04-06.md

1. Lezione del 06/04 – Chiusure e Polimorfismo

Abbiamo parlato esaustivamente di come passare ad una funzione un riferimento e dei vantaggi che ciò comporta. Tuttavia il borrow checker esegue vari controlli a riguardo, tipo assicurarsi che quel dato rimanga valido per tutta l'esecuzione della funzione chiamata, potenzialmente anche dopo se viene riutilizzata.

Questo controllo è molto stringente, e per un programmatore non è difficile convivervi fino a quando le nostre funzioni vanno a lavorare su un singolo dato. Ma nel caso che una funzione accetti più parametri, esso dovrebbe assicurarsi che entrambi rimangano validi per il tempo necessario, e questo puo' essere difficile da capire anche per il compilatore.

Puo' quindi capitare che esso non riuscendo ad intendere quanto una variabile debba rimanere valida, chieda al programmatore di esplicitare il tempo di vita. Precisiamo che nel momento in cui più dati abbiano tempi di vita diversi, si intende che i dati debbano rimanere validi almeno per il tempo di vita più breve tra quelli ricevuti.

Discordo ancor più complicato per una funzione che ritorna a sua volta un riferimento, in quanto anche per tale riferimento è necessario conoscerne il tempo di vita. Per come detto prima, si assegna a tale rifeirmento il tempo di vita del dato più breve accettato come parametro, ma non è detto che questo sia sufficiente per i nostri scopi, ed è uno degli errori più frustranti che si ottengono all'interno del linguaggio.

Ad esempio:

```
fn f(s: &str, v: &mut Vec<&str>){
    v.push(s)
}
```

Il codice qui descritto genererebbe un errore. Qui andiamo a vincolare il tempo di vita dei due dati, `s` e `v`, senza fornire garanzie che fintanto che `v` avrà vita lo stesso varrà per `s`.

Per definire il tempo di vita di un riferimento in Rust si utilizzano le etichette, ovvero una serie di caratteri che identificano un tempo di vita della referenza, e si fa utilizzando un apice seguito dalla sequenza di caratteri. Ad esempio, `&'rlt var` indica che il tempo di vita di `var` è da chimarsi `rlt`, ma non si va ad esplicitarne la durata in quanto è dedotta dal compilatore.

Potremmo provare ad obbligare il compilatore a controllare che il tempo dei dua dati sia identico come:

```
fn f(s: &'a str, v: &'a mut Vec<&str>){
    v.push(s)
}
```

Qui otterremmo un errore subdolo. Infatti è vero che il dato `s` ha tempo di vita `'a`, identico al contenitore `v`, ma non abbiamo spiegato al compilatore che anche le altre stringhe contenute in `v` abbiano tempo di vita `'a`.

```
fn f(s: &'a str, v: &'a mut Vec<&'a str>){
    v.push(s)
}
```

Ora i tempi di vita dei dati sono congurenti, eppure non basta. Di fatto, abbiamo esplicato la vita delle variabili all'interno delle funzioni, ma non come si comportano al di fuori di essa, ovvero nel codice chiamante. Ad esempio:

```
fn main(){
    let mut v: Vec<&str> = Vec::new();
```

```
    {
        let s = String::from("ciao");
        v.push(&s)
    }
    println!("{:?}", v); // Qui l'errore
}
```

Se facessimo questo tipo di chiamata il compilatore si arresterebbe perchè `s` ha un tempo di vita minore di `v` e quindi la verifica di validità dei riferimenti non è soddisfatta.

Precisiamo che questi indicatori non sono un modo di complicarsi la vita ma un modo per spiegare al compilatore come ci attendiamo che vengano gestiti i dati a basso livello. Seppur omessi, il compilatore genera per sé questi riferimenti, e questo è solo un modo di esercitare un controllo su loro da parte del compilatore.

Quanto detto puo' essere esteso alle strutture, poichè anche in esse posso avere questa necessità. Questo perchè è possibile che le strutture a loro volta possano contenere dei riferimenti.

```
struct Utente<'a> {
    id : u32,
    nome: &'a str
}
```

Così facendo vincolo il tempo di vita di `Utente` al tempo di vita della `str` che essa contiene.

Passiamo ora a parlare delle chiusure, componente tipica dei linugaggi moderni e che quindi anche Rust implementa. Una funzione puo' esser vista a basso livello come essenzialmente un indirizzo di memoria che contiene del codice, quindi nel momento in cui lo si voglia eseguire è sufficiente spostare a tale indirizzo.

Nello stesso modo in cui distinguiamo come si utilizza una variabile dal tipo che essa ha, allo stesso modo distinguiamo l'uso di una funzione dalla sua firma, che altro non è che l'ordine ed il tipo di parametri che accetta assieme al valore che essa va eventualmente a ritornare.

Il linugaggio permette quindi di definire degli oggetti funzionali, puntatori a indirizzi di memoria che saranno predisposti ad un corretto funzionamento compatibilmente alla firma dichiarata, in quanto ad esempio bisogna assicurarsi di allocare abbastanza memoria all'indirizzo puntato per accogliere i parametri passati.

Vediamo un esempio:

```
fn oggetto_funzionale(i: i32, f: f64) -> f64 {
    return i as f64 * f;
}

let contenitore: fn(i32, f64) -> f64;
contenitore = oggetto_funzionale;
// ^ per qualche motivo OK anche se contenitore non è mutabile
contenitore(2, 3.14);
```

La reale utilità degli oggetti funzionali è che, a differenza delle funzioni, permettono di implementare funzioni dipendenti da uno stato, mentre una chiamata a funzione fornirà sempre lo stesso risultato dati gli stessi dati in ingresso, e sono pertanto senza stato.

Il loro reale punto debole è l'esser definite in un punto e utilizzate in un altro. Un'alternativa è rappresentata dalle funzioni definite in loco, anche dette lambda, e come gli oggetti funzionali anche loro sono assegnabili ma ci permettono di non preoccuparci della firma, che viene invece dedotta dal compilatore.

La loro caratteristica fondamentale è l'esecuzione di un meccanismo di cattura, che è funzionale all'implementazione delle chiusure, ovvero quando una funzione cambia il suo comportamento in base al contesto in cui viene definita. La cattura avviene per riferimento a meno che il movimento non sia esplicitamente richiesto. Se il riferimento deve anche essere mutabile, la lambda va dichiarata come mutabile e il borrow checker eseguirà i controlli anche su esso.

```
let f1 = |x| { x + y.f() + z }; // Per riferimento
let f2 = move |x| { x + y.f() + z }; // Per valore
```

Anche queste funzionalità sono ovviamente legate a dei tratti. I due più autoesplicativi sono `Fn` ed `FnMut`, che nel primo caso indica una funzione che cattura i parametri per riferimento e nel secondo per riferimento mutabile. L'ultimo caso, dal nome non così immediato, è `FnOnce`, che lascia al compilatore la scelta su come eseguire il passaggio dei parametri, se per movimento o per referenza, mutabile o meno.

Questo avviene in quanto `FnOnce` è supertratto di `Fn` ed `FnMut`, quindi una variabile che implementa uno di quei due tratti puo' essere usata in un contesto in cui ci si aspetta un `FnOnce`, ma per correttezza dobbiamo aspettarci che il le variabili usate in `FnOnce` siano sempre ottenute per movimento e che il contesto di esecuzione catturato sia consumato dopo la prima esecuzione, vale a dire come il nome suggerisce che quella funzione possa essere utilizzata una sola volta.