COMP 737011 - Memory Safety and Programming Language Design

Lecture 7: Rust Type System

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Basic Concept
- 2. Basic Types in Rust
- 3. Generic and Trait
- 4. Subtype in Rust

1、Basic Concept

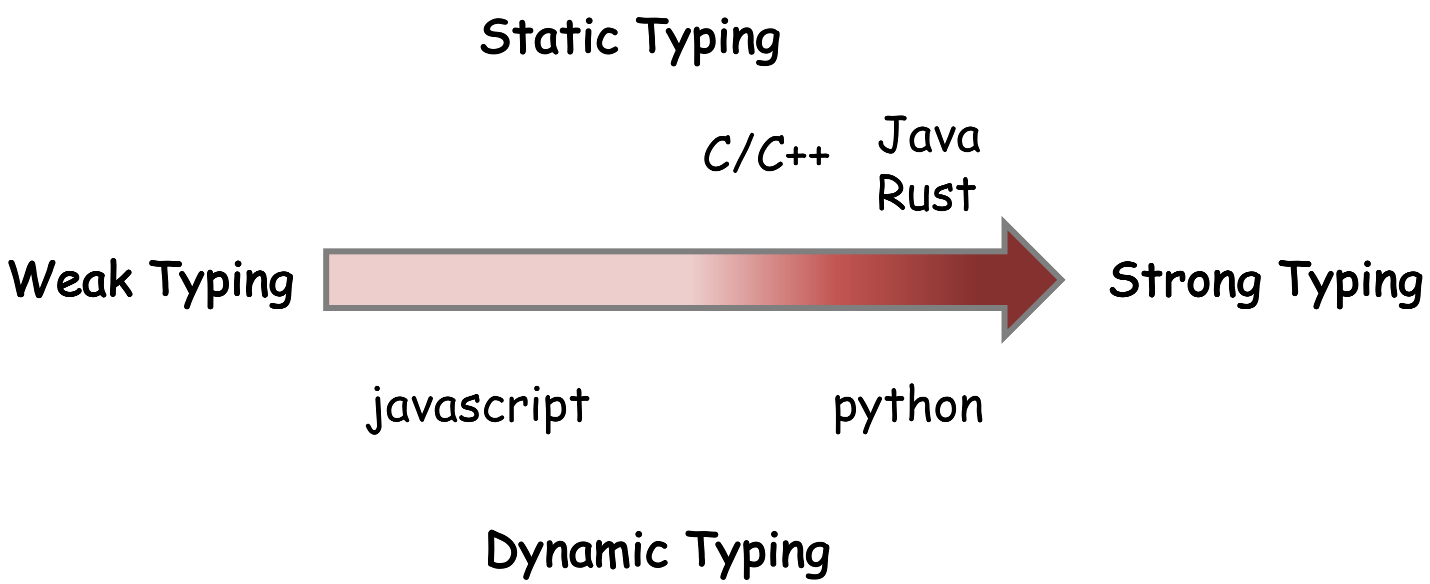
Type System

- Types:
  - Primitive types: basic types from which all other data types are constructed.
    - int, char, bool, float...
  - Custom/composite types: struct
- Rules of typing
  - e.g., requirement of operand types for each operator
- How to justify type equivalence?
  - Same name
  - Same structure

Objective of Type System

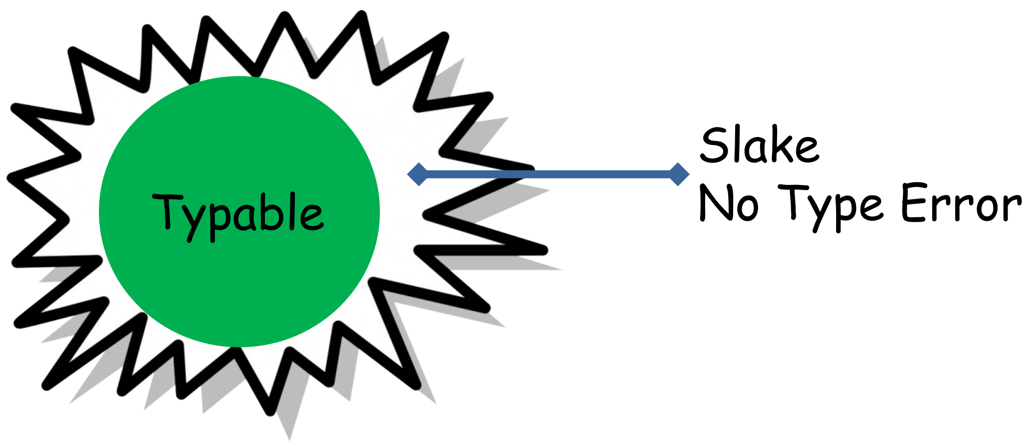
- Type soundness/safety
  - a well-typed program should not include any undefined operation.
- Expressiveness or usability
  - Example features
    - Implicit cast (may undermine type safety)
    - Overloading

Taxonomy of Type Systems



How to Prevent Type Error

- Type checking
  - explicitly declare the type of each variable and check the consistency
- Type inference
  - typeability: infer the type of variables given a context
  - Damas-Hindley-Milner algorithm
    - Based on constraint modeling and solving
    - Widely used, e.g., ML、Haskell、Ocaml



Sample Type System

- Basic types

$\tau \rightarrow$ int	int
$\mid$ & τ	pointer
$\mid$ $(\tau,\ldots,\tau) \rightarrow \tau$	function

- More derived types

(int, &int) → &&int

Sample Rules of Constraint Extraction

	Code	Type Constraint
1	`I`	`⟦I⟧ = int`
2	`E1 op E2`	`⟦E1⟧ = ⟦E2⟧ = ⟦E1 op E2⟧ = int`
3	`E1 == E2`	`⟦E1⟧ = ⟦E2⟧ = ⟦E1 == E2⟧ = int`
4	`input`	`⟦input⟧ = int`
5	`X = E`	`⟦X⟧ = ⟦E⟧`
6	`Output E`	`⟦E⟧ = int`
7	`if(E){S}`	`⟦E⟧ = int`
8	`if(E){S1}else{S2}`	`⟦E⟧ = int`
9	`while(E){S}`	`⟦E⟧ = int`

10	`X(X1,...,Xn){`
	`   ...`
	`   return E;`
	`}`
		`⟦X⟧ = (⟦X1⟧...⟦Xn⟧)→⟦E⟧`

11	`E(E1,...,En)`	`⟦E⟧ = (⟦E1⟧...⟦En⟧)→⟦E(E1...En)⟧`
12	`alloc E`	`⟦alloc E⟧ = &⟦E⟧`
13	`&X`	`⟦&X⟧ = &⟦X⟧`
14	`*E`	`⟦E⟧ = &⟦*E⟧`
15	`*X=E`	`⟦X⟧ = &⟦E⟧`

Applying the Rules

```
main(){
    var x, y, z;
    x = input;
    y = alloc x;
    *y = x;
    z = *y;
    return z;
}
```

Extracted Constraint

$[\![main]\!] = () \to [\![z]\!]$

$[\![x]\!] = [\![input]\!] = int$

$[\![y]\!] = [\![alloc\ x]\!] = \&[\![x]\!]$

$[\![y]\!] = \&[\![x]\!]$

$[\![z]\!] = [\![*y]\!],\ [\![y]\!] = \&[\![*y]\!]$

Solution

$[\![x]\!] = int$

$[\![y]\!] = \&int$

$[\![z]\!] = int$

$[\![main]\!] = () \to int$

2. Basic Types in Rust

Literals of Primitive Types

- Pure value (may need type inference)
  - 1 (int), 0.1 (float), true (boolean)
  - 'a' (char), "a" (string)
- Explicit: prefix + value + type
  - 12i8
  - 0xabcd_u32
  - 0b01110000
  - 0o100

Type Conversion

- Primitive types can be converted to each other, except
  - int/float => bool (unsupported yet?)

```
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
```

Integer Overflow

- Rust compiler checks integer overflow by default

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

```
note: `#[deny(arithmetic_overflow)]` on by default
```

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: [i32; 500] = [0; 500];

    println!("{},{},{}", a[0], a.len(), mem::size_of_val(&a));
    println!("{},{},{}", b[500], b.len(), mem::size_of_val(&b));
}
```

Compiler error: out-of-bound

```
#: ./array
1,5,20
0,500,2000
```

Slice

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

```
fn f1(s: &[i32]) {
    println!("{},{},{}", s[10], s.len());
}

fn main(){
    let a: [i32; 5] = [1, 2, 3, 4, 5];
    let b: [i32; 500] = [0; 500];
    f1(&a);
    f1(&b);
}
```

```
#: ./slice
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 10', slice.rs:4:26
```

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 name, named fields, and methods

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

impl List {
    fn from(a:&[i32]) -> List {
        let mut l = Some(Box::new(List{val:a[0], next:None}));
        let mut cur = &mut l;
        for i in 1..a.len()-1 {
            …
        }
        *l.unwrap()
    }
    fn print(&self) { }
}
```

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> to declare the generic types to be used

```
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?
Could incur 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

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

impl<T:Copy+fmt::Debug> List<T> {
    fn from(a:&[T]) -> List<T> {
        let mut l = Some(Box::new(List{val:a[0], next:None}));
        let mut cur = &mut l;
        for i in 1..a.len()-1 {
            match cur {
                None => {}
                Some(ref mut _node) => {
                    _node.next = Some(Box::new(List{val:a[i], next:None}));
                    cur = &mut _node.next;
                }
            }
        }
        *l.unwrap()
    }
}
```

Trait

- Some developers may call it Objective Rust
- The functionality can be shared/reused among multiple types

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

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

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

Implement a Trait

```
trait Countable{ fn getcount(&self) -> u32; }
struct MyList{ val:i32, next:Option<Box<MyList>>, }

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

Trait Bound for Generics

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

impl Countable for MyList {
    fn getcount(&self) -> u32 {
        ...
    }
}

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

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

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(Debug,Clone)]
struct List{
    val: i32,
    next: Option<Box<List>>,
}

let mut l = List::from(&a);
let mut lc = l.clone();
if (l == lc) {
    lc.val = 100;
    println!("{:?}",lc);
}
```

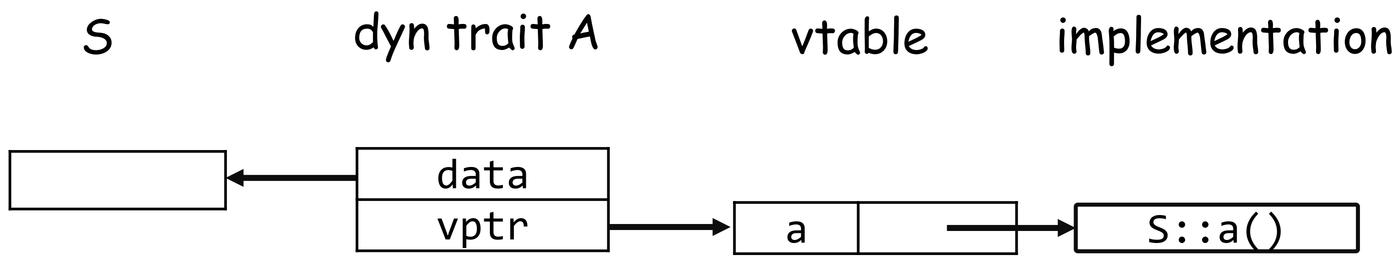
Dynamic Trait

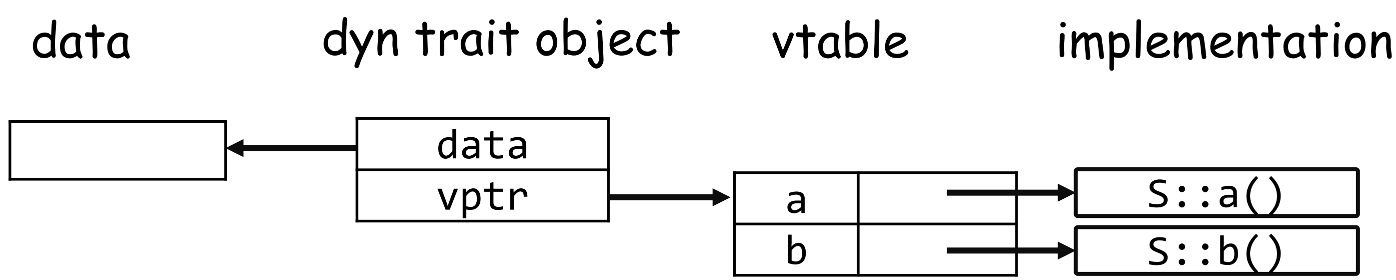
- Any type that implements the trait
- Based on vtable, similar to C++ virtual functions

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

impl A for S { }
//impl B for S { }

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

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

Mechanism of Dynamic Trait

- Based on vtable





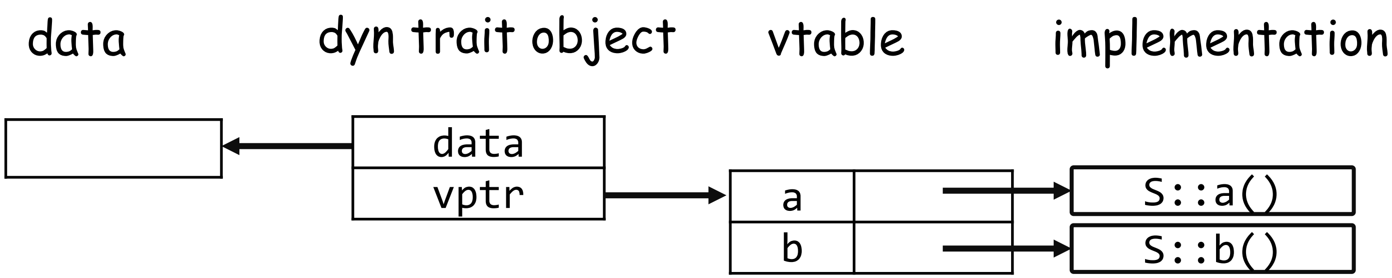
4. Subtype in Rust

Subtype

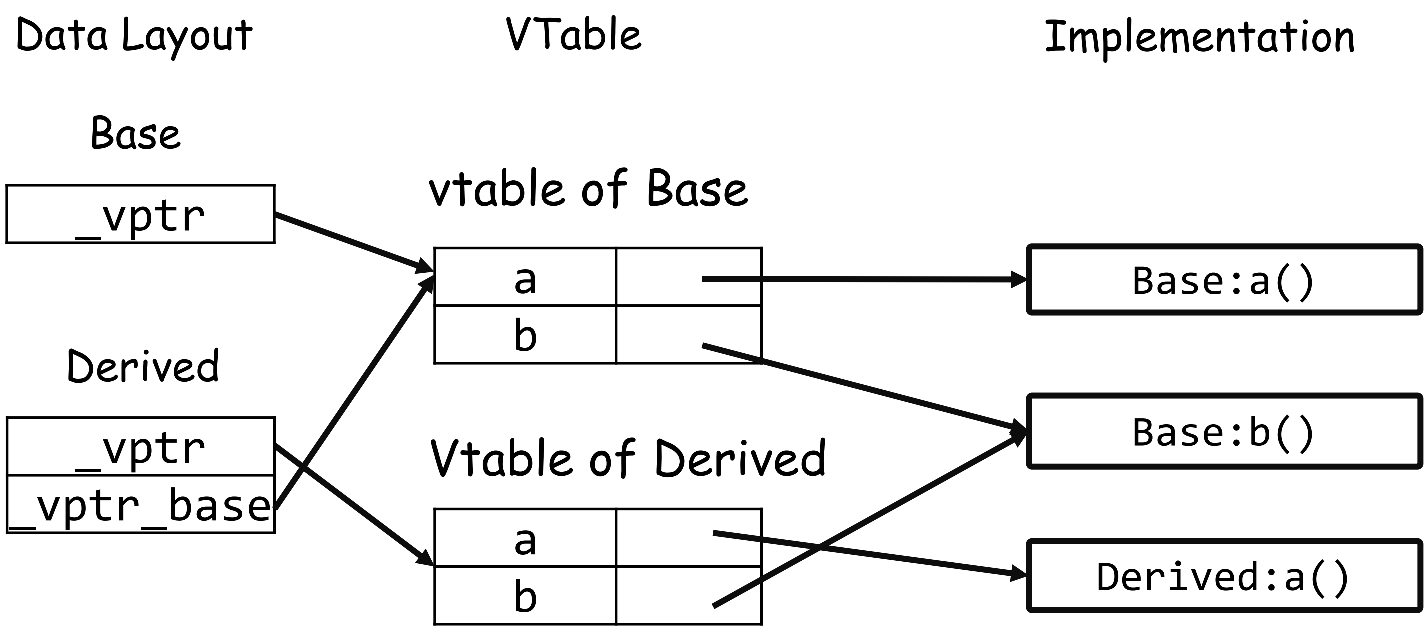
- Partial order like $X \leq Y$
  - X is a subtype of Y
  - Y is a supertype of X
- Properties of subtype
  - Self-reflective: $X \leq X$
  - Communicative: $X \leq Y,\ Y \leq Z \Rightarrow X \leq Z$;
- When requiring a specific type, any of its subtype can be used.
  - Subtype is compatible to be used with its super type

Upcast and Downcast

- Upcast: If X>Y, cast Y to X
  - Generally safe, allowed by default (C++)
  - Rust trait cannot be upcasted (not subtype)
- Downcast: If X>Y, cast X to Y
  - May incur undefined behaviors, should be checked



Comparison with C++ Vtable



Rust Support Subtype?

- If the lifetime of s>t, s is a subtype of t
- You may think trait could have partial order:
  - B:A => B<A
  - impl<T> B for T where T:A { } => B<A
- But trait is not type
  - B is not a subtype of A
  - impl A for S does not imply S>A

```
struct S { }
trait A { }
trait B : A { }
impl A for S { }
impl B for S { }

fn makeacall<T:A>(s: &T){ s.a() }

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

Valid as S implements A

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

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

Invalid: T is not a subtype of S although it implements more traits

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; 500] = [0; 500];
    longer(&a,&b);
}
```

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.

More Reference

- Chapter 3 Type Analysis, Static Program Analysis, Anders Møller etc.
- https://doc.rust-lang.org/nomicon/subtyping.html