COMP 737011 - Memory Safety and Programming Language Design

Lecture 7: Rust Type System

Hui Xu
xuh@fudan.edu.cn

Outline

1. Type System
2. Basic Types in Rust
3. Advanced Types in Rust

1. Type System

Type System

- Primitive types: basic types supported by the compiler
  - scalar types: integer, char, bool, float...
  - compound types: array, tuple
- Composite types: struct/union/enumerate

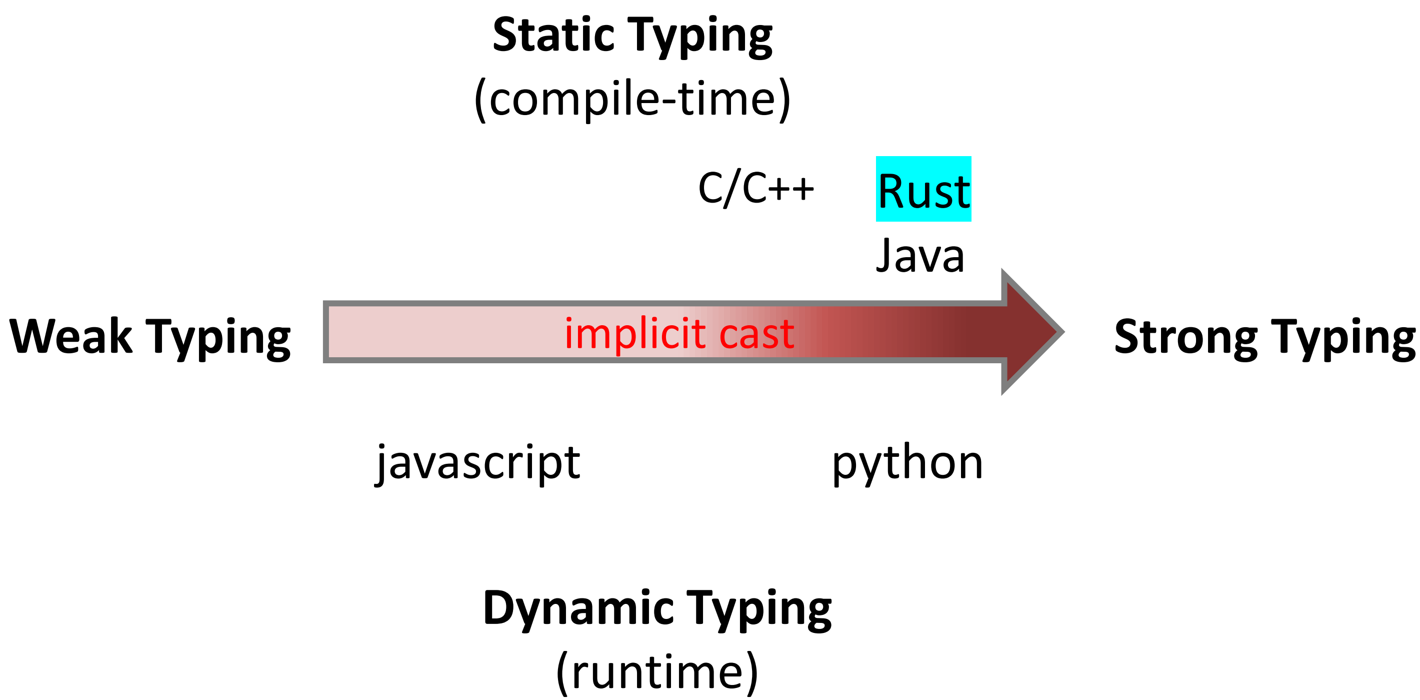
Type System

- Rules of typable code
  - operators have requirements on operand type
  - functions have requirements on the type of arguments and return values
- How to determine whether types are equivalent?
  - same name or same structure?

Objective of Type System Design

- Type safety
  - a well-typed program should not introduce undefined behaviors
  - basic requirement for achieving memory safety
- Expressiveness or usability
  - operator overloading
  - allow implicit type cast (coercion): may undermine type safety

Taxonomy of Type Systems



2. Basic 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: Integer

- Unsigned integers
  - Extend: short => long
  - Truncate: long => short
    - not saturated

```
assert_eq!(1u8 as u32, 1u32);
assert_eq!(257u32 as u8, 1u8);
```

- Signed integers
  - Extend: short => long
  - Truncate: long => short

```
assert_eq!(1i8 as i32, 1i32);
assert_eq!(257i32 as i8, 1i8);
```

- Unsigned <=> Signed
  - Bit reinterpretation (transmute)

```
assert_eq!(255u8 as i32, 255i32);
assert_eq!(256u32 as i8, 0i8);
assert_eq!(255u8 as i8, -1);
assert_eq!(-1i8 as u8, 255u8);
```

Type Conversion: Bool

- Bool to integer: extend
- Integer to bool: not allowed, why?

```
assert_eq!(true as u8, 1u8);
assert_eq!(1u8 as bool, true);
```

```
error[E0054]: cannot cast `u8` as `bool`
help: compare with zero instead
    | assert_eq!(1_u8 != 0, true);
```

Integer <=> Float

- Integer => float: pick the nearest floating-point numbers

```
assert_eq!(u32::MAX, 4294967295);
println!("{:.0}", u32::MAX as f32); // 4294967296
println!("{:.0}", 4294967301u64 as f32); // 4294967296
```

- Float => integer: if the number exceeds the target type's range
  - Saturated casting

```
assert_eq!(1000.0_f32 as u8, 255);
assert_eq!(-1000.0_f32 as u8, 0);
```

  - Modular casting

```
//LLVM fptoui/fptosi may incur undefined behavior
assert_eq!(unsafe{1000.0_f32.to_int_unchecked::<u8>()}, 232);
assert_eq!(unsafe{(-1000.0_f32).to_int_unchecked::<u8>()}, 24);
```

Transmute

- Reinterpret the memory of one type as another type
  - Transmute to bool

```
assert_eq!(unsafe {transmute::<u8, bool>(1)}, true);
assert_eq!(unsafe {transmute::<u8, bool>(2)}, false);
```

  - Transmute floating-point numbers to integers

```
let a = 1.1f32;
//00111111100011001100110011001101 in IEEE-754 format
let b = unsafe {std::mem::transmute::<f32, i32>(a)};
assert_eq!(b, 1066192077);
```

Integer Overflow

- Compiler check: only for very simple cases
- Run-time check
  - enabled by default in debug mode
  - not enabled in release mode

```
let x = std::i32::MAX + 7;
```

Compiling error

```
let mut x = 1;
for _i in 1..10000 {
    x += x;
}
```

Runtime error in debug mode

Analyse the mir to see what happens

```
bb5: {
    _8 = copy ((_5 as Some).0: i32);
    _9 = copy _1;
    _10 = AddWithOverflow(copy _1, copy _9);
    assert(!move (_10.1: bool), "attempt to compute `{} + {}`, which would overflow", copy _1, move _9)
}
```

Array

- A collection of values of the same type in a contiguous memory.
- The memory is allocated on stack.
- The memory must be initialized when created.

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

Runtime error: out-of-bound

Vec (not primitive type)

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

```
let mut a = vec![1, 2, 3, 4, 5];
a.push(6);
println!("{:?}", a); // [1, 2, 3, 4, 5, 6]
a.pop();
println!("{:?}", a); // [1, 2, 3, 4, 5]
```

Slice

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

```
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[1..5], 2); // the third element of the slice is 4
}
```

The Problem of Slice

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

Slice is unbound to the lifetime of borrowing

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

s could live out the lifespan of v;

dangling pointer!!!

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> {
    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 PhantomData

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!("{}, {}", r.0, r.1);
    let tot = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);
    println!("{:?}", tot);
}
```

tuple of tuples

Struct

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

```
#[derive(Debug)]
struct List {                                    struct definition
    val: u64,
    next: Option<Box<List>>,
}
impl List {                                      implementation
    fn new(v: u64) -> Self {
        List { val: v, next: None }
    }
    fn append(&mut self, v: u64) { ... }
}

let l1 = List{ val:1, next:None };               employ the literal constructor
let mut l2 = List::new(1);                       call the constructor
l2.append(2);
println!("{:?}", l1);
println!("{:?}", l2);
```

Enum Type

- An enumerate type has one or several different variants.
  - such as Option<T> and Result <T, E>
- Unwrap the object of an enumerate type via match-case.
  - _=> means match the rest patterns
  - (), do nothing

```
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) => (),
}
```

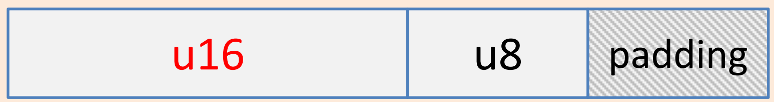
Layout of Types: Alignment + Size + Padding

- Alignment is always a power of 2. $\forall x \in N, align = 2^x$
- The address of a type should be a multiple of its alignment.

$$\&T \; \% \; align = 0$$

- Size: The size of a value is the offset in bytes between successive elements in an array with that item type including padding.

```
struct MyStruct {
    a: u16,
    b: u8
} // alignment: 2
mem::size_of::<MyStruct>(); //? 3 or 4
```



https://doc.rust-lang.org/reference/type-layout.html

3. Advanced Types in Rust

Generic Type

- Defining functions, structs, and enums with type parameters.
- Enables code reuse while maintaining type safety.
- To achieve parameter polymorphism (similar as C++ template).
- When used, generic types are monomorphized to concrete types.

Functions with Type Parameters

- Use <T:Bound> to declare the generic types to be used.
- All types satisfying the trait bound are valid.

```
fn larger<T: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>
...
```

Structs with Type Parameters

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

```
struct List<T> {
    val: T,
    next: Option<Box<List<T>>>,
}
impl<T> List<T> {
    fn new(v: T) -> Self {
        List { val: T, next: None }
    }
    fn append(&mut self, v: T) { ... }
}
```

Trait

- A trait contains shared behaviours among multiple types.
- Traits are not types because traits cannot have fields.
- Some people may call it objective Rust.
  - Traits can be inherited.
  - Traits may have default implementations which can be overloaded.

```
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 inherits from Person

Adult inherits from Person

Common Usage of Traits in Rust

- Primitive Traits
  - Comparison: Eq/PartialEq/Ord/PartialOrd.
  - Print: Display/Debug
  - Duplication: Copy/Clone
  - Concurrency: Send/Sync
- Implement Traits for struct:
  - Automatically derive if all fields implement the trait via #[derive].
  - Manual implementation

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

Not allowed, why?

Manual Implementation

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

impl<T: fmt::Display> fmt::Display for List<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.val)?;
        if let Some(ref next) = self.next {
            write!(f, " -> {}", next)?;
        }
        Ok(())
    }
}
```

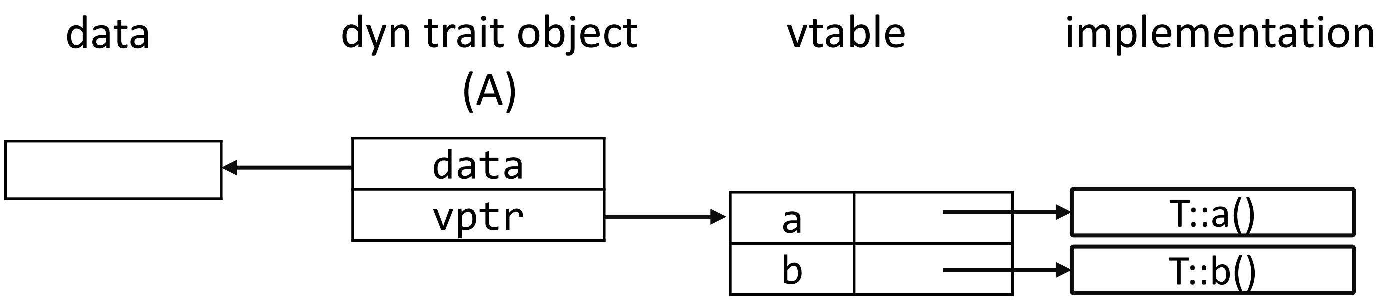
Dynamic Trait

- Any type that implements the trait.
- To achieve dynamic dispatch, similar to C++ virtual functions.

```
trait A {
    fn a(&self) { println!("default A"); }
}
struct S { }
struct T { }

impl A for S { }
impl A for T {fn a(&self) { println!("new A"); } }

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

fn main() {
    makeacall(&S {}); // default A
    makeacall(&T {}); // new A
}
```

Mechanism of Dynamic Trait

- Based on vtable




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

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

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

Trait vs Subtype

- Liskov substitution principle: when requiring a specific type, any of its subtype can be used
- Subtypes are partial order relationships:
  - $X \le Y$: X is a subtype of Y
  - Self-reflective: $X \le X$
  - Communicative: $X \le Y$, $Y \le Z \Longrightarrow X \le Z$；
- 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

Traits are not Subtypes

- Traits could have partial order, *e.g.*, B:A => B<A
- But traits are not types, so B is not a subtype of A
- Subtype in Rust: lifetime
  - If the lifespan s>t, s is a subtype of t

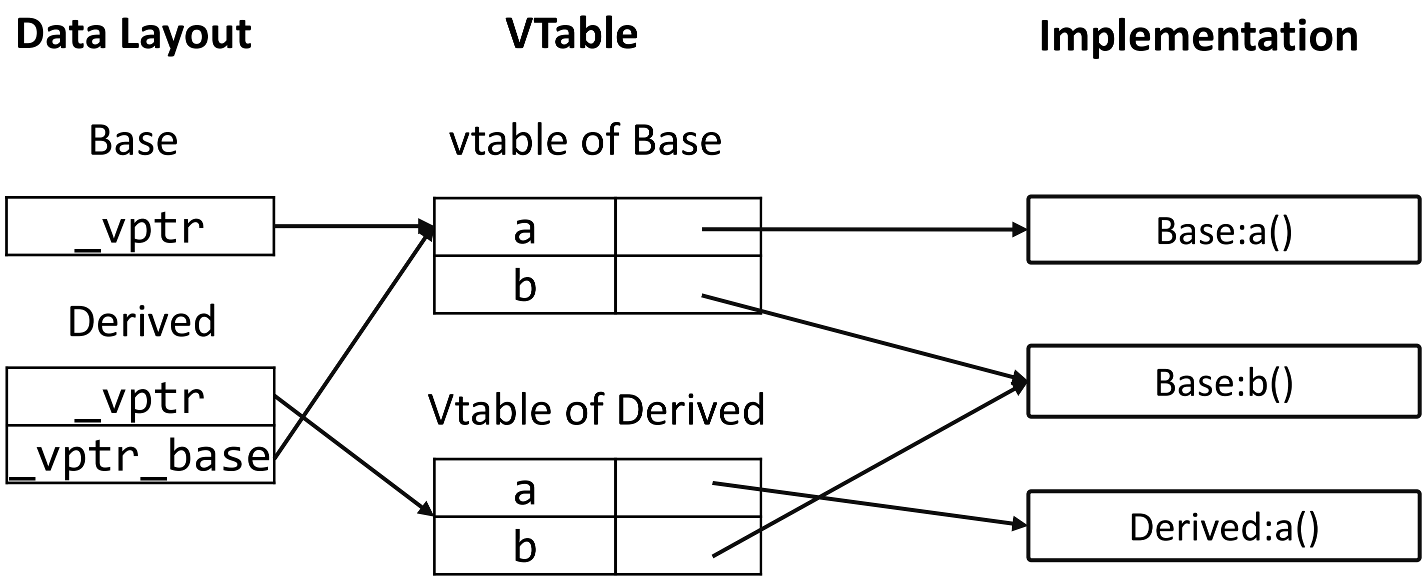
```
struct S { }
struct T { }
trait A { }
trait B:A { }
impl A for S { }
impl B for T { }
fn makeacall(s: &S){ }

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

- C++ classes can be upcasted
- Trait cannot be upcasted



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: no relationship

```
fn longer<'a, T>(a: &'a [T], b: &'a [T]) -> &'a [T] {
    if a.len() > b.len() {
        return a;
    }
    return b;
}
fn main(){
    let  a: [i32; 5] = [1, 2, 3, 4, 5];
    let  b: [i32; 6] = [0; 6];
    longer(&a,&b);
}
```

In-Class Practice

- Extend your binary search tree to support generic parameters
- Implement traits for your binary search tree
  - PartialEq
  - PartialOrd
  - Debug
  - Display
  - Clone
- Design and implement a new trait Count for the tree, that outputs the number of nodes.