Análisis Comparativo Entre Distintos Aspectos de Programación Funcional de 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 y las funciones de orden superior. Luego de cada aspecto, se hará una conclusión parcial de los resultados obtenidos en el experimento. Finalmente, se expresará una conclusión 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 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 siga el step 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 loop explícito en el cual se van 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

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

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.

Nota: Hay más puntos que puede tener una función lambda,

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

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 o v_lambdas_2 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 loop. 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 approach 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]

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 };
```

[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.

```
        //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.

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

```
#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 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.

C++

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

```cpp
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.

PYTHON

Equivalentemente, en Python se resolvería de la siguiente manera:

```python
#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.