Subtyping & Variance

CIS 198 Lecture 16

Higher-Rank Trait Bounds

```rust
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F> where F: Fn(&(u8, u16)) -> &u8 {
    fn call(&self) -> &u8 {
        (self.fun)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let c = Closure { data: (0, 1), func: do_it, };
    c.call();
}
```

Higher-Rank Trait Bounds

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F> where F: Fn(&'??? (u8, u16)) -> &'??? u8 {
    fn call<'a>(&'a self) -> &'a u8 {
        (self.fun)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
        let c = Closure { data: (0, 1), func: do_it, };
        c.call();
    }
}
```

Higher-Rank Trait Bounds

```
impl<F> Closure<F> where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8 {
    fn call<'a>(&'a self) -> &'a u8 {
        (self.fun)(&self.data)
    }
}
```

Inheritance vs. Subtyping

- Rust does not support structural inheritance.
  - No classes, virtual functions (sort of), method overriding, etc.
- Subtyping in Rust derives exclusively from lifetimes.
- Lifetimes may be partially ordered based on a containment relation.

Lifetime Subtyping

- Lifetime subtyping is in terms of the containment relationship:
  - If lifetime `a` contains (outlives) lifetime `b`, then `'a` is a subtype of `'b`.
  - In Rust syntax, this relationship is written as `'a: 'b`.

Type Variance

- Variance is a property of type constructors with respect to their arguments.
- Type constructors in Rust can be either *variant* or *invariant* over their types.
- Variance: `F` is *variant* over `T` if `T` being a subtype of `U` implies `F<T>` is a subtype of `F<U>`.
  - Subtyping "passes through".
- Invariance: `F` is *invariant* over `T` in all other cases.
  - No subtyping relation can be derived.

Type Variance

- `&'a T` is variant over `'a` and `T`
  - As is `*const T`.
- `&'a mut T` is variant over `'a` but invariant over `T`.
- `Fn(T) -> U` is invariant over `T` but variant over `U`.
- `Box<T>`, `Vec<T>`, etc. are all variant over `T`
- `Cell<T>` and any other types with interior mutability are invariant over `T`
  - As is `*mut T`.

Type Variance

- Why do the above properties hold?

`&'a mut T` Type Invariance

```
fn overwrite<T: Copy>(input: &mut T, new: &mut T) {
    *input = *new;
}

let mut forever: &'static str = "hello";
{
    let s = String::from("world");
    overwrite(&mut forever, &mut &*s);
}
println!("{}", forever);
```

- In general, if variance would allow you to store a short-lived value into a longer-lived slot, you must have invariance.

Type Variance

- `Box` and `Vec` are variant over `T`, even though this looks like it breaks the rule we just defined.
- Because you can only mutate a `Box` via an `&mut` reference, they become invariant when mutated!
  - This prevents you from storing shorter-lived types into longer-lived containers.
- Cell types need invariance over `T` for the same reason `&mut T` does.
  - Without invariance, you could smuggle shorter-lived types into longer-lived cells.

Type Variance

- What about the variance of types you define yourself?
- Struct Foo contains a field of type A...
  - Foo is variant over A if all uses of A are variant.
  - Otherwise, it's invariant over A.

Type Variance

```rust
struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H> {
    a: &'a A,     // variant over 'a and A
    b: &'b mut B, // variant over 'b, invariant over B
    c: *const C,  // variant over C
    d: *mut D,    // invariant over D
    e: Vec<E>,    // variant over E
    f: Cell<F>,   // invariant over F
    g: G,         // variant over G
    h1: H,        // would be variant over H...
    h2: Cell<H>,  // ...but Cell forces invariance
}
```

Type Variance

- A good way to think about lifetime variance is in terms of ownership.
- If someone else owns a value, and you own a reference to it, the reference can be variant over its lifetime, but might not be variant over the value's type.