Part 3: Type Systems and Concurrency

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

November 4, 2023

University of Oslo

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}$ `int`	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	$\texttt{chan}_{?1,!1}$ `int`	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	$\texttt{chan}_{!.?.0+?.!.0}$ `int`	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants
Binary ST	$\texttt{chan !int.?string.0}$	Implicit at declaration	Explicit in spec.	No DL on single channels	Medium effort Consistency checked	Complex protocols with branching between 2 participants
Multi-Party ST	$\texttt{chan}\ p \to q : \texttt{int.0}$	Extra mechanism	Explicit in spec.	No DL on single channels	Considerate Consistency checked	Complex protocols with branching between n participants

Uniqueness Types

Linear Types

So far, we have used linear types for channels, and limited the use of *reading* and *writing*. Passing the channel around was no problem, but required to split the type environment.

- What about other kinds of usages?
- What about limiting the use of *passing*?

Uniqueness Types

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

- Sometimes used interchangeably with linear types, when every use of a variable is considered creating a new reference

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

```
func drive(param T) = ...
car := ...
drive(car); //passes reference out, uses up car
drive(car); //error
```

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

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

Background

How to use types and concurrency models in development?

External Tool Support

- Developers mixing libraries lose guarantees
- Studies in Scala show that developers very often mix libraries

  [Why do scala developers mix the actor model with other concurrency models?, Tasharofi et al., 2013]
- Maintenance of libraries becomes a problem

Practical Types for Concurrent Systems

Rust and Go both focus on practical implementations of types and concurrency patterns

- Motivated by memory safety and concurrency: ownership, goroutine, channels
- Session types still external, but Rust library integrates with type system

Connecting Syntax and Semantics

There are several ideas to connect syntax and semantics for memory management.

- Linear types, RAII, RBMM, etc.
- All face similar issues

Aliasing

Aliasing occurs if multiple references to one value/object exist

- Makes reasoning about the program more difficult
- Especially in concurrency: is there another *active* reference?
- Separates (semantic) value from (syntactic) variable

Excursus for Object-Orientation: Ownership Types

- Each object has one owner to enforce hierarchy of accesses
- Basis for further analyses, have elegant formalization as type system
- *Name Warning:* Rust's ownership system is not based on Ownership types.

Connecting Syntax and Semantics

Resource Allocation Is Initialization (RAII)

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

```
class C { //C++ example
  int* p;
  C() { p = new int[4]; }
  ~C() { delete[] data; }
  ...};
void f() {
  C v();
  v.f();
} //v goes out of scope, gets deallocated and calls the destructor
```

Connecting Syntax and Semantics

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

Linear types, uniqueness types, RBMM and alias analyses relate values, their lifetime at run-time 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

Ownership

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, 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, i.e., memory will be deallocated.

Ownership

- Reassignment of ownership *(as in let b = a)* is a move
- Affinity is considered with respect to moves
- Once ownership has been given away, a variable can no longer be used
- *a* is “used up” and therefore unusable
- Values with copy trait and literals are not moved, but copied

Rust

```
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'
}
```

Ownership

- Reassignment of ownership *(as in let b = a)* is a move
- Affinity is considered with respect to moves
- Once ownership has been given away, a variable can no longer be used
- *a* is “used up” and therefore unusable
- Values with copy trait and literals are not moved, but copied

Rust

```
fn main() {
let a = 1:
let b = a;                    // not a move: ‘a‘ is copied!
println!("{0}␣{1}", a, b); //works
}
```

Passing Ownership

Passing a value also passes ownership of the value

Rust

```
fn make_vec() -> Vec<i32> {
  let mut vec = Vec::new();
  vec.push(1);
  vec  // transfer ownership back 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>) { // vec is owned by print_vec
  for i in vec.iter()
    println!("{}", i)
 }  // now, vec is deallocated
```

Passing Ownership

Rust

```
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*
- More than just “discipline”: the vector has already been *deallocated* at this point!

Mutability, Borrowing, References

Mutability

Do we need the **mut** modifier?

Mutability

Do we need the **mut** modifier?

Rust

```
fn main() {
    let numbers = vec![1, 2, 3];
    numbers.push(4); // ERROR: cannot borrow as mutable
    println!(...);
};
```

Mutability

Do we need the **mut** modifier?

Rust

```
fn main() {
  let mut numbers = vec![1, 2, 3];
  numbers.push(4); // no error
  println!(...);
};
```

Mutability

Do we need the **mut** modifier?

Rust

```
fn main() {
  let mut numbers = vec![1, 2, 3];
  numbers.push(4); // no error
  println!(...);
};
```

- Rust distinguished between references for reading and writing (for non-literals)
- Just owning a value does not enable one to change it
- 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

Rust

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

Borrowing Vs Ownership : same example cont'd

Rust

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

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

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

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Lifetime

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Rust

```
fn main() {
  let rf;                        //---+ Lifetime of ref
  {                              //   |
    let vec = vec![1, 2, 3];     //-+ | Lifetime of vec and value
    rf = &vec;                   // | | ref borrows read access
  }                              //-+ | vec goes out-of-scope, value deallocated
  println!("{}", (*rf)[0]);      //   | access invalid!
}                                //---+
```

Lifetime

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Rust

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

Referencing in Rust

A reference to a value cannot outlive the owner

Rust

```
let v = vec![1 , 2];
let x=&v[0];
let v2=v;        // Owner changes from v to v2!
let y = *x + 1 // ERROR - x refers to v, but v is not an owner!
```

A value can have one mutable reference or many immutable references

Rust

```
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;           // removing this line fixes the example!
```

Data Races and Race Conditions in Rust

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

Excursus: Race condition in Rust

Rust

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0)); //ARC manages shared ownership
let other = idx.clone();                  //shared ownership btw. idx and other

thread::spawn(move || {
  other.fetch_add(10, Ordering::SeqCst);
}); // we can mutate idx since its atomic it cannot cause a Data Race.

if idx.load(Ordering::SeqCst) < data.len() {
  unsafe{ println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst))); }
}
```

Channels

Rust and Channels

- Rust uses *transmitter(tx)* and *reciever(rx)* for channel communication
- Channels created with *mpsc::channel* (MPSC = multiple producer, single consumer)
- Channel can have multiple sending endpoint, but only one receiving endpoint

Rust

```
fn main() {
  let (tx, rx) = mpsc::channel(); // two channel endpoints
  thread::spawn(move || { // || creates closure
    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

Rust

```
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>*.

Message passing : Ownership Transfer

Rust

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

- We try to print *val* after we sent it down the channel via *tx.send*
- Results in *error*, since once the value has been sent to another thread, that thread(i.e.,function *recv*) takes the ownership
- Same mechanism as with function valls

Creating Multiple Producers by Cloning

Rust

```
let tx1 = tx.clone();
thread::spawn(move || {
  let vals = vec![ ... ];
  for val in vals {
    tx1.send(val).unwrap();
    thread::sleep(Duration::from_secs(1));}});

thread::spawn(move || {
  let vals = vec![ ... ];
  for val in vals {
    tx.send(val).unwrap();
    thread::sleep(Duration::from_secs(1));}});

for received in rx {
  println!("Got: {}", received); }
```

Creating Multiple Producers by Cloning

- 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.

Rust and Session Types

We have seen safety for shared memory so far.

session_types

A rust crate (=package/library) for binary session types.
Equipped with numerous macros.

(see auxiliary material session0.rs and session1.rs)