



UNIVERSIDAD AUTÓNOMA DE MANIZALES
TEORÍA Y TRADUCCIÓN DE LENGUAJES

ABSTRACCIONES PROCEDIMENTALES EN EL LENGUAJE RUST.
PROFESOR: MAURICIO FERNANDO ALBA CASTRO

REALIZADO POR:
DIEGO GEOVANNY MEDINA CARDENAS
JESUS DAVID CRUZ RONCALLO

MANIZALES, OCTUBRE 2019

Abstracciones Procedimentales En El Lenguaje Rust.

1. Abstracciones procedimentales (procedimientos y funciones)

que contempla el lenguaje Rust.

Funciones:

Rust permite la creación de funciones para abstraer el cálculo de expresiones, como por ejemplo:

```
fn main () {  
    let n = 5;  
    let x = factorial(n);  
    println!("factorial de {} es: {}", n, x);  
}  
  
fn factorial(n: u128) → u128  
{  
    if n == 0  
    {  
        1  
    }  
    else  
    {  
        let result = n * factorial(n-1); //c  
        result  
    }  
}
```

Imagen 1. Ejemplo Función factorial.

```
Compiling hello_cargo v0.1.0 (C:\Users\excelente\Documents\proj  
o_cargo)  
Finished dev [unoptimized + debuginfo] target(s) in 3.80s  
Running `target\debug\hello_cargo.exe`  
Factorial de 5 es: 120
```

Esta función permite obtener el factorial de un número de manera recursiva.

Procedimientos:

También se pueden crear procedimientos, por ejemplo:

```
fn main()  
{  
    let mut edad = 5;  
    actualizar_edad(&mut edad);  
    println!("{}", edad);  
}  
fn actualizar_edad(edad: &mut i32){  
    *edad = *edad + 1;  
}
```

Imagen 2. Ejemplo Procedimiento con abstracción.

```
Compiling hello_cargo v0.1.0 (C:\Users\excelente\Documents\  
o_cargo)  
Finished dev [unoptimized + debuginfo] target(s) in 0.80s  
Running `target\debug\hello_cargo.exe`  
6
```

Como se puede observar este procedimiento nos permite actualizar la variable edad pasada por referencia mutable.

2. Rust cumple con el principio de abstracción?

Rust si cumple con el principio de abstracción ya que podemos usar abstracciones de costo cero a través de varias herramientas diseñadas en rust por ejemplo: “ Los cierres e iteradores son características de Rust inspiradas en ideas de lenguaje de programación funcional. Contribuyen a la capacidad de Rust para expresar claramente ideas de alto nivel

con un rendimiento de bajo nivel. Las implementaciones de cierres e iteradores son tales que el rendimiento del tiempo de ejecución no se ve afectado. Esto es parte del objetivo de Rust de esforzarse por proporcionar abstracciones de costo cero”. [\[6\]](#)

También podemos crear abstracción de datos a través de las referencias y préstamos que nos ofrece Rust para hacer referencia a distintas variables creadas, existen dos tipos de referencias la mutable (&mut) y las inmutables (&) más adelante se explicaran estos tipos de referencias y préstamos. Por ahora se plantea un ejemplo de abstracción de acceso de variables:

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

6

Imagen 3. Ejemplo abstracción de acceso a variable. Tomado de [\[7\]](#).

Como podemos ver se accede a la variable x a través de una referencia mutable que se guarda en la variable local y, algunas consideraciones que se deben tener en cuenta es que los “{}” se deben a una de las normas de las referencias mutables, que se refiere a que su alcance debe terminarse para devolver su referencia a x, además de esto debemos indicar “*” para poder acceder al valor de la referencia.

3. Mecanismos de copia de parámetros que contempla Rust.

Rust está basado en C y C++, y estos lenguajes solo soportan el mecanismo copy-in parameters o value parameter (Programming Language Design Concepts, pág.124) [\[2\]](#), entonces Rust también comparte esta idea, y permite que la copia sea por copy-in parameters, o parámetros por valor, además también Rust maneja parámetros por referencia(&).

Ejemplo:

```
fn main() {  
    let n_main: usize = 100;  
    println!("{}", inc(n_main));  
}  
  
fn inc(n_inc: usize) -> usize {  
    n_inc + 1  
}
```

Imagen 4. Ejemplo mecanismo copy-in parameters. Tomado de [\[1\]](#).

En el ejemplo podemos ver como a la función inc, se le pasa como parámetro n_Main, el cual pasa como valor, y nos retorna el valor de 101.

4. Mecanismos de parámetros por referencia que contempla Rust.

Como se mencionó anteriormente Rust también maneja parámetros por referencia. [\[3\]](#), y al igual que C++, soporta el mecanismo variable parameters directly (Programming Language Design Concepts, pág.126) [\[2\]](#). Como se puede ver en el siguiente ejemplo.

Ejemplo:

```
fn main() {  
    let mut s = String::from("hello");  
  
    change(&mut s);  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world");  
}
```

Imagen 5. Ejemplo del mecanismo de parámetros por referencia, variable parameter. Tomado de Rust Reference. [\[3\]](#)

```
1 ▾ fn main() {  
2     let mut s = String::from("hello");  
3     println!("Antes: {}",s);  
4     change(&mut s);  
5     println!("Luego: {}",s);  
6  
7  
8 }  
9  
10 ▾ fn change(some_string: &mut String) {  
11     some_string.push_str(", world");  
12 }
```

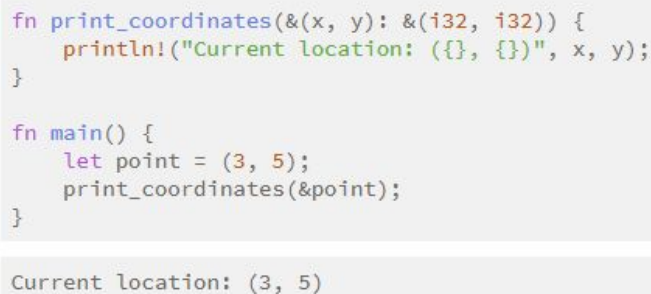
Imagen 6. Ejemplo mecanismo parámetro por referencia.

Se agrega unos println al ejemplo(Imagen 5), para mostrar los cambios y nos da el siguiente resultado:

```
Antes: hello  
Luego: hello, world
```

Imagen 7. Resultado del ejemplo.

Con este ejemplo se puede ver, como en Rust se maneja el mecanismo de parámetros por referencia, variable parameters, enviando una variable como parámetro referenciado, la cual puede modificar el valor inicial, al tener la referencia de este. Además de este mecanismo también se puede dar el caso del mecanismo de un “constant parameter”, cuando la variable no es mutable(mut), porque se podría acceder a la variable pero no modificarla. Como se puede ver en el siguiente ejemplo:

The image shows a code editor with Rust code. The code defines a function `print_coordinates` that takes a tuple `(x, y)` by reference and prints it. The `main` function creates a `point` tuple `(3, 5)` and calls `print_coordinates` with its reference. Below the code, the output shows `Current location: (3, 5)`.

```
fn print_coordinates(&(x, y): &(i32, i32)) {  
    println!("Current location: ({} , {})", x, y);  
}  
  
fn main() {  
    let point = (3, 5);  
    print_coordinates(&point);  
}
```

Current location: (3, 5)

Listing 18-7: A function with parameters that destructure a tuple

Imagen 88. Ejemplo mecanismo de constant parameter. Tomado de Rust Reference.[\[4\]](#)

El mecanismo de parámetros por referencia, variable parameters, se presenta también en los métodos, cuando se accede a una variable por fuera del bloque de este, y debido a que esté siempre su primer parámetro será `self`[\[5\]](#), podrá con este acceder a la instancia donde es llamado.

5. Rust cumple con el principio de correspondencia?

Según lo que establece el principio de Correspondencia: “For each form of declaration there exists a corresponding parameter mechanism”(*Programming Language Design Concepts*, pág.128)[\[2\]](#).

Luego de haber visto los mecanismo de parámetros en los puntos 3 y 4, y ver que para cada forma de declaración existe un mecanismo o dos, como por ejemplo la referencia con el “&” para el mecanismo de referencia Constant parameter o el “`self`” en los métodos o con el “`mut + &`” para el mecanismo de referencia Variable parameter , se puede concluir que se cumple en cierto grado este principio, ya que puede existir casos en los que no sea así por la sintaxis o ambiente y también por no presentar los otros mecanismos. También se puede presentar el mismo caso que en C++, como lo mencionan en la pág. 129 del libro *Programming Language Design Concepts*.[\[2\]](#)

Referencias

- [1] Ryan Levick, R. L. (2018, 9 diciembre). Rust: Pass-By-Value or Pass-By-Reference? 🍌 - You Learn Something New Everyday 🧠.
Recuperado 27 octubre, 2019, de <https://blog.ryanlevick.com/posts/rust-pass-value-or-reference/>.
- [2] .Watt, D. A. (2004). Programming Language Design Concepts (2ª ed.). Southern Gate, Chichester, England: Wiley.
- [3] Rust. (2018). References and Borrowing - The Rust Programming Language.
Recuperado 27 octubre, 2019, de <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html?highlight=refe#mutable-references>.
- [4] Rust. (2018). All the Places Patterns Can Be Used(Function Parameters) - The Rust Programming Language. Recuperado 27 octubre, 2019, de <https://doc.rust-lang.org/book/ch18-01-all-the-places-for-patterns.html#function-parameters>.
- [5] Rust. (2018e). Method Syntax - The Rust Programming Language. Recuperado 27 octubre, 2019, de <https://doc.rust-lang.org/book/ch05-03-method-syntax.html>.
-

- [6] Rust. (2018e). Summary of Comparing Performance: Loops vs. Iterators - The Rust Programming Language. Recuperado 27 octubre, 2019, de <https://doc.rust-lang.org/book/ch13-04-performance.html>
<https://doc.rust-lang.org/1.30.0/book/first-edition/references-and-borrowing.html>
- [7] Rust. (2018e). References and borrowing - The Rust Programming Language. Recuperado 27 octubre, 2019, de <https://doc.rust-lang.org/1.30.0/book/first-edition/references-and-borrowing.html>
-