Análisis Comparativo Entre Distintos Aspectos de Programación Funcional en Python y C++

A. Vekselman, *Instituto Tecnológico de Buenos Aires*

***Abstract*—En este trabajo se propone realizar un análisis comparativo entre los aspectos de programación funcional (FP, por las siglas en inglés) de C++ y de Python. Ambos lenguajes poseen características funcionales, como la mayoría de los lenguajes de programación actuales, pero ¿cuál de los dos es el indicado para la resolución de problemas de manera funcional? ¿Depende del problema o alguno de los dos tiene características definitivamente superiores? Para responder esto, se aplicarán distintas técnicas de FP en uno y otro lenguaje, y se evaluarán puntos como desempeño, facilidad de escritura, lectura y chequeo de errores, y potencial de uso. Se estudiarán aspectos como las expresiones lambda, las comprensiones (particularmente de listas), los functores, las funciones de orden superior, la vinculación de argumentos y las funciones variádicas. Luego de cada aspecto, se hará una conclusión parcial de los resultados obtenidos con dicha herramienta. Finalmente, se expresará una conclusión general a modo de cierre, en la cual se intentará responder a las preguntas planteadas previamente.**

I. Introducción

Para empezar a responder a preguntas como "¿Qué lenguaje de programación, entre Python y C++, debo elegir si quiero llevar a cabo un desarrollo funcional?" y "¿Hay uno realmente mejor que el otro?", es necesario primero tener en claro algunos conceptos relacionados a la programación funcional (FP).

En primer lugar, ¿qué es FP? Según Graham Hutton, de la Universidad de Nottingham:

> *La programación funcional es un estilo de programación que pone énfasis en la evaluación de expresiones, en lugar de la ejecución de comandos. Las expresiones en estos lenguajes son formadas usando funciones para combinar valores básicos.*[1]

¿Qué quiere decir esto? Esta expresión quiere decir que, a diferencia de la programación imperativa tradicional de C++, se intenta describir a los problemas en términos de funciones, más que en términos de órdenes. Al paradigma FP se lo suele considerar como un paradigma declarativo, en el cual no se le explica al procesador qué pasos debe seguir para resolver un problema, sino que se definen los parámetros del problema. Así, los programas se estructuran de forma tal que los pasos que se siguen son evaluaciones de funciones, para las cuales el procesador tiene "una explicación" dada por el programador.

A. Vekselman estudia en la Universidad Tecnológica de Buenos Aires (ITBA).

.

II. Conceptos Previos

Veamos un ejemplo de lo explicado anteriormente:

```
// Factorial con llamada a sí mismo.
int factorial_recursivo(int num) {
  if (num <= 1) return 1;

  return num * factorial_recursivo(num - 1);
}

// Factorial "tradicional".
int factorial(int num) {
  int tot = 1;

  for (int i = 2; i <= num; i++) tot *= i;
  return tot;
}

int main() {

// Imprime 720.
  cout << factorial_recursivo(6) << endl;

// Imprime 720.
  cout << factorial(6) << endl;
}
```

Fig. 1. Aplicación en C++ de la función factorial de forma imperativa y recursiva.

Vemos que, en el primer caso, la función se llama a sí misma. No se le dice expresamente a la función cómo se calcula el factorial de un número, sino que simplemente se le dice que evalúe los valores y devuelva el resultado de llamarse a sí misma con el número decrementado en uno, hasta que, si se lo programó correctamente, el programa llegará al caso base y se efectuarán los sucesivos *return* hasta salir de la función. En resumen: se le dice al procesador qué es un factorial, pero no se le dice expresamente cómo calcularlo. Es un estilo de programación claro, conciso y declarativo. Sin embargo, trae consigo un agregado de *overhead* o "costo extra", ya que se deben ir apilando las llamadas a funciones.

En el segundo caso, la función se escribe de forma imperativa, con un ciclo *for* explícito en el cual se van

[1] Graham Hutton, "Frequently Asked Questions for comp.lang.functional", University of Nottingham, Nottingham, Inglaterra, 2002

multiplicando los números sucesivos y luego sale, cuando la variable i deja de cumplir la condición especificada. Como se puede ver, se debe agregar una variable *tot* que mantenga el valor actual de la cuenta, lo cual no es necesario en la versión FP.

Nota: se dejó de lado el chequeo de errores en el input del número a evaluar, con el fin de simplificar el código.

A. *Aspectos Principales de FP*

Retomando el tema de FP, en general, los programas que se escriben suelen seguir ciertas reglas generales:

- Utilizar principalmente funciones puras, que son funciones que sólo pueden usar (pero no modificar) los argumentos que se les pasan con el fin de calcular el resultado.[2] Esto quiere decir que una función llamada con los mismos argumentos debería devolver siempre el mismo valor, independientemente del estado del programa que la está llamando.
- Mantener el valor de las variables: también conocido como inmutabilidad. Las funciones no deben modificar variables externas a ella, o que "no le pertenezcan". Estrictamente, una función no debería tener mutabilidad incluso dentro de sí misma.

Siguiendo en principio estas reglas, se podría comenzar a escribir con el paradigma de FP código cada vez más corto, fácil de leer, manejable y con un mínimo de puntos propensos a errores. Al definir los problemas en términos de sí mismo, o en términos de evaluaciones de funciones que no modificarán su entorno, la posibilidad de errores se minimiza y la detección de los errores que puedan ocurrir es mucho más veloz.

B. *Funciones Lambda*

Las funciones lambda son funciones anónimas que se escriben en la parte del código donde se desean usar (Čukić,2018). A fines del trabajo actual, eso es lo más importante que se necesita saber. Están basadas en el cálculo lambda, una piedra angular de la programación funcional. Siguiendo las características de este paradigma de programación, las funciones lambda facilitan la lectura y escritura del código, permitiendo crear funciones de corta duración. Como explica Čukić en su libro, no tendría sentido ir a una parte externa al código y nombrar funciones que serán usadas sólo una vez. Para eso aparecen las funciones lambda.

Nota: en este trabajo se hará uso indistinto de "funciones lambda" y "expresiones lambda", aunque técnicamente las soportadas por Python y C++ son las últimas.

C. *Comprensión*

Una herramienta muy usada en FP es la comprensión. En este trabajo, se ejemplificará esencialmente con listas, pero se puede extender a la mayoría de las colecciones. La comprensión es un método por el cual se crea una colección (a partir de ahora, repito, se hablará exclusivamente de listas) a partir de otra, en una sola línea, de manera concisa.

¿Qué ventajas trae esto? Como la gran mayoría de lo relacionado FP, trae consigo claridad en el código, reducción de la probabilidad de errores y una escalabilidad y maleabilidad mucho mayores.

D. *Functores*

Los functores son el punto en el que FP se encuentra con la programación orientada a objetos (OOP, por sus siglas en inglés). C++ es un lenguaje principalmente del estilo OOP, por lo cual ese aspecto está ampliamente desarrollado. Python, por su parte, también tiene un manejo de objetos que es comparable con el nivel de C++, con muchas similitudes y algunas marcadas diferencias. Los functores son simplemente clases que tienen sobrecargado el operador ( ). Esto permite que estos objetos sean "como funciones", en el sentido de que pueden ser llamados con argumentos, y se ejecutará una serie de pasos (o evaluaciones en funciones, si se plantea de manera funcional), para luego devolver un valor final (o no, dependiendo del caso).

E. *Funciones de Orden Superior*

Por último, otro concepto clave que se necesitará conocer es el de funciones de orden superior (HOF, por las siglas en inglés). Éstas son funciones que reciben o devuelven otras funciones. Un ejemplo:

```
template <class T, typename F>
void apply(T& iter, const F& func) {
  for (auto i = iter.begin(); i != iter
.end(); i++)
      *i = func(*i);
}
```

Fig. 2. Función en C++ que recibe como parámetro una función.

En la Figura 2 se puede observar un *template* de función que recibe un iterable de tipo T y una función de tipo F, y aplica la función a cada uno de los valores del iterable. Dada la versatilidad de los *templates* y la practicidad de FP, esta combinación resulta muy efectiva a la hora de resolver problemas que necesiten aplicar HOF. Como se observa, el código es extremadamente conciso y fácil de entender.

Entendidos estos conceptos, el siguiente paso será comparar las aplicaciones de FP en los dos lenguajes previamente mencionados: C++ y Python.

III. Análisis

A. *Funciones Lambda*

C++

La sintaxis en C++ consiste en, principalmente, los siguientes aspectos:

- [ ]: indicador de captura. Dice cuáles y de qué forma (valor o referencia) serán utilizadas las variables del entorno.
- ( ): indicador de parámetros. Los parámetros que recibe la función lambda. Sintaxis igual a la de funciones tradicionales.
- { }: indicador de cuerpo. Las acciones que lleva a cabo la función.

[2] Ivan Čukić, "Functional Programming in C++", Manning Publications, 2018.

Nota: Hay más puntos que puede tener una función lambda, como la mutabilidad de las variables pasadas por valor, un aviso sobre los *throw* que puedan aparecer dentro de la expresión y una explicitación del tipo de dato de retorno en caso de haber múltiples *return*, pero para el análisis actual no son tan relevantes.

Veamos un ejemplo:

```
int add1(int num) { return num + 1; }
int main() {

/*Defino los vectores a aplicarles las funciones.*/
  std::vector<int> v_function{1, 2, 3, 4};
  std::vector<int> v_lambda{1, 2, 3, 4};

  // v_function = {2,3,4,5}
  apply(v_function, add1);

  // v_lambda = {2,3,4,5}
  apply(v_lambda, [](int num) { return num + 1; });
                                                  }
```

Fig. 3. Ejemplo de expresiones lambda en C++.

En la Figura 3 se utilizó el *template apply* escrito en la Figura 2, pero aplicado de dos formas distintas a dos vectores distintos. En una primera instancia, se lo aplica con una función tradicional, definida afuera del *main*. En el segundo caso, se aplicó una expresión lambda, escrita en el momento.

¿Qué ventajas trajo este modo de escritura?

Para empezar, código más legible. Se puede mirar a la función y ver exactamente lo que está haciendo, escrito en el momento.

Por otro lado, código más manejable. Si se quiere modificar lo aplicado en esa llamada a *apply*, basta con ir a dicho punto y modificarlo. De la otra forma, habría que ir afuera del *main* y modificar la función add1, pero teniendo en cuenta a los puntos que potencialmente la puedan haber llamado desde afuera. Alternativamente, se podría definir otra función f que pusiera en práctica la nueva funcionalidad requerida, pero en ambos casos la alternativa lambda suena mucho más concisa y al punto.

Sin embargo, es importante notar que ambas formas de escritura son equivalentes. Los distintos paradigmas de programación logran convivir en C++, permitiendo aplicar uno u otro de manera que se considere conveniente en cada caso y para la aplicación específica.

PYTHON

Veamos un ejemplo de expresiones lambda en Python:

```
def add1(num):
    return num + 1

#v_function y v_lambda son [1,2,3,4]
v_function = list(range(1, 5))
v_lambda = list(range(1, 5))

#v_function = [2,3,4,5]
v_function = list(map(add1, v_function))

#v_lambda = [2,3,4,5]
v_lambda = list(map(lambda x: x + 1, v_lambda))
```

Fig. 4. Ejemplo de expresiones lambda en Python.

Nota: la función *map* es equivalente a la función *apply* de la Figura 2, donde se le brinda una función y un iterable. Devuelve un iterable del tipo *map*, construido a partir de la aplicación de la función a cada uno de los miembros del iterable.

Como se observa, la sintaxis es incluso más concisa que en C++. Consta simplemente de la palabra 'lambda' seguida por la lista de parámetros, dos puntos y el *return*. Las funciones lambda de Python también permiten aplicar discriminaciones if/else en su cuerpo, variando el *return* (o no) en cada caso.

Nuevamente, las ventajas son fácilmente apreciables entre las dos formas de escribir el mismo programa. Mayor facilidad de lectura, mayor versatilidad y menos posibilidad de errores.

COMPARACIÓN

Se vio que tanto C++ como Python tienen expresiones lambda, cada lenguaje con su sintaxis propia. Pero ¿qué diferencias hay entre ellas? Por un lado, veamos el siguiente ejemplo en Python:

```
#Cargamos la lista con las expresiones lambda.
v_lambdas = []
for i in range(1, 5):
    v_lambdas.append(lambda x: x + i)

#Imprime 9 9 9 9, porque i fue capturado del
entorno por referencia.
for lam in v_lambdas:
    print(lam(5), end=' ')

print('\n')

i = 0

#Imprime 5 5 5 5, porque el cambio de i afecta
a las funciones.
for lam in v_lambdas:
    print(lam(5), end=' ')
print('\n')
```

Fig. 5. Ejemplo de valores por referencia en lambda en Python.

Como se observa en la Figura 5, además de que en Python no hay forma de especificar modos de captura de los valores del entorno, los valores capturados son siempre por referencia. Esto, en muchas ocasiones, puede resultar beneficioso: no hay que llamar a los constructores innecesariamente, por ejemplo. Sin embargo, en otras ocasiones puede resultar perjudicial, como se ve en el ejemplo. Uno esperaría que cada una de las funciones lambda guardadas en *v_lambdas* devolvieran el argumento más el valor de la posición. Pero como i fue

capturado por referencia, todas hacen referencia al mismo i, por lo que cualquier cambio en el valor de éste afecta a todas por igual. Es por esto por lo que todas terminan devolviendo, en primera instancia, el argumento más el último valor que tuvo *i* al salir del ciclo *for*. Por otro lado, se ve que al configurar *i* = 0, todas pasan a devolver simplemente el argumento.

En cambio, en C++:

```
int main() {
/*Creamos tres vectores de funciones que reciben
un int y devuelven un int.*/
        vector<function<int(int)>> v_value;
        vector<function<int(int)>> v_ref;
        vector<function<int(int)>> v_no_capture;
        int i;
        for (i = 0; i < 4; i++) {
//Cargamos v_value con lambdas que reciben i por
valor.
                v_value.emplace_back([=](int
num) {return num + i; });

//Cargamos v_ref con lambdas que reciben i por
referencia.
                v_ref.emplace_back([&](int num)
{return num + i; });

        /*No compila, i no está definida.
                v_no_capture.emplace_back([](
int num) {return num + i; });*/
        }

//Imprime 5 6 7 8, porque i fue capturado por
valor.
        cout << "Value: ";
        for (auto x : v_value)
                cout << x(5) << ' ';

        cout << endl;


//Imprime 9 9 9 9, porque i fue capturado por
referencia.
        cout << "\nReference: ";
        for (auto x : v_ref)
                cout << x(5) << ' ';

        cout << endl;

        i = 0;

//Imprime 5 5 5 5, porque ahora i vale 0.
        cout << "\nNew reference: ";
        for (auto x : v_ref)
                cout << x(5) << ' ';

        cout << endl;
}
```

Fig. 6. Ejemplo de capturas por valor y referencia en lambda en C++.

Se ve en la Figura 6 que, haciendo uso de las clausuras ([ ]) se pueden configurar distintas formas de capturar los valores del entorno, haciendo que la función lambda sea mucho más versátil. En un primer caso, se toma *i* por valor, por lo que todas las expresiones lambda devuelven el argumento más la posición. El segundo ejemplo es equivalente a lo que en Python ocurre por defecto, que *i* está capturada por referencia, por lo que todas las funciones devuelven lo mismo. Comentado hay también un tercer caso, en el que no se especifica captura de ninguna variable del entorno, por lo que la expresión escrita no compilará, al no haber sido definida *i*.

Por otro lado, las funciones lambda de Python son una mezcla entre buenas prácticas funcionales y malas prácticas funcionales.

Adhieren menos al paradigma funcional en el sentido de que no hay protecciones en cuanto a la modificación de variables del entorno. Esto va directamente en contra del concepto de inmutabilidad y de puridad de funciones, ya que las expresiones lambda no deberían ni depender de variables del entorno ni poder modificarlas.

Sin embargo, también son “más funcionales”, es decir, adhieren más al paradigma funcional, en el sentido de que las expresiones lambda de Python son estrictamente declarativas. Son una declaración simple, especificando sus parámetros y sus valores de retorno. No se pueden poner múltiples acciones dentro de una expresión lambda (como hacer un *print* y luego devolver un valor). Para ello, se deben usar las funciones tradicionales, definidas con *def*.

Sin embargo, en C++ se puede hacer algo así:

```
int main() {
        int value = 0;

//Recibe por referencia a value. Equivalente, en
este caso, a poner sólo &.
        auto func = [&value](auto& x) {

                cout << x << endl;
                if (value)
                        x = x + x;
                return x + x;
        };

        string s = "Hola";
        int num = 3;

        /*Imprime 3.
        res1 = 6. */
        auto res1 = func(num);

        //Modificar value modifica a func.
        value = 1;

        /*Imprime "Hola".
        s = "HolaHola".
        res2 = "HolaHolaHolaHola".*/
        auto res2 = func(s);

}
```

Fig. 7. Ejemplo de programación imperativa en lambda en C++.

Vemos en la Figura 7 cómo las expresiones lambda de C++ permiten realizar cualquier acción que se podría realizar dentro de una función normal, con la clara ventaja de que los parámetros pueden ser del tipo auto, es decir, las funciones

lambda se convierten en un *template*.

Por último, cabe destacar que no sólo las funciones lambda, sino todas las funciones en Python actúan como *template*, en el sentido de que no es necesario especificar el tipo de dato, por lo que una misma función podría recibir como argumento distintos tipos de dato y funcionar correctamente. Ejemplo:

```
#Devuelve x+x si x es entero o lista, o devuelve
x en otro caso.
func = lambda x: x+x if isinstance(x,int) or isin
stance(x,list) else x

#Imprime 2.
print (func(1))

#Imprime [1,2,3,1,2,3]
print (func([1,2,3]))

#Imprime {'Hola' : 1, 'Chau' : 2}
print (func({'Hola' : 1, 'Chau' : 2}))

#Imprime range(0,5)
print (func(range(5)))

#Imprime "Hola"
print (func("Hola"))

#Imprime 1.6
print (func (1.6))

#Imprime <function<lambda>>
print (func(lambda x: x+1))
```

Fig. 8. Ejemplo de lambda aplicada a varios tipos de dato en Python.

Se observa claramente en la Figura 8 cómo una misma función lambda funciona correctamente para enteros, listas, diccionarios, rangos, *strings*, *floats* y hasta para otra función lambda. Esta versatilidad es de gran utilidad en la programación funcional, porque permite reutilizar aún más las mismas funciones y reducir aún más la proporción de errores que se puedan llegar a cometer.

Nota: No se ahondará en claridad y legibilidad del código, tiempo de ejecución ni ocupación de memoria, porque son características propias de cada lenguaje y no tanto de la forma que tenga el lenguaje de expresar las funciones lambda.

B. *Comprensión*

C++

La comprensión no es una característica nativa de C++. Si bien hay librerías *open source* que intentan recrearla, principalmente a partir del uso de la librería *Boost::Phoenix*[3] o *range_v3* (que luego dio lugar a *C++20 Ranges*)[4] y algoritmos como *transform* o *for_each*, no se tocará el tema por irse del alcance de la monografía. Pero resulta relevante mencionar al menos su existencia, la cual no se está pasando por alto.

Python

En Python, por otro lado, la comprensión es moneda de uso corriente. Al ser un lenguaje orientado principalmente a *Data Science*, Python está preparado para trabajar con grandes cantidades de datos. En ese proceso, generalmente se busca convertir esos datos de un formato a otro, aplicándoles en medio alguna función, y guardarlo rápidamente en otra lista. Para evitar crear funciones extra, listas vacías e iteraciones innecesarias, se utiliza la comprensión de listas. Ejemplo de ambas formas:

```
#Lista1 = [0,1,2,3,4]
lista1 = list(range(5))

#Se debe declarar lista vacía primero.
lista2 = []

#Lista2 = [2,4,6,8,10]
for value in lista1:
    lista2.append((value+1) * 2)

#Equivalentemente...
#En una línea, se logró que lista3 = [2,4,6,8,10]
lista3 = [2*(i+1) for i in lista1]

#Otra forma, con lambdas y comprensión.
my_func = lambda x: 2*(x+1)

#Lista4 = [2,4,6,8,10]
lista4 = [my_func(i) for i in lista1]
```

Fig. 9. Comprensión de listas en Python.

En la Figura 9 se observa la facilidad de escritura de la comprensión de listas en Python, tanto así que el código en sí suele ser más explicativo que los comentarios al respecto. En una línea se pudo escribir lo que de otra forma se habría tenido que escribir en tres, y además se lo hizo de forma mucho más expresiva. Cabe destacar que las comprensiones de listas tienen la misma sintaxis que una función lambda, por lo que podrían aplicarle lógicas condicionales dentro de la comprensión.

Comparación

En este caso, no se puede llevar a cabo una comparación, porque en Python las comprensiones son tan poderosas como fáciles de usar, mientras que en C++ implican inclusión de librerías y contenido que todavía está en vías de desarrollo. En este sentido, Python tiene superioridad en cuanto al paradigma FP, permitiendo usar esta herramienta que es extremadamente común en la resolución de problemas de manera funcional.

C. *Functores*

¿Qué ventajas tienen los functores? Dado que las implementaciones tanto en Python como en C++ son similares, veamos su uso con un ejemplo.[5]

[3] Joel de Guzmán, Dan Marsden, Thomas Heller, John Fletcher, "Phoenix 3.2.0", Boost Software, 2015.

[4] Autor desconocido, Ranges Library, CPP Reference, https://en.cppreference.com/w/cpp/ranges, 2020.

[5] Ejemplo tomado de Autor desconocido, Functors in C++, GeeksforGeeks, https://www.geeksforgeeks.org/functors-in-cpp/, sin fecha especificada.

C++

Problema: Partiendo de la función apply de la Figura 2, si se quisiera hacer una función que sume uno, una forma de hacerlo sería la siguiente:

```
int increment (int num){ return num + 1; }
int main() {
        vector<int> v{ 1,2,3,4 };

        //v = {2,3,4,5}
        apply(v, increment);
}
```

Fig. 10. Planteo del problema que los functores resuelven.

Sin embargo, esto se complica cuando se quiere hacer un incremento de otro valor, ya que la función que se le debe pasar a apply sólo puede recibir un argumento. Si se quisiera incrementar los valores en cinco, por ejemplo, se debería hacer lo siguiente:

```
int increment (int num){ return num + 1; }
int add5 (int num){ return num + 5; }
int main() {
        vector<int> v{ 1,2,3,4 };

        //v = {2,3,4,5}
        apply(v, increment);

    //v = {7,8,9,10}
    apply(v, add5);
}
```

Fig. 10.1. Parte 2 del problema que los functores resuelven.

Esto puede resultar tedioso, ya que justamente el paradigma de FP trata sobre la reducción de excesos y la no implementación de funciones innecesarias. Con functores, la solución sería así:

```
//Functor.
class Adder {
public:
        Adder(const int& num_) : num(num_) {};

        int operator()(const int& nn) const {
return nn + num; }

private:
    int num;
};

int main() {
        vector<int> v{ 1,2,3,4 };

    //Ahora v = {2,3,4,5,6}
        apply(v, Adder(1));

     //Ahora v = {7,8,9,10,11}
        apply(v, Adder(5));
}
```

Fig. 11. Solución del problema con functores en C++.

Así, se soluciona el problema, lográndolo generalizar a través de objetos que sirven como funciones. Además, esto posee otra ventaja: los objetos pueden guardar más información que sólo el constructor y la sobrecarga del operador ( ). Potencialmente, esta propiedad tiene múltiples usos, ya que se pueden mantener registros de los procedimientos que se van haciendo, o se pueden realizar procedimientos en paralelo con los datos obtenidos.

Python

Equivalentemente, la situación en Python sería la siguiente:
Problema:

```
increment = lambda x: x+1

add5 = lambda x: x+5

#Lista1 = [1,2,3,4]
lista1 = list(range(1,5))

#Lista1 = [2,3,4,5]
lista1 = list(map(increment,lista1))

#Lista1 = [7,8,9,10]
lista1 = list(map(add5,lista1))
```

Fig. 12. Planteo del problema que resuelven los functores en Python.

Solución:

```
#Functor
class Adder:
    def __init__(self,num):
        self.num = num

    def __call__(self,nn):
        return self.num + nn

#Lista1 = [1,2,3,4]
lista1 = list(range(1,5))

#Lista1 = [2,3,4,5]
lista1 = list(map(Adder(1),lista1))

#Lista1 = [7,8,9,10]
lista1 = list(map(Adder(5),lista1))
```

Fig. 13. Solución del problema con functores en Python.

Comparación

Como se mencionó antes, vemos claramente que la solución del problema es muy similar en ambos lenguajes, limitándose a la creación de una clase *Adder* con un constructor que recibe un número y lo guarda en una variable propia, y la sobrecarga del operador ( ), con la cual devuelve la suma entre el número pasado como argumento y su propio número.

En cuanto a ventajas de Python, nuevamente vuelve a aparecer la generalidad que se obtiene al no tener que especificar tipos de datos. En C++, para implementar el mismo caso con *float*, se debería sobrecargar nuevamente el operador ( ), o escribir un *template* para la clase *Adder*, donde se haga de

manera más genérica. Ambas soluciones son más tediosas que en Python, donde nativamente no se especifica tipo de dato. Aunque, cabe destacar, en el caso particular de estudio, el operador suma que se aplica en la sobrecarga al operador ( ) ya es en parte restrictivo, dado que su correcto funcionamiento dependerá de con qué tipo de dato se construyó la instancia de la clase.

Sin embargo, una clara desventaja de Python radica en su manejo de los métodos y datos miembro de las clases. En las clases de Python, y en particular en los functores, no existen restricciones sobre quién puede acceder a los datos miembro y a sus métodos, y más aún, cualquiera puede crear nuevos datos miembro y nuevos métodos desde afuera de la clase. Si bien esto puede resultar ventajoso en muchas ocasiones, en este caso resulta más que nada un impedimento para el funcionamiento "hermético", puro, que se busca en la programación funcional. Si se espera que cada vez que se llame a una función, el resultado sea el mismo, entonces es lógico pensar que no se deberían poder modificar los pasos que sigue dicha función.

Nota: Si bien existen formas de hacer que en apariencia haya métodos o datos miembro que sean inaccesibles desde afuera de una clase, estas formas son fácilmente *bypasseables*, es decir, existen maneras de salteárselas.[6] Por este motivo, a fines de esta monografía, se considera que no existen tales maneras.

Ejemplos:

```
#Functor
class Adder:
    def __init__(self,num):
        self.num = num

    def __call__(self,nn):
        return self.num + nn

#Ahora Adder es un restador, no un sumador.
Adder.__call__ = lambda adder,nn: nn - adder.num
```

Fig. 14. Sobreescritura de método __call__ en Python.

Se puede ver en la Figura 14 un ejemplo de un problema leve que podría ocasionar la falta de seguridad en las clases de Python. En este caso, la clase *Adder* que se había definido, pasó en una sola línea a tener una funcionalidad opuesta a la que tenía originalmente. Problemas de este estilo son los que C++ intenta evitar con las declaraciones *private*, *public*, *protected* y sus variaciones.

```
#Functor
class Adder:
    def __init__(self,num):
        self.num = num

    def __call__(self,nn):
        return self.num + nn

#Sobreescritura de método __call__.
Adder.__call__ = lambda x: x+1

add3 = Adder(5)

#El programa se rompe, porque ahora Adder.__call__
#está recibiendo dos parámetros, pero espera uno
print (add3(1))
```

Fig. 14.1. Otra sobreescritura de método __call__ en Python.

```
#Functor
class Adder:
    def __init__(self,num):
        self.num = num

    def __call__(self,nn):
        return self.num + nn

#Sobreescritura de método __init__.
Adder.__init__ = lambda adder, num: None

add3 = Adder(5)

#El programa se romperá, porque add3.num ahora
#no está definida.
print (add3(1))
```

Fig. 14.2. Sobreescritura de método __init__ en Python.

En los casos de la Figura 14.1 y 14.2, los problemas que puede ocasionar esta falta de seguridad son mucho más severos que en la figura 14. En estos casos, el programa directamente se romperá.

En la figura 14.1, el programa se romperá porque se está intentando llamar a la sobrecarga *__call__* con dos parámetros (la instancia y el número), pero la sobreescritura hizo que ahora *__call__* tome sólo un parámetro. Por lo tanto, el programa correrá con errores.

En la figura 14.2, el programa se romperá porque se sobreescribió el método *__init__*, haciendo que nunca se cree la variable add3.num. Por lo tanto, al querer utilizarla en el método *__call__*, habrá un error.

En resumen, la utilización de functores es una solución práctica y efectiva para el problema planteado en las figuras 10 a 13, pero su efectividad dependerá del caso de uso y del lenguaje. En el caso de C++, posee ventajas por ser un lenguaje destinado a OOP, por lo cual el manejo de objetos será mucho más eficiente. Sin embargo, su versatilidad se ve reducida por la obligación de declarar tipo de dato (o la utilización de *templates*, que aún así tienen limitaciones). Por otro lado, Python posee la versatilidad de que sus funciones son ya de por sí una especie de *template*, pero falla en cuanto al manejo de los objetos, pues se incrementa la probabilidad de errores al no tener una forma "a prueba de balas" de disponer tanto los métodos como los datos miembro de la clase.

D. Funciones de orden superior

Veamos el problema del ejemplo anterior, pero resuelto con HOF.

[6] nikhilaggarwal3, Private Methods in Python, GeeksforGeeks, https://www.geeksforgeeks.org/private-methods-in-python/, sin fecha especificada.

C++

En C++, otra forma de resolver el problema que se aplicó para functores sería la siguiente:

```
int main() {
  /*Recibe un parámetro y devuelve una función que recibe un parámetro y devuelve la suma del parámetro nuevo y el original.*/
  auto HOF = [](const auto& adder) {
    return [=](const auto& changeable) { return changeable + adder; };
  };

  vector<int> v{1, 2, 3, 4};

  // v = {2,3,4,5}. Sumó 1.
  apply(v, HOF(1));

  // v = {7,8,9,10}. Sumó 5.
  apply(v, HOF(5));
}
```

Fig. 15. Solución al problema planteado para functors en C++ usando HOF.

Como se ve en la Figura 15, una forma alternativa de resolver el problema es con HOF. Se declara una función HOF, que recibe un parámetro *adder* y devuelve otra función. Esa función que devuelve a su vez recibe un argumento y devuelve la suma del parámetro *adder* y el argumento.

Además de la aplicación con funciones lambda, C++ también permite la manipulación de punteros a funciones, los cuales pueden ser tanto pasados como parámetros a funciones y devueltos por ellas.

Python

Equivalentemente, en Python se resolvería de la siguiente manera:

```
#Recibe un parámetro y devuelve una función
#que recibe otro parámetro y devuelve la suma
#de ambos parámetros.
HOF = lambda x: lambda y: x+y

#Lista1 = [1,2,3,4]
lista1 = list(range(1,5))

#Lista1 = [2,3,4,5]. Sumó 1.
lista1 = list(map(HOF(1),lista1))

#Lista1 = [7,8,9,10]. Sumó 5.
lista1 = list(map(HOF(5),lista1))
```

Fig. 16. Solución al problema planteado para functor con HOF en Python.

Se observa en la figura 16 que, al igual que en el caso de C++, HOF es una función lambda que recibe como parámetro un valor y devuelve una función, que a su vez recibe como parámetro otro valor y devuelve la suma del valor original y el último.

Las funciones definidas con *def* también se pueden pasar como parámetro a funciones y ser devueltas.

Comparación

Si bien a primera vista se podría pensar que la implementación en Python es más conveniente, dado (sí, nuevamente) que no hay que especificar tipo de dato, la realidad es que C++ tiene con qué competir. A partir de C++11, el *header* <*functional*>[7] de la STL pasó a incluir muchas características de la programación funcional que facilitan mucho el uso de funciones en este lenguaje. Ejemplos de estas características son el *template* de clase *std::function*, que sirve de *wrapper* para guardar cómodamente funciones dentro de una variable, o la función *std::bind*, de la cual se hablará más en detalle en el próximo apartado.

Sin embargo, a pesar de las facilidades que brinda este *header*, todavía sigue siendo complejo del manejo de HOF debido a que muchas veces no hay conversiones definidas entre un tipo de dato y otro, por lo cual las implementaciones pueden no ser compatibles.

Ejemplo:

```
typedef int(*function_ints) (const int&);

/*HOF recibe un entero (num) y devuelve una función que recibe un número y devuelve la suma entre ese número y num.
function_ints HOF(const int& num) {

        function<int(const int&)> temp_func = [=](const int& num2) {return num + num2; };

/*No compila, porque no hay conversión definida entre function<int(const int&)> y function_ints.*/
        return temp_func;
}
```

Fig. 17. Ejemplo de falta de conversión entre tipos de funciones similares.

En el ejemplo de la figura 17, se intentó convertir una variable del tipo *function* que envuelve a una función que recibe un *const int&* y devuelve un *int*, a un puntero a función con mismas especificaciones. Como se ve en la figura, el programa no compila, ya que no hay una conversión definida entre esos tipos de dato.

En cambio, como ya se ha visto antes, en Python esto no habría tenido problema, ya que el programador no debe hacer declaración explícita del tipo de dato que se manejará en una función.

Además, como suele ser el caso, la sintaxis es mucho más amigable en Python, ya que se intenta que sólo se deba escribir lo estrictamente necesario, abstrayendo al programador de los procesos internos que pueda estar realizando cada función. Sin embargo, un aspecto que no se ha mencionado hasta ahora sobre Python es el hecho de que su facilidad y simpleza de código vienen en detrimento de velocidad de procesamiento y memoria ocupada. Al ser un lenguaje interpretado (en contraposición con

[7] Header <functional>, CPlusPlus, http://www.cplusplus.com/reference/functional/, sin autor especificado.

C++, que es compilado), el procesador debe realizar muchas más acciones para la ejecución de cada acción, lo cual, como se mencionó, resulta en programas mucho más lentos. Es por esto que en muchas aplicaciones en las que se tiene una cantidad limitada de memoria y/o se necesita que las velocidades de procesamiento sean altas, se le da preferencia a lenguajes como C++ en lugar de a lenguajes como Python.

E. Vinculación de argumentos

La vinculación de argumentos es una nueva definición de alguna función, en la cual alguno o algunos de sus argumentos están fijos en algún valor. Se utiliza mucho en programación funcional para evitar repetición múltiple de funciones que son similares, salvo con algún o algunos argumentos ya fijos.

C++

En C++, la vinculación de argumentos es una herramienta muy poderosa. Está soportada en el *header <functional>*, a través de la función *std::bind*, que recibe una función, seguida de parámetros a fijar o *std::placeholders*, que hacen referencia a la posición de la llamada a función.

Veamos un ejemplo, para que se entienda mejor:

```
//Recibe dos números y devuelve su suma.
int adder(const int& num1, const int& num2) {
        return num1 + num2; }

//Recibe un número y devuelve adder con primer
parámetro fijo en num.
auto HOF(const int& num) {
        return bind(adder, num, placeholders::_1);}

int main() {

        auto increment = HOF(1);

        auto add5 = HOF(5);

        //Imprime 6.
        cout << increment(5) << endl;

        //Imprime 10.
        cout << add5(5) << endl;
}
```

Fig. 18. Solución al problema planteado para functores en C++ usando HOF y std::bind.

Como se ve en la figura 18, ésta es otra alternativa a la solución del problema de los functores, utilizando *std::bind*. Como está explicado en el código, HOF ahora es una función que devuelve una "versión" de la función *adder*, pero con el primer parámetro ya fijado en el parámetro que se le pase a HOF. Eso convierte al valor de retorno en una función que, como se buscaba, recibe un número y devuelve la suma entre ése y el número con el que se la haya construido. Allí, *placeholders::_1* hace referencia al primer valor de la llamada a función (en este caso, el único).

[8] Functools, Documentaion of Functional Programming Modules of the Python Standard Library, Python.org, https://docs.python.org/3/library/functools.html, sin autor especificado.

Python

En Python, la vinculación de argumentos no es una herramienta tan común. Al tener tanta facilidad y libertad para crear y modificar funciones, evidentemente no se consideró que fuera una herramienta tan necesaria. Sin embargo, existe, aunque tiene sus limitaciones. Éstas serán explicadas en el apartado de comparación, cuando se contrapongan con la implementación en C++.

La vinculación de argumentos en Python está soportada por la libería *functools*[8], a través de la función *partial*. Veamos un ejemplo, resolviendo nuevamente el problema de los functores:

```
#Devuelve la suma de dos números.
adder = lambda x,y: x+y

#Devuelve adder con el primer parámetro fijado
en num.
HOF = lambda num: partial(adder,num)

increment = HOF(1)
add5 = HOF(5)

#Imprime 6.
print (increment(5))

#Imprime 10.
print (add5(5))
```

Fig. 19. Solución al problema planteado para functors en Python usando HOF y partial de functools.

Se observa en la figura que la función *partial* está devolviendo un *wrapper* que envuelve a la función *adder*, y fija el primer parámetro en 1 y 5, respectivamente.

Su sintaxis es idéntica al llamado de las funciones a las que se les quiera fijar los parámetros, es decir, se pueden fijar los argumentos tanto posicionalmente como con palabra clave. Ejemplo:

```
def divider(num1, num2):
    return num1/num2

HOF_first = lambda num: partial(divider,num1=num)
HOF_second = lambda num: partial(divider,num2=num)

#Función que recibe num y devuelve 1/num.
divide_1 = HOF_first(1)

#Imprime 0.2.
print (divide_1(num2 = 5))

#Función que recibe num y devuelve num/5
divide_by_5 = HOF_second(5)

#Imprime 0.2.
print (divide_by_5(1))
```

Fig. 20. Ejemplo de partial en Python con parámetros de palabra clave.

En la figura, se ve que haciendo uso de las palabras clave

num1 y num2, se puede elegir qué parámetro se desea fijar (equivalente al número de *placeholder* en C++).

Comparación

Si bien se ha visto que en ambos lenguajes la fijación de parámetros es relativamente similar, en el ejemplo de Python de la figura 20 se puede ver que al llamar a divide_1, se debe especificar que el argumento que se intenta pasar es num2. ¿Por qué ocurre esto? Aquí entra en juego las diferencias en la implementación de fijación de parámetros en C++ y Python.

En C++, se podrían aplicar funciones del siguiente estilo sin ningún problema:

```
void printer(int num1, int num2, int num3) {
        cout << num1 << num2 << num3 << endl;
}

int main() {
        auto first_fixed = bind(printer, 5, placeholders::_1, placeholders::_2);

        //Imprime 5 1 2.
        first_fixed(1, 2);

        auto second_fixed = bind(printer, placeholders::_1, 5, placeholders::_2);

        //Imprime 1 5 2.
        second_fixed(1, 2, 3, 4);

        auto all_first = bind(printer, placeholders::_1, placeholders::_1, placeholders::_1);

        //Imprime 1 1 1.
        all_first(1, 2, 5);

        auto all_binded = bind(printer, 1, 2, 5);

        //Imprime 1 2 5.
        all_binded(4, 5, 6, 7);
}
```

Fig. 21. Aplicaciones de std::bind en C++.

Se puede ver en la figura 21 que la función *std::bind*, en conjunto con los *std::placeholders* son herramientas sumamente poderosas en C++, permitiendo hacer todo tipo de combinaciones y vinculaciones de argumentos. El primer ejemplo de la figura 21 es un ejemplo tradicional, donde se fija el primer argumento y los otros dos se dejan con *std::placeholders::_1* y *std::placeholders::_2*, respectivamente. Esto quiere decir que cuando se llame a *first_fixed(num1,num2)*, num1 irá a parar a donde se puso *std::placeholders::_1* y num2 irá a donde se puso *std::placeholders::_2*. En el segundo caso, se ve una implementación similar al caso anterior, pero fijando el segundo parámetro, y además mostrando que se puede pasar una cantidad de argumentos cualquiera, y el programa simplemente los ignorará, porque ya se dejaron en claro con los *placeholders* qué posiciones se desea tomar y qué se desea hacer con ellas. Ésta es una característica muy buscada en FP, ya que se busca que los errores que se puedan generar sean los menos posibles. El tercer ejemplo muestra que si en todos los parámetros se pone, por ejemplo, el *placeholders::_1*, entonces en todos los valores se utilizará el valor que se ponga en la primera posición, independientemente de lo que se ponga en el resto. Por último, se ve en el último ejemplo un caso de vinculación de todos los parámetros, por lo que ahora *all_binded* es una función que no recibe argumentos. También se pueden declarar un *std::bind* dentro de otro, siguiendo ciertas reglas con respecto a los *placeholders*, pero siempre con una libertad y control bastante positivos.

En contrapartida, Python, sorprendentemente, ofrece muchas menos posibilidades en el campo de la vinculación de parámetros. Como se mencionó, en el ejemplo de la figura 20 sólo se podía llamar a divide_1 si se especificaba que el valor que se quería pasar era num2. ¿Y qué pasaría si no se especificara? Veamos:

```
    print (divide_1(5))
TypeError: divider() got multiple values for argument 'num1'


Process finished with exit code 1
```

Fig. 22. Ejemplo de limitaciones de partial en Python.

Como se puede ver en la figura 22, se obtiene un *TypeError* con la leyenda “La función *divider*() recibió múltiples valores para el argumento num1”. ¿Por qué ocurrió esto? Porque al declarar:

```
divide_1 = HOF_first(1)
```

Lo que estamos haciendo es decirle que el primer parámetro (es decir, num1) valdrá 1. Al no haber *placeholders*, no teníamos forma de decirle al intérprete que, si bien estábamos vinculando num1, queríamos que el primer valor pasado en la llamada a función fuera el relacionado con num2. Entonces, como sigue realizando la llamada tradicional a función, asume que, en el siguiente código, el número que se está intentando pasar también es el que va para num1:

```
print (divide_1(5))
```

Por lo tanto, aparece el *TypeError* indicando que, en esa llamada a función, le pedimos que num1 valiera tanto 1 como 5. Solucionar esto implicaría no especificar que el parámetro que se quiere fijar es num1, pero esto sólo es aplicable para el primer parámetro y no para el resto, que deben especificarse sí o sí por palabra clave (en caso de que se quieran fijar parámetros salteados, y no los *n* primeros). Así, como se observa, si bien la sintaxis de los *placeholders* puede ser tediosa debido a los largos *namespaces*, éstos son extremadamente útiles y logran resolver de manera muy versátil el problema que en Python pareciera no tener solución.

Sin embargo, como se mencionó antes, la vinculación de parámetros en Python tiene sintaxis idéntica a la de la llamada a función. Por lo tanto, utilizando los argumentos por palabra

clave, el problema quedaría solucionado (con un poco más de código, sí, y debiendo conocer los nombres de los argumentos). Pero esto no es del todo cierto. A partir de Python 3.8, la última versión, se permitió una forma de sintaxis que puede bloquear el uso de argumentos con palabra clave, es decir, se puede especificar en la declaración de una función qué argumentos se quiere que sean exclusivamente posicionales. ¿Cómo? Veamos:

```
def show(num1, num2):
    print ("First: ", num1, "Second: ", num2)


def show_posic(num1,num2,/):
    print ("First: ", num1, "Second: ", num2)


#Imprime:
#First: 5 Second: 1
show(num2=1,num1=5)

#Arroja TypeError.
show_posic(num2=1, num1=5)
```

Fig. 22. Argumentos exclusivamente posicionales en Python.

Como se ve en la figura 22, la utilización de "/" en la definición de la función le avisa al intérprete que todos los argumentos a la izquierda del símbolo "/" deben ser exclusivamente posicionales. Por lo tanto, al intentar llamar a la función *show_posic* con argumentos de palabra clave, el programa colapsa. Así, la solución que habíamos encontrado para el problema de *partial* en Python no funciona en el caso de que los argumentos sean sólo posicionales. ¿Y por qué nos importaría este caso particular? Porque muchas funciones actuales de la librería estándar de Python están siendo escritas de esta forma, siendo el ejemplo típico el caso de la función pow. Ésta recibe un argumento x y un argumento y (y un argumento z, pero no nos importa para el ejemplo), y devuelve x elevado a la potencia de y. Sin embargo, estos argumentos son estrictamente posicionales, entonces se puede hacer:

```
    #Recibe un número num y devuelve 2^num.
        elevate_two = partial(pow,2)
```

Pero no se puede hacer:

```
#Debería recibir un número num y devolver num^2.
        squared = partial(pow,y=2)
```

Este último ejemplo arroja un *TypeError*, avisando que *pow* recibió un argumento por palabra clave, pero sus argumentos son posicionales.

Se puede notar que, a partir del uso de *std::bind* y *std::placeholders*, <*functional*> de C++ es claramente superior en este aspecto que la librería *functools* de Python. Logra ofrecer un rango de posibilidades mucho más amplio, permitiendo al programador escribir código en un nivel más alto de FP y hacer uso de poderosas herramientas de este paradigma de programación.

F. *Funciones variádicas*

El último aspecto de programación funcional del cual se hablará en esta monografía son las funciones variádicas. Éstas son funciones que pueden recibir una cantidad variable de argumentos. Si bien no son un concepto tan importante de la programación funcional, definitivamente vale la pena mencionarlas, porque pueden llegar a resultar muy útiles a la hora de escribir un programa genérico y a prueba de errores.

Veamos su implementación en Python y C++.

C++

En C++, las funciones variádicas no son de uso tan común, ya que es parte de su filosofía el "tener que especificar lo que se usa", sin dejar lugar a que el compilador tenga que adivinar, y reduciendo la cantidad de cálculos que se tienen que hacer en tiempo de ejecución. Sin embargo, las funciones variádicas sí están soportadas en C++, incluyendo el *header* <*stdarg.h*>[9]. Su implementación es la siguiente:

```
//Recibe letras y devuelve la palabra junta.
string join(int cant, ...) {
        va_list args;
        va_start(args, cant);
        string res;
        for (int i = 0; i < cant; i++) {
                res.append(args);

                va_arg(args, const char*);
        }

        va_end(args);
        return res;
}

int main() {
        //Imprime "Hola".
        cout << join(4, 'H', 'o', 'l', 'a') << ' ';

        //Imprime "Mundo".
        cout << join(5, 'M', 'u', 'n', 'd', 'o')  <<
endl;
}
```

Fig. 23. Implementación de funciones variádicas en C++.

Como se observa, la función *join* está recibiendo una cantidad variable de parámetros, y el programa está funcionando correctamente. En C++, los argumentos variables de una función se denotan con "…" en su definición, siempre como último argumento. En este ejemplo, se solicitó un parámetro *cant* para saber cuántas letras se tendrían que unir a la palabra final. Con <*stdarg.h*>, la lista de argumentos de tamaño variable se guarda en una clase *va_list*. Luego, se llama al método *va_start*, que carga en la *va_list* los argumentos. Para iterar sobre la lista, se utiliza *va_arg*, que carga en la *va_list* el valor del próximo argumento. Por último, se llama a *va_end* para liberar la memoria.

[9] Header <cstdarg> (stdarh.h), CPlusPlus, http://www.cplusplus.com/reference/cstdarg/, sin autor especificado.

PYTHON

```python
#Recibe un valor y una serie de funciones y
#devuelve el resultado de aplicarle
#sucesivamente las funciones a ese valor.
def HOF (value, *args):
    temp = copy.deepcopy(value)
    for func in args:
        temp = func(temp)

    return temp

def squared(num):
    return num**2

def sqrt(num):
    return num**0.5

#num2 = 3*2 + 6 = 12
num2 = HOF(3,lambda x: x*2, lambda x: x+6)

#num3 = (12/4)^2 + 12/4 = 12
num3 = HOF(12,lambda x: x/4, squared, lambda x
: x + sqrt(x))
```

Fig. 24. Implementación de funciones variádicas en Python.

En Python, la implementación de las funciones variádicas se hace especificando un parámetro que comienza con asterisco, generalmente identificado con la palabra "args". Vemos en la figura 24 que se le pueden pasar tanto dos como tres, o la cantidad de parámetros (funciones, en este caso) que se quiera (incluso ninguna) y de cualquier tipo. En el contexto de FP, esta característica suele ser atractiva y útil, ya que no sólo evita tediosos chequeos de errores, sino que además reduce código y hace que los resultados que se puedan obtener sean mucho más generales.

Nota: No se habló de los parámetros de tamaño variables generalmente identificados como ***kwargs*, pero sólo difieren de **args* en que estos últimos son parámetros sin nombre, mientras que los primeros tienen una palabra clave que los identifica. Y en lugar de llegar en forma de tupla, como llegan los parámetros de **args*, llegan en forma de diccionario. **args* y ***kwargs* suelen aparecer juntos.

COMPARACIÓN

A diferencia de en C++, en Python esta característica es usada en casi todos los códigos que se jacten de ser "correctos", ya que las buenas prácticas indican que así debe ser. Tanto en la librería estándar como en librerías más utilizadas para *Data Science* (como *NumPy*, *Pandas*, *SciPy*, *MatplotLib*), se suele encontrar que al final de los prototipos de las funciones aparece la palabra clave **args* (y ***kwargs*), dejando lugar a que el usuario ingrese la cantidad de parámetros que desee, también evitando así que el programa se rompa por ingresar cantidades no esperadas. Esto se debe principalmente a su facilidad de uso, ya que Python de forma nativa soporta la implementación de funciones variádicas.

Nótese, además, que se puede iterar sobre la tupla sin necesidad de conocer la cantidad de argumentos, por lo que el parámetro *cant* que se tuvo que proveer en C++ no es necesario en Python. Si bien en C++ también existen los bucles basados en rango, que implican iterar sobre justamente un elemento de tipo rango (listas, arreglos, vectores, entre otros) sin conocer su longitud, esta característica no está soportada por el tipo de dato *va_list*, ya que no es una lista real, sino que apunta a cada valor de la lista de argumentos a medida que se va llamando a *va_arg*. Esto no sólo quita mucha versatilidad, sino que además es una gran fuente de errores, porque se está accediendo a memoria confiando en que el usuario haya ingresado el número correcto referenciando a la cantidad de parámetros. No sería un problema leer de menos, causando simplemente errores lógicos en el código; pero sí podría ser un potencial problema leer demás, ya que se está accediendo a memoria con contenido desconocido.

Así, Python tiene superioridad en cuanto a las funciones variádicas, teniéndolas como característica nativa y extensamente desarrollada, tanto en funcionalidad como en simplicidad de escritura y evasión de errores.

IV. CONCLUSIÓN

A lo largo de este trabajo se han analizado distintas aplicaciones del paradigma de programación funcional en Python y C++.

La programación funcional es una filosofía de la programación que se basa en un modelo declarativo, diciéndole al procesador qué partes están en juego en un problema. Así, en lugar de darle una serie de pasos a seguir para resolver dicho problema, simplemente se lo define, bastando esto para que la computadora sepa cómo resolverlo.

Este paradigma, abreviado como FP por sus siglas en inglés, tiene características importantes como la inmutabilidad de las variables y la puridad de las funciones, conceptos que son clave para entender por qué este estilo de programación es exitoso. Respetando una serie de reglas, se logra estructurar al programa para que resuelva los problemas en una cantidad mucho menor de código, y con declaraciones más expresivas.

Dentro de C++ y Python, se analizaron aspectos como las funciones lambda, que son funciones anónimas de corta duración; la comprensión, una forma compacta de declarar una lista como transformación de otra; los functores, objetos con el operador ( ) sobrecargado; las funciones de orden superior, que reciben y/o devuelven otras funciones; la vinculación de argumentos, que son nuevas funciones que surgen de fijar ciertos argumentos de otras funciones; y las funciones variádicas, que son funciones que pueden recibir una cantidad variable de argumentos.

En lo que respecta a las funciones lambda, si bien la sintaxis es mucho más amable en Python, ya que en C++ se deben especificar múltiples símbolos como paréntesis, corchetes y llaves, es justamente esa especificación de "cosas" la que hizo que C++ lograra la superioridad. Por ejemplo, al poder definir explícitamente el modo de captura de las variables del entorno, el programador consigue un control mucho más exhaustivo sobre lo que está haciendo el programa, y por lo tanto logra una libertad mayor. En Python, por el contrario, al no poder especificar tipo de captura, el programador se ve obligado a capturar las variables del entorno por referencia, sometiéndose

a posibles errores y, punto no menor, yendo directamente en contra del concepto de puridad de funciones e inmutabilidad de variables de FP.

En cuanto a comprensión, la "victoria" la consiguió Python, dado que en C++ la comprensión es una funcionalidad para la cual hay poca documentación, principalmente en librerías de uso no común, y que todavía están en vías de desarrollo. En Python, por el contrario, la comprensión es una herramienta muy poderosa, nativa y de uso muy regular, que adhiere perfectamente a los conceptos planteados por FP.

Cuando se habló de los functores, se vio que en un principio ambas implementaciones parecían tener características similares en ambos lenguajes. Sin embargo, al inspeccionar un poco más de cerca, se vio que la superioridad la tenía C++, gracias a su desarrollado y útil manejo de clases y objetos. Al tener un control fuerte y definitivo sobre los permisos para acceder a métodos y datos miembro dentro de una clase, C++ logra evitar cierto tipo de errores que en Python son altamente probables, y que pueden ir desde muy leves hasta muy serios (dentro de la seriedad que Python permite que exista, dado que gran parte del manejo de lo "sensible" es invisible para el programador).

En materia de funciones de orden superior, HOF, el lenguaje con mejores características es Python. Para empezar, la sintaxis se vuelve cada vez más oscura en C++ a medida que se intentan realizar programas cada vez más complejos, principalmente con punteros a funciones. La solución a ese problema, el *header* <*functional*>, trae consigo nuevos problemas, como se vio en el análisis hecho en el apartado correspondiente. Estos problemas se basan en la compatibilidad de tipo de dato, y aparecen si no se utiliza esta librería en todo el programa. Combinar punteros a función con funciones del tipo *std::function*, entonces, resulta en potenciales errores de compilación. Si bien no es un problema tan grave, es suficiente para que Python lleve la ventaja.

Al analizar el aspecto de vinculación de argumentos, se vio que C++ tenía una ventaja casi indiscutible. En Python, esta funcionalidad la provee la librería *functools*, pero con un alcance muy acotado y con una libertad de acción casi nula. Si bien se plantearon algunas soluciones para estos problemas, rápidamente se vio que otros nuevos surgían. En cambio, <*functional*> de C++, a través de herramientas como *std::bind* y *std::placeholders*, permite que la vinculación de argumentos sea de gran ayuda para resolver problemas de manera funcional, adhiriendo a la filosofía de FP.

Por último, cuando se habló de funciones variádicas, se vio que éstas eran poco usuales en C++ y tenían una funcionalidad más bien acotada y restrictiva, mientras que en Python son extremadamente usuales. Tanto así, que incluso son consideradas "buena práctica" en este lenguaje. C++ permite su uso a partir del *header* <*stdarg.h*>, mientras que Python las soporta de forma nativa. Por su parte, C++ requiere que se le diga en el llamado a función la cantidad de argumentos que se pasarán, mientras que en Python esto se evita con los bucles basados en rango. Así, las funciones variádicas son una herramienta para la cual el lenguaje elegido debería ser Python, y es de los dos lenguajes el que más se acerca a lo que idealmente plantea el paradigma funcional.

Cabe destacar que, a lo largo del trabajo, se han hecho otros tipos de comentarios no relacionados con los aspectos funcionales, sino con características generales de cada lenguaje. Por ejemplo, se habló de que C++ es mucho más veloz que Python, y requiere mucha menos memoria, mientras que Python tiene una sintaxis mucho más clara, debiendo escribir menos código y dando una versatilidad más grande por su estilo "cuasi-*template*".

¿Cuál de los dos lenguajes se debería elegir para hacer un desarrollo basado en FP? ¿Cuál de los dos lenguajes adhiere más a estos conceptos? La conclusión a la que se llega en este trabajo es que el lenguaje que se debería elegir es C++.

¿Por qué? Principalmente, porque los aspectos en los que Python es superior son aspectos en los que dicho lenguaje logra hacer alguna funcionalidad de manera más sencilla, pero C++ aún así soporta dicha funcionalidad de alguna forma alternativa. Por ejemplo, Python fue el elegido para las comprensiones de listas, pero aun así sería muy sencillo hacer una herramienta en C++ que en cuanto a funcionamiento sea similar, aunque requeriría más código y escritura de diversas funciones extra. Otro ejemplo son las HOF, donde C++ perdió por su baja versatilidad, la cual se puede elevar a partir del uso de *templates* en combinación con <*functional*>. Nuevamente, si bien Python ganó por cuestión de facilidad, esto no implica que C++ no soporte dicha herramienta.

Sin embargo, esto no es válido en sentido inverso. En las funciones lambda, C++ permite controlar al detalle los métodos de captura, mientras que Python no ofrece tal funcionalidad. Esto atenta directamente contra la inmutabilidad, característica casi central del concepto FP. Otro ejemplo es el de los functores, los cuales en C++ tienen protección frente a posibles intentos de manipulación de su contenido, y por lo tanto a la variación de su funcionamiento, mientras que en Python dichas protecciones no existen. Esto dificulta la puesta en práctica del concepto de puridad de funciones que plantea FP. Un último ejemplo es la vinculación de parámetros, que puede resultar muy útil para reusar funciones y moldear código fácilmente. C++ ofrece posibilidades casi ilimitadas sobre lo que se puede hacer con este concepto, mientras que en Python esta implementación es muy pobre y, por lo tanto, muy poco usada.

Así, luego del análisis realizado, se da por concluida la comparación. Nuevamente, se reitera que C++ es el lenguaje que los resultados obtenidos indican que se debería elegir para un mejor desarrollo funcional, ya que posee características que en conjunto son más útiles para poner en práctica el paradigma FP. Habiendo dicho eso, es necesario remarcar que ambos lenguajes, obviamente, tienen sus ventajas y desventajas. Ambos son lenguajes extremadamente versátiles y útiles, siempre y cuando se encuentre la aplicación adecuada y para la cual fue diseñado cada uno.

V. Bibliografía

- Dan Marsden, Joel de Guzmán, John Fletcher, Thomas Heller, "Phoenix 3.2.0", Boost Software, 2015.

- nikhilaggarwal3, Private Methods in Python, GeeksforGeeks, https://www.geeksforgeeks.org/private-methods-in-python/, sin fecha especificada.

- Autor desconocido, Header <functional>, CPlusPlus, http://www.cplusplus.com/reference/functional/, sin fecha especificada.

- Autor desconocido, Higher-order Functions and Operations on Callable Objects, Functools, Documentation of Functional Programming Modules of the Python Standard Library, Python.org, https://docs.python.org/3/library/functools.html, sin fecha especificada.

- Autor desconocido, Functors in C++, GeeksforGeeks, https://www.geeksforgeeks.org/functors-in-cpp/, sin fecha especificada.

- Autor desconocido, Header <cstdarg> (stdarh.h), CPlusPlus, http://www.cplusplus.com/reference/cstdarg/, sin fecha especificada.

- Autor desconocido, Ranges Library, CPP Reference, https://en.cppreference.com/w/cpp/ranges, 2020.

- Graham Hutton, “Frequently Asked Questions for comp.lang.functional”, University of Nottingham, Nottingham, Inglaterra, 2002.

- Ivan Čukić, “Functional Programming in C++”, Manning Publications, 2018.