La Programación Funcional: Un Poderoso Paradigma

Gustavo Ramiro Rivadera [1]

gritadera@ucasal.net

Resumen

El presente artículo pretende introducir al lector en el paradigma de la programación funcional, utilizando el lenguaje Haskell, un lenguaje funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcionales.

Palabras Claves: programación funcional, paradigma funcional, abstracción de datos, funciones de orden superior, evaluación perezosa, transparencia referencial, tipos de datos

1. Introducción

Hace ya bastante tiempo, cuando cursaba mis últimos años en la Universidad, formé parte de un proyecto de investigación en arquitecturas de computadoras. En este proyecto analizábamos y clasificábamos todas los posibles tipos de arquitecturas existentes, desde las comunes y ubicuas PC a las lejanas (para nosotros) supercomputadoras paralelas. En ese proyecto, a la par de descubrir que no todas las computadoras eran PC, y que no todas tenían un solo procesador y del mismo tipo, me di cuenta asimismo que los lenguajes de programación que utilizaban tampoco eran similares. Claro que ya había aprendido el omnipresente Pascal, algo de C, C++, y la nueva ola de los lenguajes orientados y basados en objetos, pero lo que me llamó la atención, no fueron las variaciones sobre estos últimos lenguajes,

[1] Ingeniero en Computación, desarrollador independiente de software, docente de las Cátedras de Modelos y Simulación, Análisis Estratégico de Datos y Bases de Datos III, en la Facultad de Ingeniería e Informática, UCASAL. Actualmente cursa la Maestría en Ingeniería del Software en el Instituto Tecnológico de Buenos Aires (ITBA).

sino otros radicalmente distintos, diseñados muchas veces en forma especial para máquinas específicas. Para comenzar, estos lenguajes no ordenaban sus computaciones (instrucciones) en forma secuencial, sino que el orden estaba dado por, en el caso particular de los funcionales, la aplicación de funciones a otras funciones, resultando por tanto, innecesaria la existencia de variables. Finalmente, este proyecto me llevó a investigar sobre los diferentes paradigmas de programación.

Un paradigma de programación es una forma específica de realizar las computaciones. Un lenguaje de programación siempre sigue un paradigma o una mezcla de varios; por ejemplo el paradigma procedimental es seguido por la mayoría de los lenguajes actuales, tales como JAVA, Pascal y C++. También podemos encontrar lenguajes con la influencia de dos paradigmas, por ejemplo el antes mencionado C++, que tiene su origen procedimental y al cual se le ha agregado el paradigma orientado a objetos.

El paradigma del que trata este artículo se denomina funcional, programación funcional o FP.

2. Orígenes

Los orígenes de la programación funcional pueden rastrearse al matemático Alonzo Church, que trabajaba en la Universidad de Princeton, y, al igual que otros matemáticos de allí, estaba interesado en la matemática abstracta, particularmente en el poder computacional de ciertas máquinas abstractas. Las preguntas que se hacía eran por ejemplo: si dispusiésemos de máquinas de un ilimitado poder de cómputo, ¿qué tipos de problemas se podrían solucionar?, o ¿se pueden resolver todos los problemas?

Para contestar este tipo de preguntas, Church desarrolló un lenguaje abstracto, denominado Cálculo Lambda, que el cual sólo realizaba evaluación de expresiones usando funciones como mecanismo de cómputo. Este lenguaje abstracto no tenía en cuenta limitaciones concretas de implementación de ningún tipo.

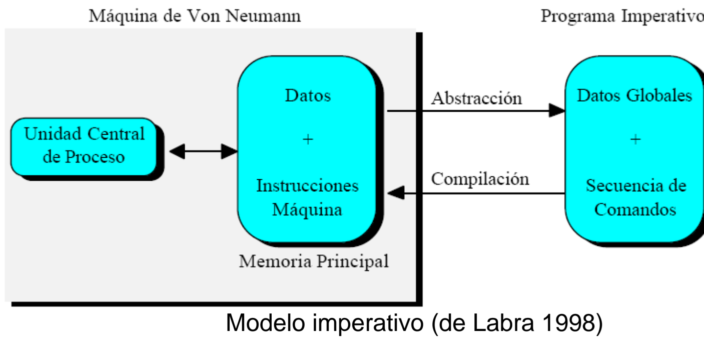
Al mismo tiempo que Church, otro matemático, Alan Turing, desarrolló una máquina abstracta para intentar resolver el mismo tiempo de problemas planteados por Church. Después se demostró que ambos enfoques son equivalentes.

Las primeras computadoras digitales se construyeron siguiendo un esquema de arquitectura denominado de Von Neumann, que es básicamente una implementación de la máquina de Turing a una máquina real. Esta máquina forzó de alguna manera el lenguaje en el cual se escriben sus programas, justamente el paradigma procedimental, el cual, como menciona Backus en un muy famoso artículo que escribió al recibir el premio Turing en 1978 (Backus 1978), tiene tantísimos defectos, que muchos programadores padecemos aun hoy.

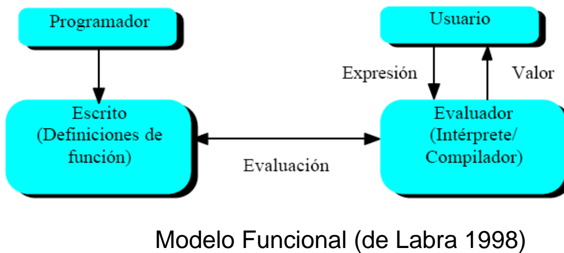
La programación funcional se aparta de esta concepción de máquina, y trata de ajustarse más a la forma de resolver el problema, que a las construcciones del lenguaje necesarias para cumplir con la ejecución en esta máquina. Por ejemplo, un condicionamiento de la máquina de Von-Neumann es la memoria, por lo cual los programas procedimentales poseen variables. Sin embargo en la programación funcional pura, las variables no son necesarias, ya que no se considera a la memoria necesaria, pudiéndose entender un programa como una evaluación continua de funciones sobre funciones. Es decir, la programación funcional posee un estilo de computación que sigue la evaluación de funciones matemáticas y evita los estados intermedios y la modificación de datos durante la misma.

Hoy en día existen diversos lenguajes funcionales. Se podría considerar como uno de los primeros lenguajes funcionales al LISP, que actualmente sigue en uso, sobre todo en áreas de la inteligencia artificial. También un pionero de este paradigma es APL desarrollado en los años 60 (Iverson 1962). El linaje funcional se enriqueció en los años 70, con el aporte de Robin Milner de la Universidad de Edimburgo al crear el lenguaje ML. Éste se subdividió posteriormente en varios dialectos tales como Objective Caml y Standard ML. A fines de los años 80, a partir de un comité, se creó el lenguaje Haskell, en un intento de reunir varias ideas dispersas en los diferentes lenguajes funcionales (un intento de estandarizar el paradigma). Este año Microsoft Research ha incluido un nuevo lenguaje (funcional), denominado F#, a su plataforma .NET.

La Figura 1 compara gráficamente ambos paradigmas.



Modelo imperativo (de Labra 1998)



Modelo Funcional (de Labra 1998)

Figura 1. Comparación entre los modelos imperativo y funcional (Labra 98).

3. Qué es la programación funcional

En este breve artículo, intentaré explicar la utilidad y potencia de la programación funcional, por medio de pequeños ejemplos, para comprender más rápidamente esta filosofía de programación.

Dado el nombre del paradigma, sabemos básicamente que lo central en el mismo es la idea de función, que podríamos decir es análoga a lo que conocemos de funciones de la matemática. Por ejemplo, podemos escribir en el lenguaje funcional Haskell:

```
Factorial :: int -> int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Es decir, la última es una función sencilla, parecida a la que conocemos de las matemáticas de la secundaria, que permite calcular el factorial de un número entero (ver definición de la función factorial más abajo).

Comparemos esa línea de código de Haskell con la siguiente en un lenguaje como C#:

```
unsigned factorial (unsigned n)
{
  int product = 1;     // valor inicial
  while (n > 1)
  {
    product *= n--;    // acumulador
  }
  return product;      // resultado
}
```

Este ejemplo es muy sencillo y los dos fragmentos de código son muy parecidos. Sin embargo, la definición de Haskell es mucho más cercana a la matemática:

```
0!   =    1
n!   =    n * (n - 1)!
```

La pregunta que nos hacemos ahora, es si podemos construir programas complejos usando simplemente funciones. Esta pregunta se puede contestar fácilmente si describimos las características principales de este tipo de lenguajes. A modo de ejemplo de introducción, veamos como se escribiría el programa de ordenamiento QuickSort en Haskell:

```
quicksort []     = []
quicksort (x:xs) = (quicksort [ y | y <- xs, y <= x ])
                   ++ [x] ++
                   (quicksort [ z | z <- xs, z  > x ])
```

En este caso, conseguimos un programa muy sencillo y corto, que basa su potencia en la habilidad de Haskell de manipular listas ([ ]) y especificar funciones recursivas. Quicksort se define recursivamente, a partir de la misma definición matemática del algoritmo. Este algoritmo utiliza una estrategia de divide y conquista (*divide and conquer*), la cual divide a una lista en dos sublistas, la primera con elementos menores o iguales que uno dado, denominado *pivot*, y la segunda con elementos mayores. En la primera línea del cuerpo (línea 2), se define la primera sublista, en la segunda línea el *pivot*, y la tercera línea la tercera sublista. El operador predefinido ++ concatena dos listas. Cada llamada recursiva a QuickSort en el cuerpo especifica que se concatenará el nuevo elemento (y o z) a la lista de argumento siempre y cuando se cumpla con la condición de más a la derecha (ej. y <= x).

Comparemos ahora nuestro sencillo programa funcional con uno en un lenguaje procedimental, tal como C (Wikipedia 2008):

```
//Programa Quicksort en C
void quicksort(int* array, int left, int right)
{
    if(left >= right)
        return;

    int index = partition(array, left, right);
    quicksort(array, left, index - 1);
    quicksort(array, index + 1, right);
}

int partition(int* array, int left, int right)
{
    findMedianOfMedians(array, left, right);
    int pivotIndex = left, pivotValue = array[pivotIndex], index =
left, i;

    swap(&array[pivotIndex], &array[right]);
    for(i = left; i < right; i++)
    {
        if(array[i] < pivotValue)
        {
            swap(&array[i], &array[index]);
            index += 1;
        }
    }
    swap(&array[right], &array[index]);
    return index;
}

int findMedianOfMedians(int* array, int left, int right)
{
    if(left == right)
        return array[left];

    int i, shift = 1;
    while(shift <= (right - left))
    {
        for(i = left; i <= right; i+=shift*5)
        {
            int endIndex = (i + shift*5 - 1 < right) ? i + shift*5
- 1 : right;
            int medianIndex = findMedianIndex(array, i, endIndex,
shift);
            swap(&array[i], &array[medianIndex]);
        }
        shift *= 5;
    }
    return array[left];
```

```
}

int findMedianIndex(int* array, int left, int right, int shift)
{
    int i, groups = (right - left)/shift + 1, k = left +
groups/2*shift;
    for(i = left; i <= k; i+= shift)
    {
        int minIndex = i, minValue = array[minIndex], j;
        for(j = i; j <= right; j+=shift)
            if(array[j] < minValue)
            {
                minIndex = j;
                minValue = array[minIndex];
            }
        swap(&array[i], &array[minIndex]);
    }
    return k;
}

void swap(int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

4. Principales características

Para demostrar las principales características de los lenguajes de programación funcionales modernos, vamos a utilizar el lenguaje Haskell. Este es un lenguaje funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa (*lazy evaluation*), tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones (*pattern matching*), y definiciones de listas por comprensión. Tiene además otras características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos. Aunque seria imposible explicar con detalle cada una de estas características en este artículo, daremos una breve explicación de alguna de ellas. El lector interesado puede recurrir a un excelente libro en castellano (Ruiz 2004) y otros dos en inglés (Thompson 1999; Doets & van Eijck 2004) para profundizar en el tema.

4.1. Tipos de datos, tuplas y listas

Los lenguajes funcionales, en particular Haskell, tienen un rico conjunto de datos atómicos predefinidos, tales como los numéricos *int*, *integer* (de mayor precisión que el anterior), *float*, *double*, etc., y además los tipos *char* y *bool*.

El sistema de tipos de Haskell es uno de los más sofisticados que existen. Es un sistema polimórfico, que permite una gran flexibilidad de programación, pero a la vez mantiene la correctitud de los programas. Contrariamente a la mayoría de los lenguajes de programación procedimentales actuales, Haskell utiliza un sistema de inferencias de tipos, es decir sabe el tipo resultante de una expresión, por lo que las anotaciones de tipo en un programa son opcionales.

La parte más interesante de Haskell en relación con los tipos son los constructores, las tuplas y las listas. Una tupla es un dato compuesto donde el tipo de cada componente puede ser distinto. Una de las utilidades de este tipo de datos es cuando una función tiene que devolver más de un valor:

```
predSuc :: Integer → (Integer,Integer)
predSuc x = (x-1,x+1)
```

Las listas son colecciones de cero o más elementos de un mismo tipo (a diferencia de las tuplas que pueden tenerlos de diferentes). Los operadores utilizados son el [] y (:). El primero representa una lista vacía, y el segundo denominado *cons* o constructor, permite añadir un elemento al principio de una lista, construyendo la lista en función de agregar elementos a la misma, por ejemplo [Thompson99]:

```
4 : 2 : 3 : []
```

da lugar a la lista [4, 2, 3]. Su asociatividad es hacia la derecha. Un tipo particular de lista son las cadenas de caracteres.

Para terminar diremos que el constructor utilizado para declarar el tipo correspondiente a las distintas funciones es el símbolo →.

4.2. Patrones

Como vimos anteriormente, una función puede ser definida como:

```
f <pat1> <pat2> . . . <patn> = <expresión>
```

donde cada una de las expresiones *<pat1> <pat2> . . . <patn>* representa un argumento de la función, al que también podemos denominar como patrón. Cuando una función está definida mediante más de una ecuación, será necesario evaluar uno o más argumentos de la función para determinar cuál de las ecuaciones aplicar. Este proceso se llama en castellano encaje de patrones. En los ejemplos anteriores se utilizó el patrón más trivial: una sola variable. Como ejemplo, considérese la definición de factorial:

```
fact n = product [1..n]
```

Si se desea evaluar la expresión "fact 3" es necesario hacer coincidir la expresión "3" con el patrón "n" y luego evaluar la expresión obtenida a partir de "product [1..n]" substituyendo la "n" con el "3". Una de los usos más útiles es la aplicación de los patrones a las listas. Por ejemplo la siguiente función toma una lista de valores enteros y los suma:

```
suma :: [Integer] → Integer
suma []       = 0                         -- caso base
suma (x : xs) = x  +  suma xs             -- caso recursivo
```

Las dos ecuaciones hacen que la función esté definida para cualquier lista. La primera será utilizada si la lista está vacía, mientras que la segunda se usará en otro caso. Tenemos la siguiente reducción (evaluación):

```
    suma [1,2,3]
⇒ { sintaxis de listas}
    suma (1 : (2: (3: [])))
⇒ { segunda ecuación de suma (x ← 1, xs ← 2 : (3 : [])) }
    1 + suma (2 : (3 : []))
⇒{ segunda ecuación de suma (x ←2, xs ← 3 : []) }
    1 + (2 + suma (3 : []))
⇒{ segunda ecuación de suma (x ← 3, xs ← []) }
    1 + (2 + (3 + suma []))
⇒{ primera ecuación de suma}
    1 + (2 + (3 + 0))
⇒{ definición de (+) tres veces}
    6
```

Existen otros tipos de patrones

- Patrones Anónimos: Se representan por el carácter (_) y encajan con cualquier valor, pero no es posible referirse posteriormente a dicho valor. Ejemplo:

```
cabeza (x:_) = x
cola (_:xs) = xs
```

- Patrones con nombre: Para poder referirnos al valor que está encajando, por ejemplo, en lugar de definir *f* como

```
f (x:xs) = x:x:xs
```

podría darse un nombre a ***x:xs*** mediante un patrón con nombre

```
f p@(x:xs) = x:p
```

- Patrones *n+k*: encajan con un valor entero mayor o igual que *k*. El valor referido por la variable *n* es el valor encajado menos *k*. Ejemplo:

```
x ^ 0 = 1
x ^ (n+1) = x * (x ^ n)
```

En Haskell, el nombre de una variable no puede utilizarse más de una vez en la parte izquierda de cada ecuación en una definición de función. Así, el siguiente ejemplo:

```
son_iguales x x = True
son_iguales x y = False
```

no será aceptado por el sistema. Podría ser introducido mediante *if*:

```
son_iguales x y = if x==y then True else False
```

4.3. Funciones de orden superior

Los lenguajes funcionales modernos utilizan una poderosa herramienta de programación: las funciones de orden superior. Haskell en particular considera que las funciones pueden aparecer en cualquier lugar donde aparezca un dato de otro tipo (por ejemplo permitiendo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados). Por ejemplo consideremos la siguiente función (Ruiz 04):

```
dosVeces :: (Integer → Integer) → Integer → Integer
dosVeces f x = f ( f x )
```

Entonces al evaluar las siguientes funciones, obtenemos:

```
MAIN> dosveces (*2) 10
40:: Integer

MAIN> dosveces inc 10
12:: Integer
```

También podemos agregar a estas funciones los llamados combinadores, que son funciones de orden superior que capturan un esquema de cómputo. Pongamos por ejemplo el combinador *iter*:

```
Iter :: (Integer → Integer → Integer) → Integer →
     (Integer → Integer)
Iter op e = fun
     where
     fun 0 = e
     fun m@(n+1) = op m (fun n)
```

Entonces podríamos definir las funciones factorial y sumatoria simplemente proporcionando los dos primeros argumentos en forma correcta:

```
factorial :: Integer → Integer
factorial = iter (*) 1

sumatoria :: Integer → Integer
sumatoria = iter (+) 0
```

Obsérvese que la estructura de ambas funciones es idéntica, salvo el operador (*,+).

4.4. Polimorfismo

Algunas funciones pueden tener argumentos de más de un tipo de datos, por ejemplo, la función identidad, que definimos en Haskell como:

```
Id :: a → a
```

donde *a* puede ser cualquier tipo. Entonces tenemos:

```
MAIN> id ‘d’
‘d’ :: char
MAIN> id true
true :: bool
```

Otro ejemplo más complejo es una función que calcule la longitud de una lista:

```
long ls = IF vacia(L) then 0
else 1 + long(cola(L))
```

El sistema de inferencia de tipos infiere el tipo *long::[x] -> Integer*, indicando que tiene como argumento una lista de elementos de un tipo *a* cualquiera y que devuelve un entero. En un lenguaje sin polimorfismo sería necesario definir una función *long* para cada tipo de lista que se necesitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir porciones de código para estructuras similares.

Otro ejemplo, esta vez de una función estándar, definida en el PRELUDE[2], es la función *map*, que aplica una función a todos los elementos de una lista, devolviendo una lista con los resultados:

```
map  :: (a → b) → [a] →lb]
mapf []  = []
map f (x : xs) = f x : map f xs

PRELUDE> map (i 2) [1,2,3] [1,4,9] :: [Integer]
PRELUDE> map toUpper "pepe" "PEPE" :: String
PRELUDE> map ord "pepe" [112,101,112,101] :: [Int]
```

4.5. Programación con evaluación perezosa

El método de evaluación (la forma en que se calculan las expresiones) se llama evaluación perezosa (*lazy evaluation*). Con la evaluación perezosa se calcula una expresión (parcial) solamente si realmente se necesita el valor para calcular el resultado. El opuesto es la evaluación voraz (*eager evaluation*). Con la evaluación voraz se calcula directamente el resultado de la función, si se conoce el parámetro actual.

Dada la evaluación perezosa del lenguaje es posible tener listas infinitas. En lenguajes que usan evaluación voraz (como los lenguajes imperativos), esto no es posible.

El siguiente ejemplo, tomado de Fokker (95), muestra la potencia de este concepto. Supongamos la siguiente función, para saber si un número es primo:

[2] El PRELUDE es el modulo principal del lenguaje, que contiene todas las funciones y operadores predefinidos, y el único que se carga por defecto cuando se ingresa al interprete.

```
primo :: Int → Bool
primo x = divisores x == [1,x]
```

Esta función es totalmente perezosa. Por ejemplo, si evaluamos *primo 30* pasa lo siguiente: Primero se calcula el primer divisor de 30: 1. Se compara este valor con el primer elemento de la lista [1,30]. Para el primer elemento las listas son iguales.

Después se calcula el segundo divisor de 30: 2. Se compara el resultado con el segundo valor de [1,30]: los segundos elementos no son iguales. El operador == ‘sabe’ que las dos listas nunca pueden ser iguales si existe un elemento que difiere. Por eso se puede devolver directamente el valor *false*, y así los otros divisores de 30 no se calculan.

4.6. Listas por comprensión

La notación de listas por comprensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. Esta notación está adaptada de la teoría de conjuntos de Zermelo-Fraenkel. Sin embargo en Haskell se trabaja con listas, no con conjuntos. El formato básico de la definición de una lista por comprensión es:

```
[ <expr> | <cualif_1>, <cualif_2> . . . <cualif_n> ]
```

donde cada ***<cualif_i>*** es un cualificador. Existen dos tipos:

- Generadores: Un cualificador de la forma *pat<-exp* es utilizado para extraer cada elemento que encaje con el patrón *pat* de la lista *exp* en el orden en que aparecen los elementos de la lista. Un ejemplo simple sería la expresión:

```
? [x*x | x <- [1..10]]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Filtros: Una expresión de valor booleano, el significado de una lista por comprensión con un único filtro podría definirse como:

```
[e |condición ] = if condición then [e] else []
```

5. Conclusión

A mi parecer, la programación funcional es un paradigma de programación muy poderoso, que nos permite pensar de una forma completamente diferente cuando tenemos que resolver un problema,

más parecido a la lógica y matemática de alto nivel, que a las abstracciones artificiales de la máquina de Von Neumann. Digo que se debería enseñar en los primeros años de las universidades, y que nos abre definitivamente un camino para mejorar la Ingeniería del Software, ya que sus programas carecen de efectos colaterales, y otros desafortunados productos de la arquitectura tradicional de las computadoras, siendo sencillamente evaluaciones. Sin embargo, queda mucho todavía por recorrer para que este tipo de lenguajes sea de utilización en proyectos cotidianos, pero creo que un primer paso es la enseñanza de este paradigma. Hay experiencias variadas en este sentido, inclusive algunas en nuestro país (Szpiniak 1998), y en los ámbitos académicos de otros países (Chakravarty 2004, Thompson 1997, FDPE 2008), por lo que creo vale la pena su promoción.

Bibliografía

Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of the ACM*, August 1978, Vol 21 (8)

Chakravarty, M. M. and Keller, G., The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming* 14, 1 (Jan. 2004), 113-123

Doets, K. y van Eijck, J. *The Haskell Road to Logic, Math and Programming*, King's College Publications, Londres, 2004.

FDPE 08: Proceedings of the 2008 international workshop on Functional and declarative programming in education, ACM, New York, USA, 2008.

Fokker, J. *Functional Programming*, Department of Computer Science, Utrecht University, 1995

Iverson, K. *A Programming language*, John Wiley and Sons, New York, 1962

Labra G., J. *Introducción al Lenguaje Haskell*, Universidad de Oviedo, Departamento de Informática, 1998

Ruiz, B., Gutiérrez, F., Guerrero, P. y Gallardo, J.E. *Razonando con Haskell, Un curso sobre programación funcional*, Thompson-Paraninfo, Madrid, 2004

Szpiniak, A. F., Luna, C. D., and Medel, R. H. Our experiences teaching functional programming at University of Río Cuarto (Argentina). *SIGCSE Bulletin,* 30(2) (Jun. 1998), 28-30

Thompson, S. Where Do I Begin? A Problem Solving Approach in teaching Functional Programming. In H. Glaser, P. H. Hartel, and H. Kuchen, Eds., *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Special Track on Declarative Programming Languages in Education* (September 3-5, 1997). Lecture Notes in Computer Science, vol. 1292. Springer-Verlag, London, 323-334

Thompson, S. *Haskell: The Craft of Functional Programming*, Addison-Wesley, Boston, MA, 1999

Wikipedia contributors, "Quicksort," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Quicksort&oldid=250850445 (consultada 24 Noviembre 2008).