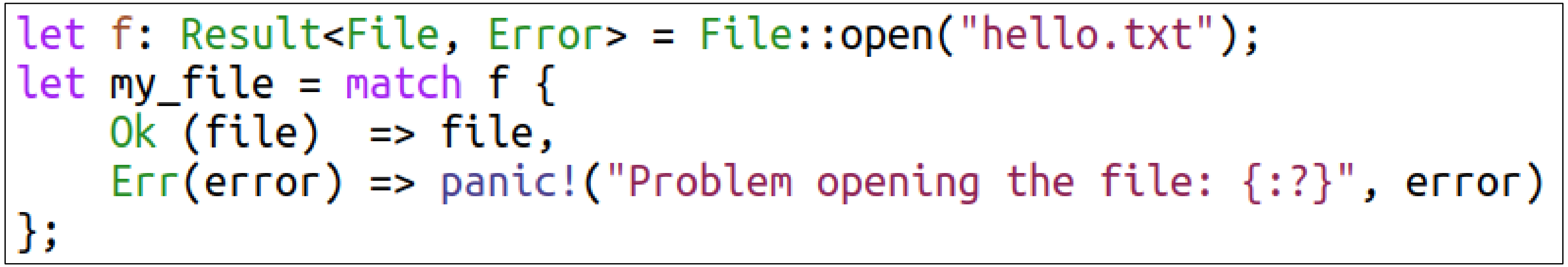
GRAMMATECH

Existing Rust-Style Error Handling

Rust has a “union” type

```
enum Result<T,E> { Ok(T), Err(E) }
```

= a two-variant type encapsulating “ok” and error results:



```
let f: Result<File, Error> = File::open("hello.txt");
let my_file = match f {
    Ok (file)  => file,
    Err(error) => panic!("Problem opening the file: {:?}", error)
};
```

GRAMMATECH

Throwing and Propagating Exceptions

Idea: Treat exceptions as part of a function's return value.

1.a Throwing:

- Perform *may-throw* analysis
- Change all functions that *may throw* to return `Result<T,str>`
- `return x` becomes `return Ok(x)`
  `throw!("Error!")` becomes `return Err("Error!")`

1.b Propagation: change calls to *may-throw* `f()` **outside** a `try` block:

`f()` →

```
match f() {
    Ok (s) => s,
    Err(t) => return Err(t)
}
```

Rust helps us out here:
`f()` → `f()?`

Trying and Catching Exceptions

Idea: Abstract try block into a function.

2.a Wrap try code into *helper function*.

2.b Call the helper.

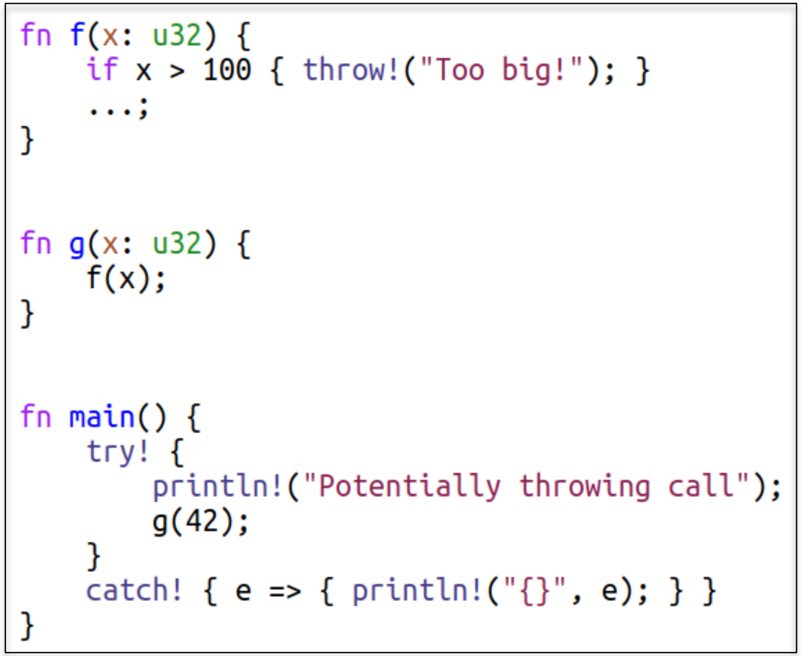
2.c Pattern-match on the result.

```
try!   { stmts1; }
catch! { e => stmts2; }
```

```
let helper = || -> Result<(), str> {
    stmts1;
    Ok(())
};
match helper() {
    Ok (_) => (),
    Err(e) => stmts2
}
```

GRAMMATECH

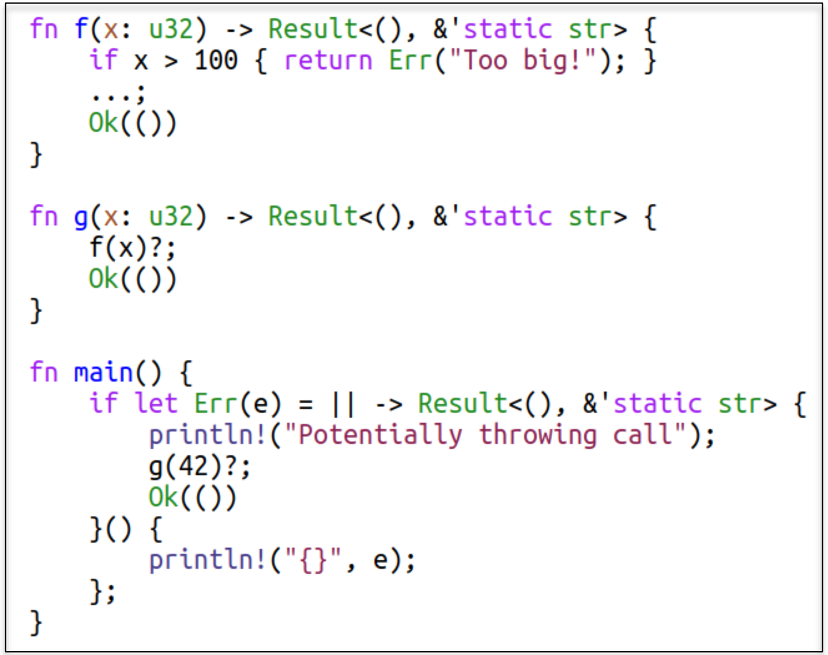
Roof to Rust Transpilation: Result



```
fn f(x: u32) {
    if x > 100 { throw!("Too big!"); }
    ...;
}


fn g(x: u32) {
    f(x);
}


fn main() {
    try! {
        println!("Potentially throwing call");
        g(42);
    }
    catch! { e => { println!("{}", e); } }
}
```



```
fn f(x: u32) -> Result<(), &'static str> {
    if x > 100 { return Err("Too big!"); }
    ...;
    Ok(())
}

fn g(x: u32) -> Result<(), &'static str> {
    f(x)?;
    Ok(())
}

fn main() {
    if let Err(e) = || -> Result<(), &'static str> {
        println!("Potentially throwing call");
        g(42)?;
        Ok(())
    }() {
        println!("{}", e);
    };
}
```

GRAMMATECH

DISCUSSION

GRAMMATECH

Trade-Offs of our Exception Handling

Our Exception system is currently binary:

A function either throws or it doesn't throw.

<u>In reality:</u>

- Exception types form hierarchies.
- Binary matching `Ok(_)` vs. `Err(e)` should really be (sub-)type checking

  → we need *inheritance*.

Inheritance in Rust

1. Simple data and method inheritance:

- Turn “is a" relationship into “has a”:

```
class Tree: public Plant → class Tree { Plant p; … }
```

- Works for multiple inheritance, too

2. Virtual methods: can be implemented using Rust’s *trait* mechanism:

- Capture virtual methods in a trait (function body = default implementation)
- Wrap a `Box` pointer around variables of base type: “dynamic dispatch”

```
Box<dyn Plant>
```

GRAMMATECH

Summary: OO Programming in Rust

1. Can we do it?
   - “OOP” means different things to different people
   - Core OOP concepts can be implemented fairly naturally in Rust

2. Do we need it?
   - Rust certainly has its own design patterns.
   - OOP comes with a baggage of 50+ years of history (“legacy concept”)
   - New Rust programmers with C++/Java background will appreciate

GRAMMATECH