Instituto de Ciencias de la Educación

Introducción a la programación funcional

(Parte II)

Santiago Higuera de Frutos

Universidad Politécnica de Madrid

(Versión de fecha 19 de mayo de 2024)

A los alumnos y profesores de la Universidad Politécnica de Madrid y al oficio de profesor, que tan pocos reconocimientos recibe, a pesar de su importancia en la creación de una sociedad mejor.
Santiago

CONTENIDO

Prólogo

La programación funcional no es nueva. El lenguaje LISP (LISt Processing) está considerado uno de los primeros lenguajes de alto nivel que se desarrollaron, junto a Fortran y Cobol. Durante mucho tiempo se ha considerado un paradigma de programación demasiado teórico y alejado de la resolución de los problemas habituales en la programación real.

Durante ese tiempo, la Programación Orientada a Objetos (POO) se consideraba el paradigma base necesario para la programación en cualquier lenguaje. Con el tiempo, la POO ha mostrado algunas de sus debilidades, originadas por el mecanismo de la herencia entre tipos. Ya en 1994 "*The gang of four*", en su famoso libro "*Design Patterns: Elements of Reusable Object-Oriented Software*", indicaban que había que primar la composición frente a la herencia al construir los programas [1].

Los principios que rigen la programación funcional son muy simples. En los últimos años la programación funcional ha adquirido especial relevancia. Todos los lenguajes principales han ido incorporando algunos de los elementos que caracterizan a la programación funcional, como los iteradores, las funciones de primera clase, las closures, la inmutabilidad de variables u otros. Es posible escribir programas con un estilo funcional casi en cualquier lenguaje.

Rust es un lenguaje de programación con solo unos años de vida pero que está teniendo un crecimiento muy importante debido la seguridad del manejo de memoria que proporciona, a la velocidad de ejecución del código generado, a la calidad de las herramientas que proporciona y a las buenas sensaciones que proporciona al programar. Rust no es un lenguaje funcional estricto, pero proporciona suficientes elementos para poder ser utilizado como tal.

A lo largo del curso, se va a hacer una introducción a la utilización del lenguaje Rust y a los conceptos que sustentan el paradigma de la programación funcional. Se utilizará principalmente el lenguaje Rust, aunque también se mostrarán ejemplos en otros lenguajes, como Java, C o Javascript.

Al final del curso, el alumno comprenderá los conceptos que subyacen bajo el paradigma de la programación funcional, de forma que podrá utilizar algunos de ellos en el lenguaje que utilice habitualmente. Ademas, conocerá los conceptos fundamentales de la programación con Rust, lo que le permitirá adentrarse en el lenguaje, si lo considera de interés.

Acerca del autor

Santiago Higuera de Frutos

Doctor Ingeniero de Caminos por la Universidad Politécnica de Madrid (UPM). Ha sido profesor en la Escuela de Ingenieros de Caminos y actualmente en Teleco Campus Sur. Es autor de los libros "*Programacion en Rust*" y "Programación y métodos numéricos para Ingeniería con MATLAB y Octave", publicados por la Editorial Garceta, así como de numerosos artículos y conferencias sobre programación, geometría de las carreteras y simuladores de conducción, todo ello en el ámbito del software de código abierto.

CAPÍTULO 1

Funciones de orden superior, closures e iteradores

En matemáticas y en ciencias de la computación, se denominan funciones de orden superior a las que cumplen al menos una de las siguientes condiciones:

- *Tomar una o más funciones como parámetro de entrada.*
- *Devolver una función como salida.*

En matemáticas, un ejemplo lo podría constituir la función derivada, que toma una función como parámetro y devuelve otra función.

En programación funcional, las funciones son consideradas elementos de primera clase, esto es, pueden usarse como argumentos de otras funciones, como valores devueltos y también asignarse a un símbolo o variable. La mayoría de los lenguajes han adoptado ya esta premisa y dan un tratamiento de primera clase a las funciones.

En este capítulo se va a hablar de funciones, de cómo asignarlas a variables y de cómo utilizarlas como argumento o como valor devuelto de otras funciones.

También se hablará de las closures, que son un tipo especial de funciones definidas dentro de otras funciones, que pueden acceder al contexto en el que se han definido.

Por último, se hablará de los iteradores y cómo encadenar procesos utilizando iteradores y funciones de orden superior.

1.1 Funciones de primera clase

En programación, se denominan *elementos de primera clase* a los elementos del lenguaje que se pueden asignar a variables y se pueden utilizar como argumentos o como valor devuelto por otras funciones.

No todos los elementos del lenguaje son elementos de primera clase. Por ejemplo, en muchos lenguajes, los operadores aritméticos como el $+$, el $-$ y otros, no son de primera clase. También sucede algo parecido con algunas construcciones, como los bucles o las bifurcaciones.

Es importante entender la diferencia entre el concepto de *declaración* y el de *expresión*. Un elemento de un lenguaje se considera que es una *declaración*

cuando no devuelve ningún valor; una *expresión*, en cambio, es un elemento que devuelve un valor.

En los lenguajes funcionales se da prioridad a las expresiones que devuelven valores. Así se hace en Rust. Algunas construcciones que en otros lenguajes son declaraciones, en Rust son expresiones. Es el caso de las bifurcaciones *if* o de los bucles *loop, for* o *while*, como se indicó en los apartados **??** y **??**. En cambio, el operador *let* es una declaración que no devuelve ningún valor.

Se podría realizar un código como el siguiente:

```
fn main() {
   let x = 2;
   println!("{}", suma(if x==2 {3} else {4}, 5));

   let mut n = 1;
   println!("{}", loop {
      n=n+1;
      if n>3 {
         break 10
      }
   });
}
fn suma(x: i32, y: i32) -> i32 {
   x+y
}
```

En el programa anterior, se utiliza una bifurcación *if* como argumento de la función *suma()* y un bucle *loop* como argumento de la macro *println!()*. Observe que, en realidad, el argumento es el valor que devuelven dichas expresiones.

Actualmente, numerosos lenguajes se han adherido a la línea marcada por los lenguajes funcionales y consideran las funciones como elementos de primera clase, esto es, permiten que las funciones se puedan asignar a variables y se puedan utilizar como parámetros o como valor devuelto por otras funciones. En Rust, las funciones son objetos de primera clase.

Se denominan *funciones de primer orden* o también *funciones ordinarias* aquellas en las que ni los parámetros ni el valor devuelto son otras funciones. Por contra, las *funciones de orden superior* son aquellas en las que o bien alguno de los parámetros o bien el valor devuelto son una función.

Hay varias maneras de definir funciones en Rust:

- Dentro de un módulo o fichero *.rs*, con la visibilidad que le corresponda y acceso a otras funciones y a sus variables locales .
- Dentro de otra función o bloque de código, con visibilidad restringida a dicho bloque y acceso a otras funciones, pero no a variables del contexto del bloque en el que se ha definido.
- Cómo función asociada a un tipo de datos personalizado, como es el caso de las estructuras y las enumeraciones.
- Como closure definida dentro de otra función o bloque de código. Su visibilidad estará restringida al bloque en el que se ha definido, pero las closures sí que tienen acceso a las variables existentes en el contexto del bloque en el que se haya definido.

Ya se comentó este tema en el Apartado **??**. También se habló allí de los tres tipos de funciones que considera Rust, en función del uso que hacen de la propiedad de los parámetros: el tipo *Fn*, para funciones ordinarias, el tipo *FnMut*, para funciones con parámetros mutables y el tipo *FnOnce* para funciones que solo se pueden invocar una vez.

1.1.1 Asignación de funciones a variables

Cualquiera de los tipos de funciones presentes en Rust se pueden asignar a una variable. Luego, se puede utilizar la variable como si fuera la función.

El siguiente código asigna a una variable de nombre *raiz2* la función *sqrt* de los números *f64*. Como resultado, la variable *raiz2* es del tipo *fn(f64) ->f64*, esto es, una función que recibe un argumento del tipo *f64* y devuelve un valor *f64*. Observe la forma de calcular la raíz cuadrada utilizando la variable así definida:

```
fn main() {
   let raiz2  = f64::sqrt;
   let y = raiz2(4.0); // Equivale a 4.0f64.sqrt()
   println!("{}", y);
}
```

Se podría asignar una función previamente codificada, como se hace en el siguiente código:

```
fn main() {
   let f = saludo;
   f(); // Imprime Hola, ¿qué tal?
}
fn saludo() {
   println!("Hola, ¿qué tal?");
}
```

En este caso, la variable *f* es del tipo *fn()*, esto es, del tipo de las funciones que no reciben argumentos y no devuelven ningún valor.

Las closures también se pueden asignar a variables, como se hace en el siguiente ejemplo:

```
fn main() {
   let cuadrado = |x| x*x;
   println!("{}", cuadrado(3.0)); // Imprime 9
}
```

Este tipo de funciones que no se definen dentro de un bloque *fn* con nombre, se denominan *funciones anónimas*. En este caso, la función anónima se ha asignado a la variable *cuadrado*.

También se suele decir que la closure se ha codificado *inline*. El término *inline* se refiere a que la codificación de la función se ha hecho dentro de la línea de código de otra instrucción. Es habitual también codificar closures *inline* cuando se utilizan como parámetros de otras funciones.

1.1.2 Utilización de funciones como parámetros de otras funciones

Observe el código siguiente:

```
fn main() {
   let pi = std::f64::consts::PI;
   println!("{}", operador(f64::sin, pi/2.0)); // Imprime 1
   println!("{}", operador(f64::cos, pi)); // Imprime -1
   println!("{}", operador(|x| x*x, 4.0)); // Imprime 16
   println!("{}", operador(duplica, 5.0)); // Imprime 10
}
fn operador(f:fn(f64) -> f64, x: f64) -> f64 {
   f(x)
}
fn duplica(x: f64) -> f64 {
   x*2.0
}
```

En el código anterior, la función *operador* es una función de orden superior que se ha definido con la siguiente signatura:

```
fn operador(f:fn(f64) ->f64, x: f64) ->f64
```

Cualquier función con un único parámetro del tipo *f64* y que devuelve un valor del tipo *f64*, se podrá utilizar como argumento en la llamada a la función *operador()*. La función *operador()* devolverá el resultado de aplicar la función *f* que recibe como primer argumento al valor *x* del tipo *f64* que recibe como segundo argumento.

En algunos lenguajes, a la función que se pasa como argumento, para que la función de orden superior la invoque, se le denomina *función de callback*.

1.1.3 Uso de funciones como valor devuelto por otras funciones

En el siguiente código se define una función de orden superior, llamada *operador*, que recibe un número entero como argumento. Si el valor recibido es mayor que cero, *operador()* devuelve la función *f64::sin*; en caso contrario, devuelve la función *f64::cos*.

```
fn main() {
   let pi = std::f64::consts::PI;
   let seno = operador(3);
   println!("{}", seno(pi/2.0)); // Imprime 1
   let coseno = operador(-2);
   println!("{}", coseno(pi)); // Imprime -1
}
fn operador(n: i32) -> fn(f64) -> f64 {
   if n>0 {f64::sin} else {f64::cos}
}
```

Otra forma de indicar el tipo de función que se pasa como parámetro o que se devuelve como resultado es utilizar la cláusula *impl* y el nombre del *Trait*, que en este caso es parecido pero poniendo *Fn* en mayúsculas:

```
fn operador(n: i32) -> impl Fn(f64) ->f64
```

1.2 Errores de ejecución

El mecanismo de excepciones es la forma habitual que ofrecen muchos lenguajes para gestionar los errores que se producen durante la ejecución de los programas. Pueden ser errores debidos a la entrada de un dato incorrecto, al intento de acceso a un fichero que no existe, el intento de acceder a una página web sin tener conexión u otros tipos de errores de ejecución. En muchas ocasiones, el error de ejecución proviene de la existencia de un puntero nulo en alguna parte del programa.

El invento del puntero nulo se atribuye a Tony Hoare, durante la implementación de los tipos de datos para el lenguaje ALGOL W, a mediados de la decada de 1960. Años después, él mismo lo llamó *el error de los mil millones de dólares*[1]:

> *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

[1] Esta cita se ha tomado de la Wikipedia

En Rust no existe el puntero *null*. Tampoco existe un mecanismo de excepciones para el tratamiento de los errores de ejecución. En Rust se distingue entre dos tipos de errores de ejecución:

- *Errores recuperables*: por ejemplo, se pide al usuario el nombre del fichero que se pretende abrir y se comprueba que dicho fichero no existe. En esos casos, lo que se puede hacer es informar del error al usuario y volver a pedirle el nombre del fichero.
- *Errores no recuperables:* por ejemplo, se necesita acceder a un dispositivo y el dispositivo no existe. En estos casos se interrumpe la ejecución del programa, quizás imprimiendo un mensaje que informa al usuario del error que se ha producido.

En Rust, cuando el error da lugar a la interrupción del programa se suele decir que el programa *ha entrado en pánico*, por analogía con el nombre de la macro *panic!()* que se suele utilizar para forzar interrupciones.

La macro *panic!()* recibe un mensaje de texto como argumento, interrumpe el programa y muestra en la consola dicho mensaje junto con otras informaciones.

Puede probar el siguiente código:

```
fn main() {
    panic!("¡Programa interrumpido!")
}
```

Para gestionar los errores recuperables se suelen utilizar las enumeraciones *Option<T>* y *Result<T>*, en la forma que se va a explicar en los siguientes apartados.

1.3 El *enum Option*

La enumeración *Option* tiene la siguiente declaración:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

El tipo *T* puede ser cualquiera. Es fácil definir un valor del tipo *Option*. Por ejemplo, el siguiente código declara dos variables del tipo *Option<u32>*, una con cierto valor y otra sin valor:

```
let n = Some(3u32);
let m: Option<u32> = None;
```

Observe que, en el caso de la variable *n* que sí tiene valor, el tipo de datos viene definido por el valor en sí, mientras que en el caso de la variable *m* hay que especificr el tipo de datos del *Option*.

El tipo *Option<T>* se utiliza para encapsular un valor que puede no existir. Procede de los lenguajes funcionales y del concepto de *mónada*. En estos lenguajes es habitual encontrar el tipo *Maybe* (“tal vez un”), representando un valor que es o un valor del tipo *T*, o ninguno.

Los métodos *is_some()* e *is_none()* permiten saber si determinada variables del tipo *Option* es la variante *Some* o la variante *None*:

```
fn main() {
   let n = Some(3);
   let m: Option<u32> = None;
   assert!(n.is_some());
   assert!(m.is_none());
}
```

Cuando la variante de la variable es *Some*, se puede extraer su valor con el método *unwrap()*:

```
let n = Some(3);
assert_eq!(n.unwrap(), 3);
```

La función *unwrap()* interrumpe la ejecución del programa con *panic!*, si la variante es *None*. Por ello, se suele utilizar más el método *unwrap_or()*, que proporciona un valor del mismo tipo que el contenido en el *Option* como valor devuelto en el caso de que la variante sea *None*:

```
assert_eq!(Some("car").unwrap_or("bike"), "car");
assert_eq!(None.unwrap_or("bike"), "bike");
```

También existen unos métodos *unwrap_or_else()* y *unwrap_or_default()*, que puede consultar en la documentación del tipo *Option*.

Es habitual utilizar el tipo *Option* como resultados de funciones. Suponga una función que reciba dos parámetros enteros y devuelva la división de uno entre el otro. En Java, la solución podría ser la siguiente:

```
static int dividir(int num, int den) {
   return num/den;
}
```

En Java, si el denominador es cero, se lanzará una excepción y se interrumpirá el programa. En principio hay dos posibilidades para gestionar una función de ese tipo: capturar la excepción o interceptar los denominadores cero. [2].

Como se ha dicho, Rust no dispone de tipo *null* ni de ningún mecanismo de excepciones para la gestión de errores. En Rust, en cambio, se puede imponer que la función *dividir()* devuelva un *Option<i32>* en lugar de un valor entero. El código podría ser el siguiente:

```
fn main() {
   let n = dividir(3, 0);
   assert_eq!(n, None);
}
fn dividir(num: i32, den: i32) -> Option<i32> {
   match den {
      0 => None,
      _ => Some(num/den)
   }
}
```

Una ventaja de que la función devuelva un *Option* es que el programa está obligado a gestionar la posibilidad de que el valor devuelto no exista. En el caso

[2]No se podría hacer que la función devuelva *Infinity* pues no existe para valores del tipo *int*.

del programa Java, nada indica en la signatura de la función ni en el cuerpo de la misma que cabe la posibilidad de que se genere una excepción. Ello podría dar lugar a errores de ejecución no previstos. En el caso resuelto con Rust, el programa que llama a la función sabe que puede obtener como resultado la variante *None* y el compilador le obliga a gestionar los dos posibles resultados[3].

La ventaja de devolver resultados de funciones con *Option* es más evidente cuando hay que encadenar varias funciones en las que una o más de ellas pueden devolver resultados no válidos. En el Apartado 1.6 se mostrarán ejemplos de encadenamiento de funciones con iteradores.

1.4 Gestión de errores con *Result*

Como se ha mencionado, en *Rust* no existe el mecanismo de excepciones que se utiliza en otros lenguajes para resolver las situaciones en las que una operación pueda dar lugar a un error de ejecución. Tampoco existe un valor *null*. El módulo *result* de la librería *std* de *Rust* proporciona la enumeración *std::result::Result* para gestionar dichas situaciones. Cuando una operación pueda dar lugar a un resultado correcto o erróneo, lo que se hace es devolver una instancia de *Result*.

La enumeración *std::result::Result* se define de la siguiente manera:

```
pub enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

Las dos variantes de esta enumeración son:

- *Ok(T)*: representa el éxito de una operación y proporciona un valor del tipo de datos genérico *T*.
- *Err(E)*: representa que se ha producido algún error en la operación y proporciona un valor del tipo de datos genérico *E*.

Se va a desarrollar a continuación una función para dividir dos números del tipo *f64*. Para resolver la situación de que el divisor sea cero, la función devuelve un valor de la enumeración *Result* que se acaba de describir:

[3] Java ofrece también un tipo *Optional<T>* que, junto a otros tipos, permite acometer éste y otros aspectos de la programación funcional.

```
fn div(x: i32, y: i32) -> Result<i32, ()> {
   match y {
      0 => Err(()),
      _ => Ok(x / y)
   }
}
fn main() {
   let d = div(10, 0);
   match d {
      Ok(value) => println!("{}", value),
      Err(()) => println!("Error, denominador cero")
   }
}
```

Como se verá en el apartado 1.5, otra forma de gestionar los errores devueltos en un *Result* es utilizar el operador *?* para propagarlos hacia arriba, hacia la función que llama.

Un problema asociado a los lenguajes que utilizan el mecanismo de excepciones es que se pueden capturar las excepciones e ignorarlas. El tipo *Result* está definido con la anotación *#[must_use]*, que indica al compilador que se emita un mensaje de aviso si se ignora el resultado de una función que devuelva un *Result*. El lector puede probar a sustituir el código de la función *main()* del ejemplo anterior por el siguiente:

```
fn main() {
   div(10.0, 0.0);
}
```

Si ahora trata de compilar el programa, el compilador emitirá un aviso de que el *Result* que devuelve la función no se ha utilizado.

El lector puede comprobar también que, si el resultado de la función *div()* se asigna a una variable, no se genera el aviso *must_use*. Pruebe a compilar el siguiente código:

```
fn main() {
   let _x = div(10.0, 0.0);
}
```

En el código anterior, el nombre de la variable se ha iniciado con un guion bajo. Si no se hace así, el compilador genera otro aviso, el de que la variable *x* no se utiliza. Es una demostración más de las numerosas comprobaciones que hace el compilador de *Rust*.

1.5 El operador *?*

Una manera habitual de tratar el objeto *Result* que devuelven algunos métodos es mediante una construcción *match*, como se ha visto en el apartado anterior. Dentro de una función que devuelva un *Result* o un *Option*, se puede sustituir dicha construcción por el operador *?*, escrito a continuación de la función que devuelve el *Result*. Con el operador *?*, si el resultado es *Ok*, el operador *?* hace el *unwrap()*; si el resultado es *Err*, se vuelve de la función en la que esté, devolviendo el error. De esta manera, se deja la gestión del error para la función que llamó. Si se está dentro de *main()*, se devuelve al sistema operativo. Este mecanismo se denomina *propagación de los errores hacia arriba*. El siguiente ejemplo utiliza el operador *?*:

```
fn main() -> Result<(), ()> {
   let _d = div(10.0, 0.0)?;
   Ok(())
}
```

Como se ve en el código anterior, ahora la función *main()* tiene que devolver un *Result*. Si se ejecuta el programa anterior, la división genera un error, que se muestra en el sistema operativo como *Error: ()*. En el ejemplo, si no hay error, se devuelve la variante *Ok()* de *Result*.

1.6 Iteradores

Un *Iterador* es un objeto que permite recorrer ordenadamente los elementos de una fuente de datos. En muchas ocasiones, la fuente de datos es una colección de objetos, pero también se pueden utilizar iteradores para leer ficheros línea a línea o para leer streams procedentes de un puerto de comunicaciones, por ejemplo.

Cuando un tipo de datos implementa el *trait Iterator*, se convierte en una especie de colección, que es posible recorrer de manera ordenada. Por ello, es habitual referirse a los elementos de dicho tipo de datos como *los elementos del iterador*, como si se tratase de una colección.

El uso de iteradores es un patrón de diseño muy utilizado en la programación funcional y que se ha incorporado progresivamente en todos los lenguajes de programación. Su origen se marca en el lenguaje CLU del año 1973 [2].

Técnicamente, un iterador en Rust es un tipo de datos que implementa el *trait Iterator*, que impone la existencia del método *next()*. Este método, si hay más elementos por iterar, devuelve *Some(E)*, donde *E* es el tipo de datos del iterador; si no quedan más elementos por iterar, devuelve *None*.

Las colecciones que ofrece el lenguaje Rust —arrays, vectores, hashmaps y otras— permiten obtener un iterador sobre los elementos de la colección mediante el método *into_iter()*, o bien un iterador sobre referencias a los elementos de la colección utilizando el método *iter()*. La diferencia fundamental entre ambos métodos es que, si se utiliza el método *into_iter()*, se *consume* la colección, pasando la propiedad al proceso que la esté utilizando; el método *iter()*, en cambio, utiliza referencias y no *consume* la propiedad de la colección.

El siguiente ejemplo utiliza el método *iter()* con un vector. Tras recorrer el vector, los elementos de la colección siguen estando disponibles:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.iter() {
      println!("{}", num);
   }
   println!("{}", v[0]);
}
```

En cambio, si en el ejemplo anterior se utiliza el método *into_iter()*, se consume el vector en el bucle y los elementos de la colección ya no estarán accesibles después de la ejecución del mismo:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.into_iter() { // into_iter() mueve la propiedad
      println!("{}", num);
   }
   println!("{}", v[0]); // Error, v se movió al bucle
}
```

Es posible utilizar un bucle *for* para recorrer una colección sin llamar explícitamente a ninguno de los dos métodos. El problema será similar: si se recorre la colección, se pasa la propiedad al bucle, mientras que si se recorre utilizando una referencia, la colección no se consume en el bucle.

El siguiente ejemplo utiliza una referencia a la colección y los elementos del vector siguen disponibles tras finalizar el bucle:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   for num in &v { // Se utiliza una referencia &v
      println!("{}", num);
   }
   println!("{}", v[0]); // v sigue disponible
}
```

En cambio, en el siguiente ejemplo, el bucle utiliza directamente la colección y consume la propiedad de la misma. Al finalizar el bucle, no se puede acceder a los elementos de la colección:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    // Se recorre la colección, se pasa la la propiedad de v
    for num in v {
        println!("{}", num);
    }
    // Ahora v ya no está disponible
    println!("{}", v[0]); // Error
}
```

En Rust, al definir los iteradores, se hace referencia al tipo de elementos que recorre, pero no al tipo de colección o a la fuente de los datos que los alberga. Ésto permite definir funciones que describen la intención de lo que se quiere conseguir al procesar una fuente de datos, de manera independiente a la forma que adopten dichos datos. De esta forma, se podría cambiar la fuente de datos o su forma interna de organización, manteniendo las funciones que procesan dichos datos. Se trata de una codificación más declarativa que los bucles.

Los iteradores proporcionan numerosos métodos que permiten operar con los elementos que se iteran. Básicamente hay dos tipos de métodos:

- Métodos que devuelven un nuevo iterador obtenido al realizar alguna operación sobre los elementos del iterador original. Se suelen llamar *adaptadores* o también *iteradores internos*. Ejemplos de estos métodos serían *map()* y *filter()*.
- Métodos que realizan alguna operación directamente sobre los elementos del iterador. Se suelen denominar *iteradores externos*. Un ejemplo de estos métodos sería *for_each()*.

Los métodos adaptadores permiten encadenar operaciones haciendo que la salida de cada uno ellos, sea la entrada del siguiente. Ésto favorece la *canalización del procesamiento* de los datos (*processing pipeline*). La codificación obtenida es declarativa y muestra de manera explícita qué se quiere hacer con los datos, en contra de lo que sucede al realizar el procesamiento a base de bucles.

La totalidad de los métodos que proporcionan los iteradores se pueden consultar en la documentación de la librería estándar, en la siguiente dirección:

`https://doc.rust-lang.org/std/iter/trait.Iterator.html`

Una característica de los iteradores en Rust es que actúan de manera perezo-

sa, esto es, las operaciones asociadas a sus métodos no se ejecutan hasta que son necesarias. En concreto, es habitual tras una serie de operaciones encadenadas utilizando los métodos del iterador, querer convertir el iterador resultante en algún tipo de colección, como un vector u otra. Para ello, se suele utilizar el método *collect()*, que permite obtener una colección a partir de un iterador. En ese momento se ejecutarán las operaciones. En los ejemplos que siguen se verá cómo utilizar el método *collect()* en diferentes situaciones.

Muchos de los métodos que se utilizan cuando se trabaja con iteradores son más o menos estándar y están disponibles en todos los lenguajes que hacen uso de iteradores, si bien, en algunos casos pueden utilizar nombres diferentes.

Los métodos de los iteradores es frecuente que sean ejemplos paradigmáticos de funciones de orden superior: reciben una función como argumento que es la que se utilizará para operar sobre los elementos del iterador. También es frecuente que la función que se pasa como argumento sea una closure.

Tres de estos métodos que podríamos decir que son estándar en cualquier lenguaje que utilice iteradores son los métodos *map*, *filter* y *fold*:

- *map()*: este método recibe como argumento una función que se aplica sobre cada uno de los elementos del iterador. El método *map* devuelve un nuevo iterador cuyos elementos no tienen por qué ser del mismo tipo que los del iterador original.
- *filter()*: recibe como argumento una función que al aplicarla a los elementos del iterador devuelve *true* o *false*. A esta función que recibe como argumento se le llama el *filtro*. El método *filter* devuelve un nuevo iterador con los elementos del iterador original en los que el filtro devuelve *true*.
- *fold()*: este método acumula los valores del iterador en un valor que puede ser de otro tipo diferente que el de los elementos del iterador. También utiliza una función como argumento. Es posible encontrar este método en otros lenguajes con el nombre *reduce*. En Rust también hay un método *reduce*, similar a *fold()*.

En los siguientes apartados se van a tratar en profundidad estos tres métodos.

1.7 El método *map()*

Como se ha dicho, el método *map()* recibe como argumento una función y la aplica a los elementos del iterador para obtener un nuevo iterador. La función que se pasa como argumento a *map()*, toma como argumento un elemento del

iterador original y devuelve otro elemento, que puede ser del mismo tipo o de otro. El resultado final es un nuevo iterador con elementos que no tienen por qué ser del mismo tipo que los del iterador original. La Figura 1.1 muestra el esquema de funcionamiento de *map()*.

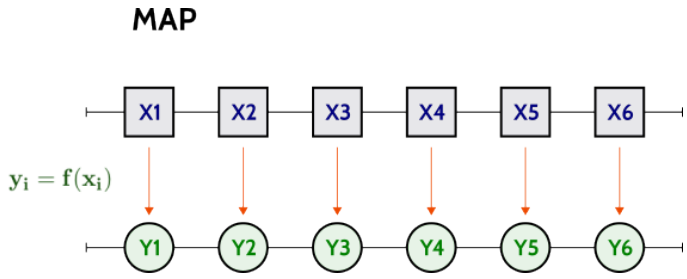

Figura 1.1: Esquema de funcionamiento de *map()*

En el siguiente código se crea un vector *v* de números enteros y se obtiene un nuevo vector formado por los cuadrados de dichos números.

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let iterador_original = v.into_iter();
   let new_iterador = iterador_original.map(|x| x*x);
   let new_vector: Vec<i32> = new_iterador.collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Observe los pasos que se han seguido:

1. Se obtiene un iterador a partir del vector *v* utilizando el método *into_iter()*.
2. Se obtiene un nuevo iterador aplicando el método *map* al iterador original. El método *map* recibe como argumento una closure que, para cada valor *x* del iterador original, devuelve su cuadrado.
3. El iterador resultante de la aplicación de *map()* se convierte de nuevo en un vector utilizando el método *collect()*. Al método *collect* hay que indicarle el tipo de colección que se quiere obtener, que en este caso es *Vec<i32>*.

Aunque en el ejemplo se ha desglosado cada paso para hacerlo más didáctico, lo habitual es hacer todas las operaciones encadenadas. También es habitual

ir poniendo las sucesivas operaciones en diferentes líneas, utilizando la tabuladores para dar claridad al código. El siguiente ejemplo es equivalente al anterior:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let new_vector: Vec<i32> = v.into_iter()
   .map(|x| x*x)
   .collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Como se ha dicho, el iterador resultante de la aplicación de *map()* puede tener elementos de distinto tipo que el iterador original. En el siguiente ejemplo, el iterador original es de cadenas de caracteres y se obtiene un iterador con el número de caracteres de cada una de esas cadenas:

```
fn main() {
   let v = vec!["Uno", "Dos", "Tres", "Cuatro", "Cinco"];

   let new_vector: Vec<i32> = v.into_iter()
   .map(|x| x.len() as i32)
   .collect();

   println!("{:?}", new_vector); // Imprime [3, 3, 4, 6, 5]
}
```

1.8 El método *filter()*

El método *filter()* recibe como argumento un función de retorno lógico: *true* o *false*. El parámetro de la función es del tipo de los elementos del iterador. El resultado es un nuevo iterador formado por los elementos del iterador original que *pasan el filtro*, esto es, los elementos en los que la función filtro devuelve *true*. La Figura 1.2 esquematiza el funcionamiento de *filter()*.

El siguiente ejemplo parte de un vector de tuplas con las de dos coordenadas *f64* de una serie de puntos. Se aplica un filtro para extraer los puntos del vector

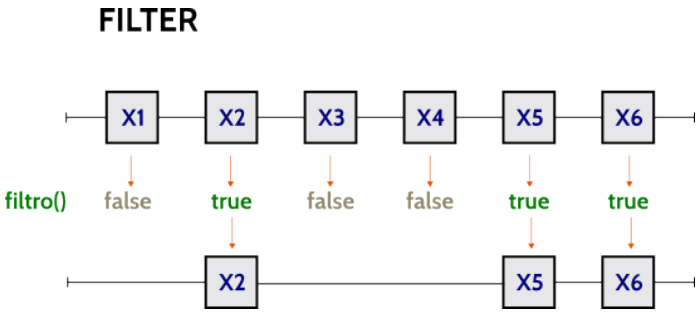

Figura 1.2: Esquema de funcionamiento de la función *filter()*

original que son interiores a un círculo de radio *1* centrado en el origen.

```
fn main() {
   let v: Vec<(f64, f64)> = vec![
   (1., 1.), (0.5, -0.25),
   (2., 0.1), (-0.5, -0.5)
   ];

   let new_vector: Vec<(f64, f64)> = v.into_iter()
   .filter(|(x, y)| (x * x + y * y).sqrt() < 1.0)
   .collect();

   println!("{:?}", new_vector); // [(0.5, -0.25), (-0.5, -0.5)]
}
```

Observe que en la función *filter*, a diferencia de lo que ocurría con la función *map()*, los elementos del iterador resultante son del mismo tipo que los del original. De hecho, son un subconjunto de los elementos del iterador original.

1.9 El método *fold()*

El método *fold()* aplica la función que recibe como argumento a los valores del iterador original, acumulando los valores en un solo valor que no tiene por qué ser del mismo tipo. A diferencia de *map* y *filter*, el método *fold()* tiene dos parámetros: el valor inicial del acumulador y la función que se utilizará para realizar los cálculos. La signatura de *fold()* es la siguiente:

```
fn fold(valor_inicial_acumulador: B, f)->B)
```

La función *f* es del tipo *FnMut*, recibe dos parámetros: el elemento del iterador

original que se está procesando y un valor del tipo de datos *B* del acumulado; la función debe devolver un valor del tipo de datos *B* del acumulado. Llamando *A* al tipo de datos de los valores del iterador original y *B* al tipo de datos del valor acumulado, esto se puede expresar así:

```
f: FnMut(B, T) ->B
```

Seguramente, un ejemplo ayude a comprender el funcionamiento de la función *fold()* mejor que tanta explicación. El siguiente código, calcula la suma de los elementos de un vector de números enteros:

```
fn main() {
   let v: Vec<i32> = vec![1, 2, 3, 4, 5];

   let suma: i32 = v.iter()
   .fold(0, |acc, x| acc+x);
   println!("{suma}"); // Imprime 15
}
```

Observe que, en este caso, el valor inicial del acumulador es *0*. La función acumuladora declara dos parámetros: *acc* y *x*. En cada iteración, *acc* contendrá el actual valor del acumulador. La función devuelve la suma de *acc* y *x*.

Observe también que, para obtener el iterador del vector, se ha utilizado el método *iter()*, por lo que las *x* de la función acumuladora son referencias a los valores del vector original y no se consume el vector.

La variable *acc* que se utiliza para el acumulador necesita ser mutable, de ahí que se haya dicho que la función acumuladora es del tipo *FnMut*.

En el siguiente ejemplo, los valores del iterador original son del tipo *char*, mientras que el valor acumulado es del tipo *String*:

```
fn main() {
   let v: Vec<char> = vec!['h', 'o', 'l', 'a'];

   let suma: String = v.into_iter()
   .fold(String::new(), |mut acc, c| {
      acc.push(c);
      acc
   }
   );
   println!("{suma}"); // Imprime hola
}
```

En este caso, ha sido necesario declarar que el parámetro *acc* de la closure es mutable. También ha sido necesario operar en dos pasos, pues la función *push()* modifica la cadena original pero devuelve el valor unitario *()*.

Otro ejemplo clásico es el cálculo del máximo de una serie de números:

```
fn main() {
   let v: Vec<i32> = vec![-1, 2, 5, 1, -2];

   let maximo: i32 = v.iter()
   .fold(v[0], |acc, x | if *x>acc {*x} else {acc});
   println!("{}", maximo); // Imprime 5
}
```

Un ejemplo un poco más complejo podría ser el calculo de la longitud de una polilínea definida mediante las coordenadas de sus puntos:

```
fn main() {
   let v: Vec<(f64, f64)> = vec![(0., 0.), (1., 1.), (2., 2.)];

   let acumulado: (f64, f64, f64) = v.iter()
   .fold((v[0].0, v[0].1, 0.0), |acc, p |{
      let d = ((p.0 - acc.0)*(p.0 - acc.0) +
      (p.1 -acc.1)*(p.1 -acc.1)).sqrt();
      (p.0, p.1, acc.2 + d)
   }
   );
   println!("{:.4}", acumulado.2); // Imprime 2.8284
}
```

Observe que, en este caso, cada iteración necesita las coordenadas del punto anterior, de ahí la técnica que se ha utilizado para definir los valores acumulados como una tupla de tres componentes.

1.9.1 Ejemplo: cálculo del máximo

1.10 Herramientas de los Iteradores

Rust ofrece otras funciones que se pueden aplicar a los iteradores.

- **for_each():** devuelve la lista resultante de aplicar la closure que recibe como argumento a cada elemento de la lista original. Es equivalente a un bucle *for*, aunque en determinadas situaciones puede ser más rápido.
- **filter():** devuelve la lista resultante de filtrar los elementos de la lista original utilizando para ello la closure que recibe como argumento. La closure acepta como parámetro un elemento del tipo de datos de los elementos de la lista y devuelve *true* o *false*. La lista resultante contendrá solo los elementos en los que la closure devuelva *true*.

  El siguiente ejemplo extrae los elementos pares del array original:

```
fn main() {
   let lista = [1, 2, 3, 4, 5];
   let nueva_lista: Vec<&i32> = lista.iter().filter(|&x| *x%2==0).
   println!("{:?}", nueva_lista); // [2, 4]
}
```

En el ejemplo anterior, la función *filter()* opera sobre un iterador de referencias a valores, por lo que primero hay que convertir el array original en un iterador de referencias a números enteros usando el método *iter()*. La closure recibe una referencia a un elemento como parámetro, *&x*, y comprueba si el resto de dividir por 2 el valor al que apunta dicha referencia es igual a 0. Sucede que, a priori, no se conoce el tamaño del array resultante y el tamaño de un array hay que conocerlo en tiempo de compilación, por lo que se utiliza el método *collect()* para convertir el iterador resultante en un vector de referencias a números enteros.

La totalidad de las funciones aplicables a los iteradores se pueden consultar en el siguiente enlace:

(Fuentes:

- `https://blog.jetbrains.com/rust/2024/03/12/rust-iterators-beyond-the-`
- Module iter: `https://doc.rust-lang.org/std/iter/`
- Processing a Series of Items with Iterators: `https://doc.rust-lang.org/book/ch13-02-iterators.html`
- `https://mustafabugraavci.blog/2024/04/23/mastering-iterators-in-rust-`

CAPÍTULO 2

Funciones puras, inmutabilidad

Contenido

Como se indicó en el Capítulo **??***, una de las características distintivas de la Programación Funcional es su preferencia por la utilización de las funciones puras, esto es, funciones sin efectos secundarios. Algunos autores utilizan la denominación cálculos, para referirse a las funciones puras y acciones, para referirse a las funciones con efectos secundarios [3]. Esta será la denominación que se utilizará preferentemente en el texto a partir de este momento.*

Otro de los pilares de la programación funcional es la utilización de las estructuras de datos inmutables, lo que proporciona ventajas a la hora de hacer determinadas optimizaciones en el código compilado y facilita la programación concurrente y en paralelo.

En este capítulo se va a ahondar en la utilización de las funciones puras y variables inmutables. Las técnicas que se van a explicar, que proceden en su mayoría de patrones de diseño de la Programación Funcional, no dependen del lenguaje de programación que se utilice y su aplicación produce beneficios al código generado, haciéndolo más fácil de comprender, favoreciendo su reutilización y facilitando el proceso de pruebas y depuración. Se pueden aplicar por igual a la programación orientada a objetos o a la programación basada en procedimientos. En realidad, se podría decir que se trata simplemente de buenas prácticas de programación.

Hay dos ideas principales que subyacen en todas estas técnicas:

- *Separar las acciones de los cálculos y de los datos.*
- *Propiciar el uso de abstracciones de orden superior.*

2.1 Distinguir entre datos, cálculos y acciones

El código de los programas se puede dividir en tres categorías:

- Datos.
- Cálculos (funciones puras, funciones sin efectos secundarios).
- Acciones (funciones impuras, funciones con efectos secundarios).

Los *datos* son la plasmación de hechos que han sucedido. Los datos no producen efectos secundarios ni dan lugar a ningún resultado. Son código inerte. Lo que sí admiten, son diferentes interpretaciones. Un mismo dato, por ejemplo una cadena de texto con una palabra, puede admitir diferentes interpretaciones según el programa concreto que haga uso de ella.

Las *funciones puras* son aquellas que no producen efectos secundarios, no dependen de ningún estado exterior y siempre proporcionan el mismo resultado para el mismo valor de los argumentos de entrada (ver Figura 2.1). Se suelen denominar también *funciones matemáticas* o *cálculos*. En este texto, siguiendo la denominación utilizada por Eric Normand, se denominarán frecuentemente *cálculos* [3].

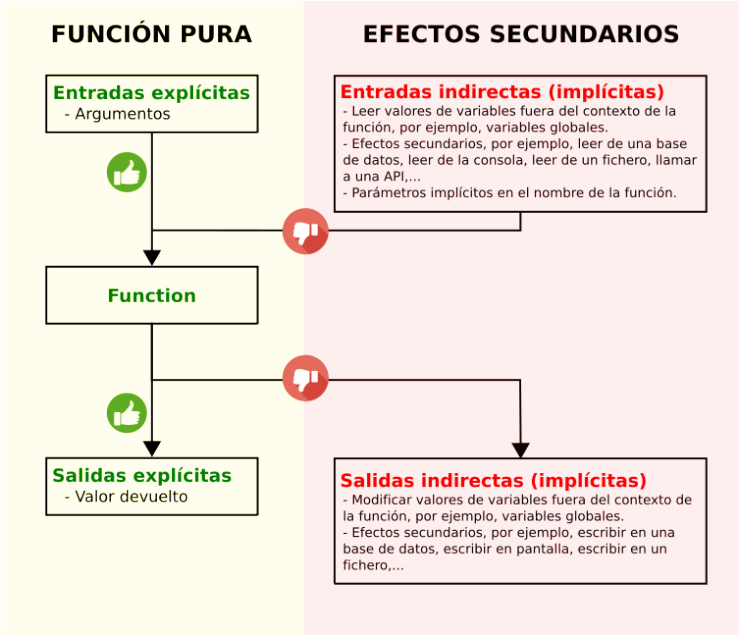

Figura 2.1: Funciones puras y efectos secundarios.

Se entiende por *efecto secundario* cualquier acción del programa que no sea un cálculo. Por ejemplo, modificar el estado de una variable, mostrar un mensaje en una pantalla o enviar un correo electrónico serían ejemplos de efectos secundarios. Cuando una función produce algún efecto secundario, se dice que es una *función impura*, una *función con efectos secundarios* o, siguiendo la denominación de Eric Normand, una *acción*. Así se denominarán frecuentemente en el texto a partir de ahora.

No es posible hacer un programa de utilidad sin producir efectos secundarios. De hecho, los efectos secundarios suelen ser la razón de ser de los programas. Lo que sí se puede hacer es codificar los programas de forma que las acciones siempre se lleven a cabo dentro de ámbitos controlados y bajo una supervisión minuciosa.

Al inspeccionar cualquier fragmento de código, hay que distinguir las partes que son acciones, de las que son cálculos y de las que son simplemente datos. En general, se debe dar preferencia a la utilización de datos sobre los cálculos y a los cálculos sobre las acciones.

Un ejemplo de cálculo sería la función siguiente:

```
fn cuadrado(x: f64) -> f64 {
   x*x
}
```

La función anterior, recibe como argumento un número del tipo *f64* y devuelve el cuadrado de dicho número. El código dentro de la función no afecta a nada externo a la misma, no tiene ninguna entrada de información que no sea a través de su parámetro ni ninguna salida que no sea a través de su valor devuelto; el valor que devuelve, siempre será el mismo si recibe el mismo valor como argumento, independientemente de en qué momento se llame a la función o cuántas veces se llame a la misma.

Las acciones, en cambio, son funciones con efectos secundarios. Los efectos secundarios pueden tomar distintas formas dentro del código. Por ejemplo:

- Llamadas a otras funciones o métodos: `println!("Hola, qué tal")`
- Constructores: `new Date()`
- Expresiones: `x` (si `x` es mutable).
- Declaraciones: `x = 3` (si se muta `x`).

Las funciones que son acciones, pueden estar compuestas por otras acciones, cálculos y datos. Una función que sea un cálculo puede estar compuesta por varios cálculos menores. Los datos solo pueden contener otros datos.

A continuación, se van a explicar con más detalle las características de los datos, los cálculos y las acciones.

2.1.1 Datos

Los datos son hechos resultantes de acontecimientos que han sucedido a lo largo de la ejecución del programa. Sus características principales son las siguientes:

- Se implementan utilizando los tipos de datos que ofrezca el lenguaje de programación que se esté utilizando.
- Las estructura de los datos encierra parte de su significado, por ejemplo, el orden de los elementos en una lista.
- Los datos en sí mismos, son inmutables. Los datos de un acontecimiento no varían, lo que puede suceder es que se produzcan nuevos acontecimientos que den lugar a nuevos datos.

- *Ventajas de los datos:* son serializables, se pueden comparar para comprobar su igualdad, están abiertos a distintas interpretaciones.
- *Desventajas:* el hecho de que admitan distintas interpretaciones puede ser un arma de doble filo. Un cálculo es útil, aunque no entendamos cómo está hecho, pero un dato no tiene ningún significado sin una interpretación adecuada.

En los programas, conviene separar la generación de los datos de su utilización. Por ejemplo, si un programa tiene que enviar unos correos a una serie de personas, conviene proceder primero a generar los correos, guardarlos en una estructura de datos y luego enviarlos.

Es preferible hacerlo así, que ir generando los correos uno a uno e ir enviando cada correo que se genera. Esto, facilitará la realización de test: es más fácil probar una función que genera una lista de datos que una función que envía un correo.

Si la lista de correos fuera muy grande, siempre se podría repetir el procedimiento anterior dividiendo en subgrupos los correos que se tienen que enviar.

2.1.2 Cálculos: funciones puras

Los cálculos son las funciones puras, también llamadas funciones matemáticas. A partir de una entrada se genera una salida. En una función que sea un cálculo, la entrada se produce a través de los parámetros de la función, la salida es el valor que devuelve y el cálculo es el cuerpo de la función.

Los cálculos gozan de *trasparencia referencial*, esto es, en cualquier lugar del código en el que aparece la llamada a una función que es un cálculo, se puede sustituir dicha llamada por el resultado que se obtiene. Cuando el compilador identifica un cálculo, puede proceder a optimizaciones sustituyendo la llamada a la función por su resultado. Esto no es posible garantizarlo en el caso de que la función sea una acción.

Las características de los cálculos:

- En los programas, los cálculos se implementan utilizando funciones.
- El significado que encierran es el cálculo que hacen.
- Las ventajas de los cálculos frente a las acciones son:
    - Es más fácil hacer test y probarlos.
    - Los test se pueden automatizar más fácilmente.
    - Es posible componer unos cálculos con otros.
    - No hay que preocuparse de otros cálculos que se pudieran estar pro-

duciendo al mismo tiempo, el resultado solo depende del valor de los argumentos de entrada (programación en paralelo).
    - Por el mismo motivo, no hay que preocuparse de lo que haya sucedido antes de llamar al cálculo o de lo que pueda suceder después.
    - No hay que preocuparse de cuántas veces se haya llamado anteriormente al cálculo.
- *Desventajas:* al igual que sucede con las acciones, no se puede estar seguro de lo que hacen si no se ejecuta el código. Lo único que se está seguro de lo que hacen son los datos, que no hacen nada.

2.1.3 Acciones: funciones con efectos secundarios

Las funciones que tienen entradas o salidas implícitas, son *acciones*, también llamadas *funciones con efectos secundarios* o *funciones impuras*. Una acción es cualquier cosa que tenga un efecto en el mundo exterior a la función o que sea afectada por el mundo exterior.

Sus características principales son:

- Las acciones se implementan en el código utilizando funciones.
- Se propagan por el código: si una función llama a otra función que es una acción, ella misma se convierte en una acción.
- Las acciones dependen o pueden depender de cuándo se ejecutan (del orden de ejecución) o de cuantas veces se ejecutan (repetición).
- El significado de las acciones es el efecto que producen en el mundo exterior. Hay que asegurarse de que producen el efecto deseado.
- Las acciones son inevitables. De hecho, son la razón de ser de los programas.

La Programación Funcional propone diversas técnicas para gestionar de manera adecuada las acciones:

- Reducir el número de acciones a las estrictamente necesarias.
- Mantenerlas reducidas a su mínima expresión, extrayendo de ellas todos los elementos que sean cálculos o datos.
- Restringir las acciones a las interacciones con el exterior. En el interior, idealmente, todo son cálculos y datos.
- Reducir la dependencia del tiempo. Hacer que las acciones dependan menos de cuándo se jecutan o de cuántas veces se ejecutan.

2.2 Tipos de entradas y salidas de las funciones

Todas las funciones tienen entradas y/o salidas. Se pueden clasificar en implícitas y explícitas:

- *Entradas explícitas*: a través de los parámetros.
- *Salidas explícitas*: a través del valor devuelto.
- *Entradas o salidas implícitas*: cualquier otra forma de entrada o salida de información de la función.

Las entradas *explícitas* son los valores de los argumentos que recibe la función. Cualquier otra entrada de información a la función se considera una entrada *implícita*. Las salidas *explícitas* son los valores devueltos. Cualquier otra salida de información de la función se considera una salida *implícita* (ver Figura 2.1).

La existencia de entradas o salidas implícitas convierten una función en impura. Una *entrada implícita* puede ser leer el valor de una variable externa a la función, por ejemplo una variable global. También sería un entrada implícita la consulta de datos provenientes de una base de datos. Las *salidas implícitas* son cualquier información que salga de la función y que no sea el valor devuelto, por ejemplo, modificar el valor de una variable externa o enviar un email o escribir informacion en una base de datos o un fichero.

Cualquier entrada o salida implícita de información a la función convierte la función en impura, en una acción.

Si en una función se eliminan todas las entradas y salidas implícitas, se convierte en un cálculo. Las entradas implícitas, hay que convertirlas en parámetros y las salidas implícitas hay que convertirlas en valores devueltos.

2.3 Separar las cosas

Conviene recordar el concepto de *refactorizar*. Consiste en hacer modificaciones en el código organizándolo de otra manera sin alterar su comportamiento. Hay diversas técnicas de refactorización que se han estandarizado, por ejemplo, la denominada *extraer subrutina* consiste en extraer una parte de una función a otra función independiente.

En general, la refactorización da lugar a un código con más líneas, pero es más comprensible, más reusable y es más fácil hacer los test.

Una técnica de diseño reconocida consiste en *separar las cosas* (*pull things apart*). Ésta técnica también es conocida como el *Principio de Responsabilidad única*: que cada función haga solo una cosa y que la haga bien. Da lugar a un

código con más funciones y más pequeñas, pero facilita la reutilización y hace el código más fácil de comprender. En la Programación Funcional se promueve la utilización de funciones más pequeñas, separando las cosas. Los procesos complejos se consiguen mediante la *composición* de dichas funciones. La figura 2.2 trata de esquematizar está técnica.

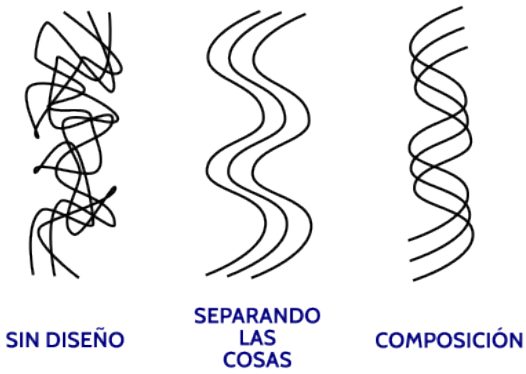

Figura 2.2: Visualización de la separación de diferentes cosas y su composición

2.4 Extraer los cálculos de las acciones

Observe la función *grabar_caudal()* que se muestra en el siguiente fragmento de código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();
   let caudal = 1.8 * velocidad.powf(3.2);
   save_caudal_to_database(caudal);
}
```

La función lee el valor de la velocidad de un sensor, realiza el cálculo del caudal asociado a esa velocidad y graba el resultado en una base de datos. La función es una acción: no tiene ningún argumento de entrada ni devuelve ningún resultado, pero tiene una entrada de datos implícita a través de la función *leer_sensor()* y una salida de datos implícita que graba datos en la base de datos utilizando la función *save_caudal_to_database()*.

Además, la función incluye un cálculo, que aquí se ha expresado mediante una operación relativamente sencilla, pero que podría ser más complicado.

Si se quisiera hacer algún test para comprobar que el cálculo se realiza de manera correcta, no sería fácil. Una opción sería habilitar el sensor correspondiente y la base de datos asociada mientras se hacen los test. Para comprobar que el resultado calculado es correcto, seguramente habría que consultar el valor grabado en la base de datos, pues la función no devuelve ningún valor. Si se quisiera probar el cálculo con diferentes valores de velocidad, habría que *trucar* el sensor, lo que puede resultar complicado.

El problema está en que se están mezclando dos cosas: una acción (en ralidad, dos) y un cálculo. La solución adecuada es extraer el cálculo a una función independiente, como se muestra en el siguiente código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();

   let caudal = calcula_caudal(velocidad);

   save_caudal_to_database(caudal);
}
fn calcula_caudal(velocidad: f64) -> f64 {
   let resultado = 1.8 * velocidad.powf(3.2);
   resultado
}
```

Ahora hay dos funciones: una acción y un cálculo. Es fácil realizar los test para comprobar que los cálculos se hacen de manera adecuada. Se puede probar la función *calcula_caudal()* para cualquier valor de entrada de manera sencilla.

Los pasos que se han seguido se podrían esquematizar de la siguiente forma:

- En la función original, seleccionar el fragmento de código a extraer.
- Ponerlo en una nueva función, sustituyéndolo en la función original por una llamada a la función recién creada. Es posible que haya que añadir algún parámetro.
- Identificar todas las entradas y salidas en la nueva función, convirtiendo las entradas en parámetros y las salidas en valores devueltos, que habrá que asignar a alguna variable en la función original.
- Los argumentos y lo valores devueltos de la función recién creadas tienen que ser inmutables, de forma que la nueva función sea un cálculo.

2.5 Ejemplo: eliminar entradas implícitas

En numerosas ocasiones, una misma función hace más de una cosa. Suponga que se tiene la siguiente función:

```
fn main() {
   facturar(100.0);
}
fn facturar(precio: f64) {
   let descuento = 0.10 * precio;
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

La función *facturar()* recibe como argumento el precio de un artículo, calcula el descuento a aplicar y muestra el resumen de la factura en pantalla. Está haciendo dos cosas: calcular el descuento y mostrar los resultados en pantalla. El hecho de mostrar los resultados en pantalla es un efecto secundario, una salida implícita que convierte a la función en una acción. Pero, además, tiene una entrada implícita de información. El valor 0.10 del porcentaje de descuento que se aplica es una especie de variable global, correspondiente a la lógica de negocio de la aplicación.

No es posible eliminar la acción, de hecho, es el objetivo del programa: mostrar la factura en pantalla, una vez que se ha aplicado el descuento. Es difícil diseñar un test para probar la función. Por una parte hay que probar que los cálculos son correctos y, por otra, que la salida de la función también es correcta. Tenga en cuenta que, aunque la salida se ha planteado simplemente mostrando unos mensajes en pantalla, podría tratarse de la actualización de ciertos registros en una base de datos o cualquier otra salida. En el caso de hacer la salida a una base de datos, para probar los cálculos, habría que habilitar y tener disponible la base de datos.

Lo correcto es separar el cálculo del descuento de la salida de resultados. Pero, además, la entrada implícita del porcentaje de descuento hay que convertirla en un parámetro de la función que calcula el descuento. Podría hacerse de la siguiente forma:

```
fn main() {
   let porcentaje_descuento = 0.10;
   let precio = 100.0;
   let descuento =
   calcula_descuento(precio, porcentaje_descuento);
   facturar(precio, descuento);
}
fn calcula_descuento(precio: f64, porcentaje: f64) -> f64 {
   precio*porcentaje
}
fn facturar(precio: f64, descuento: f64) {
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

Ahora, la función *calcula_descuento()* es una función pura, un cálculo. Sus únicas entradas se producen a través de los argumentos que recibe y su salida es el valor que devuelve. No importa cuándo se llame a la función o cuántas veces se la llame: si recibe los mismos argumentos siempre devolverá el mismo resultado. Es fácil plantear los test para probar que realiza los cálculos de manera adecuada.

Además, el porcentaje del descuento que se tiene que aplicar se ha independizado de cualquiera de las funciones. Está centralizado en una variable accesible para otras funciones del programa. Si la empresa decidiera cambiar su política de descuentos, solo tendría que modificar el valor de la variable en un sitio y se actualizaría la política en cualquier función que haga uso de ella.

Por otra parte, la función *facturar()* sigue siendo una acción, pero independiente de otras consideraciones. Si se decidiera cambiar la forma de salida de la pantalla a una base de datos, por ejemplo, solo habría que actuar en dicha función, despreocupándose del cálculo de los valores asociados. Aun siendo una acción, se han eliminado sus entradas implícitas, convirtiéndolas en parámetros.

En el ejemplo, se han realizado varias refactorizaciones:

- Extraer un cálculo de una acción.
- Convertir una entrada implícita en un parámetro.
- Separar las cosas en funciones independientes.

2.6 Inmutabilidad. Copy-on-write

Al inicializar una variable, se cambia su valor. Si en las funciones solo se inicializan variables locales, nada fuera de la función ve dichos cambios, salvo que lo vean a través del valor devuelto.

Ejemplo sin copy-on-write:

```
fn main() {
   let mut list = vec![1, 2, 3];
   list.push(4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
```

Sin copy-on-write, pero con función:

```
fn main() {
   let mut list = vec![1, 2, 3];
   modify_list(&mut list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn modify_list(list: &mut Vec<i32>, x: i32) {
   list.push(x);
}
```

Con copy-on-write:

```
fn main() {
   let list = vec![1, 2, 3];
   let list = copy_on_write(list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn copy_on_write(list: Vec<i32>, x: i32) -> Vec<i32> {
   let mut new_list = list;
   new_list.push(x);
   new_list
}
```

No se modifica la lista original, se transforma en una nueva lista que incorpora las modificaciones. No hay pérdida de rendimiento, la propiedad de los valores se va trasmitiendo, sin modificar su posición en memoria.

Variante usando un *RefCell*:

```
fn main() {
    let list = RefCell::new(vec![1, 2, 3]);
    let list = modify_cell(list, 4);
    assert_eq!(list.take(), vec![1, 2, 3, 4]);
}
fn modify_cell(list: RefCell<Vec<i32>>, x: i32)
-> RefCell<Vec<i32>> {
    let new_list = list;
    new_list.borrow_mut().push(x);
    new_list
}
```

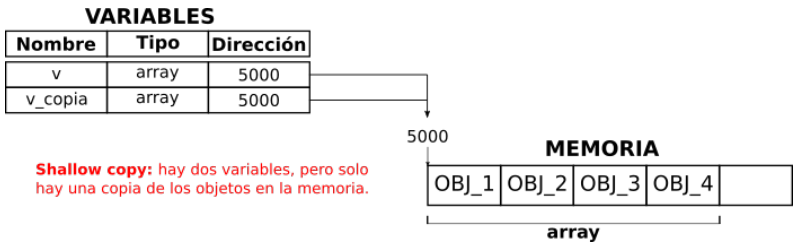

Figura 2.3: Esquema del mecanismo *shallow-copy* de un *array* en Java: se copian las referencias a los objetos, no los objetos en sí mismos. Al final hay dos variables apuntando a los mismos objetos en memoria.

CAPÍTULO 3

Colecciones

CAPÍTULO 4

Patrones de diseño en la Programación Funcional

CAPÍTULO 5

Grokking Simplicity

Contenido

Apuntes del Libro «Grokking Simplicity», de Eric Norman [3]

5.1 Welcome to Grokking simplicity

5.2 Functional thinking in action

5.3 Distinguishing actions, calculations and data

Los programadores funcionales clasifican cualquier fragmento de código como acción, cálculo o datos. Esta clasificación puede recibir otras denominaciones, pero el concepto es el mismo.

- **Acciones:** son todo aquello que dependa de en qué momento se ejecuta o de cuántas veces se ejecuta. Por ejemplo, enviar un correo es una acción.
- **Cálculos:** son las tareas que solo dependen de los valores de entrada para generar un resultado. Los cálculos siempre devuelven el mismo resultado, si los valores de entrada son los mismos. Además, los cálculos nunca afectan a nada que esté fuera de ellos. Esto hace que los cálculos sean fáciles de testear y que su uso sea seguro, pues no hay que preocuparse de cuántas veces se utilicen o en qué orden sean invocados: si los parámetros de entrada son los mismos, el resultado será siempre el mismo.
- **Datos:** los datos son información registrada acerca de los acontecimientos. Tienen propiedades conocidas. Un mismo dato se puede interpretar de manera diferente según el contexto de ejecución en el que se encuadra. Por ejemplo, la factura de una cena la puede utilizar el cliente para llevar la cuenta de sus gastos mensuales o la puede utilizar el propietario del restaurante para determinar los gustos favoritos de sus clientes.

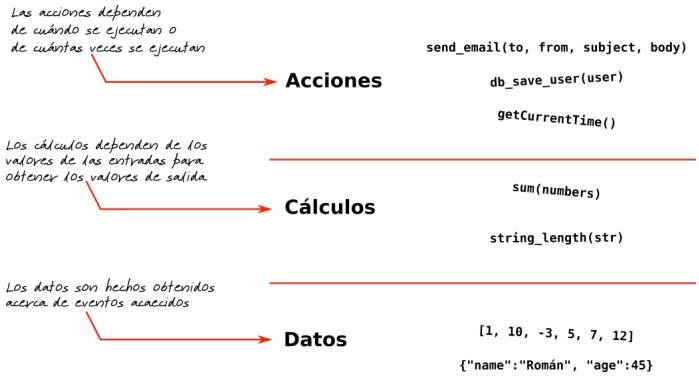

Figura 5.1: Ejemplos de código: Acciones, Cálculos y Datos

La proliferación de los sistemas distribuidos en los que intervienen diferentes dispositivos y peticiones cuasi simultáneas de información a las bases de datos,

el problema de organizar de manera adecuada el código se hace indispensable. En general, el código de las acciones será más difícil de comprender y de testear que el código de los cálculos, ya que estos últimos no dependen del número de veces que se invocan o del orden de dichas invocaciones. Es por ello que se hace importante separar en la medida de los posible el código que corresponde a las acciones del código que corresponde a los cálculos.

La programación funcional proporciona herramientas para el correcto tratamiento de cada una de estas categorías de código. En el caso de las acciones, es importante gestionar la forma en que cambia el estado de las variables del programa a lo largo del tiempo, garantizando el número de veces y el orden en el que se realiza cada acción. Los cálculos tienen su propia estrategia para comprobar su buen funcionamiento, a veces basada en técnicas matemáticas. En el caso de los datos, es importante organizarlos en estructuras que faciliten un acceso eficiente a la información que se quiere extraer de los mismos.

Hay dos técnicas fundamentales que permiten abordar los programas con un enfoque funcional:

- Distinguir en el código las acciones, de los cálculos y los datos.
- Utilizar abstracciones de primera clase.

A lo largo del curso se tratará de trasmitir el razonamiento funcional a la hora de abordar un problema de codificación. El objetivo es que las técnicas que se aprendan sean independientes del lenguaje que se utilice para programar y que sean de aplicación inmediata, tanto para la realización de un programa nuevo, como para refactorizar partes de un código ya existente.

Para clasificar el código en acciones, cálculos y datos es útil seguir los principios del *diseño estratificado*, separando el código en diferentes capas. Para organizar el orden en el que se ejecutan las acciones son de utilidad los *diagramas de tiempos* y la utilización de funciones de *primera clase*, que permiten utilizar otras funciones como parámetros o resultados.

En el diseño estratificado, se organiza el código en diferentes capas, ordenadas en función de la mayor o menor probabilidad de cambios en el código correspondiente a lo largo de la vida útil de la aplicación. Se suelen considerar tres capas principales:

- **Nivel técnico:** correspondería a la parte de la aplicación que tiene menos probabilidades de cambiar. Por ejemplo, el lenguaje de programación de la aplicación o las estructuras de datos que se utilizarán para almacenar la información.

- **Reglas del dominio de la aplicación:** si por ejemplo se está haciendo una aplicación para gestionar unos cultivos de hortalizas, las distancia optima a la que hay que poner las plantas en el terreno o la cantidad de humedad que necesitan corresponden al campo de conocimiento de dicho dominio técnico y es difícil que cambien durante la vida útil de la aplicación.
- **Reglas del negocio:** se consideran aquí reglas que vienen marcadas por la aplicación concreta que se esté desarrollando. En el ejemplo de los cultivos podrían ser los precios de los factores de producción o la disponibilidad de determinados recursos.

Cada capa de la aplicación se desarrolla sobre las demás y solo debe depender de las capas que hay situadas por debajo de ella. De esta forma, si se produce una modificación en algún elemento, se sabe que solo puede afectar a los elementos que estén situados en la misma capa o en las capas superiores. La Figura 5.2 muestra un ejemplo de organización en capas de una aplicación para gestionar cultivos.

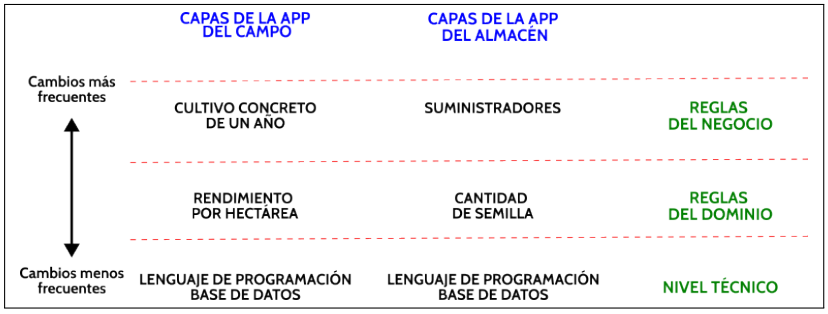

Figura 5.2: Esquema de capas en el diseño estratificado

5.3.1 Líneas de tiempos

En todas las aplicaciones hay que establecer el orden en el que se tienen que realizar las acciones. En aplicaciones sencillas, una distribución secuencial de las acciones puede ser suficiente. Pero, en sistemas distribuidos, en los que distintas tareas pueden correr a cargo de distintos componentes que pueden trabajar en paralelo o de forma concurrente, es importante establecer el orden en el que se tienen que realizar todas las acciones y los puntos en los que determinadas acciones no pueden ejecutarse si antes no se ha finalizado determinada acción anterior. Hay que cortar la línea de tiempos en algunos puntos para indicar que la ejecución no puede continuar hasta que se completen todas las tareas anteriores.

La Figura 5.3 muestra la línea de tiempos de una aplicación en la que, para poder ejecutarse las acciones *D* y *E* es necesario que primero se hayan completado las tareas llevadas a cabo por los componentes *A*, *B* y *C*.

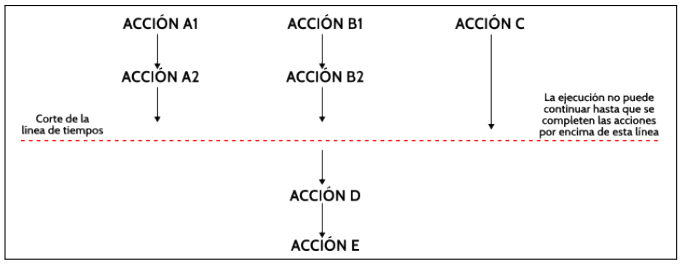

Figura 5.3: Cortando la línea de tiempo para garantizar el orden de ejecución de las acciones

La línea de tiempos ayuda a coordinar distintas acciones y a determinar los puntos en los que se pueden producir cuellos de botella durante la ejecución de los programas en sistemas distribuidos.

5.4 Distinción entre acciones, cálculos y datos

Antes de ponerse a codificar una aplicación nueva, conviene razonar .para distinguir qué elementos del programa serán acciones, cálculos o datos. En general, el orden de implementación consistirá en definir primero los datos, luego los cálculos y, por último, las acciones.

Cuando se está analizando un código ya existente, habrá que identificar igualmente las funciones que incluyen acciones. En muchos casos, una misma función incluirá acciones y cálculos. En esos casos, conviene refactorizar para separar las acciones de los cálculos.

Como se ha comentado, las acciones dependen de cuándo se ejecutan o de cuantas veces se ejecutan. Las funciones que implementan acciones se suelen denominar *funciones impuras* o *funciones con efectos secundarios*. Enviar un correo o leer datos de una base de datos serían ejemplos de acciones.

Por el contrario, los cálculos solo dependen del valor de los parámetros de entrada y no producen efectos secundarios. Se denominan *funciones puras* o *funciones matemáticas*. Calcular el máximo de una serie de números o comprobar si determinada dirección de correo es válida podrían ser ejemplos de funciones puras.

Se dice que los cálculos son *referencialmente trasparentes*. La trasparencia referencial significa que se puede sustituir el cálculo por el resultado sin que el

programa se vea afectado. Por ejemplo, si una función realiza la suma de dos números, en los lugares del programa donde se hace la llamada a la función se puede sustituir ésta por el resultado de la suma y el programa no se verá afectado.

Los datos son hechos.

5.4.1 Entradas y salidas implícitas

Los parámetros de una función son el procedimiento de *entrada explícita* de datos a la función. El valor devuelto por la función es la salida *explícita*. Cuando una función es impura, tiene entradas o salidas implícitas. Se denomina *entrada implícita* a la entrada de datos a la función que no procede de un parámetro. Se denomina *salida implícita* al valor que sale de la función sin hacerlo a través del valor devuelto por la misma.

El código del ejemplo muestra una función denominada *contador_impuro()* que no tiene parámetros ni devuelve ningún valor. En cambio, la función recibe como entrada implícita el valor de un contador existente en la base de datos, a través de un método llamado *get_contador_from_database()*, y realiza una salida también implícita reescribiendo en la base de datos el valor incrementado de dicho contador a través del método *update_contador_in_database()*. La salida implícita de la función es, además, un efecto secundario de la misma, convirtiendo a dicha función en una *acción*.

Ejemplo 5.1 Entradas y salidas implícitas

```
fn contador_impuro() {
   let contador = get_contador_from_database(); // Entrada implícita
   update_contador_in_database(contador+1); // Salida implícita
}
```

El acceso a variables globales dentro de una función es una forma de entrada o salida implícita.

En la medida de lo posible, hay que evitar las entradas y salidas implícitas, pues complican la trazabilidad y la facilidad de testeo de las funciones. En muchas ocasiones, las entradas implícitas se pueden sustituir por parámetros de la función. De la misma forma, las salidas implícitas es posible sustituirlas por valores devueltos por las funciones.

5.5 Ejemplo

Ejemplo 5.2 Ejemplo carrito

```
struct Producto {
   name: String,
   precio: f64,
}
fn main() {
   let mut carrito = Vec::<Producto>::new();
   let mut total_compra: f64 = 0.0;
   let producto = Producto{name: "Sandalias".to_string(), precio: 12.5
   add_producto(producto, &mut carrito, &mut total_compra);
}
fn add_producto(producto: Producto, carrito: &mut Vec<Producto>, total
   carrito.push(producto);
   calc_total_carrito(carrito, total_compra);
}
fn calc_total_carrito(carrito: &mut Carrito, total_compra: &mut f64) {
   *total_compra = 0.0;
   for producto in &carrito.items {
      *total_compra = *total_compra + producto.precio;
   }
   actualiza_web_total_compra( *total_compra);
}
fn actualiza_web_total_compra(total_compra: f64) {  }
```

CAPÍTULO 6

Functional Programming made easier (Scalfani)

6.1 1.- Discipline is freedom

6.1.1 Global State

El uso de varibles globales conlleva determinados inconvenientes:

- Cualquiera desde cualquier módulo puede cambiar el valor de las variables globales.
- Se producen acoplamientos de las variables globales entre sí y de unos módulos con otros.
- En programación concurrente no hay garantías respecto de la modificación de las variables.
- Colisión de nombres entre las variables globales y otros identificadores en cualquier módulo.

En el caso de la programación orientada a objetos se produce la misma circunstancia con el patrón *Singleton*.

Los lenguajes de PF prohíben la existencia de variables globales. Rust permite solo la existencia de constantes globales.

6.1.2 Mutable State

La posibilidad de que las variables puedan cambiar de valor también tiene algunos inconvenientes. Por ejemplo, es más difícil razonar sobre el código, pues los valores que pueden cambiar pueden hacer cambiar también la semántica del programa, o hacer el código más frágil.

En los lenguajes funcionales, las variables son inmutables y ello da lugar a algunas consecuencias.

- Las expresiones del tipo $x = x + 1$ no tienen sentido. Una vez que se asigna un valor a x, no se puede cambiar.

- Las asignaciones como $x = 20$ son *expresiones referencialmente trasparentes*, esto es, en cualquier parte del programa se puede utilizar indistintamente x o 20 con la seguridad de que el programa seguirá funcionando igual. La trasparencia referencial permite una evaluación *lazy* de la sustitución de x por su valor en el código.

Si las variables no pueden cambiar de estado, no es posible hacer bucles. En los lenguajes funcionales, los bucles se sustituyen por la recursividad. Cualquier bucle se puede ejecutar mediante recursividad y viceversa.

Por ejemplo, la definición matemática del factorial de un número se podría hacer de la siguiente forma:

$$n! = 1.2...(n-2).(n-1).n \quad \forall n \in \mathbb{N} \tag{6.1}$$

Con esta definición, parecería inmediato resolver el problema con un bucle, como se hace en el Ejemplo 6.1.

Ejemplo 6.1 Factorial calculado con un bucle

```
fn factorial_bucle(n: u32) -> u32 {
   let mut prod = 1;
   for i in 1..=n {
      prod = prod*i;
   }
   prod
}
```

Si en la Expresión 6.1 se cambia el orden del producto, los términos se podrían agrupar de la siguiente manera:

$$n! = n.(n-1)...2.1 \quad = n.(n-1)! \quad \forall n \in \mathbb{N} \tag{6.2}$$

Con esta definición surge el problema de calcular $0!$, pero su valor se puede deducir. De la Expresión 6.1 se sabe que $!1 = 1$. Se podría operar de la siguiente forma:

$$\begin{aligned} 1! &= 1 \\ 1! &= 1.(1-1)! = 1.0! \\ 1 &= 1.0! = 0! \end{aligned}$$

Con lo que finalmente queda que:

$$0! = 1 \tag{6.3}$$

Una vez calculado el valor de $0!$, se puede proceder a definir el factorial de cualquier número natural con la siguiente definición recursiva:

$$\begin{aligned} 0! &= 1 \\ n! &= n.(n-1)! \quad \forall n \in \mathbb{N} \end{aligned} \tag{6.4}$$

Esta definición de factorial se podría codificar como se hace en el Ejemplo 6.2.

Ejemplo 6.2 Factorial calculado de manera recursiva

```
fn factorial_recursivo(n: u32) -> u32 {
   match n {
      0 => 1,
      _ => n*factorial_recursivo(n-1)
   }
}
```

6.2 Variables globales

Vamos a ver en primer lugar cómo no se pueden usar las variables globales en Rust. Podríamos estar tentados de usar sentencias *let*, como en la variables locales de cualquier función:

```
use chrono::Utc;

let START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El código anterior no compila, no se pueden usar asignaciones *let* en el ámbito global, pues la instrucción *let* crea una variable en la memoria stack, que no se inicializa hasta que se ejecuta el programa. Las variables globales solo se

pueden crear utilizando las cláusulas *const* o *static*, que utilizan la memoria del segmento de datos (*data segment*) del programa.

Tampoco se puede compilar la siguiente inicialización:

```
use chrono::Utc;

static START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El compilador avisa que no se pueden utilizar funciones no constantes en la inicialización de variables globales. No se puede ejecutar ningún tipo de código antes de que el programa comience. El valor de una variable global debe ser conocido en el momento de la compilación, antes de la ejecución.

Tampoco valdría declarar la variable global y tratar de asignarle valor en *main()*:

```
use chrono::Utc;

static START_TIME: String;

pub fn main() {
   START_TIME =  = Utc::now().to_string();
   println!("{}", START_TIME);
}
```

El código anterior tampoco compila, el compilador nos avisa de que hay que asignar algún valor a la variable *static*. Podríamos pensar en asignarle un valor *None*

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   START_TIME = Some(Utc::now().to_string());
   println!("{}", START_TIME);
}
```

Pero el código anterior tampoco compila, el compilador nos avisa de que una variable global mutable solo se puede utilizar dentro de un bloque inseguro *unsafe*. Finalmente, si encerramos el código dentro de *main()* en un bloque *unsafe*, sí que compila:

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   unsafe{
      START_TIME =  Some(Utc::now().to_string());
      println!("{}",  START_TIME.clone().unwrap());
   };
}
```

Ahora, el código compila y se puede ejecutar, pero no parece una forma muy cómoda de utilizar, aunque en algunas ocasiones pudiera ser útil.

Vamos a ver otro problema relacionado con la extensión del dominio de las variables. En un programa como el anterior, no se necesitaría declarar la variable como global, se podría declarar dentro de la función *main()*. Pero suponga que lo que se quiere es utilizar la variable en un hilo diferente creado dentro de *main()*

```
use chrono::Utc;

pub fn main() {
   let start_time = Utc::now().to_string();

   let thread_1 = std::thread::spawn(||{
      println!("Started {}, called thread 1 {}", &start_time, Utc::now());
   });

   thread_1.join().unwrap();
}
```

Si tratamos de compilar este código, el compilador nos dirá que el hilo *thread_1* podría vivir más tiempo que la variable *start_time* que se ha creado en el marco de datos en el stack de la función *main()* y no está garantizado que la variable sobreviva lo suficiente. Nosotros, viendo el código, sabemos que al hacer el *join()* antes de salir de *main()* estamos garantizando esa supervivencia, pero el compilador no nos deja compartir con otros hilos variables que no tengan una vida útil *&static*.

Hay un par de soluciones posibles sin utilizar variables globales. La primera sería clonar la variable *start_time* y pasar a la closure la propiedad del valor clonado:

```
pub fn main() {
   let start_time = Utc::now().to_string();
   let cloned_start_time = start_time.clone();
   let thread_1 = std::thread::spawn( move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time, Utc::
   });
   thread_1.join().unwrap();
}
```

Esta solución puede servir para una variable de cadena de caracteres como la anterior, pero si hubiera que clonar una variable de mayor tamaño podría no ser la solución óptima. En esos casos se podría envolver la variable con un puntero *Arc*:

```
use chrono::Utc;
use std::sync::Arc;

pub fn main() {
   let start_time = Arc::new(Utc::now().to_string());
   let cloned_start_time = Arc::clone(&start_time);
   let thread_1 = std::thread::spawn(move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time,
   });

   thread_1.join().unwrap();
}
```

Si además se necesitara mutabilidad interior de la variable, se podría envolver en un `Arc<Mutex<String>>`.

6.2.1 Valor conocido en tiempo de compilación

Cuando el valor de la variable global se conoce en tiempo de compilación, hay básicamente dos soluciones:

- **const:** valores constantes que se conocen en tiempo de compilación. No permiten la mutabilidad interior. El compilador resuelve sustituyendo el valor en línea (inline)[1].
- **static:** las variables reciben un espacio de memoria en el segmento de datos. Es posible la mutabilidad interior.

Si se necesita mutabilidad interior, para los tipos primitivos se pueden utilizar valores *atomic* [2] y, para tipos más complejos, se pueden usar *locks* en la forma *read-write lock*, *RwLock* o en la forma *mutual exclusion lock*, *Mutex*.

Si lo que se necesita es calcular el valor en tiempo de ejecución, las soluciones con *const* y *static* no sirven.

[1] Conviene consultar el apartado de la cláusula *const* en la documentación en `https://doc.rust-lang.org/std/keyword.const.html` y del concepto de *expresiones constantes* en `https://doc.rust-lang.org/reference/const_eval.html`

[2] Ver libro en líne de Mara Bos `https://marabos.nl/atomics/`

CAPÍTULO 7

Railway Oriented Programming

El *Railway Oriented Programming* es un término acuñado por Scott Wlaschin que propone un modelo del tratamiento de errores en la programación funcional.

- Railway Oriented Programming: `https://fsharpforfunandprofit.com/rop/`
- Monoids without tears: `https://fsharpforfunandprofit.com/posts/monoids-wi`

7.1 Monoides

Morfismo: en varios campos de las matemáticas, se llaman *morfismos* (u *homomorfismos*) a las aplicaciones entre estructuras matemáticas que preservan la estructura interna. Por ejemplo, en teoría de conjuntos, los morfismos son las aplicaciones entre conjuntos; en álgebra lineal, las transformaciones lineales; y en topología, las funciones continuas. En *teoría de las categorías*, el morfismo tiene una noción más general.

Categoría: una categoría viene dada por dos tipos de datos: una clase de objetos y, para cada par de objetos X e Y, un conjunto de morfismos desde X hasta Y. En el caso de una categoría concreta, X e Y son conjuntos de cierto tipo y un morfismo *f* es una función desde X a Y que satisface alguna condición.

Los morfismos se representan frecuentemente como flechas entre los objetos. Esto origina la notación:

$$f : X \to Y \tag{7.1}$$

CAPÍTULO 8

NaN an Inf

https://stackoverflow.com/questions/14682005/why-does-division-by-zero

8.1 Artículos para recomendar a los alumnos

The Mediocre Programmer's Guide to Rust:

https://www.hezmatt.org/~mpalmer/blog/2024/05/01/the-mediocre-programmers-guide-to-rus
html

REFERENCIAS

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1994. ISBN 978-0-201-63361-0.

[2] Wikipedia. CLU (programming language). *Wikipedia*, January 2024.

[3] Eric Normand. *Grokking Simplicity: Taming complex software with functional thinking*. Manning Publications, July 2021.