LOS LENGUAJES FUNCIONALES - NUESTRA EXPERIENCIA CON KRC

Miguel Santana - Fabienne Carrier
Institut IMAG - *Francia*
Eva Rodríguez
Universidad Central de Venezuela - *Venezuela*

1. INTRODUCCION

La importancia actual de los lenguajes funcionales puede ser atribuída en gran parte a la crisis del software, que ha hecho pensar en estos lenguajes como una alternativa a los lenguajes de programación clásicos.

Efectivamente, debido a su poder de expresión, los lenguajes funcionales corresponden bien a las necesidades de las aplicaciones recientes de la Informática: sistemas expertos, comunicación hombre-máquina, ingeniería del software, etc. Por otro lado, son capaces de explotar las posibilidades de cálculo paralelo de las arquitecturas hardware modernas, gracias a su paralelismo intrínseco.

Sin embargo, estas características no han bastado para que estos lenguajes se impongan. Efectivamente, ciertos defectos les son atribuídos por los programadores habituados a los lenguajes clásicos; se les reprocha en particular su ineficiencia en los computadores actuales así como la dificultad para escribir una aplicación real.

El objetivo de este artículo es de estudiar la adecuación de los lenguajes funcionales a la expresión de problemas informáticos clásicos; para ésto, nos servimos de un lenguaje funcional moderno sobre el cual hemos trabajado durante los dos últimos años: KRC (Kent Recursive Calculator). Nuestro trabajo forma parte de un proyecto más ambicioso [Briat 85], cuyo objetivo final es la realización de un sistema de producción de software, en base a un lenguaje de la quinta generación y a una arquitectura paralela (16 a 1024 procesadores).

La sección 2 de este artículo describe de manera informal el lenguaje KRC, ilustrándolo con múltiples ejemplos. La sección 3 presenta una evaluación de KRC y, a través de ella, de los lenguajes funcionales en general; dos aspectos constituyen la parte más importante de nuestro análisis: el poder de expresión de estos lenguajes y su aplicación eventual a un contexto real de producción de software. Luego, describimos en la sección 4 la implementación de KRC que realizamos para llevar a cabo nuestras experiencias. Finalmente, presentamos en la sección 5 las características que nos parece indispensable agregar a los lenguajes funcionales para volverlos *utilisables*, antes de concluir sobre el interés de nuestro trabajo y sobre las perspectivas de desarrollo futuro.

2. DESCRIPCION DEL LENGUAJE KRC.

KRC es un lenguaje funcional puro, en el cual no existen las nociones de efectos secundarios (la afectación, en particular) ni de flujo de control; es un lenguaje simple, basado en las ecuaciones recursivas de orden superior. Fue diseñado en la Universidad de Kent con un objetivo preciso: la enseñanza de la programación funcional [Turner 81, Turner 82].

KRC es además une lenguaje interactivo; efectivamente, la definición de ecuaciones así como la evaluación de expresiones son efectuadas dinámicamente, tal como sucede en los sistemas LISP. Por otro lado, el orden de definición de las ecuaciones no tiene ninguna importancia; sólo es necesario que una ecuación ya haya sido definida cuando la evaluación de la expresión en curso lo requiere.

Un programa KRC es un conjunto de ecuaciones; estas ecuaciones constituyen una definición matemática de los objetos usados por el programa y de las relaciones existentes entre estos objetos. KRC dispone de dos tipos de ecuaciones: las definiciones y las expresiones simples. Las primeras asocian un nombre a un valor o a una función, que puede luego ser usado en otras ecuaciones. Las expresiones simples son evaluadas inmediatamente y el resultado impreso en la pantalla; estas expresiones corresponden por consiguiente a la ejecución de un programa.

Los diferentes aspectos del lenguaje son presentados en los párrafos que siguen. Nuestra presentación es voluntariamente concisa y se apoya en ejemplos simples para ilustrar cada aspecto.

2.1. OBJETOS DEL LENGUAJE.

KRC dispone de dos clases de objetos: elementales y estructurados. Estos últimos son definidos por el usuario gracias a ciertas construcciones del lenguaje y son de dos tipos: listas y funciones. Cabe indicar que un objeto estructurado puede ser manipulado como cualquier otro objeto y, en particular, ser pasado como parámetro o devuelto como resultado de una función.

2.1.1. OBJETOS ELEMENTALES.

Cuatro tipos de objetos predefinidos existen en KRC: enteros, reales, cadenas de caracteres y lógicos. Todas las operaciones aritméticas, lógicas y de comparación usuales pueden ser usadas con estos objetos. Las expresiones que siguen constituyen un ejemplo de lo que puede ser escrito:

```
-100 + 5654
1.3 / 12e2 * -8.0
"string1" < "string2"  |  15e3 >= 7685
```

2.1.2. LISTAS.

Una lista KRC es un conjunto ordenado de objetos. Estos objetos, llamados elementos de la lista, pueden ser de cualquier tipo; el orden entre los elementos es definido por sus posiciones respectivas dentro de la lista. Una lista es definida por enumeración de sus elementos:

```
[]      [1,2,3,4,5]        ["a","b"]
[ "David", [15,"Janvier",85], true]
```

Un intervalo de enteros puede ser definido usando una forma sintáctica más agradable: `[-2..2]` (equivalente a `[-2,-1,0,1,2]`). Es igualmente posible definir un intervalo infinito de enteros: `[1..]` (enteros positivos).

Una gran cantidad de operadores y de funciones de manipulación de listas existen en

KRC, entre ellos:

- adición de un elemento al comienzo de una lista (*cons*):
  `"a": ["e","i","o"]  0: [1..]`
- acceso al elemento n^{o} i de una lista (indexación):
  `index( ["a","b","c"], 2 )` cuyo valor es `"b"`.
- longitud de una lista:
  `# [1..5]` cuyo resultado es 5.
- concatenación de dos listas:
  `["a","b","c"] ++ ["d","e"]` igual a `["a","b","c","d","e"]`
- diferencia de dos listas:
  `[1..] -- [1..5]` equivalente a `[6..]`
- cabeza y cola de una lista:
  `hd( ["e","i","o"] )` cuyo valor es `"e"`
  `tl( ["e","i","o"] )` cuyo valor es `["i","o"]`

etc.

Además, dos listas pueden ser comparadas usando los operadores de igualdad y de desigualdad usuales:

```
[1,2,3,4,5]   =  [1..5]
["e","i","o"] <> tl( ["e","i","o"] )
```

La construcción de una lista es efectuada según un mecanismo de evaluación perezosa [Friedman 76]. La construcción efectiva es por consiguiente retardada hasta el primer acceso a la lista, momento en el cual son construídos los elementos de la lista implicados en ese acceso y sólo ellos; por esta razón, una lista infinita no conduce necesariamente a una iteración infinita.

2.1.3. FUNCIONES

Una función es definida por una o varias ecuaciones. Cada ecuación está constituída de un encabezamiento, que define su nombre y sus parámetros, y de un cuerpo, que define el valor de la ecuación. Todas las ecuaciones de una misma función deben tener el mismo nombre y el mismo número de parámetros; si este último es igual a cero, se trata de la definición de una constante.

La notación usada para designar la aplicación difiere ligeramente de aquella que usa Turner. Efectivamente, hemos preferido que los parámetros sean especificados entre paréntesis y separados por comas, en vez de usar la yuxtaposición simple, como Turner. Esta notación otorga una mejor lisibilidad y sobre todo, facilita la tarea de análisis sintáctico (ambiguedades, recuperación de errores). Los ejemplos que siguen ilustran las diferentes posibilidades:

```
% Suma de los n primeros enteros %
def suma(n) = ((n+1)*n) / 2.
suma(4).   % suma de los 4 primeros enteros %
```

```
% Días de la semana %
def laborables = ["lunes","martes","miércoles","jueves","viernes"].
def weekend    = ["sábado","domingo"].
def semana     = laborables ++ weekend.
index(semana,3).   % tercer día de la semana %
```

Como una función puede tener varias ecuaciones, el usuario puede especificar las condiciones de aplicación de cada ecuación, usando dos tipos de construcciones disponibles en KRC:

a) Las guardas, expresiones lógicas asociadas al cuerpo de una ecuación. Estas guardas permiten de escoger entre las diferentes expresiones que componen el cuerpo de una ecuación; su semántica es similar a la del *cond* LISP: una expresión es evaluada únicamente si su guarda es verdadera. El orden de evaluación de estas guardas no es impuesto por el lenguaje; sin embargo, una sola expresión debe ser seleccionada y ejecutada. La última expresión de la ecuación no debe tener una guarda, lo que podría conducir a una aplicación indefinida. Ejemplo:

```
% El más grande común divisor %
def mgcd(i,j) = i,          i=j,
                mgcd(i-j,j), i>j,
                mgcd(i,j-i).
```

b) Los filtros, que corresponden a restricciones sobre los valores de los parámetros de la ecuación. Estas restricciones son definidas mediante una especie de *pattern matching*; un parámetro puede ser restringido a un valor dado o a una forma determinada de lista. Todos los filtros de una ecuación tienen que ser respetados para que la ecuación pueda ser aplicada. Ejemplos:

```
% Factorial %
def fact(0) = 1;
    fact(1) = 1;
    fact(n) = n*fact(n-1).

% Suma de los elementos de una lista %
def suma([])          = 0;
    suma(primero:resto) = primero + suma(resto).

% Multiplicación de números complejos %
def mult([r1,i1],[r2,i2]) = [r1*r2-i1*i2,r1*i2+i1*r2].
```

Ninguna limitación existe sobre el tipo de los parámetros; una función puede perfectamente recibir o retornar otra función. Los ejemplos que siguen ilustran esta posibilidad:

```
% Aplicación de una función a los elementos de una lista %
def map(f,[])     = [];
    map(f,head:tail) = f(head): map(f,tail).

% Identificación de los comandos de un editor. El resultado
  es el nombre de la función que efectuará el tratamiento %
def comando(clave) = insert, clave="i",
                     delete, clave="d",
```

```
...
unknown.
```

2.2. CONJUNTOS.

Otra facilidad disponible en KRC para la construcción de listas son los conjuntos. Esta facilidad permite expresar las nociones de conjunto y de cuantificación, sin tener que escribir explícitamente una recursión. Es necesario precisar que el resultado obtenido con esta construcción no es verdaderamente un conjunto sino una lista; un valor puede así aparecer varias veces en este resultado.

Los conjuntos KRC constituyen una notación muy poderosa; así, por ejemplo, el conjunto de enteros pares positivos puede ser definido de una manera muy simple:

```
{ i; i <- [1..], i mod 2 = 0 }
```

Como se puede ver en este ejemplo, la definición de un conjunto comprende tres partes:
- una expresión de definición i, que especifica los elementos del conjunto,
- un generador i <- [1..], que precisa el dominio de valores en el cual está definida la variable i,
- una guarda i mod 2 = 0, que define la condición que deben respectar los valores generados para pertenecer al conjunto.

Ejemplos:

```
% Otra definición de la función map %
def map(f,lista) = { f(elemento); elemento <- lista }

% Conjunto de números primos (método de Eratosteno) %
def primos      = filtro( [2..] ).
def filtro(p:r) = p: filtro( {e; e<-r, (e div p) > 0} ).
```

La definición de un conjunto puede contener varios generadores. El orden de definición de estos generadores es importante porque es él quien define la visibilidad de la variable asociada; efectivamente, una variable sólo puede ser usada en un generador que no es el suyo, si ya ha sido definida por un generador anterior. En virtud de esta característica, la guarda de un generador puede constituir una condición global, si hace intervenir variables de generadores precedentes. Cabe indicar que la expresión de definición del conjunto puede usar cualquier variable. Ejemplo:

```
% Parejas de números pares cuya suma es igual a 10 %
{ [i,j]; i <- [1..10], i mod 2 = 0;
         j <- [1..10], j mod 2 = 0  &  i+j = 10 }
```

Para terminar, es importante indicar que un conjunto puede ser usado en lugar de una lista, sin restricción alguna.

2.3. EJEMPLOS.

El objetivo de este párrafo es de ilustrar el poder de expresión de KRC. Para ello, presentamos la versión KRC de algoritmos clásicos, que tienen la doble virtud de ser ampliamente conocidos y de escritura relativamente sucinta.

2.3.1. FUNCIONES CLASICAS.

```
% FUNCION DE ACKERMANN %

def ackermann( 0, n ) = n + 1;
    ackermann( m, 0 ) = ackermann( m-1, 1 );
    ackermann( m, n ) = ackermann( m-1, ackermann(m,n-1) ).

% NUMEROS DE FIBONACCI %

% Definición standard %
def fib( 1 ) = 1;
    fib( 2 ) = 1;
    fib( n ) = fib( n-1 ) + fib( n-2 ).

% Versión optimizada: cada número es calculado una vez %
def fib'( 1 ) = 1;
    fib'( 2 ) = 1;
    fib'( n ) = index( n-1, fiblist ) + index( n-2, fiblist ).
def fiblist   = fibl( 1 ).
def fibl( i ) = fib'( i ): fibl( i+1 ).
```

2.3.2. FUNCIONES DE ORDEN SUPERIOR. GENERICIDAD.

```
% DOBLE APLICACION DE UNA FUNCION %

def twice( f, val ) = f( f(val) ).

def sqr( i ) = i * i.
def power4   = twice( sqr ).

% APLICACION DE UNA FUNCION CUALQUIERA A UNE LISTA %

def fold( f, init, [] )  = init;
    fold( f, init, p:r ) = f( p, fold(f,init,r) ).

def add( op1, op2 ) = op1 + op2.
def sum             = fold( add, 0 ).
```

2.3.3. APLICACION DE LOS CONJUNTOS.

```
% PRODUCTO CARTESIANO %

def cartesiano( l1, l2 ) = { [i,j]; i <- l1; j <- l2 }.

% NUMEROS PRIMOS %

def primos =
       { primo; primo <- [2..50],
         { divisor; divisor <- [ 2..round( sqrt(primo+1) ) ],
                    primo mod divisor = 0 }
         =
         [] }.
```

2.3.4. QUICKSORT.

```
% SEGMENTACION + FUSION %

def quicksort( [p] )     = [];
    quicksort( i:resto ) =
       fusion( i, segmentar(i,resto,[],[]) ).

def fusion( pivot, [parte1,parte2] ) =
       quicksort( parte1 ) ++ pivot: quicksort( parte2 ).

def segmentar( pivot, [], parte1, parte2 ) =
       [ parte1, parte2 ];
    segmentar( pivot, i:resto, parte1, parte2 ) =
       segmentar( pivot, resto, i:parte1, parte2 ), i <= pivot,
       segmentar( pivot, resto, parte1, i:parte2 ).

% DEFINICION POR CONJUNTOS %

def quicksort'( [] )  = [];
    quicksort'( p:r ) = quicksort'( { v; v<-r, v<=p } )
                        ++
                        p: quicksort'( { v; v<-r, v>p } ).
```

2.3.5. PROBLEMA DE LAS 8 REINAS.

```
% SOLUCION CLASICA %

def OchoReinas (0) = [ [] ];
    OchoReinas (n) = {  ListaReinas ++ [reina];
                        reina <- [1..8];
                        ListaReinas <- OchoReinas (n-1),
                        correcta (reina, ListaReinas) }.
```

```
def correcta (reina, lista) =
       and ( { ~test (reina,lista,i); i <- [1..#lista] } ).

def test (reina, lista, i) =
       reina = index (i,lista)
       |  abs (reina - index (i,lista)) = (#lista-i)+1.
```

3. EVALUACION DE KRC.

A primera vista, KRC constituye un lenguaje sumamente atractivo y con un gran poder de expresión, como lo muestran los ejemplos presentados. Sin embargo, un análisis más detallado incita a preguntarse si podría ser usado con éxito en la realización de una aplicación real. Para tratar de responder a esta pregunta y a otras similares, nos pareció necesario efectuar una evaluación de este lenguaje, tratando de separar claramente sus virtudes y sus límites; fue así como decidimos escribir en KRC un cierto número de aplicaciones, pertenecientes en su mayor parte al área del Software de base y de la Ingeniería del Software. Los párrafos que siguen presentan las conclusiones más importantes de esta experiencia; para mayores precisiones, el lector puede consultar [Carrier 85].

3.1. PODER DE EXPRESION.

Dos aspectos de KRC nos parecen de un gran interés: su notación ecuacional y su mecanismo de filtrado de parámetros; estos dos aspectos se amoldan perfectamente a la expresión de una cantidad apreciable de problemas. Por este motivo, KRC ofrece un nivel de abstracción bastante elevado, cercano de la especificación; efectivamente, la definición KRC de un problema recuerda muchas veces la formulación matemática del mismo. Este nivel de abstracción permite al usuario de expresar su algoritmo sin preocuparse de los detalles relativos a su implementación: representación de sus datos, administración de la memoria, flujo de ejecución, etc. (nociones indispensables en un lenguaje imperativo).

El mecanismo de filtrado ofrece dos ventajas importantes, que facilitan enormemente la labor del programador. Por un lado, permite distinguir los diferentes casos de una función según la structura de los parámetros; por otro lado, reduce el uso de las funciones de acceso, mejorando la lisibilidad del programa y reduciendo los errores (muchas veces delicados) correspondientes a este tipo de acceso. Las dos definiciones siguientes ilustran estos aspectos:

```
def suma3 (triple)  =
          hd(triple) + hd(tl(triple)) + hd(tl(tl(triple))).

def suma3 ([a,b,c]) = a + b + c.
```

Nos han igualmente seducido por su poder de expression los conjuntos KRC. Efectivamente, estos conjuntos permiten especificar de una manera simple y concisa una noción importante de la programación: la aplicación de un mismo tratamiento a una colección de datos (la iteración en los lenguajes clásicos); esta posibilidad es además enriquecida por un mecanismo de filtrado, que permite seleccionar el sub-conjunto de la colección al cual se quiere aplicar el tratamiento. Por esta razón, los conjuntos KRC se adaptan muy bien a la

expresión de una gama importante de algoritmos, que se pueden resumir en términos de *el conjunto de objetos que verifican tal propiedad.*

Otro aspecto interesante de KRC es la semántica no estricta de sus funciones, lo que les permite retornar un resultado inclusive cuando ciertos parámetros no son definidos; en particular, la evaluación de un parámetro que no es usado y que conduciría a una evaluación infinita no es jamás efectuada. Esta propiedad garantiza el término de la ejecución de una función cuando existe por lo menos una solución de ésta. Este mismo principio es aplicado a las listas, lo cual otorga la posibilidad de definir estructuras de datos potencialmente infinitas.

Finalmente, ciertas propiedades de KRC, y de los lenguajes funcionales en general, nos parecen igualmente de suma importancia. La más importante de ellas es la transparencia de las funciones: dado un mismo conjunto de parámetros, una función siempre da el mismo resultado; esta propiedad está ligada a la ausencia de afectación en el lenguaje y al hecho que una variable representa un valor y no un objeto, sujeto a modificaciones. La importancia de esta propiedad es doble:

a) Posibilidad de efectuar diversos tratamientos formales: prueba de igualdad entre dos funciones, transformaciones, prueba de programas, etc. Esta posibilidad se ve reforzada por la solidez de las bases teóricas sobre las cuales reposan estos lenguajes: lambda-cálculo, lógica combinatoria, etc.

b) Posibilidad de efectuar una implementación paralela. Diferentes partes de un programa pueden efectivamente ser efectuadas en paralelo: evaluación de los argumentos de una función, construcción de los elementos de una lista, etc.

En conlusión, podemos decir que KRC constituye un lenguaje poderoso, que ofrece un poder de expresión y un nivel de abstracción bastante elevados. Sin embargo, ciertos aspectos de KRC nos parecen contrarios a un uso real del lenguaje; el análisis de estos inconvenientes constituye el objeto del párrafo siguiente.

3.2. APLICACION A UN CONTEXTO DE PRODUCCION.

La producción industrial de software tiene sus imperativos y exigencias en cuanto a la calidad y cualidades de las herramientas de desarrollo; por esta razón, un lenguaje como KRC sólo sería utilizado en este contexto si respeta todas estas condiciones. Las líneas que siguen tratan de identificar, por un lado, las características de KRC que se adaptan a este contexto y, por otro lado, aquellas que son contrarias a él o que son simplemente ausentes.

Una primera característica importante de KRC es la notación clara y natural que utiliza, lo que le da una excelente lisibilidad.. Esta característica constituye una ventaja importante en relación a otros lenguajes funcionales; no estamos frente a la gran cantidad de paréntesis de LISP o de caracteres especiales de FP. Este aspecto reviste una gran importancia en un contexto de producción de software, debido a la evolución constante de éstos (mantenimiento y nuevas versiones).

El nivel de abstracción de KRC, bastante cercano de la especificación, constituye otra ventaja importante de este lenguaje. Efectivamente, esta similitud nos hace pensar en una reducción del ciclo de producción de un software y, por consiguiente, de su costo.

Sin embargo, estas virtudes aliadas a su poder de expresión no confieren a KRC la categoría de *lenguaje de producción*. Muchas características, usadas intensivamente en la producción de software, son completamente ausentes o demasiado simples en KRC y, generalizando, en los lenguajes funcionales:

a) La pobreza de las estructuras de datos disponibles: en la mayoría de casos, la lista constituye la única estructura de datos existente. Esta pobreza dificulta considerablemente la programación de ciertas aplicaciones; en particular, un gran número de parámetros son a menudo necesarios para transmitir a una función las informaciones que necesita.

b) La manipulación eficaz de estructuras de datos. Estas operaciones han sido definidas por lo general en torno a la noción de variable: agrupamiento de variables, lectura o modificación de una de estas variables, etc. Esta definición es totalmente incompatible con los conceptos de base de los lenguajes funcionales, en los cuales una estructura de datos es considerada como un todo. Esta característica es una fuente importante de ineficiencia, tanto desde el punto de vista ocupación de memoria como de tiempo de ejecución, pues, toda modificación implica la reconstrucción completa de la estructura de datos.
Para resolver este problema, dos soluciones son posibles. La primera consiste en transgredir los principios de estos lenguajes, autorizando la manipulación de variables; es la solución adoptada por LISP o ML. La segunda solución consiste en efectuar una administración más fina de la memoria, encargada entre otras cosas de efectuar las modificaciones de estructuras de datos de una manera más eficiente [O'Donnell 85]; esta solución ha sido muy poco estudiada y merecería una mayor atención.

c) Las entradas/salidas. La noción de entrada/salida está en conflicto total con las propiedades de base de los lenguajes funcionales; efectivamente, las entradas/salidas han sido definidas como si se tratara de afectaciones a, o de, una variable externa (archivo, periférico). Una consecuencia de esta definición es la falta de transparencia de toda función de entrada/salida; así, por ejemplo, la función *read* no retorna siempre el mismo valor sino los elementos consecutivos de un archivo. Otra consecuencia es la incompatibilidad entre la secuencialidad de las entradas/salidas y la ausencia de orden de evaluación de los lenguajes funcionales, que puede conducir a resultados sorprendentes; así, por ejemplo, es difícil decir en que orden serán ejecutadas las llamadas de la función *read* en la expresión *f(read(), read())*.
El tratamiento de las entradas/salidas nunca ha sido completamente resuelto en los lenguajes funcionales. Una idea que comienza a popularizarse consiste en tratar un archivo como un flujo de datos (*stream*), cuya evaluación es efectuada por intermedio del mecanismo de evaluación perezosa. Esta idea parece sumamente interesante pero necesita ser desarrollada un poco más.

Otros problemas son específicos a KRC. Entre ellos podemos citar:

a) Imposibilidad de definir variables locales a una función. Esta carencia, aunada a la imposibilidad de definir el cuerpo de función como una secuencia de expresiones, hacen que la programación en KRC sea bastante enredada cuando una función tiene que calcular un valor intermediario. Ciertos problemas simples se convierten de esta manera en verdaderos rompecabezas, en los que el programador se ve obligado de introducir funciones intermediarias que *simulan* estas posibilidades. La introducción de una construcción de tipo *let* permitiría de resolver este problema.

b) Ausencia de funciones anónimas (el equivalente de las expressiones lambda de LISP). Esta construcción es particularmente necesaria cuando uno desea pasar como parámetro o retornar como resultado una función.

c) Ambiguedades e imprecisiones en la definición del lenguaje: exclusividad entre las guardas de una función, orden de evaluación de los generadores de un conjunto, etc. Estos problemas son sin embargo de poca importancia y pueden ser fácilmente resueltos.

d) Noción de tipo. Ningún mecanismo de elaboración ni de verificación de tipos existe en KRC. Cuando uno sabe que muchos errores de programación son detectados por estos mecanismos, sólo puede lamentarse de esta ausencia.

Como se puede observar, la mayor parte de las carencias propias de KRC son menores con respecto a las que se puede imputar a los lenguajes funcionales en general. Una buena implementación del lenguaje puede resolver estos pequeños problemas, a partir de una definición precisa y de unas cuantas extensiones al lenguaje.

Todos los problemas señalados no deben llevarnos a rechazar estos lenguajes sino, por el contrario, a tratar de mejorarlos y de encontrar las técnicas y métodos apropiados para imponerlos. La sección que sigue presenta los puntos que, en nuestra opinión, deben ser estudiados y desarrollados en prioridad para lograr este objetivo.

4. NUESTRA IMPLEMENTACION DE KRC.

La realización de nuestro proyecto necesitaba una experimentación real, para poder obtener resultados significativos. Como el único sistema KRC existente no podía ser instalado en nuestro laboratorio por razones materiales, decidimos efectuar nuestra propia implementación de KRC. Esta decisión se vió corroborada por nuestro deseo de experimentar, por un lado, con las técnicas de implementación de los lenguajes funcionales y, por otro lado, con el lenguaje mismo.

Nuestra implementación fue realizada en un VAX-780, usando Franzlisp como lenguaje de desarrollo. Esta sección presenta los diferentes aspectos de esta implementación; una parte importante es dedicada a la técnica de evaluación utilizada, por la importancia que reviste en cuanto a la eficiencia del sistema.

4.1. TECNICA DE EVALUACION.

La semántica de KRC impone un orden normal de evaluación, tanto en lo que concierne las listas (evaluación perezosa) como en la transmisión de parámetros (pasaje por nombre o por necesidad). La técnica de los combinadores [Turner 79] se adapta muy bien a un tal lenguaje, pues ofrece directamente un orden normal de evaluación. En cambio, la lambda-reducción corresponde a un orden aplicativo, el costo de su adaptación a un orden normal resultando relativamente elevado.

Esta razón, aunada a nuestra deseo de experimentar con este nuevo método de evaluación, nos condujo a escoger la técnica de los combinadores. Esta técnica se basa en ciertos resultados

de la lógica combinatoria y especialmente en la demostración [Curry 58] de que las variables no son necesarias al cálculo de una ecuación y pueden ser reemplazadas por un cierto tipo de constantes, llamadas combinadores. Un algoritmo de transformación (abstracción de variables), capaz de transformar cualquier expresión en su equivalente sin variables, fue igualmente propuesto. La evaluación de estas expresiones puede enseguida ser efectuada aplicando un conjunto de reglas de reducción, que definen la semántica de los combinadores usados.

La técnica de los combinadores fue propuesta por Turner como una alternativa a la técnica clásica de evaluación de los lenguajes funcionales [Landin 64, Baker 78]. Ha sido usada en la implementación del lenguaje SASL así como en la realización de la máquina SKIM [Clarke 80, Stoye 84]; en ambos casos, los resultados obtenidos han sido sumamente interesantes tanto desde el punto de vista de la eficiencia como de la sencillez de la realización. Los párrafos que siguen presentan los diferentes aspectos de esta técnica.

4.1.1. DESCRIPCION GENERAL.

En el método de Turner, todo programa es traducido en un conjunto de expresiones, compuestas exclusivamente de combinadores, de constantes y de aplicaciones de funciones. La evaluación de una expresión consiste en reducir su forma combinatoria en un valor elemental.

Los dos combinadores de base de Curry son usados:

```
S f g x = f x (g x)
K x y   = x
```

Según las conclusiones de Curry, estos combinadores son suficientes para calcular cualquier expresión funcional. Sin embargo, otros combinadores son necesarios esencialmente por razones prácticas; efectivamente, estos nuevos combinadores reducen considerablemente la talla de las expresiones combinatorias y disminuyen al mismo tiempo el número de pasos de reducción. Se trata en realidad de optimizaciones de ciertas formas de S y de K:

```
I x     = x
B f g x = f (g x)
C f g x = f x g
```

Así, por ejemplo, la expresión combinatoria

```
S (C (K +) ((* 4) 2)) (K 5)
```

equivalente a

```
4*2 + 5    o    ((+ ((* 4) 2)) 5)  (forma currificada)
```

sería evaluada por la serie de reducciones siguiente:

```
S (C (K +) ((* 4) 2)) (K 5) nil
=> (C (K +) ((* 4) 2) nil) (K 5 nil)    REDUCCION S
=> (K + nil ((* 4) 2)) (K 5 nil)        REDUCCION C
=> (+ ((* 4) 2)) (K 5 nil)              REDUCCION K
=> (+ 8) (K 5 nil)                      EVALUACION *
=> (+ 8) 5                              REDUCCION K
=> 13                                   EVALUACION +
```

El método de Turner puede ser descompuesto en dos etapas, descritas a continuación.

4.1.2. ABSTRACCION DE VARIABLES.

Esta etapa consiste a suprimir las variables usadas por la expresión tratada. La supresión es efectuada por un algoritmo conocido con el nombre de abstracción de variables; la operación de abstracción puede ser definida de la manera siguiente:

Sean la expresión E y la variable x. La abstracción de E respecto de x, cuya notación es [x] E, es una expresión que no contiene ninguna ocurrencia de x y que respeta la ecuación:

([x] E) x = E

La abstracción es por lo tanto la operación inversa de la aplicación.

Esta operación puede ser usada para suprimir los parámetros de una función, en la expresión que constituye su cuerpo. De esta manera, la ligazón entre parámetros efectivos y parámetros formales es inútil, lo cual conduce a una mayor eficiencia respecto del lambda-cálculo.

El algoritmo de abstracción es definido por un conjunto de reglas de transformación, que permiten traducir una expression lambda (tipo LISP) en una expresión combinatoria. La expresión inicial tiene que estar en una forma currificada: toda aplicación de función debe tener un solo argumento. Las reglas de transformación son las siguientes:

```
[x] (E1 E2) => S ([x] E1) ([x] E2)
[x] x       => I
[x] y       => K y
```

donde x es la variable que se quiere abstraer e y una constante o una variable; la aplicación es considerada asociativa por la izquierda.

Además, dos reglas de optimización son usadas por el algoritmo:

```
S (K E1) (K E2) => K (E1 E2)
S (K E1) I      => E1
```

Finalmente, los combinadores B y C constituyen dos casos particulares del combinador S, en los cuales la variable x está ausente:

```
S (K E1) E2 => B E1 E2
S E1 (K E2) => C E1 E2
```

El conjunto de reglas debe ser aplicado en el orden definido para obtener un mejor resultado.

Otras reglas de optimización fueron definidas como consecuencia de una observación simple: la aplicación repetitiva de la abstracción provoca la aparición frecuente de ciertas expresiones tipo. Otras formas más simples y eficientes de estas expresiones pueden ser usadas:

```
S (B EK E1) E2 => S' EK E1 E2
B (EK E1) E2   => B' EK E1 E2
C (B EK E1) E2 => C' EK E1 E2
```

donde EK es una expresión constante y E1, E2 son cualquier tipo de expresiones. Estas optimizaciones son aún más interesantes si el número de variables a abstraer es importante.

El algoritmo de abstracción tiene que ser aplicado tantas veces como variables hay. Así, por ejemplo, la abstracción de la función:

```
f(x, y, z) = E
```

corresponderá a la expresión:

```
[z] ([y] ([x] E))
```

La abstracción de variables puede ser extendida al nombre de la función e inclusive a las variables y funciones globales.

La expresión obtenida puede ser representada como una forma arborescente, en la cual todos los nodos internos corresponden a la operación de aplicación y las hojas a una constante, un combinador o un nombre de función.

4.1.3. REDUCCION DE UNA EXPRESION.

La etapa de reducción corresponde al cálculo del valor de una expresión. Este cálculo es realizado por un algoritmo de transformación de grafo; efectivamente, la forma arborescente de la expresión es transformada por aplicación sucesiva de las reglas de reducción que definen los combinadores. Este algoritmo implementa de manera natural una evaluación perezosa.

La evaluación es efectuada según un orden normal de reducción. Una etapa de evaluación consiste a transformar la expresión mediante la regla de reducción asociada a la hoja ubicada más hacia la izquierda; la reducción es efectuada hasta obtener un valor elemental.

En el orden normal de reducción, los parámetros son transmitidos según el esquema de llamada por nombre; la ventaja de este esquema es que garantiza que la ejecución se termina, siempre y cuando sea posible. En la técnica de los combinadores, este esquema es reemplazado por la llamada por necesidad, esquema equivalente pero mucho más eficiente. Efectivamente, en este esquema los parámetros efectivos son únicamente apuntadores hacia la verdadera expresión, lo cual permite compartirla y sobre todo limitarse a evaluarla una sola vez como máximo; no hay que olvidar que la reducción transformará esta expresión en un valor elemental en la primera ocasión. Gracias a este mismo mecanismo, las expresiones estáticas solo serán evaluadas una sola vez durante toda la existencia del programa.

El algoritmo de reducción usa una pila de evaluación. Este algoritmo consiste a empilar el nodo tratado mientras se trate de una aplicación y a tratar luego el hijo izquierdo; cuando se trata de una hoja, tres casos pueden presentarse:

a) La hoja es una constante, que es retornada directamente.

b) La hoja es un combinador. La regla de reducción correspondiente es aplicada; los parámetros necesarios se encuentran en la pila de evaluación.

c) La hoja es el nombre de una función. La hoja es en este caso sustituída por el cuerpo de la función y la reducción reactivada sobre éste. Las funciones predefinidas son tratadas como combinadores especiales, por razones de eficiencia.

4.2. DESCRIPCION DEL SISTEMA REALIZADO.

Nuestra implementación ha sido realizada en base a un intérprete, que se encarga de traducir las formas KRC en un árbol de combinadores y de reducirlo mediante las reglas presentadas en el párrafo anterior. El intérprete está acompañado por un ambiente de trabajo que permite conservar las informaciones y comunicar con el exterior: definición de funciones, evaluación de expresiones, entradas/salidas, etc.

Tres elementos principales constituyen el intérprete:

a) Análizador, encargado del análisis lexical y sintáctico de las ecuaciones KRC así como de ciertas verificaciones contextuales. La ecuación analizada es traducida en una representación intermediaria, que constituye una versión currificada y listificada de la ecuación ; esta representación es mucho más adaptada a las tareas de abstracción y optimización. La sintaxis usada es de tipo LL(1), lo que nos permitió usar un análisis de tipo descendente recursivo.

b) Abstracción, cuyo objetivo es la supresión de las variables de la ecuación analizada y su transformación en una expresión combinatoria; esta expresión es optimizada a medida que va siendo construída. La abstracción es efectuada al momento de definición de una función o de un conjunto, con respecto a las variables que éstos contienen. Todos los combinadores presentados en el párrafo "Técnica de evaluación" son utilizados por nuestro algoritmo.

c) Reducción, que se encarga de la evaluación de las expresiones simples. La evaluación es efectuada mediante el algoritmo de reducción presentado anteriormente ; cabe indicar que la reducción de un combinador es realmente implementada por una transformación física del árbol que corresponde a la expresión. Los operadores de base del lenguaje son tratados como si fueran combinadores ; es interesante indicar que éstos han sido implementados de manera *inteligente*, tratando de retardar lo máximo posible la evaluación de los elementos de una lista.

El ambiente de trabajo está igualmente compuesto por tres elementos:

a) Nivel superior (*toplevel*). Se trata del elemento de control del intérprete, encargado de leer y de ejecutar las ecuaciones tipeadas por el usuario ; esta función es implementada, a la imagen del *toplevel* LISP, mediante un loop compuesto de tres operaciones: *read* (lectura de una ecuación), *eval* (evaluación de la expresión o definición de la función) y *print* (impresión del resultado). La comunicación con el usuario es efectuada a través del terminal pero puede ser redirigida hacia un archivo.

b) Biblioteca de funciones. Esta biblioteca contiene el conjunto de funciones standard de KRC más algunas funciones propias a nuestra implementación. La mayor parte de estas funciones han sido escritas en Franzlisp por razones de eficiencia. Cabe indicar que todas estas funciones pueden ser redefinidas por el usuario.

c) Comandos. Estos comandos permiten al usuario dos tipos de acción: la comunicación con el exterior (redirigir las entradas, salvar la sesión en un archivo, lanzar un *shell*, etc.) y la consulta o modificación de ciertas variables propias al sistema KRC (nivel y longitud de impresión de una lista, pila de llamadas, etc).

4.3. MEJORAS Y EXTENSIONES EFECTUADAS.

Las experiencias efectuadas nos llevaron a cuestionar ciertos aspectos de KRC y a proponer ciertos otros. La mayor parte de estas mejoras y extensiones fueron agregadas a nuestra implementación:

a) Definición precisa de la noción de conjunto.

b) Construcción de tipo *let*, que permite definir un conjunto de variables locales a una expresión.

c) Funciones anónimas (expresiones lambda), que pueden ser definidas y usadas en una expresión cualquiera.

d) Generalización del mecanismo de filtrado: enriquecimiento de los tipos de filtro existentes y extensión de este mecanismo a la definición de variables de un conjunto o de un *let*.

e) Implementación de las entradas/salidas usando la técnica de flujos (*stream*). Nuestra implementación de esta técnica es bastante simple y tendrá que ser completamente rediseñada en una versión futura.

Otras extensiones no han sido tomadas en cuenta por razones de factibilidad. Entre ellas podemos mencionar: nuevas estructuras de datos, funciones de memorización, funciones de modificación parcial de una estructura de datos, etc.

5. HACIA UN LENGUAJE FUNCIONAL "UTILISABLE".

Los lenguajes funcionales constituyen en nuestra opinión una solución posible a la crisis que sufre actualmente el área de producción de software. Sus bases sólidas, su alto poder de expresión y su paralelismo potencial son tres virtudes inapreciables; KRC constituye una prueba fehaciente de nuestra aseveración. Sin embargo, consideramos que aún queda mucho camino por andar para que esta posibilidad se convierta en realidad; tal vez, la tarea más urgente es la adaptación de estos lenguajes a las necesidades específicas de la producción de software, necesidades que han sido por lo general ignoradas durante el diseño de estos lenguajes.

Un primer imperativo que deberá ser respetado es la eficiencia. Efectivamente, nos parece prácticamente imposible llegar a explotar las aplicaciones tradicionales de la Informática con las performances que ofrecen hoy en día los lenguajes funcionales; este problema es aún más crítico en lo que concierne el software de base y las nuevas aplicaciones informáticas. Por lo tanto, nos parece indispensable mejorar sustancialmente esta eficiencia y, si es posible, llevarla hasta un nivel comparable al de los lenguajes actuales de producción (Cobol, Pascal, C, etc). Las investigaciones realizadas actualmente en este sentido son bastante positivas y nos permiten esperar que este objetivo será alcanzado en un plazo relativamente corto; nuestro trabajo se sitúa en esta línea.

Desde el punto de vista de la programación, un número importante de construcciones y de facilidades hacen falta en los lenguajes funcionales para que aplicados a un contexto de producción. Aquellas que nos parecen más importantes son las siguientes:

a) Mecanismos de abstracción que permitan definir y manipular con mayor facilidad las estructuras de datos: modificarlas parcialmente, compartirlas, etc. Agregar estos mecanismos presenta un problema serio de incompatibilidad entre las propiedades de los lenguajes funcionales y la eficiencia necesaria a la producción de software. Un compromiso es por consiguiente necesario.

b) Nociones de objeto y de módulo. Estas nociones nos parecen indispensables pues sólo ellas permiten definir programas que ofrecen *servicios* a otros programas. Estas nociones están igualmente en conflicto con las propiedades de base de los lenguajes funcionales pues implican la noción de estado.

c) Mecanismos más completos para el tratamiento de las entradas/salidas. Los mecanismos existentes son por lo general bastante pobres, debido a sus características poco funcionales. Limitar estos mecanismos equivaldría a excluir muchos tipos de software del área de aplicación de los lenguajes funcionales.

d) Interface con el mundo exterior. Los sistemas funcionales disponibles hoy en día son excesivamente cerrados sobre ellos mismos; efectivamente, son raros los lenguajes funcionales que permiten usar las rutinas de servicio del sistema operativo y, menos aún, programas escritos en otros lenguajes.

e) Mecanismos de tratamiento de excepciones. Estos mecanismos son bastante útiles en cierto tipo de aplicaciones, en las cuales es necesario tratar directamente los diferentes casos de error, sin la intervención del lenguaje de desarrollo o del sistema operativo.

Otro aspecto que deberá ser estudiado con mucho cuidado es el ciclo de producción que corresponde a este tipo de lenguajes, así como las herramientas necesarias a cada una de las etapas de este ciclo. Este trabajo de exploración y definición deberá, en nuestra opinión, inspirarse de los trabajos efectuados en el área de los ambientes de programación.

Finalmente, consideramos que es también indispensable definir una metodología de programación propia a los lenguajes funcionales. Efectivamente, los programadores acostumbrados a los lenguajes algorítmicos encuentran muchas dificultades cuando programan en un lenguaje funcional; una metodología apropiada les permitiría adaptarse con mayor facilidad y beneficiar al mismo tiempo de una escritura más sistemática de sus programas.

6. CONCLUSION.

Hemos presentado en este artículo los resultados de un proyecto de investigación sobre la adecuación de los lenguajes funcionales a la expresión de problemas informáticos clásicos. Este proyecto ha sido realizado en torno al lenguaje KRC, usado como soporte de experimentación en todas nuestras experiencias ; este lenguaje fue escogido principalmente por su elegancia y sencillez. Cabe indicar que este proyecto es el fruto de una colaboración entre la Escuela de Computación de la Univ. Central de Venezuela y el Laboratorio de Ingeniería Informática de Grenoble.

Dos aspectos esenciales han sido desarrollados en nuestra presentación. El primero corresponde a una evaluación de KRC y, por medio de él, de los lenguajes funcionales; numerosas aplicaciones fueron escritas en este lenguaje para efectuar la evaluación; nuestro artículo presenta únicamente los resultados finales de estas experiencias. El segundo aspecto corresponde a la implementación de KRC realizada para efectos de nuestro proyecto.

Por otro lado, nuestro artículo presenta los puntos que nos parece indispensable desarrollar para que estos lenguajes sean realmente *utilisables* en un contexto de producción de software. La falta de eficiencia es probablemente una de las conclusiones más importantes, lo que nos conduce a pensar que su mejora constituye la tarea más urgente; nuestro trabajo se orienta actualmente en esta dirección. Dos aspectos son estudiados por el momento: la compilación y la evaluación paralela del lenguaje; otro aspecto que podría ser estudiado es una implementación hardware (micro-programada o VLSI) de la técnica de los combinadores.

BIBLIOGRAFIA

[Baker 78] Baker H.G, "Shallow binding in Lisp 1.5", CACM 21-7, Julio 1978, pp. 565-569

[Briat 85] Briat J, Carrier F, Rouzaud Y, Santana M, "A la recherche du langage de programmation de la cinquième génération", Congreso AFCET, Octubre 1985, París

[Carrier 85] Carrier F, Rodriguez E, Santana M, "Agréments et péripéties de la programmation fonctionnelle. Notre expérience avec KRC", T.R. 562, LGI/IMAG, Noviembre 1985, Grenoble

[Clarke 80] Clarke T.J.W, Gladstone P.J.S, MacLean C.D, Norman A.C, "SKIM - The S, K, I Reduction Machine", Proceedingsd of the 1980 Lisp Conference, Agosto 1980, Stanford, pp. 128-135

[Curry 58] Curry H.B, Feys R, "Combinator logic", 1958, North-Holland, Amsterdam

[Friedman 76] Friedman D, Wise D, "CONS should not evaluate its arguments", Automata language and programming, ed. Michaelson and Milner, 1976, Edimburgh Univ. Press, pp. 257-284

[Landin 64] Landin P.J, "The mechanical evaluation of expressions", Computer Journal 6-4, Enero 1964, pp. 308-320

[O'Donnell 85] O'Donnell J.T, "An architecture that efficiently updates associative aggregates in applicative programming languages", IFIP Conference on Functional Languages and Computer Architecture, Septembre 1985, Nancy, pp. 164-189

[Rodriguez 85] Rodriguez E, "Mise en oeuvre de langages fonctionnels. Application á KRC", Memoria de DEA, Universidad de Grenoble, Junio 1985, Grenoble

[Stoye 84] Stoye W.R, Clarke T.J, Norman A.C, "Some practical methods for rapid combinator reduction", ACM Symposium on Lisp and Functionnal Programming, Austin, Agosto 1984, pp. 159-166

[Turner 79] Turner D.A, "A new implementation technique for applicative languages", Software-Practice & Experience 9-9, Setiembre 1979, pp. 31-49

[Turner 81] Turner D.A, "The semantic elegance of applicative languages", ACM Conf. on Functionnal Prog. Languages and Computer Architecture, Portsmouth, Octubre 1981, pp. 85-92

[Turner 82] Turner D.A, "Recursion equations as a programming language", en Functionnal Programming and its Applications, ed. Darlington/Henderson/Turner, Indiana University, 1982