COMP 737011 - Memory Safety and Programming Language Design

Lecture 8: Rust Type System

徐 辉
xuh@fudan.edu.cn

Outline

1. Type System
2. Types in Rust
3. Generic and Trait
4. Special Types

1. Type System

Type System

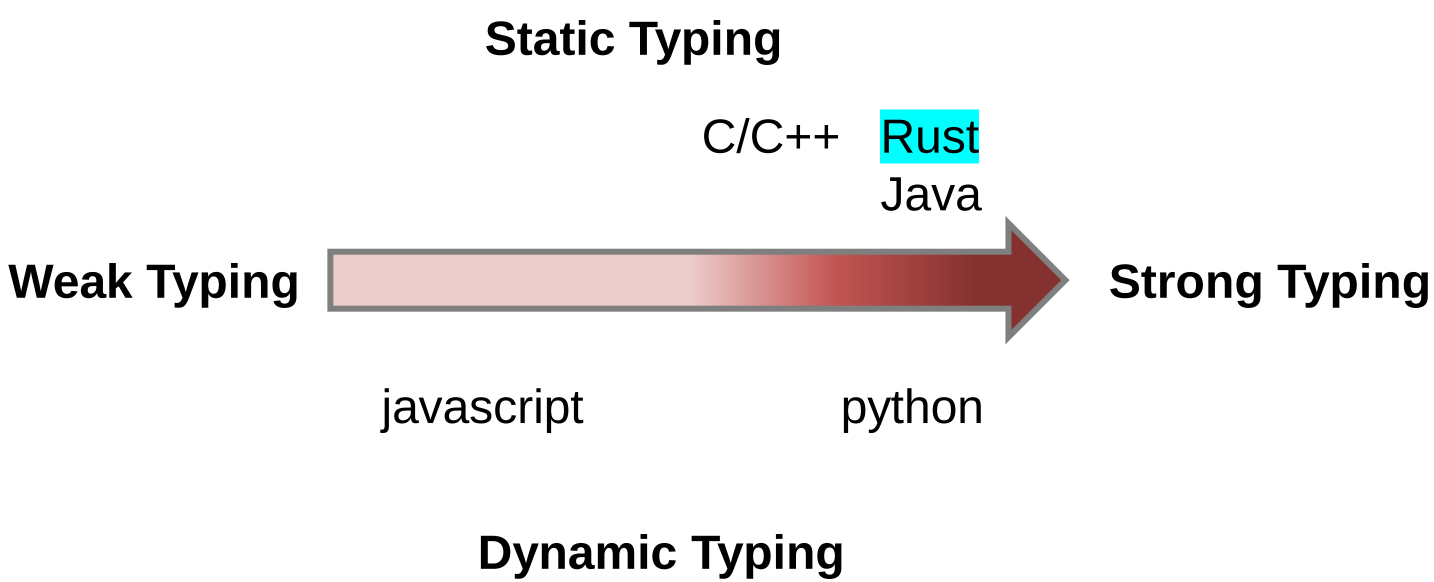
- Primitive types: basic types from which all other data types are constructed
  - scalar types: int, char, bool, float...
  - compound types: array, tuple
- Composite types: struct
  - abstract data type: tree, list,...

- Rules of typable code
  - function signature
  - operands of the operator
- How to justify type equivalence?
  - same name or same structure?

Objective of Type System Design

- Type soundness/safety
  - a well-typed program should not include any undefined operation
  - required for achieving memory safety
- Expressiveness or usability
  - implicit cast (coercion): may undermine type safety
  - overloading

Taxonomy of Type Systems



2. Types in Rust

Primitive Types

- Scalar types: literals (prefix + value + type)
  - 0xabcd_u32
  - 12i8 (no prefix)
  - 0b01110000 (no type: need type inference)
  - 1 (only value)
- Compound types:
  - array
  - tuple

Type Conversion

- Primitive types can be converted to each other
- Except int/float => bool (why?)

```
assert_eq!(true as i32, 1);

assert_eq!(255_u8 as i8, -1_i8);
assert_eq!(-1_i8 as u8, 255_u8);

assert_eq!(-1_i8 as i16, -1_i16);
assert_eq!(1024_i16 as u8, 0_u8);

assert_eq!(1.1_f32 as i32, 1_i32);
assert_eq!(-0.1_f32 as f64, -0.1_f64); //fail

assert_eq!(1_u8 as bool, true);
```

error[E0054]: cannot cast as `bool`
help: compare with zero instead: `1_u8 != 0`

Integer Overflow

- Compiler check (default)
- Run-time check (default in debug mode)
- Unchecked (default in release mode)

```
fn main() {
    let x = std::i32::MAX + 7;
    println!("{}",x);
}
```

Compile error

```
fn main() {
    let mut x = 1;
    for _i in 1..10000 {
        x += x;
    }
}
```

Runtile error in debug mode

Check what happens with the release mode: cargo build --release

cargo rustc --release -- --emit=mir

Array

- A collection of values of the same type in a contiguous memory

```
fn main(){
    let a: [i32; 5] = [1, 2, 3, 4, 5];
    let b = [0; 5];
    println!("{},{},{}", a[5], a.len(), mem::size_of_val(&a));
}
```

Runtime error: out-of-bound

std::Vec (not primitive type)

- A collection of values of the same type on heap
- The size can be dynamically adjusted

```
fn main(){
    let a = vec![1, 2, 3, 4, 5];
    a.push(6)
    println!("{},{},{}", a[5], a.len(), mem::size_of_val(&a));
}
```

Runtime error: out-of-bound

Slice

- Slices are similar to arrays, but their length is unknown at compile time (dynamically sized type)
- Two field: a pointer to the data, length

```
fn foo(s:&[i32], x:usize) {
    println!("{},{},{}", s[x], s.len(), mem::size_of_val(s));
}

fn main(){
    let a: [i32; 5] = [1, 2, 3, 4, 5];
    foo(&a, 2);
}
```

Tuple

- A collection of values of different types
- An anonymous strut without named fields

```
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    let (a, b) = pair;
    (b,a)
}
fn main(){
    let t = (1, true);
    let r = reverse(t);
    println!("tuple: ({}, {})", r.0, r.1);
    let tot = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);
    println!("tuple: {:?}", tot);
}
```

tuple of tuples

Struct

- Struct has a name, named fields, and methods
- Objects can be initialized via struct literals or constructors

```
struct MyList{
    val: i32,
    next: Option<Box<MyList>>,
}

impl List {
    fn from(a:&[i32]) -> MyList {
        ...
    }
    fn print(&self) { }
}
let mut l = Some(Box::new(MyList{val:a[0], next:None}));
let a: [i32; 6] = [1, 2, 3, 4, 5, 6];
let l = MyList::from(&a);
```

struct literal

constructor

Enum

- Which could be one of several different variants.
  - such as Option<T> and Result <T, E>
- Match
  - _=> means match the rest patterns
  - (), do nothing

```
pub enum Option<T> {
    None,
    Some(T),
}
```

```
match a {
    Some(ref value) => (),
    _ => (),
}
```

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

```
match r {
   Ok(v) => (),
   Err(e) => (),
}
```

3. Generic and Trait

Generic Type

- For parameter polymorphism (similar as C++ template)
  - Function with generic type parameters
  - Struct with generic type parameters
- Generic types can be monomorphized to concrete types when used

Generic Functions

- Use <T:Bound> to declare the generic types to be used
- All types satifing the (trait) bound are valid

```
fn larger<T:std::cmp::PartialOrd>(a:T, b:T) -> T {
    if(a > b) {
        return a;
    }
    return b;
}

fn main(){
    assert!(larger(100, 200) == 200);
    assert!(larger('a', 'b') == 'b');
    //assert!(larger('a', 100) == 100);
}
```

T is i32

T is char

Is T char or i32? compilation error!!!

Monomorphization

```
0000000000001fb0 <_ZN11genericfunc4main17hfd44a73acdc5c880E>:
    1fb0:        push   %rax
    1fb1:        mov    $0x64,%edi
    1fb6:        mov    $0xc8,%esi
    1fbb:        callq  1e30 <_ZN11genericfunc6larger17h937a6d14a36a7b9cE>
    1fc0:        mov    %eax,0x4(%rsp)
    1fc4:        mov    0x4(%rsp),%eax
    1fc8:        cmp    $0xc8,%eax
    1fcd:        sete   %cl
    1fd0:        xor    $0xff,%cl
    1fd3:        test   $0x1,%cl
    1fd6:        jne    1fec <_ZN11genericfunc4main17hfd44a73acdc5c880E+0x3c>
    1fd8:        mov    $0x61,%edi
    1fdd:        mov    $0x62,%esi
    1fe2:        callq  1ef0 <_ZN11genericfunc6larger17hfeeca0519db784d8E>
    1fe7:        mov    %eax,(%rsp)
    1fea:        jmp    2006 <_ZN11genericfunc4main17hfd44a73acdc5c880E+0x56>
...
```

Generic Structs

- Monomorphized to concrete types when instantiated
- Declare the trait bound with method implementations

```
struct MyList<T>{
    val: T,
    next: Option<Box<List<T>>>,
}

impl<T:Copy> MyList <T> {
    fn from(a:&[T]) -> MyList<T> {
        ...;
    }
}
let a: [i32; 6] = [1, 2, 3, 4, 5, 6];
let l = List::<i32>::from(&a);
```

Generic parameter

Constructor from slice

Trait

- Shared behavior among multiple types
- Traits are not types; traits can't have fields
- Some people may call it Objective Rust

```
trait Person {
    fn speak(&self);
    fn eat(&self);
}

trait Kid: Person {
    fn play(&self);
}

trait Adult: Person {
    fn work(&self);
}
```

A trait can have multiple methods

Kid inheritance from Person

Adult inheritance from Person

Implement Traits for Structs

- Separate the data and behavior

```
trait Count { fn getcount(&self) -> u32; }
struct MyList<T> { val:T, next:Option<Box<MyList<T>>>, }

impl Count for MyList<T> {
    fn getcount(&self) -> u32 {
        let mut r = 1;
        let mut cur = &self.next;
        loop {
            match cur {
                Some(x) => { r += 1;}
                _ => {break;}
            }
        }
        return r;
    }
}
```

Trait Bound for Generics

```
trait Count { fn getcount(&self) -> u32; }
struct MyList<T> { val:T, next:Option<Box<MyList<T>>> }

impl Count for MyList<T> {
    fn getcount(&self) -> u32 { ... }
}

fn foo<T:Count>(t: T) {
         println!("Count: {:?}", t.getcount());
}

impl<T:Copy> MyList <T> {
    fn from(a:&[T]) -> MyList<T> {
        ...;
    }
}

impl<T:Count> MyList<T> {
    fn dosth(&self){
        println!("Count:{}", self.val.getcount());
    }
}
```

Common Usage of Traits in Rust

- Comparison: Eq/PartialEq/Ord/PartialOrd.
- Print: Display/Debug
- Duplication: Copy/Clone
- Concurrency: Send/Sync
- Some traits can be derived via #[derive]

```
#[derive(Copy)]
#[derive(Debug,Clone)]
struct MyList<T> {
    val: T,
    next: Option<Box<List>>,
}
```

Not allowed, why?

Dynamic Trait

- Any type that implements the trait
- Based on vtable, similar to C++ virtual functions

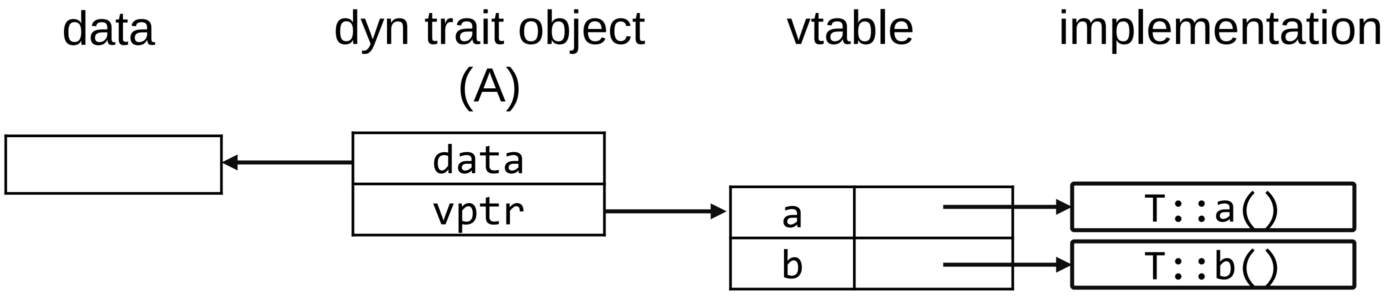
```
trait A {
    fn a(&self) {
        println!("super a");
    }
}
struct S { }
impl A for S { }

fn makeacall(dyna: &dyn A){
    dyna.a();
}

fn main() {
  let s = S {};
  makeacall(&s);
}
```

differences from generic functions?

```
fn makeacall<T:A>(a:T);
```

Mechanism of Dynamic Trait

- Purpose: dynamic dispatch
- Based on vtable




```
trait B : A{
    fn a(&self) { println!("sub a"); }
    fn b(&self) { println!("sub b"); }
}
struct S { s:i32 }
struct T { t:i32 }

impl A for S { }
impl B for T { }

fn makeacall(dyna: &dyn A){ dyna.a(); }
```

Trait versus Subtype

- Liskov substitution principle: when requiring a specific type, any of its subtype can be used
- Subtypes are partial order relationships:
  - $X \leq Y$: X is a subtype of Y
  - Self-reflective: $X \leq X$
  - Communicative: $X \leq Y$, $Y \leq Z \Longrightarrow X \leq Z$;
- Upcast: If X>Y, cast Y to X
  - Generally safe, allowed by default (C++)
- Downcast: If X>Y, cast X to Y
  - May incur undefined behaviors, should be checked

Rust Supports Subtype?

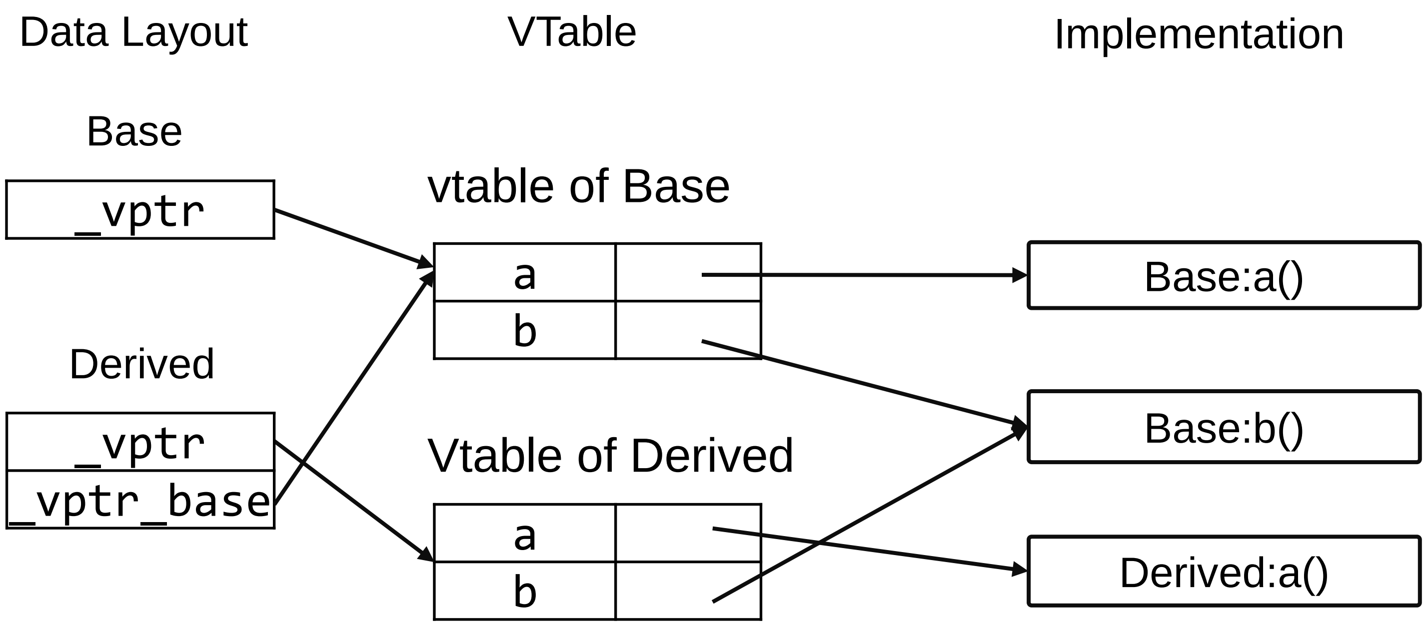
- You may think traits could have partial order:
  - B:A => B<A; impl<T> B for T where T:A { } => B<A
  - But traits are not types, B is not a subtype of A
  - Trait cannot be upcasted (not subtype)
- Subtype in Rust: lifetime
  - If the lifetime s>t, s is a subtype of t

```
struct S { }
struct T { }
trait A { }
trait B:A { }
impl A for S { }
impl B for T { }
fn makeacall(s: &S){ }

fn main() {
  let t = T {};
  makeacall(&t);
}
```

T implements more traits than S

Invalid: T is not a subtype of S

Comparison with C++ Vtable



Covariance

- Covariance: if t1 is a subtype of t2, g(t1) is a subtype g(t2)
  - e.g., i32 is a subtype of T, [i32] is a subtype of [T]
- Other relationships
  - contravariant: e.g., F(T) is a subtype of F(i32)
  - invariant

```
fn longer<'a, T>(a:&'a [T], b:&'a [T]) -> &'a [T]{
    if(a.len() > b.len()) {
        return a;
    }
    return b;
}

fn main(){
    let mut a: [i32; 5] = [1, 2, 3, 4, 5];
    let mut b: [i32; 6] = [0; 6];
    longer(&a,&b);
}
```

4. Special Types

PhantomData

Zero Sized Types

Sized vs Dynamic Sized Types

Pin/Unpin

The Problem of Slice

```
struct Slice<'a, T: 'a> {
    ptr: *const T,
    len: uszie,
}
```

Slice is unbound to the lifetime of T

Substitute raw pointers with reference?

```
let s;
{
    let mut v = [1, 2, 3];
    s = &mut v;
}
s[1] = 0;
```

s could live out the lifespan of v;

Incur dangling pointers!!!

PhantomData To Rescue

- A special marker type that consumes no space
- Simulate a field for lifetime inference
- Common patterns for raw pointers that own an allocation

```
struct Slice<'a, T: 'a> {
    ptr: *const T,
    len: uszie,
    _marker: PhantomData<&'a T>,
}
```

Similar Issues for Vec

```
struct Vec<T> {
    data: *const T,
    len: usize,
    cap: usize,
    _marker:marker::PhantomData<T>,
}
```

Raw pointer does not own T

T will not be reclaimed automatically

Own T via Phantom Data

Exotical Sized Types: Zero Sized Types

- Tell the compiler that the load/store of the value can be optimized
- Example: implement HashSet<T> as HashMap<T, ()>
- Not new in Rust; Java/C++/Go also have ZST

```
struct Zst;
struct Zst ();
```

Exotical Sized Types: Dyn Sized Types

- All types should have a constant size at compile time
- Dyn Sized Types: Use T: ?Sized to relax the bound
  - Dynamic trait
  - Slice

```
struct MyStruct<T: ?Sized> {
    data: T,
}

fn main() {
    let sized = MyStruct {data:[0; 8],};
    let dyn: &MyStruct<[u32]> = &sized; //type coercion
}
```

Problem of Self-Referential Struct

- If we move a SelfRef object, the pointer point to itself could be invalidated.

```
struct SelfRef { data: String, ptr: *const String, }
impl SelfRef {
    fn new(s: &str) -> Self {
        SelfRef {
            data: String::from(s),
            ptr: std::ptr::null(),
        }
    }
    fn init(&mut self) {
        self.ptr = &self.data as *const String;
    }
}
```

```
let mut a = SelfRef::new("123");
a.init();
let mut b = SelfRef::new("456");
b.init();
std::mem::swap(&mut a, &mut b);
println!("a = {:?}, b = {:?}", unsafe{&*a.ptr}, unsafe{&*b.ptr});
```

Use Pin

- Use Pin to wrap a type that is !Unpin
  - enable the compiler to check if a pined value is moved
  - has no effect if the type is Unpin

```
struct SelfRef {
    data: String,
    ptr: *const String,
    _pin: PhantomPinned,
}
```

Inform the compiler that the struct is !Unpin
Suffer compiling error if moving the objects of !Unpin

```
impl SelfRef {
    fn new(s: &str) -> Self {
        SelfRef {
            data: String::from(s),
            ptr: std::ptr::null(),
            _pin: PhantomPinned,
        }
    }
    fn init(self: Pin<&mut Self>) {
        unsafe{
            self.get_unchecked_mut().ptr = &self.data as *const String;
        }
    }
}
```

In-Class Practice

- Extending your binary search tree or double-linked list to support generic parameters
- Implement the PartialEq and PartialOrd traits for your struct