IN3040

Types for Concurrency and the Rust Ownership System

Eduard Kamburjan

25. 11. 2022



Beyond Data Types

- Data types (OO, records, ints, etc.) describe sets of values
- Behavioral types describe *allowed behavior* using a similar underlying formalism
  - Linear types: this value should be used once
  - Security types: this value should not be exposed

Uniqueness Types

A uniqueness type system ensures that every value (channel etc.) has at most one usable reference pointing to it.

- Same basic idea as data types: connect syntax (variable) with semantics (value)
- Specification and analysis of intent

Uniqueness Types

Every value is associated with a single variable

```
C i = ...; //type of i : C
C j = i;  //uses up i, type of i : Used
C k = i;  //error, Used is not a subtype of C
```

Calls are considered creating a new, external reference

```
fun (Car param) = ...
var car = ...
drive(car); //passes reference out, uses up car
drive(car); //error
```

```
var car = ...
car = drive(car); //passes reference out, uses up car
                  //gets new reference!
drive(car); //allowed
```

Uniqueness and Threads

In a concurrent setting, uniqueness types are used to ensure that only one thread has access to a shared resource.
Trivially removes data races, as only one thread can modify an any time – all other references are considered used up.

```
var car = ...
thread::spawn( car.wheels = 5 ); //passes reference out
                                 //also uses up car!
car.wheels = 6;          //error
```

Types for Concurrent Systems

- Tool support external to language and standard library
  - Developers mixing libraries lose guarantees
  - Maintenance of libraries becomes a problem

Practical Types for Concurrent Systems

Rust, and to a smaller degree Go, have practical implementations

- Motivated by memory safety and concurrency
- Some types (Session types) still external, but Rust library sensible

Remark

Ownership types (introduced by David Clarke) are types for object oriented languages with similar basic ideas. Rust ownership system is not based on Clarke's ownership types.

Resource Allocation Is Initialization (RAII)

- Memory management for local (=stack) instances
- Memory used by class allocated by constructor
- Memory deallocated by destructor
- No explicit deallocation needed, destructor called upon leaving the stack scope

```
class C { //C++ example
  public:
    int* p;
    C() { p = new int[4]; }
    ~C() { delete[] data; }
    ...
};

void f() {
  C c();
  c.f();
}
```

Region/Scope Based Memory Management (RBMM)

- RAII is mostly used to refer to OO, RBMM is more general
- Associate lexically-scoped part of the program with a *region* in the heap
- Region deallocated once scope exited
- Type checker ensures that no external pointers into region exist

Shared Themes

Uniqueness types, RBMM and alias analyses relate values, their lifetime at runtime and the syntactic structure of the program.

- Keep track of number of possible (types) references to reason about concurrent operations
- Prevent general errors from faulty memory management

Memory management in Rust

Rust combines many ideas to guarantee memory and thread-safety, as well as static memory management without garbage collection

Ownership is how Rust manages memory

Ownership In Rust

- Each value (a String, i32, Vec, etc.) is owned by a single variable, called *owner*
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership

```
fn main() {
    let a = vec![1, 2, 3];    // a vector
    let b = a;                // move: ‘a‘ can no longer be used
    println!("{0} {1}", a[0], b[0]);    //error : borrow of moved value : 'a'
}
```

- Reassignment of ownership (as in let b = a) is a move
- Once ownership has been given away, a variable can no longer be used
- a is “used up” and therefore unusable
- Works for values with copy trait, and literals

```
fn main() {
    let a = 1:
    let b = a;                // not a move: ‘a‘ is copied!
    println!("{1} {2}", a, b);    //works
}
```

Passing Ownership

Passing a value also passes ownership of the value

```
fn make_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
    vec    // transfer ownership to the caller }

fn use_vec() {
 let vec = make_vec();  // take ownership of the vector
 print_vec(vec);        // pass ownership to print_vec
}

fn print_vec(vec: Vec<i32>) {
 // the vec is owned by print_vec
    for i in vec.iter() {
    println!("{}", i) }
  // now, vec is deallocated }
```

Passing Ownership

```
fn use_vec() {
    let vec = make_vec(); // take ownership of vector
    print_vec(vec);  //pass ownership to print_vec‘

    for i in vec.iter() { ERROR: // continue using ‘vec‘
        println!("{}", i * 2)
    }
}
```

- Ownership is not transferred again by print_vec, vec it destroyed here.
- Trying to use the vec again gives an error
- The vector has already been deallocated at this point!

Mutability

```
fn main() {
    let numbers = vec![1, 2, 3];

    numbers.push(4);  // ERROR: cannot borrow as mutable

    println!();
}
```

- Rust distinguishes between references/variables for reading and writing
- Just owning a value does not enable to change it
- Not relevant for literals

Mutability

```
fn main() {
let mut numbers = vec![1, 2, 3];

numbers.push(4);    //mutable Vec supports push

println!("numbers: {:?}", numbers);
}
```

- Capability of changing the state of a value is known as *mutability*
- Mutability constrains your ability to borrow references
- Variables own *immutable* values by default. We can override this behavior by preceding a variable with the mut keyword.

Borrowing Vs Ownership : same example cont'd

```
fn use_vec() {
  let vec = make_vec(); // take ownership of vector
  print_vec(vec);  //pass ownership to print_vec‘
  for i in vec.iter() { //ERROR: continue using ‘vec‘
    println!("{}", i * 2)
  }
}
```

You can *borrow* the access to functions you call

- Grant print_vec temporary access to the vector, and then continue using the vector afterwards
- To borrow a value, you make a reference to it

```
fn use_vec() {
  let vec = make_vec();// take ownership of vector
  print_vec(&vec);   // lend access to ‘print_vec‘
  for i in vec.iter() {  // continue using ‘vec‘
    println!("{}", i * 2)}
 // vec is destroyed here
}
```

Referencing in Rust

References come in two flavors:

- Immutable references `&T`, which allow sharing but not mutation. There can be multiple `&T` references to the same value simultaneously, but the value cannot be mutated while those references are active.

- Mutable references `&mut T`, which allow mutation but not sharing. If there is an `&mut T` reference to a value, there can be no other active references at that time, but the value can be mutated.

Rust checks these rules at compile time; borrowing has no runtime overhead.

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime not respected due to scoping

```
fn main() {
  let ref;                          //---+
  {                                 //   |
    let vec = vec![1, 2, 3];        //-+ |
    ref = &vec;                     // | |
  }                                 //-+ |
  println!("{}", (*ref)[0]);        //   |
}                                   //---+
```

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime respected

```
fn main() {
  let vec = vec![1, 2, 3];     //---+
  let ref = &vec;              //-+ |
  println!("{}", (*ref)[0]);   // | |
}                              //-+-+
```

A reference to a value cannot outlive the owner

```
let v = vec![1, 2];
let x=&v[0];
let v2=v;
let y = *x +1 // ERROR - x refers to v, but v is dead
```

A value can have one mutable reference or many immutable references

```
let mut v = vec![1, 2];
let x=&v[0];    //immutable borrow here
Vec::push(&mut v, 3); // ERROR: mutable borrow here
let y = *x +1; // x still alive
```

Data Race

A data race occurs on a value in memory if

- two or more threads are concurrency accessing memory,
- one or more of them is a write, and
- one or more of them is unsynchronised.

Data races are prevented by the ownership system/borrow checker, since we are unable to alias a mutable reference.

However Rust does not prevent general race conditions

- Since, our hardware, OS and other programs might be Racy
- Still, a race condition cannot violate memory safety in a Rust program on its own
- Only in conjunction with some other unsafe code can a race condition actually violate memory safety

Rust and Channels

- Rust uses transmitter(tx) and receiver(rx) for channel communication to send and receive over channel respectively.
- We create a new channel using the mpsc::channel function; *mpsc* stands for multiple producer, single consumer.
- Channel can have multiple sending ends producing values but only one receiving end consuming values

Message passing to transfer data between Threads

```
fn main() {
    let (tx : Sender<String>, rx : Reader<String>)
            = mpsc::channel();
    thread::spawn(move  {
        let val = String::from("Sending data");
        tx.send(val).unwrap();
    });
}
```

- Using move moves ownership of tx to new thread
- Thread must own transmitter to send messages on channel
- *send* method returns a Result<T, E> type and *unwrap* to panic in case of an error (send has nowhere to send).

Message passing to transfer data between Threads

```
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("Sending data");
        tx.send(val).unwrap();
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

- *recv* will block the main threads execution and wait until a value is sent down the channel
- Once a value is sent, recv will return it in a Result<T, E>.

Creating Multiple Producers by Cloning

```
let tx1 = tx.clone();
thread::spawn(move || {
    let val = ...;
    tx1.send(val).unwrap();
});
thread::spawn(move || {
    let val = ...;
    tx.send(val).unwrap();
});
for received in rx {
        println!("Got: {}", received); }
```

- Before creating the first spawned thread, we call clone on the transmitter
- Gives us a new transmitter we can pass to the first spawned thread.
- We pass the original transmitter to a second spawned thread which gives us two threads, each sending different messages to the one receiver.

Wrap-Up

- Beyond data types: types for behavior
- Uniqueness Types and Region-Based Memory Management
- Practical types for memory and concurrency management in Rust
- More details on types in concurrency (and more): IN5170 - models of concurrency