Enfoque Funcional Robusto con Aspectos Formales en la Enseñanza de Lenguaje C

Víctor Theoktisto
Departamento de Computación y Tecnología de la Información
Universidad Simón Bolívar, Caracas, Venezuela
vtheok@usb.ve

***Resumen*—La mayoría de las carreras de ingeniería tienen en sus pensa de estudios algún curso de Programación/Computación para Ingenieros. La naturaleza de los cursos sufre en calidad y contenido respecto a los cursos similares de Programación/Algoritmia dictados en carreras propias de Computación. Particularmente cuando se usa Lenguaje C como base de la implementación, el enfoque se dedica más a describir las características del lenguaje que en la utilización práctica del mismo para acelerar su utilización en labores prácticas de ingeniería. Se detalla como contribución una serie de encabezados (*headers*) que incorporan un conjunto de técnicas de programación y buenas prácticas surgidas e implementadas en la enseñanza de programación, y que permiten desarrollar experticias con mayor profundidad y producir un código de la mejor calidad posible: (1) un enfoque cuasi funcional desde temprano haciendo énfasis en recursión de cola y funciones de orden superior; (2) una forma limitada de especificaciones formales con generación de excepciones, (3) un énfasis temprano en recursión, específicamente Recursión de Cola; (4) implementación de Tipos de Datos Abstractos usando tipos opacos en C; (5) incorporación de Funciones de Orden Superior (6) manejo de memoria dinámica implícito y transparente usando un "recolector de basura"; y (7) uso de IDE multiplataforma con depurador incluido.**

Palabras clave*—*Programación Funcional, Estrategias de Enseñanza en Computación, Recursión, Funciones de Orden Superior.

I. Introducción

Los cursos de Computación para Ingenieros dictados para carreras de Ingeniería (y también en otras disciplinas como Ciencias) tienen una reputación no merecida de ser la hermana pobre en la enseñanza de Computación/Programación, usualmente a cargo del Departamento, Escuela o Facultad encargados de la enseñanza de cursos Algoritmos, Estructuras y Programación para la respectiva carrera en el área de Computación/Informática. En común también que carreras de Ingeniería ya consolidadas cuenten con su propio cuerpo de profesores enfocados más bien hacia cumplir un requisito de mínimos [1] que a inculcar verdaderas capacidades analíticas y de codificación a los futuros ingenieros [2].

La propuesta es cambiar enfoque sin cambiar el lenguaje, reforzando un paradigma imperativo con adiciones funcionales, incorporando estructuras formales de verificación de pre- y post- condiciones, invariantes, guardias de bucles y otros.

El artículo está organizado como sigue: Se hace un breve repaso de trabajos previos en la sección II, y de la estrategia de enseñanza en la sección III. La descripción técnica de la implementación está descrita en la IV, y las conclusiones finales se encuentran en la sección V al final, junto con dos anexos mostrando ejemplos.

II. Trabajos Previos

El lenguaje de programación C [3] es uno de los lenguajes más usados en la programación de aplicaciones en Ingeniería después de FORTRAN, con una amplia base de código instalada, librerías estándar específicas incorporadas y gran cantidad de libros y cursos disponibles para su aprendizaje, de muy alta calidad. Es el único lenguaje usado en la construcción de los núcleos (*kernels*) de los sistemas de operación más usados (MSWindows, MacOS, BSD, Linux) y sus aplicaciones.

Para su enseñanza, tradicionalmente se ha seguido el enfoque cercano a su especificación de lenguaje imperativo, quizás no la mejor manera de acercarse a su rol de herramienta de programación de algoritmos, para el cual se usan otro tipo de textos y estrategias.

El antecedente principal sobre enseñanza propia de conceptos de computación usando un enfoque constructivista apropiado para ingenieros es postulado por Ben-Ari [4], en el cual detalla el orden en que deben introducirse los elementos algorítmicos y conceptuales usando como herramienta un lenguaje de programación, con una adaptación posterior por Chesñevar *et al* [5] [6].

En general, por construcción el lenguaje C es un lenguaje imperativo y no soporta el paradigma funcional. No ha habido un enfoque que enseñe a programar algoritmos incorporando conceptos de programación funcional en ese lenguaje [7].

Por otro lado, no existe interés por parte de las coordinaciones de carrera respectivas (ingenierías no relacionadas con Ciencia de la Computación) de cambiar a un lenguaje puramente funcional en los cursos introductorios de programación. Lo que se ha planteado entonces como objetivo es incorporar de manera no intrusiva algunas fortalezas del paradigma funcional que sirvan reforzar la enseñanza del lenguaje C sin que se introduzcan conceptos adicionales en los cursos Computación I (Introductorio) [8] y Computación II (Intermedio) [9].

III. Estrategia de Enseñanza de Programación

La mayoría de los métodos de enseñanza de Programación Algorítmica basados en Lenguaje C usan como base un libro de texto que [invariablemente] comienza desarrollando

el paradigma imperativo de la forma tradicional, introduciendo en forma secuencial los siguientes ítems:

A. Concepto de variables como repositorio de valores, así como los tipos básicos, que no están basados en dominios sino en la capacidad del rango de valores a representar. Usualmente está acompañado de entrada y salida usando "funciones" de C y formatos de presentación. Se requieren instrucciones a los estudiantes de "ignorar" los conceptos de funciones, que serán explicados más adelante, al igual que el pase de variables por valor y por referencia (¡apuntadores!) en las mismas.
B. Las estructuras de control: **if/else/switch/case/while/ do while/for**. Se introducen para poder pasar inmediatamente a la implementación de algoritmos numéricos sencillos basados en bifurcación e iteración.
C. Tipos de Datos Estructurados *Homogéneos*, comúnmente llamados *arrays* (vectores, matrices y demás tensores de 1, 2 o más dimensiones), para fijar y complementar el concepto de iteración. Usualmente las cadenas de caracteres (*strings*) y el concepto de biblioteca de funciones se introduce aquí también [10].
D. Tipos de Datos Estructurados *Heterogéneos*, comúnmente llamados *registros* o **structs**, y arrays de registros. Usualmente se introduce con este tópico lectura y escritura en archivos (ficheros).
E. Métodos estructurados, funciones y procedimientos, paso de parámetros por valor y referencia. Quizás aquí se pase a definir el concepto de Tipos Abstractos de Datos (TADs), pero usualmente no da tiempo.
F. Apuntadores, uso y abuso. Manejo de memoria dinámica usando un recolector de basura (*Conservative Garbage Collector*).
G. Utilización de un Ambiente de Desarrollo Integrado (*Integrated Development Environment*) con depurador.

Transversalmente a esto se va tratando la parte algorítmica, con mayor o menor profundidad. Recursión es un concepto que rara vez se añade a los programas en Ingeniería, y justamente aquí radica la diferencia de como se enseña Programación en las carreras de Computación y afines. En esta última se hace mucho énfasis en el chequeo formal de *pre-* y *post-*condiciones, invariantes y desarrollo algorítmico formal, en el que el lenguaje usado pasa a segundo plano, aunque todos los lenguajes modernos (*C++, Java, Python, Ruby, Go, Rust, etc.*) presentan características multiparadigma (*OOP, Funcional, Imperativo*).

No se tocan temas que ya de por sí requieren énfasis particular en la práctica, tales como eficiencia, depuración (*debugging*) y gerencia de memoria dinámica, incluyendo fugas acumulativas y catastróficas causadas por apuntadores huérfanos y crecimiento del espacio no reclamado. Una situación que causa rechazo a la labor de programación por parte de un sector que podría beneficiaría más de la misma.

IV. Técnica de Adiciones Funcionales Propuesta

Para fortalecer y mejorar la experiencia de enseñanza y aprendizaje, se propone el siguiente método híbrido, que pide prestado conceptos y estructuras de otros paradigmas e implementarlos selectivamente usando construcciones propias (o macros) del lenguaje C, tales como:

(a) Un enfoque funcional desde el inicio;
(b) Especificaciones formales con pre-post condiciones e invariantes;
(c) Soluciones algorítmicas basadas en una introducción temprana de recursión;
(d) Desarrollo de Tipos Abstractos de Datos (TADs) basado en la modularidad, genericidad y opacidad de implementaciones bajo una misma API (Application Programming Interface) y uso de bibliotecas de código;
(e) Uso de funciones de orden superior (Higher Order Functions) para abstraer los TADs tradicionales (*Colección, Conjunto, Pila, Cola, Secuencia, Lista, ListaDoble*);
(f) Esconder el manejo dinámico de apuntadores usando un recolector de memoria utilizada (*garbage collector*);
(g) Uso de un IDE con depurador y validación de uso de memoria.

A continuación se describen con detalle estos puntos.

IV-A. Enfoque funcional desde el inicio

La programación funcional como paradigma implementa los siguientes conceptos:

Funciones puras: Una función pura solo mira a los argumentos que recibe, y devuelve un valor basado en el cálculo asociado a esos paramétros. No tiene efectos secundarios, como modificar los argumentos de entrada (paso por valor) o alterar variables fuera del alcance de la función. También se denomina mantener estados inmutables dentro de la función.

Funciones de primer orden: Las funciones son tipos básicos de datos con sus operadores funcionales. También implica la existencia de funciones anónimas y con functores asignables como cualquier variable:

$f = (lambda(x) : x^2 - x + 1) \Longrightarrow f(2)$ evalúa a 3.

Funciones de orden superior: Las funciones pueden ser pasadas como parámetros a otras funciones, permitiendo un grado de abstracción mayor.

Recursión: La técnica principal de programación es encontrar algoritmos recursivos que permitan usar la estrategia de dividir y conquistar. Implica la composición de funciones, el encestamiento de las mismas, y poder llamarse a sí misma.

Evaluación no estricta: (*Lazy Evaluation*) La evaluación de funciones y parámetros se retarda hasta cuando sea necesario usar un valor. Esto permite definiciones por comprensión e indefinidas que dependan de valores no calculados previamente.

Ahora bien, Lenguaje C fue diseñado como un lenguaje de programación de paradigma imperativo, y ciertamente no es un lenguaje funcional. De hecho, son dos lenguajes: Lenguaje C propio y el lenguaje *preprocesador de macros* CPP. Ambos lenguajes son "Turing completos", y con la combinación adecuada de las construcciones propias de los anteriores y sin complicarlo mucho se puede elevar a algo más sofisticado, sin cambiar el "sabor" del lenguaje.

Para la propuesta, el curso se inicia a partir del concepto matemático de función, que se se viene acarreando desde los cursos básicos de Cálculo y se traslada al lenguaje, componiendo funciones sobre otras funciones.

Sea una función genérica f tal que

$$f : \mathbb{D}_1 \times \mathbb{D}_2 \times \cdots \times \mathbb{D}_n \longrightarrow \mathbb{D}_0$$

con argumentos $f(p_1,\ p_2,\ \ldots,\ p_n),\quad p_k \in \mathbb{D}_k$, para los dominios $\mathbb{D}_k$, $k = 1 \ldots n$, con $\mathbb{D}_0$ el rango de f.

Como ejemplo, se denotan las siguientes funciones reales

$$\begin{aligned} suma: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ mult: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ calc: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \end{aligned}$$

La especificación algebraica de las funciones de arriba entonces se representará en lenguaje C tal como se muestra en el Listado 1 (*func.c*) siguiente:

Listado 1. func.c

```
Dom_0 func (Dom_1 p1, Dom_2 p2, ..., Dom_n pn) {
    Dom_0 resultado; /* Valor a calcular */
    resultado = /* código que usa p1, ... , pn */
    return resultado;
}

double suma (double x, double y) { return x + y; }

double mult (double w, double z) { return w * z; }

double calc (double a, double b ) {
    return suma(mult(a,b),suma(4,-3));
}
```

Las funciones en C pueden ser asignadas a variables y pasadas como argumentos a otras funciones mediante apuntadores a su dirección en memoria, una manera sencilla de eficiente de implementar una variante de pase de parámetros *por nombre*.

IV-B. Especificaciones Formales con pre- y post- condiciones, invariantes y cotas

Se usa el *assert()* existente en C para definir invariantes [**_inv**(*predicado*)], pre-condiciones [**_pre**(*predicado*)], post-condiciones [**_post**(*predicado*)] y cotas de índices de bucles [**_bound**(*índice*)] con limitaciones en el uso de cuantificadores, en un encabezado (*header*) de C llamado "hoare.h" (ver Apéndice A), disponible siguiendo la referencia [11].

El incumplimiento del predicado en cualquiera de los anteriores dispara la excepción correspondiente. La funcionalidad es limitada puesto que no se incorpora uso de cuantificadores para verificar correctitud. El Listado 2 (*hoare.c*) ilustra un corto ejemplo de uso:

IV-C. Énfasis Temprano en Recursión

Se hace énfasis en soluciones recursivas, especialmente *recursión de cola* como paso siguiente en el desarrollo de funciones, para aprovechar las optimizaciones del compilador.

Listado 2. hoare.c

```
#ifdef HOARE_H
    _pre(x != 0.0);           /* precondición  */
    _inv(suma == (k*k+k)/2); /* invariante    */
    _bound(j);                /* cota (j>=0)   */
    _post( factor >= N );     /* postcondición */
#endif
```

Listado 3. elevar.c

```
double elv_rc(double prod, double x, int n) {
    if (!n   )          /* Es n ==0 ?  */
        return prod; /* Fin recursión  */
    if (n & 1)          /* Es n impar ?  */
        return elv_rc(x*prod, x*x, n >> 1);
    return elv_rc(prod, x*x, n >> 1);
}  /* función auxiliar */

double elevar(double x, int n) { /* Interfaz  */
    return elv_rc(1,x,n); /* recursión de cola */
}
```

El Listado 3 (*elevar.c*) indica como elevar un número real x a una potencia n en forma de *recursión de cola*.

El compilador usado, *gcc*, bajo el estándar actual permite optimizaciones importantes del código usando las opciones de compilación `-O2` ó `-O3`. Las llamadas a función con formas *recursivas de cola* son transformadas en bucles *inline*, es decir, parámetros que son "empujados" en el *stack* se convierten al bajo nivel del lenguaje ensamblador en asignación de variables, y la llamada *call* recursiva en un *jump* condicionado hacia atrás, combinado así la manera elegante de programar con la mayor rapidez y eficiencia en memoria de un bucle iterativo equivalente. Todo transparente para el programador.

IV-D. Implementados Tipos Abstractos de Datos (TADs) en forma de Tipos Opacos

Sin pretender extenderse en el concepto de TADs, se usa el mecanismo de headers (.h) y código (.c) para separar la interfaz de la implementación usando apuntadores a tipo cuya definición se encuentra en otro sitio, permitiendo ocultar datos y forzar la operación de cada TAD sólo usando las funciones de acceso y mutabilidad, sin poder conocer o manipular su estructura interior. Para definir la interfaz del TAD Racional, esto iría en el Listado 4 (*racional.h*).

En este caso, se define el TAD Racional como un apuntador a un registro (*racional_tcd*) cuyos detalles internos no son especificados. Al volver a definir *racional_tcd* en la implementación del TAD Racional en el Listado 5 (*racional.c*), se crea una definición opaca a un Tipo de Datos Concreto (TDC), creado como una estructura dinámica por el constructor. Su contenido queda oculto y sólo es accesible por medio de los selectores, mutadores y operadores expuestos en la interfaz.

Si cambiáramos la implementación de *racional_tcd* a **int** valores[2], al igual que en los métodos definidos en *racional.c*, la interfaz *racional.h* no cambiaría.

Listado 4. racional.h

```
    /* def apuntador */
typedef struct racional_tcd* Racional;
    /* constructor   */
Racional consR(int num, int den);
    /* mutador       */
Racional simplifyR(int num, int den);
    /* mutador       */
int setNumerR(Racional q, int num);
    /* mutador       */
int setDenomR(Racional q, int den);
    /* selector      */
int getNumerR(Racional q);
    /* selector      */
int getDenomR(Racional q);
    /* operador      */
Racional sumaR(Racional x, Racional y);
    /* operador      */
Racional multR(Racional x, Racional y);
```

Listado 5. racional.c

```
#include "hoare.h"
#include "Racional.h"

/* Racional Tipo de Dato Concreto */
typedef struct racional_tcd {
    int num, den; // numerador y denominador
} Racional;

Racional consR (int num, int den) {
    Racional q = (Racional) GC_MALLOC(sizeof (
        racional_tcd));
    _pre(den != 0);  /* precond: denom no cero*/
    if (den < 0) {
        q->num = -num;
        q->den = -den;
    } else {
        q->num =  num;
        q->den =  den;
    }
    return q;
}

int setNum (Racional q, int num) {
    _pre(q); /* precond: Existe q */
    return (q->num = num);
}

Racional sumaR (Racional x, Racional y) {
    _pre(x != NULL and y != NULL); /* precond: x,y
        existen */
    return consR(x->num*y->den +  x->den*y->num,
                 x->den*y->den);
}
```

IV-E. Funciones de Orden Superior

Las funciones de orden superior (*Higher Order Functions*, HOFs) se usan en programación funcional pura para manipular y crear funciones que operan en forma genérica sobre funciones y tipos. Las más conocidas son:

map: Automorfismo, una función $f : E \to F$ se aplica en forma no destructiva a todo elemento de una secuencia S con elementos de E, produciendo una nueva secuencia de elementos del dominio F en el mismo orden).

$$map : (E \to F) \times Seq{<}E{>} \longrightarrow Seq{<}E{>}$$

filter: Automorfismo, un predicado $p : E \to \mathbf{bool}$ unario se usa para podar en forma no destructiva los elementos E de una secuencia S que cumplen el predicado p, obteniendo una nueva secuencia respetando el orden relativo.

$$filter : (E \to bool) \times Seq{<}E{>} \longrightarrow Seq{<}E{>}$$

reduce o fold-right: Acumulador, donde los elementos de una secuencia S como monoide o grupo son combinados en un valor resultante $r \in F$ al aplicar un operador binario de composición asociativo $Op : E \times F \to F$ con al menos un elemento identidad $\epsilon_0 \in F$ a toda la secuencia en el mismo orden relativo. También se implementa la variante *fold-left*.

$$reduce : (F \times E \to F) \times F \times Seq{<}E{>} \longrightarrow F$$

compose: Composición algebraica de funciones, en teoría funcional el operador de composición de funciones algebraicas ($\circ$) funciona sólo sobre tipos compatibles encadenables.

$$h(x) = (f \circ g)(x) \;=\; f(g(x))$$
$$g : \mathbb{D}_\alpha \to \mathbb{D}_\beta,\; f : \mathbb{D}_\beta \to \mathbb{D}_\gamma \Longrightarrow\; h : \mathbb{D}_\alpha \to \quad \mathbb{D}_\gamma$$

No se puede hacer una función genérica de *compose* en C sin hacer uso engorroso de macros al estilo de los *templates* de C++. Sin embargo, si limitamos las funciones al dominio de los enteros y reales ($t = i|f|d$) con 1 y 2 argumentos (extensible a 3 o más sin pérdida de generalidad), podemos definir los siguientes tipos genéricos de [apuntadores a] funciones:

$$\begin{aligned} t\_componer\_f\_gx(f,g)(x) &\equiv f(g(x)) \\ t\_componer\_f\_gxy(f,g)(x,y) &\equiv f(g(x,y)) \\ t\_componer\_f\_gx\_hx(f,g,h)(x) &\equiv f(g(x),h(x)) \\ t\_componer\_f\_gx\_hy(f,g,h)(x,y) &\equiv f(g(x),h(y)) \end{aligned}$$

También se pueden hacer HOFs específicas a un tipo particular, como mostramos para el TAD Lista en el Listado 6 (*lista.h*).

Listado 6. lista.h

```
    /* En Lista.h */
typedef struct ListaTDC_t* Lista_t;
typedef int Valor_t, Elem_t;
    /* Devuelve una Lista s vacia  */
Lista_t consEmptyEL ();
    /* Aumenta una Lista s con e al inicio  */
Lista_t consEL (Elem_t e, Lista_t s);
    /* primer Elem e de la Lista s */
Elem_t firstEL (Lista_t s);
    /* resto de la Lista s */
Lista_t restEL (Lista_t s);
    /* Inserta un Elem e en pos de Lista s */
int insertEL(Lista_t s, Elem_t e, int pos)
    /* Devuelve una copia de la Lista_t */
Lista_t mapEL (Elem_t (*fun) (Elem_t), Lista_t L);
    /* Devuelve una copia de la Lista_t */
Lista_t filterEL (bool (*pred) (Elem_t), Lista_t L);
    /* Devuelve plegado de la operación */
Valor_t reduceVEL (Valor_t (*opr)(Elem_t, Valor_t),
    Valor_t eps0, Lista_t L);
```

En el código anterior, las funciones de orden superior se usan para crear nuevas listas sin destruir las anteriores como se muestra en el Listado 7 (*lista.c*).

Listado 7. lista.c

```
typedef struct ListaTDC_t {
    Elem_t  info;
    Lista_t next;
} ListaTDC_t;

    /* Devuelve una Lista_t con un Elem_t aislado */
Lista_t consEL(Elem_t e, Lista_t S) {
    Lista_t  node = GC_MALLOC(sizeof ListaTDC_t);
    node->info = e;
    node->next = S;
    return nodo;
}
    /* Devuelve nueva Lista_t, recursivo */
Lista_t mapEL(Elem_t (*func)(Elem_t), Lista_t L) {
    if (L) return consEL((*func)(firstEL(L)),
                         mapEL(func, restoEL(L)));
    return NULL;
}
    /* Devuelve nueva Lista_t, recursivo */
Lista_t filterEL(bool (*pred)(Elem_t), Lista_t L) {
    if (!L) return NULL;
    if ( (*pred)(firstEL(L)) )
        return consEL(firstEL(L),
                      filterEL(pred, restEL(L)));
    return filterEL(pred, restEL(L));
}
    /* Devuelve plegado de la operación */
Valor_t reduceVEL(Valor_t (*opr)(Elem_t, Valor_t),
                  Valor_t eps0,
                  Lista_t L) {
    if (!L) return ident;
    return reduceVEL(opr,
                     (*opr)(eps0, firstEL(L)),
                     restEL(L) );
}

bool esImpar(Elem x) { return ODD(x); }
Elem_t cuad(Elem x)  { return SQR(x); }
Elem_t suma(Elem_t res, Valor_t val) {
    return res + val;
}

int main() {
    Lista_t a = consEmptyEL();
    Lista_t b, c;
    Valor_t val1, val2;
    a = consEL(1,consEL(2,consEL(3,consEL(4, a))));
        /* a es  {1 --> 2 --> 3 --> 4} */
    b = mapEL(sqr, a);
        /* b es  {1 --> 4 --> 9 --> 16} */
    c = filterEL(esImpar, b);
        /* c es  {1 --> 9} */
    val1 = reduceVEL(suma, 0, c);
        /* Suma es {1 --> 9}, val1 = 10  */
    val2 = reduceVEL(suma,
                     0,
                     filterEL(esImpar,
                              mapEL(cuad, a)));
    val3 = mapfilterreduce(cuad, esImpar, suma, a);
        /* suma cuadrados impares val2 = 10 */
}
```

Listado 8. lambda.c

```
#include <stdio.h>

  /* Expresión lambda que retorna una función */
#define lambda(FUNTYPE, PARAMS, ...) \
  ({FUNTYPE lambda PARAMS { __VA_ARGS__;} lambda;})

    // Podemos definir un tipo para la función y declararla
typedef int (*IFPTR_t)(int);
int (*i_componer_f_gx) (IFPTR_t, IFPTR_t, int); // Componer funciones enteras
    // O también hacer la declaración directamente de componer funciones reales
float (*f_componer_f_gx) (float (*)(float), float (*)(float), float);

int main()
{
    int (*r)(int)       = lambda( int, (int x), return x+1 );
    int (*s)(int, int) = lambda( int, (int x, int y), return x/y );
    printf( "result  = %i\n", s( r(7), r(3) ) );

        // Definimos la composición de funciones (f.g)(x) = f(g(x)) [enteras]
    i_componer_f_gx = lambda( int,
                              ( IFPTR_t f, IFPTR_t g, int x ),
                              return f(g(x)) );
    printf( "f(g(x)) = %i (enteros)\n",
            i_componer_f_gx( lambda( int, (int x), return x+x ),
                             lambda( int, (int x), return x*x ),
                             6) );
        // Definimos la composición de funciones (f.g)(x) = f(g(x)) [reales]
    f_componer_f_gx = lambda( float,
                              ( float (*f)(float), float (*g)(float), float x ),
                              return f(g(x)) );
    printf( "f(g(x)) = %f (reales)\n",
            f_componer_f_gx( lambda( float, (float x), return x*x ),
                             lambda( float, (float x), return x+x ),
                             4.0f) );

    return lambda( int, (int x), return x )(0);  // La función Identidad
}
```

IV-E1. Funciones anónimas (lambda): En programación funcional las funciones anónimas son parte de los bloques de construcción. Dado que el preprocesador de C permite trabajar con listas de argumentos de longitud indeterminada, se creó un macro muy útil que permite definir y usar funciones lambda en el código, como muestra el Listado 8.

El esquema permite definir en forma natural la composición algebraica usando funciones lambda anónimas: $i\_compose\_fg\_x(f,g,x) = f(g(x))$, con $f,g,x \in \mathbb{Z}$.

Como técnica sofisticada se incorpora de manera opcional en el curso, por tres razones: (i) No hay necesidad planteada para enseñar el uso de funciones lambda en el programa; (ii) Es una característica fuera del estándar del lenguaje, a pesar de que el compilador *gcc* permite anidar funciones (*nesting*), necesario para su ejecución; y (iii) una definición incorrecta de funciones puede dar lugar a errores difíciles de trazar.

IV-F. Apuntadores y uso del Boehm Garbage Collector

Una situación recurrente cuando se está trabajando con memoria dinámica es el problema de apuntadores a memoria dinámica que ya no referencian estructuras válidas, con el consecuente abandono de espacio que se vuelve inalcanzable.

La solución escogida fue usar un "recolector de basura" (*Garbage Collector*), particularmente el *GC* de Boehm [12]. Lo que hace *GC* es sustituir las llamadas de C que crean y manipulan memoria dinámica (*malloc(), calloc(), realloc()*) por unas más eficientes (*GC_MALLOC_ATOMIC(), GC_MALLOC(), GC_REALLOC()*) que crean y usan "apuntadores inteligentes" (*smart pointers*). No hay necesidad de usar la función *free()*, pero se puede llamar si se desea como *GC_FREE()* para provocar un vaciado, porque ahora toda la memoria dinámica es manejada de manera inteligente y bajo demanda, tal como se hace en los lenguajes funcionales más usados (Python, Scheme, Haskell, ML), y en Java.

Junto con el rastreador de memoria a nivel de bytes Valgrind de Nethercote y Seward [13] se encargan de registrar y diagnosticar el estado de la memoria de ejecución de un programa para evitar problemas graves de deterioro y desempeño.

Esto permite crear aplicaciones relativamente grandes con una manejo implícito y eficiente de memoria dinámica ***sin***

necesidad de que el programador este consciente de ello, y ajustado al último estándar del Lenguaje C y las prácticas de seguridad CERT en codificación [14].

IV-G. Uso de un IDE multiplataforma y uso del depurador incorporado

Los IDEs (Integrated Development Environments) multiplataforma como *Code::Blocks* [15] o *Eclipse* [16] son una herramienta imprescindible para un buen seguimiento del proceso de programación interactiva. Tienen depuradores (debuggers) de código fuente en tiempo real que permiten ejecutar el código paso a paso, examinar variables y estados de la memoria dinámica en cada instante. Esto permite una visión integral de la ejecución de código impensable en anteriores enfoques de la enseñanza de Computación para Ingenieros, donde el punto de corrección de errores siempre se dejaba para lo último o simplemente se ignoraba.

V. Conclusiones

Un cuestionamiento que se puede hacer a esta implementación es porque tomarse tanto trabajo con C, cuando sería más efectivo trabajar directamente con un lenguaje de paradigma funcional. La respuesta es que la decisión de escoger la herramienta de programación adecuada para el curso que deben tomar los futuros ingenieros requiere poner de acuerdo a departamentos diferentes, que no necesariamente están ganados a explorar otros enfoques o lenguajes.

Segmentar la población estudiantil para enseñar en paralelo varios lenguajes de programación no hace un uso eficiente de recursos, ya que cada disciplina de ingeniería requerirá el lenguaje más afín a su práctica, algunos de los cuales son bastante más limitados.

Aplicar este enfoque en la enseñanza del curso de Computación para Ingenieros, dividido en un curso introductorio y otro avanzado, tiene consecuencias cualitativas y cuantitativas.

Entre las cualitativas, ya van dictados dos períodos consecutivos con este enfoque, y si bien está claro que no se ha generado suficiente data en el tiempo como para hacer una afirmación soportable con estadísticas sólidas, se ha logrado subir sustancialmente la calidad y profundidad de los tópicos tratados. Los problemas y proyectos planteados como parte de ejercicios de laboratorio han incrementado en complejidad, destacando una mayor comprensión del proceso algorítmico funcional, mucho más apropiado para las disciplinas de ingeniería que ya funcionan en base a la desagregación de componentes y de su encaje funcional.

Esto ha conllevado a una mayor calidad del código generado, mejor legibilidad y crecimiento incremental de la complejidad del mismo. Permite trasladar directamente competencias adquiridas en la enseñanza de Matemáticas en Ingeniería, particularmente Cálculo Diferencial e Integral, al poderse expresar estos conceptos funcionalmente de manera directa usando las construcciones ya mencionadas.

Entre las cuantitativas, tanto el curso Introductorio como el Avanzado en los que se ha seguido esta metodología han logrado cubrir tópicos de TADs que quedaban inicialmente fuera del programa establecido, tales como especificaciones formales, recursión, entrada y salida de archivos de acceso directo, manipulación de *bitfields*, algoritmos típicos de ordenamiento y estructuras recursivas de almacenamiento.

Mirando hacia el futuro, usar este enfoque facilita plantearse la interrogante de que lenguaje actual deberían estar aplicando los ingenieros no informáticos en sus disciplinas. Lenguajes recientes como Julia, Python, Rust y Erlang permiten gran eficiencia como lenguajes de paradigmas híbridos, y sería necesario experimentar e investigar si son más apropiados para ser usados como lenguaje de programación base en la enseñanza de las ingenierías y ciencias no computacionales.

En el Apéndice A se anexa el *header* "hoare.h" usado para facilitar la correctitud formal de programas. En el Apéndice B se anexa un ejemplo sencillo de su uso en el algoritmo común que obtiene las raíces de un polinomio de segundo grado.

Referencias

[1] I. Milne and G. Rowe, "Difficulties in Learning and Teaching Programming – Views of Students and Tutors," *Education and Information Technologies*, vol. 7, no. 1, pp. 55–66, 2002.

[2] A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Upper Saddle River (N.J.): Prentice Hall, 1988.

[4] M. Ben-Ari, "Constructivism in Computer Science Education ," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.

[5] C. I. Chesñevar, M. P. González, A. G. Maguitman, and L. Cobo, "Teaching Fundamentals of Computing Theory: a Constructivist Approach," *Journal of Computer Science & Technology*, vol. 4, no. 2, pp. 91–97, 2004.

[6] P. H. Hartel and H. L. Muller, *Functional C*. Harlow, UK: Addison Wesley Longman, 1997. [Online]. Disponible: http://eprints.eemcs.utwente.nl/1077/

[7] S. Peyton-Jones and D. Lester, *Implementing functional languages: a tutorial*. Prentice Hall, 1992.

[8] V. Theoktisto. (2016) *CI-2125 Computación I. Curso Introductorio de Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Disponible: http://ldc.usb.ve/~vtheok/cursos/ci2125/aj14

[9] V. Theoktisto. (2016) *CI-2126 Computación II. Curso Intermedio de Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Disponible: http://ldc.usb.ve/~vtheok/cursos/ci2126/sd14

[10] P. J. Plauger, *The Standard C Library*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1991.

[11] V. Theoktisto. (2016) *Encabezado (header) para especificaciones formales en C "hoare.h". Curso Intermedio de Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Disponible: http://ldc.usb.ve/~vtheok/cursos/ci2126/aj14/hoare.h

[12] H.-J. Boehm, "Bounding Space Usage of Conservative Garbage Collectors," *SIGPLAN Notices*, vol. 37, no. 1, pp. 93–100, Jan. 2002. [Online]. Disponible: http://doi.acm.org/10.1145/565816.503282

[13] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 65–74.

[14] S. E. Institute, *SEI CERT C Coding Standard Rules for Developing Safe, Reliable, and Secure Systems*. Carnegie Mellon University, 2016.

[15] Code::Blocks. (2016) *Code::Blocks: the free Open Source (GPLv3) cross-platform C, C++ and Fortran IDE*. The CodeBlocks Team. [Online]. Disponible: http://www.codeblocks.org

[16] Eclipse::Neon. (2016) *The Eclipse Top-Level Project - an open source, robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools and rich client application*. The Eclipse Foundation. [Online]. Disponible: http://www.eclipse.org/ide/

APÉNDICE A
EL ENCABEZADO (HEADER) "HOARE.H"

Listado 9. hoare.h

```
/* Para ser usado en Computación I y II.
** Version 1.06 08/11/2014, Elaborado por: Xxx Yyyyyyy
/* -- specific C macro (needed for preprocessing!) */
#ifndef _HOARE_H_INCLUDED_
#define _HOARE_H_INCLUDED_

#include <stdlib.h>
#include <stdio.h>

/* --- Assert during compiling (not run-time) ---
 * CompilerAssert(exp) is designed to provide error checking at
 * compile-time for assumptions made by the programmer at design-time
 * and yet does not produce any run-time code.
 * Example: if (CompilerAssert(sizeof(int)==4)) ... */
#define CompilerAssert(Predicate) extern char _CompilerAssert[(Predicate)?1:-1]

    /* Useful definitions */
enum bool {FALSE=0, TRUE=~0};
#define and &&
#define  or ||
#define not !
#define xor ^
#define AND &&
#define  OR ||
#define NOT !
#define XOR ^
#define GLUE(a,b) a##b
#define XPREFIX(s) s
#define PREFIX(a,b) XPREFIX(a)b
#define ABS(a) ({__auto_type __a = a; __a < 0 ? -__a : __a;})
// Compiler warns when the types of x and y are not compatible
#define MAX(x, y) ({ \
    __auto_type __x = (x); __auto_type __y = (y);  \
    (void) (&__x == &__y); __x > __y ? __x : __y; })
#define MIN(x, y) ({ \
    __auto_type __x = (x); __auto_type __y = (y);  \
    (void) (&__x == &__y); __x < __y ? __x : __y; })
#define ODD(n) ((n)&1)
#define EVEN(n) (!((n)&1))

    /*--- SWAP failsafe for any size ---*/
#define SWAP(A,yvar) \
    do { \
        unsigned char temp[sizeof(A) == sizeof(B) ? (signed)sizeof(A) : -1]; \
        memcpy(temp,   &B, sizeof(A)); \
        memcpy(  &B,   &A, sizeof(A)); \
        memcpy(  &A, temp, sizeof(A)); \
    } while(0)

    /*--- Is a number a power of two ---*/
#define ISPOWEROF2(x) (!((x)&((x)-1)))
    /* _NUMCELLS() macro */
#define NUMCELLS(_arraytype) (sizeof(_arraytype)/sizeof(*_arraytype))
    /* Two's complement negation as a macro */
#define ONESCOMPLEMENT(x) ((x)^(~0)))
    /* Two's complement negation as a macro */
#define TWOSCOMPLEMENT(x) (((x)^(~0))+1)
    /* SQUARE() macro, final form */
#define SQUARE(x) ((x)*(x))
    /* fills an integer value with ONES independent of type size */
#define ALLONES ~0

    /* Distinguishing between ascii chars and wchar chars:  */
#if UNICODE
    #define dchar wchar_t
    #define TEXT(s) L##s
#else
    #define dchar char
    #define TEXT(s) s
#endif

    /* Defining macros for memory allocation */
    /* redefine malloc(sizeof(Storage)) as CREATE(Storage) using GC_MALLOC */
    /* free is optional */
#define CREATE(__Storage) (GC_MALLOC(sizeof(__Storage)))
#define NEW(__PointerVar) (__PointerVar=GC_MALLOC(sizeof(*__PointerVar)))
#define NEWARRAY(__Dim,__DataType) (GC_MALLOC(__Dim*sizeof(__DataType)))
#define DELETE(__PointerVar) (NULL)
#define DESTROY(__ReservedStoragePtr) (NULL)

    /* General assertion (Predicate) */
#define _assert(Predicate) \
     check_assert("Assertion does not hold for",Predicate)
    /* Precondition (Predicate) */
#define _pre(Predicate) \
    check_assert("Precondition does not hold for",Predicate)
    /* Postcondition (Predicate) */
#define _post(Predicate) \
    check_assert("Postcondition does not hold for",Predicate)
    /* Invariant (Predicate) */
#define _inv(Predicate) \
    check_assert("Invariant does not hold for",Predicate)
    /* Precondition (integer expression) */
#define _bound(IntExpression) \
    check_assert("Bound does not hold for",(IntExpression)>=0)

#ifdef NDEBUG
    #define check_assert(Message,Predicate) ((void)0)
#else /* Not NDEBUG.  */
    #define check_assert(Message,Predicate)  \
    ((void)((Predicate)?0:__check_assert(Message,#Predicate,__FILE__,__LINE__)))
    #define __check_assert(Message,Predicate,File,Line) \
    ((void)printf (">>> At %s:%u:  %s ' %s'\n<<< Assertion failed. "
    "Execution will stop now.\n",File,Line,Message,Predicate),exit(1),0)

#endif  /* NDEBUG */
#endif /* ifndef _HOARE_H_INCLUDED_ */
```

APÉNDICE B
EJEMPLO DE PROGRAMA USANDO "HOARE.H"

Listado 10. raizpoly2.c

```
/*
 * DESCRIPCION: El siguiente programa calcula
 * las raices de un polinomio de segundo grado
 * p(x) = Ax^2 + Bx + C = 0                    */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hoare.h"
const double epsilon = 0.000001

int main()  {
    // Es una buena práctica inicializar todas las
       variables a algún valor predefinido,
    // aunque sean sustituidas más adelante por
       otros
  double a = 1.0, b = 0.0, c = 1.0, d = 0.0, r =
      0.0;
    // Entradas
  double raiz1 = 0.0,  raiz2 = 0.0; // raíces reales
  double preal = 0.0,  pimag = 0.0; // raíces
      complejas
  int esComplejo = 0; // boolean, si es complejo

  printf("\nObtener raíces de polinomios de 2do
      grado p(x) = Ax^2 + Bx + C = 0");
  printf("\nIntroduzca el valor de 'A' (real): ");
  scanf ("%lf", &a);
  printf("\nIntroduzca el valor de 'B' (real): ");
  scanf ("%lf", &b);
  printf("\nIntroduzca el valor de 'C' (real): ");
  scanf ("%lf", &c);

  printf("\nSe leyó A = %f, B = %f, C = %f\n\n", a,
      b, c);

  _pre( a != 0.0);    //precondición modificada
  d = b*b - 4*a*c;    // discriminante
  esComplejo = (d < 0.0);  // Si d es negativo, raí
      ces complejas

  printf("\nDiscriminante D = %lf;  es_Complejo ? =
      %d\n\n", d, esComplejo);

  r = sqrt(ABS(d)); // sacar raíz cuadrada del valor
       absoluto del discriminante

  if (esComplejo) {
     preal = -b/(2*a);
     pimag =  r/(2*a);
     printf("\nRaíces complejas %lf +%lfi  y  %lf %
         lfi\n\n ", preal, pimag, preal, -pimag);
  } else {
     raiz1 = (-b + r)/(2*a);
     raiz2 = (-b - r)/(2*a);
     printf("\nRaíces reales %lf y  %lf\n\n", raiz1,
          raiz2);
  }

  _post( ((ABS(raiz1*raiz2 - c/a) < epsilon) and
          (ABS(raiz1+raiz2 + b/a) < epsilon))
     or  ((ABS(preal*preal+pimag*pimag - c/a <
         epsilon)) and
          (ABS(preal+preal + b/a) < epsilon)) );
                  //precondición modificada

  return 0;
} /* end main */
```