The lifetime parameter of Rust

Rust 함수의 lifetime parameter는 언제 써야 하고 언제 생략할 수 있나요?

Sigrid Jin (Jin Hyung Park) on '22. 4. 24.

소개

- 다른 언어와 마찬가지로, Rust의 오브젝트들도 lifetime을 갖는다.
- 하지만 다른 언어들과 다르게, Rust에서는 오브젝트에 대한 레퍼런스의 lifetime에 대해 컴파일 타임에 체크하고, 컴파일 에러를 발생시킨다.
- 또한 lifetime이 모호한 경우에도 컴파일 에러가 발생하는데, 이를 방지하기 위해 lifetime parameter라는 Rust만의 독특한 문법을 활용해야 한다.

Borrow Checker

- 하기 코드는 컴파일 에러가 발생한다.
- 그 이유는 무엇일까?

```
let p;

{
    let num = 10;
    p = &num;
}

println!("{}",p);
```

Borrow Checker (cont'd 1/n)

- `p`가 `num`을 가리키도록 되어 있는데, `println`이 호출되는 시점에 `num`의 `lifetime`이 만료되어 `p`가 `dangling reference`가 되었기 때문이다.
- 이런 상황에서 C와 같은 대부분의 언어는 컴파일 에러가 발생하지 않는다.
- Rust는 각 레퍼런스마다의 `lifetime`을 추적하며, `lifetime`이 만료된 시점에서 접근할 경우 컴파일 에러를 발생한다. Rust 컴파일러의 이러한 기능을 `Borrow checker`라고 부른다.
- Rust 컴파일러는 컴파일 에러 메시지에서, `Borrow checker`가 어느 시점에 `lifetime`이 소멸되었는지 표시해주므로, 디버깅할 때 편리하다.

Borrow Checker (cont'd 2/n)

- `Borrow checker` 덕분에 Rust에서는 각 레퍼런스의 `lifetime`을 컴파일 타임에 미리 알 수 있다. 하지만 단순 코드 상으로 `lifetime`을 추론해내는 것이 불가능한 경우가 있을 수 있다.
- 아래 함수에서는 리턴값이 `x의 레퍼런스` 일지, `y의 레퍼런스` 일지 알 수 없다.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    };
    y
}

// 5 | fn longest(x: &str, y: &str) -> &str { expected lifetime parameter
```

Declare a lifetime parameter

- 명시법은 제너릭 타입과 유사한데, 앞에 `'` 만 붙이면 된다.
- `a` 라는 `lifetime` 을 두 매개변수와 리턴값에 대해 명시해주었다.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Warning!

- `lifetime parameter`는 실제 `lifetime`에 영향을 끼치지 않는다.
- `lifetime parameter`는 컴파일러에게 `x`와 `y` 그리고 `반환값`이 최소한 `a`만큼의 `lifetime`을 갖는다는 힌트를 줄 뿐이다.
- 즉, `리턴 값의 lifetime이 최소한 x와 y 중 lifetime이 작은 것만큼은 된다` 라는 의미!

Borrow Checker (cont'd 3/n)

- 아래 함수의 `result` 는 `longest` 함수 내의 지역 변수이고, 이를 리턴하는 것은 전형적인 `dangling pointer` 의 발생 사례이다.
- `lifetime paremeter` 가 오브젝트의 `lifetime` 을 변경하는 것이 아니므로, 위 함수는 컴파일 에러가 난다.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Omitting Lifetime parameter in function

- 함수의 경우 라이프타임을 표기하는 파라미터는 생략이 가능해서 생략 규칙에 익숙하지 않으면, 함수가 의미를 이해하기 어려울 수 있다.
- 어떤 라이프타임을 생략할 수 있는지 확실치 않은 경우는 모든 레퍼런스에 파라미터를 적어주고 clippy를 사용하면 어떤 파라미터가 필요 없는지 친절하게 알려준다. 이에 따라 코드를 다듬으면 된다.
- 문제는 다른 사람이 작성한 함수를 읽을 때이다. 라이프타임에 따라 함수의 의미가 달라진다. 때문에 라이프타임을 정확히 파악하지 못하면, 코드를 잘못 이해하는 경우가 생긴다.

Omitting Lifetime parameter in function (cont'd 1/n)

입력에 사용된 라이프타임이 생략된 경우

- 전부 다른 파라미터로 생각하면 된다. 다시 말해서, 입력에 사용된 라이프타임은 다른 라이프 타임과 관계없다면 모두 생략할 수 있다.
- 예를 들어, `std::cmp::Ord::cmp(&self, other: &Self) -> Ordering` 가 2개의 레퍼런스 타입을 받지만, 라이프타임이 명시되지 않은 이유가 이 때문이다.

Omitting Lifetime parameter in function (cont'd 2/n)

출력에 사용된 라이프타임이 생략된 경우

- 첫 번째 경우는 함수의 인자에 단 하나의 라이프타임만 사용된 경우이다. 함수의 인자에 라이프타임 파라미터가 하나만 사용된 경우, 결괏값에 생략된 라이프타임 파라미터는 전부 인자의 라이프타임으로 간주한다.
- 예를 들어 `std::cell::Cell::from_mut(t: &mut T) -> &Cell<T>` 는 `T` 레퍼런스를 인자로 받아 이 레퍼런스를 감싸는 `Cell` 의 레퍼런스를 돌려주는 함수이다.
- `from_mut` 의 인자에는 `T` 의 레퍼런스를 한정하는 라이프타임만을 필요로 하므로, 결괏값에 생략된 라이프타임은 인자와 같은 라이프타임이다.
- 즉, 위의 코드는 사실 `std::cell::Cell::from_mut<'a>(t: &'a mut T) -> &'a Cell<T>` 와 같은 코드다.

Omitting Lifetime parameter in function (cont'd 3/n)

라이프타임이 레퍼런스 뿐만 아니라 타입의 파라미터로도 사용될 수 있다.

- 예를 들면 `std::cell::RefMut::map<U>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>` 의 경우 인자에 레퍼런스가 사용되지 않았지만 `RefMut` 자체가 라이프타임 파라미터를 포함하고 있는 구조체이기 때문에 `RefMut::map` 의 인자에는 하나의 라이프타임 파라미터가 사용된 것이다.
- 따라서 위의 함수는 결괏값의 `'b` 를 생략하여 `RefMut::map<U>(orig: RefMut<'b, T>, f: F) -> RefMut<U>` 이라고 적을 수도 있다.

Omitting Lifetime parameter in function (cont'd 4/n)

라이프타임이 레퍼런스 뿐만 아니라 타입의 파라미터로도 사용될 수 있다. (cont'd 2/n)

- 같은 이유로 `std::cell::Ref::clone(orig: &Ref<'b, T>) -> Ref<'b, T>` 의 결괏값에 있는 라이프타임 `'b` 는 생략할 수 없다. 명시적으로 보이는 라이프타임은 `'b` 뿐이지만, 사실 orig의 타입은 `&'a Ref<'b, T>` 로 레퍼런스의 라이프타임, `'a` 를 생략해 표현한 것이다.
- 즉, `Ref::clone` 함수는 인자에 2개의 라이프타임이 사용됐기 때문에 결괏값의 라이프타임을 생략하여 표현할 수 없다.

Omitting Lifetime parameter in function (cont'd 5/n)

결괏값의 라이프타임을 생략할 수 있는 두 번째 경우는 함수의 인자로 `self` 혹은 `mut self` 레퍼런스가 사용된 경우다.

- 이 경우 결괏값에서 생략된 라이프타임은 `&self` 와 같은 라이프타임으로 취급된다.
- `std::convert::AsRef::as_ref(&self) -> &T` 의 결괏값 `&T` 는 `self` 와 같은 라이프타임을 가진다.
- 즉, 위의 `as_ref` 는 사실 `std::convert::AsRef::as_ref<'a>(&'a self) -> &'a T` 와 같은 시그니처다.

Omitting Lifetime parameter in function (cont'd 5/n)

요약

1. 모든 레퍼런스는 lifetime을 갖는다.
2. 만약 인자로 입력받는 레퍼런스가 1개라면, 해당 레퍼런스의 lifetime이 모든 리턴값에 적용된다.
3. 함수가 self 혹은 &mut self를 인자로 받는 메소드라면, self의 lifetime이 모든 리턴값에 적용된다.

Thoughts?

- 하기 함수에서, Rust 컴파일러는 x와 y의 lifetime을 이미 알고 있다. 그렇다면, 굳이 lifetime을 명시적으로 표기할 필요 없이, 컴파일러가 알아서 x와 y 중 더 작은 lifetime을 리턴 값에 대응시키면 되는 거 아닌가?

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

- 러스트를 처음 쓰는 사람에게는, 특히 garbage collection이 있는 언어를 사용하던 사람에게 라이프타임은 과도한 제약으로 느껴질 수 있다. 하지만 라이프타임은 러스트의 안전성의 한 축을 맡는 중요한 기능이다.

Reference

- https://blog.seulgi.kim/2019/12/rust-lifetime-elision.html
- https://showx123.tistory.com/75