Using rust with bitbake and meta-rust

Steven Walter

Yocto Project Summit, May 2021

Who am I?

- Based in Central Kentucky, USA
- Senior Firmware Developer at Lexmark International
- bitbake user since 2014
- rust user and advocate since pre-1.0

What is rust?

Rust & Go Similarities

- Modern, high-level languages
- Strongly-typed and memory safe
- Compiled to machine code

Rust vs. Go

- Go: garbage collected (Rust isn't)
  - Not ideal for embedded / bare-metal code
- Go: has null pointers (Rust doesn't)
  - Rust references must point at a valid object
- Rust has algebraic data types

Algebraic Data Types

- Naively implemented data type

```
enum Fruit {
    Apple,
    Banana,
    Strawberry,
};

struct FruitDescription {
    enum Fruit type;
    int apple_diameter; // only valid when type == Apple
    int banana_weight; // only valid when type == Banana
};
```

Algebraic Data Types

- Space-optimized

```
struct FruitDescription {
    enum Fruit type;
    union {
        int apple_diameter; // only valid when type == Apple
        int banana_weight; // only valid when type == Banana
    };
};
```

Rust enum

- Similar to the tagged-union pattern
- Each enum variant can have associated data
- Pattern matching for deconstruction

Rust enum

- Equivalent Rust code

```
enum Fruit {
    Apple(u32 /* diameter */),
    Banana(u32 /* weight */),
    Strawberry,
}


fn describe_fruit(f: Fruit) {
    match f {
        Fruit::Apple(diameter) => println!("Apple with diameter {}", diameter),
        Fruit::Banana(weight) => println!("Banana weighing {}", weight),
        Fruit::Strawberry => println!("Regular old strawberry"),
    }
}
```

Rust enum

- Simple use of pattern matching

```
enum Result {
    Ok(String),
    Err(u32)
}
[...]
let result = fn_that_can_fail();
match result {
    Ok(x) => println!("The successful result was {}", x),
    Err(e) => println!("The operation failed, error code: {}", e)
}
```

No Undefined Behavior in Rust

- No type confusion
- No memory errors
  - Buffer overflow
  - Use-after-free
- No data races

Rust is strongly typed

- Unsafe type conversions forbidden

```
fn add(x: &i32) -> i32 {
    *x + 5
}

fn main () {
    let s = "Hello";
    let y: &i32 = s as &i32;
    add(&y);
}
```

Rust is strongly typed

- Unsafe type conversions forbidden

```
error[E0605]: non-primitive cast: `&str` as `&i32`
 --> src/main.rs:7:19
  |
7 |     let y: &i32 = s as &i32;
  |                   ^^^^^^^^^ an `as` expression can only
be used to convert between primitive types or to coerce to
a specific trait object
```

No buffer overflow

- Array lengths are run-time checked

```
fn get_second(x: &[i32]) -> i32 {
    x[1]
}

fn main () {
    let array = [1];
    get_second(&array);
}
```

No buffer overflow

- Array lengths are run-time checked

```
thread 'main' panicked at 'index out of bounds: the len is
1 but the index is 1', src/main.rs:2:5
```

No use-after-free

```
fn main () {
    let five = 5;
    let mut ptr = &five;
    {
        let six = 6;
        ptr = &six;
    }
    println!("{}", *ptr);
}
```

No use-after-free

- Amazingly good error messages

```
error[E0597]: `six` does not live long enough
 --> src/main.rs:6:15
  |
6 |         ptr = &six;
  |               ^^^^ borrowed value does not live long enough
7 |     }
  |     - `six` dropped here while still borrowed
8 |     println!("{}", *ptr);
  |                    ---- borrow later used here
```

No iterator invalidation

```
fn main () {
    let mut array = vec![1,2,3,4];
    for i in array.iter_mut() {
        *i +=1 ;
        if *i == 5 {
            array.remove(5);
        }
    }
}
```

No iterator invalidation

```
error[E0499]: cannot borrow `array` as mutable more than once at a time
 --> src/main.rs:7:13
  |
4 |     for i in array.iter_mut() {
  |              ----------------
  |              |
  |              first mutable borrow occurs here
  |              first borrow later used here
...
7 |             array.remove(5);
  |             ^^^^^ second mutable borrow occurs here
```

No data races

- Locks are containers

```
use std::sync::Mutex;

fn main () {
    let x = "No races!".to_string();
    let lock = Mutex::new(x);
    println!("{}", x);
}
```

No data races

- Locks are containers

```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:6:20
  |
4 |     let x = "No races!".to_string();
  |         - move occurs because `x` has type `String`,
which does not implement the `Copy` trait
5 |     let lock = Mutex::new(x);
  |                           - value moved here
6 |     println!("{}", x);
  |                    ^ value borrowed here after move
```

No data races

- Contents only accessible via lock guard

```
use std::sync::Mutex;

fn main () {
    let x = "No races!".to_string();
    let mutex = Mutex::new(x);
    let guard = mutex.lock().unwrap();
    println!("{}", *guard);
}
```

Safe vs. Unsafe rust

- Rust is “safe” by default
- Not everything can be proven by the compiler
- Potentially unsafe code goes in “unsafe” blocks

Why use rust?

- High performance
- Low overhead
- Safer than C/C++
- Rich tool ecosystem
  - cargo, crates.io, rustfmt, clippy

Cargo example

Why not use rust?

- Language is still rapidly developing
  - (But excellent backwards compatibility)
- Code size (relative to C/C++)
- Faster iteration with e.g. Python
- Not ideal for all domains

Using rust with bitbake

meta-rust

- Provides rust compiler and tooling
- Makes cargo and bitbake play nicely
- Used by meta-iotedge
- Working on integration with meta-oe

cargo-bitbake example

cargo-bitbake

- Primarily for generating executables
- Cargo builds all dependencies every time, statically linked
- Standard library can be a shared object

Thanks!

- **meta-rust maintainers:**
  - **Cody Schafer**
  - **Doug Goldstein**
  - **Tyler Hall**
  - **Paul Osborne**
  - **Derek Straka**
- **Thanks to all who have contributed!**






yocto PROJECT

THE LINUX FOUNDATION