TRANSFORMACIONES DE PROGRAMAS
DERIVACION DE VERSIONES APLICATIVAS EFICIENTES

Liliana Favre
Facultad de Ciencias Exactas
Universidad Nacional del Centro
de la Pcia. de Buenos Aires
Pinto 399-Tandil-7000
Argentina

Resumen

La construcción formal de programas por transformaciones (programación transformacional) es un método de desarrollo de software que permite derivar un programa desde una especificación formal de un problema a través de pasos de transformación que preservan la correctitud.
Las experiencias realizadas en síntesis de programas mediante programación inferencial muestran que las transformaciones por reglas en un nivel aplicativo son pasos cruciales para lograr versiones eficientes en un nivel procedural. Se presentan en este trabajo un conjunto de reglas de transformación (folding, unfolding, definición, instanciación, abstracción , embedding, reglas sobre los tipos de datos primitivos, reglas sobre el condicional, reglas sobre expresiones lógicas) y estrategias para aplicarlas (composición, generalización, tupling, y embedding funcional) que constituyen las bases para un sistema de transformaciones.

1. Introducción a la programación transformacional

Las diversas metodologías de programación organizan la construcción de un programa a través del desarrollo de una secuencia de versiones. Los criterios que definen las transiciones entre una versión y su sucesora dependen de la metodología empleada.

En las metodologías basadas en **refinamientos sucesivos** ([DIJ76]) las sucesivas versiones incrementan el grado de detalle, sin alterar la estructura exterior de las versiones previas. La verificación de correctitud es totalmente dependiente del problema.

En las metodologías basadas en transformaciones (programación inferencial) se permiten sólo transiciones formales que preservan una relación semántica. Usualmente el proceso comienza con una especificación (formal) y termina con una versión ejecutable. Las transiciones individuales entre varias versiones resultan de aplicar reglas de transformación que preservan la correctitud . La versión final es correcta por construcción. Las transiciones pueden ser descriptas por reglas esquemáticas y pueden ser empleadas en toda clase de problemas. La estructura del programa es flexible, puede ser alterada en el proceso de desarrollo por ejemplo, combinando dos recursiones en una.

Los primeros avances en transformaciones de programas estuvieron asociados a **transformaciones por esquema** de versiones recursivas en versiones iterativas([DAR74],[WAL73]).La aplicación directa de transformaciones esquemáticas limita generalmente las posibilidades de obtener eficientes versiones. Esto motivó la aparición de transformaciones por reglas más flexibles que intentan transformar versiones aplicativas en otras para las cuales existe un tratamiento esquemático eficiente.

Fueron descubiertas las reglas de **folding** y **unfolding** independientemente por Burstall y Darlington([BUR77]) y por Manna y Waldinger [MAN75]. Esta línea de trabajo fue enriquecida con posteriores aportes que precisaron otras reglas de transformación y estrategias asociadas para aplicarlas.

La transformación por folding y unfolding involucra pequeños pasos y es un serio problema controlar el proceso de transformaciones. Requiere del programador la invención de definiciones, bautizadas por Burstall y Darlington como pasos **eureka**. Son pasos **inteligentes** en el proceso de transformaciones. Pueden compararse con la invención de invariantes de un ciclo en metodologías basadas en refinamiento. Para derivar pasos eureka se emplean **estrategias**. Las mismas juegan el rol de los paradigmas que ayudan al programador a elegir la estructura de un ciclo en refinamientos sucesivos.

Si bien un desarrollo por programación transformacional puede ser hecho con lápiz y papel, existen suficientes razones para usar un sistema implementado para soportar programación inferencial:

- el sistema podría realizar la reescritura de fragmentos de programas e instanciación de ciertas partes de una versión
-el sistema podría asistir al programador con propuestas para aplicar reglas de transformación, como asimismo almacenar la experiencia de los programadores como estrategias de transformación.
-el sistema podría registrar cada decisión de diseño desde la especificación inicial hasta la versión final. Esta historia serviría como documentación, para modificar decisiones de diseño intermedias y para mantenimiento (ante la modificación de la especificación inicial podría repetirse el desarrollo registrado con la especificación modificada).

Existen variados sistemas de transformación que soportan:
-modificación de programas (optimización de estructuras de control, implementación eficiente de estructuras de datos y adaptación de programas a un estilo particular, por ejemplo aplicativo, procedural, etc.)
-síntesis de programas , la generación de un programa desde una descripción formal de un problema
- adaptación de programas a ambientes particulares.

Para un análisis comparativo de sistemas de transformación consultar [PAR84].

Importantes aportes a la programación transformacional ha efectuado el proyecto CIP ("Computer-Aided , Intuition-Guided Programming") dirigido por Bauer. Las actividades fundamentales desarrolladas en este proyecto fueron :

-el diseño y definición formal de un lenguaje de amplio espectro, CIP-L , basado en el ciclo de vida de desarrollo de programas por transformaciones [BAU85]. Este lenguaje permite que coexistan en una misma estructura sintáctica y semántica versiones descriptivas, aplicativas y procedurales.

-el diseño e implementación de un sistema de transformaciones, CIP-S [BAU87]. En principio este sistema podría ser considerado como un sistema experto con una base de conocimiento que es obtenida y extendida por razonamiento formal.

-el desarrollo de una metodología de programación transformacional [PAR90].

Las experiencias realizadas en síntesis de programas mediante programación inferencial muestran que las transformaciones por reglas en un nivel aplicativo son pasos cruciales para derivar versiones procedurales eficientes. Por ejemplo, permiten transformar una doble recursión en una recursión tail, obtener definiciones recursivas menos redundantes, o reemplazar algunos operadores por otros menos costosos. Se presentan en la sección 2. de este trabajo las bases de un sistema de transformaciones de versiones aplicativas. Se describen en 2.1 un conjunto de reglas de transformación y en 2.2 estrategias para aplicarlas. Se ejemplifican en 3. derivaciones realizadas en este contexto.

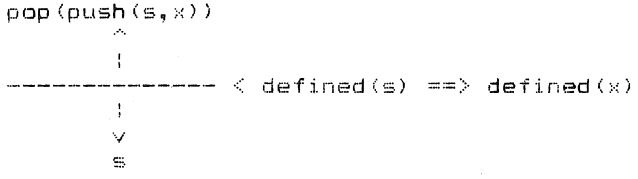
2. Transformaciones por reglas en un nivel aplicativo

(El lenguaje empleado es un subconjunto de CIP-L, el lenguaje aplicativo [BAU85]).

2.1. Reglas de transformación

Reglas generales

Incluyen reglas asociadas a los axiomas de los tipos primitivos bool, char, bit, nat, int, sequ (ver [BAU85]) y reglas originadas en el ambiente de tipos. Por ejemplo, en un ambiente que contiene la definición algebraica de un tipo STACK debería existir el axioma:
pop(push(s,x)) = s para todo s de tipo Stack y todo x de tipo nat
Este axioma se corresponde a la regla



(El predicado DEFINED caracteriza expresiones que no conducen a evaluaciones indefinidas.)

Regla de unfolding

Consiste en efectuar el reemplazo textual de una llamada a una rutina por su cuerpo bajo sustitución textual de los parámetros formales por los objetos concretos (argumentos).

Regla de folding

Consiste en el reemplazo textual de una expresión por una invocación a una rutina cuyo cuerpo es igual a la expresión después de la sustitución de los parámetros.

Regla de instanciación

Si en una invocación a una rutina un argumento es constante la operación de unfolding causa que este objeto constante aparezca en el cuerpo de la rutina originando todas las simplificaciones posibles, a esto se lo denomina **instanciación parcial** .

Regla de definición

Introduce una nueva rutina cuyo cuerpo no puede obtenerse mediante la instanciación parcial de una rutina existente.

Regla de abstracción

Sea una rutina g con diferentes subexpresiones Ei(a) de modos mi que están contenidas en una expresión G de modo w, vale para g la siguiente transformación:

```
funct g=(v  a)w: G(a,E1(a),..En(a))
                 ^
                 |
-----------------------------------
                 |
                 v
funct g =(v  a)w:G(a,S1(a),....Sn(a))
funct S1=(v  a)m1:E1(a);
.....................................
funct Sn=(v  a)mn:En(a).
```

La transformación es válida en ambas direcciones. La equivalencia está basada en el principio de sustitución([BAU82]).

Regla de embedding

Una rutina g con diferentes subexpresiones Ei(a) de modos mi(i=1,..n) que están contenidas en una expresión F de modo w, puede transformarse , bajo ciertas condiciones, mediante la siguiente transformación:

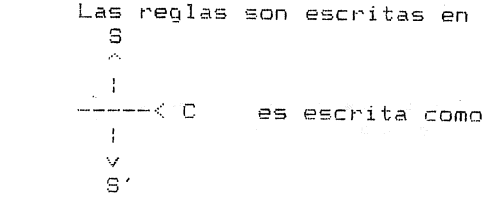
```
funct g=(v a)u:G(a,F(a,E1(a),....,En(a))
                 ^
                 |
---------------------------------------
                 |
                 v
funct g=(v a)u:G(a,f(a,E1(a),....En(a))

funct f=(v a,m1 s1,m2 s2,....,mn sn)w:F(a,s1,....,sn)
```

La nueva rutina f tiene un parámetro adicional para cada una de las subexpresiones a ser tratadas. El cuerpo de f es obtenido reemplazando cada ocurrencia Ei(a) en F por el correspondiente parámetro si. Esta forma general de la regla describe un embedding incompleto. Si la expresión encerrada es el cuerpo entero de la rutina original, el embedding es completo.

Esta transformación es válida si la terminación de g implica la terminación del sistema (g,f). Es suficiente pero no necesario que g no sea recursiva, es decir no sea invocada en alguna de las expresiones Ei(a). Además, todas las llamadas de F deben estar definidas. La transformación es válida en el sentido opuesto si cada Ei(a) está determinada([BAU82]).

Reglas asociadas a expresiones lógicas

Las reglas son escritas en notación lineal, así



El predicado DETERMINATE caracteriza expresiones determinadas. El predicado DEFINED caracteriza expresiones que no conducen a evaluaciones indefinidas.

1. (P cand Q) cand P <---> P cand Q >--< determinate(P)

2. (P and Q) cand R<-->P cand (Q cand R) >-<(wp(P)=F)==>defined(Q)

3. (P cand Q) cand R <--> P cand (Q cand R)

4. (P and Q) cand R <--> P and (Q cand R)
>--< (wp(P) =F and wp(Q) = T) ==> defined(R)

5. P and (Q cand R) <--> (P and Q) cand R
>--< (wp(P)= F and wp(Q)= T) ==> defined(R)

6. (P cand Q) and R<-->P cand (Q and R)>-<(wp(P)=F)==>defined(R)

7. (P cand Q) cand R <--> P cand (Q cand R)

8. (P and Q) and (R and S) <--> (P and R) and (Q and S)

9. P cand Q <--> Q and P >--< (wp(P)= F) ==> defined(Q)

10. P cand (Q and R) <--> Q and (P cand R)>--<wp(P)==> defined(Q)

11. P and Q <--> P cand Q >--<(wp(P)= F) ==> defined(Q)

12. P ==> Q <--> P ==> Q >--< (wp(P)= F) ==> defined(Q)

13. P OR Q <--> P cor Q >--< (wp(P) = T) ==> defined(Q)

14. P cand Q <--> P and Q >--< (wp(P)=F) ==> defined(Q)

15. P <==> Q <--> (P ==> Q) and (Q ==> P)

16. P and Q <--> Q and P

17. P <==> Q <--> Q <==> P

18. P or Q <--> Q or P

19. P cand Q <--> Q cand P
>--<(wp(P)=F)==>defined(Q)) and ((wp(Q)=F)==>defined(P))

20. P cand (Q cand R) <--> (Q cand (P cand R)
>--<((wp(P)=F)==>defined(Q)) and ((wp(Q)=F)==>defined(P))

21. P and P <--> P

22. P or P <--> P

23. P cor P <--> P

24. P cand (Q and R) <--> (P cand Q) and (P cand R)

25. P cand (Q or R) <--> (P cand Q) or (P cand R)
>--< determinate(P)

26. P cor (Q cand R) <--> (P cor Q) cand (P cor R)
>--< determinate(P)

27. (P and Q) and R <--> (P and R) and (Q and R)

28. (P and Q) and R <--> (P and (Q and R))

29.P and false <--> false >--< defined(P)

30.P and true <--> P

31.true and P <--> P

32.P <==> P >--< defined(P) and determinate(P)

33.P <==> false <--> not P

34. false and P <--> false >--< defined(P)

35.false or P <--> P

36.false cand P <--> false

37. false ===> P <--> true

38.not false <--> true

39. P or false <--> P

40.P cand false <--> false >--< defined(P)

41.P cand true <--> P

42.true and P <--> P

43.true cand P <---> P

44.(P<==>Q) cand R <--> (P <==> Q) cand S

45. P and not P = false

Reglas del condicional

1.

```
 if A then if B then E1
                else E2
           fi
      else
           if B then E3
                else E4
           fi
  fi
```

```
 if B then if A then E1
                else E3
           fi
      else if A then E2
                else E4
           fi
  fi
```

2.

```
 if A and B              then E1
 #  A and (not B)        then E2
 # (not A) and C         then E3
 # (not A) and (not C)   then E4
 fi
```

```
 true)          ; (A,B,C determinada
                v (A,B,C definidas

 if A  then  if  B  then  E1
                          else  E2
             fi
        else  if  C  then  E3
                          else  E4
             fi
   fi
```

3.

```
 F(if C then B1 else B2 fi)
```

```
 if C then F(B1)
      else F(B2) fi
```

4.

```
 F( if C then  E  fi)
```

```
   if C  then  F(E) fi
```

5.

```
 F(if A then B else C fi, D)
```

```
 if A then F(B,D)
      else F(C,D) fi
```

6.

```
  F( if C  then  E fi,P2,P3)
```

```
  if C then F(E,P2,P3) fi
```

7.

```
 F(D,if A then B else C fi)
```

```
 if A then F(D,B)
      else F(D,C) fi
```

8.

```
 if  true  then  E   fi
```

```
 E
```

9.

```
 if false then E else F fi
              ^
              |
 -------------------------
              |
              v
              F
```

10.

```
      if true  then E else F fi
               ^
               |
 -------------------------
               |
               v
               E
```

11.

```
 if C then E else E fi
              ^
              |
 ----------------------
              | ( defined(C)
              v
              E
```

12.

```
      if A  then E1 else E2 fi
                ^
                |
 -------------------------
                |
                v
 if not A then E2 else E1 fi
```

13.

```
if A then if B then E1
                else E2
             fi
     else E3
 fi
             ^
             |
 -------------------------
             |(wp(A)==>B   |
             |
             v
if A then E1 else E3 fi
```

14.

```
if  A  then  E1
       else  if  B  then  E2
                          else  E3
                  fi
 fi
                ^
                |
 ----------------------------
             |((wp(A)=F)==>
                |       not B
                v
   if A  then  E1  else E3 fi
```

15.

```
if C then if D then E1
                else E2
           fi
fi
             ^
             |
------------------------
             |
             v
if D  then if C then E1
           fi
      else if C then E2
           fi
 fi
```

16.

```
 if A  then G(F(E1,E3))
       else G(F(E2,E3))
 fi
             ^
             |
 ------------------------
             |
             v
   G(F(if A then E1 else E2 fi,E3))
```

17.

```
F(D,if A then B else C fi,E)
              ^
              |
------------------------------
              |
              v
 if A then  F(D,B,E)
      else  F(D,C,E)
 fi
```

18.

```
if A  then x
   #  B  then y
   else  x
   fi
        ^
        |
        -------------  { defined(A)
        | and wp(B)==> not A
        v
   if B then y else x fi
```

19.

```
 if C1 then   E1
 #  C2 then   E2
 ...................
 #  Ci   then Ei
 #  Ci+1 then Ei+1
 ...................
 fi

         ^
         |
 ---------------------
         | {valor(Ei)=F
         v

 if C1   then E1
 #  C2   then E2
 ...................
 #  Ci+1 then Ei+1
 ...................
 fi
```

20.

```
            if C1  then  E1
            #  C2  then  E2
            ..................
            #  Ci  then  Ei
            ..................
            #  Cj  then  Ei

            fi
                  ^
                  |
            ---------------------
                  |

                  v
            if C1      then  E1
            #  C2      then  E2
            ....................
            #  Ci OR Cj then Ei
            ....................
```

21.

```
          if A then E1
               else if B then E2
                         else if C then E3
                              #  D then E4
                              else  E5
                              fi
                    fi
          fi
                         ^
                         |
          ---------------------------------------{ A,B determinada
                         |
                         v
          if A  then E1
                else if B then E2
                         else if not A and not B and C then E3
                              #  not A and not B and D then E4
                              else E5
                              fi
                    fi
          fi
```

2.2. Estrategias para aplicar reglas de transformación

Estrategia de composición

Si una expresión f(g(x)) de modo w ocurre en un sistema de rutinas es posible definir una nueva rutina h:

funct h=(m x)w:f(g(x)).

y reemplazar las ocurrencias de f(g(x)) por h(x) en el sistema.

Estrategia de generalización

Si una expresión E(x,c) (x variable libre y c constante) de modo w ocurre en una rutina de un sistema es posible definir una nueva rutina:

funct f=(mx x,my y)w: E(x,y).

y reemplazar las ocurrencias de E(x,c) en el sistema por f(x,c).

Estrategia tupling

Si dos funciones f(x) y g(x) requieren la evaluación de la función h(x), es conveniente definir una nueva función

p(x)=<f(x),g(x)>

y usar la regla de abstracción en p(x) para calcular h(x) una sóla vez.

Estrategia de embedding funcional

Dada una rutina A es posible mediante la estrategia de generalización, definir una nueva rutina B tal que una instanciación parcial de B inmersa en otra rutina C defina una equivalencia de C con A.

3. Ejemplos

Se presentan a continuación ejemplos que pretenden mostrar el uso combinado de estrategias para aplicar reglas de transformación El Ejemplo 3 muestra que la aplicación del método planteado permite obtener versiones aplicativas eficientes en tiempo partiendo de versiones de complejidad temporal exponencial.

Ejemplo 1

Sea

```
1.   funct sqsum=( nat n) nat:
     if n=0 then  0
     else sq(n) + sqsum(n-1)
     fi.
```

2.Regla de definición- Estrategia de generalización

```
     funct gsqsum=( nat n, nat nsq)nat:
     if n=0  then  0
             else  nsq + gsqsum(n-1,nsq-2*n+1)
     fi.
```

3. Estrategia de embedding funcional

i. es equivalente al sistema:

```
     funct sqsum=(nat n)nat: gsqsum(n,sq(n))
     where
     funct gsqsum=(nat n, nat nsq)nat:
     if n=0  then  0
             else  nsq + gsqsum(n-1,nsq-2*n+1)
     fi.
```

Ejemplo 2

Derivación de un algoritmo que calcule la suma de los cuadrados de una secuencia de reales

Sea la definición de un tipo secuencia:

```
type  SEQU  = (sort m) sequ,  empty,  isempy,  make ,&  ,  top  ,
rest ,append ,bottom, upper ,stock ,length ,select ,change:
sort sequ,
sequ empty,
funct (sequ) bool isempty,
funct (m) sequ make,
funct (sequ,sequ) sequ ,&,
funct (sequ s:not isempty(s)) m top,
funct (sequ s:not isempty(s)) sequ rest,
funct (sequ, m) sequ append,
funct (sequ s: not isempty(s)) sequ upper,
funct (sequ, m) sequ stock,
funct (sequ) nat length,
funct (sequ s, nat i:1<=i and i<=length(s)) m change;
funct (sequ s: not isempty(s)) m bottom;
funct (sequ s,nat i:1<=i and i<=length(s)) m select;
laws m x, sequ r, sequ s, sequ t:
1.      (r & (s & t)) = ((r & s) & t)
2.      (s & empty) = s, (empty & s)= s,
3.      isempty(empty) = true,
4.      isempty(make(x)) = false,
5.      isempty(s & t) = (isempty(s) and isempty(t)),
6.      append(s,x) =(make(x) & s),
```

```
7.      top(append(s,x)) = x,
8.      rest(append(s,x)) = s,
9.      stock(s,x) = (s & make(x)),
10.     bottom (stock(s,x)) = x,
11.     upper(stock(s,x)) = s
12.     length(empty) = 0
13.     length(make(x)) = 1,
14.     length(s & t) = length(s) + length(t)
15.     (i <= length(s)) ==> select(s,i) = top(s),
16.    (1 < i and i<=length(s)) ==> select(s,i) =select(rest(s),i-1)
17.    (i <= length(s)) ==> change(s,1,x)=append(rest(s),x),
18.    (1 < i and i<=length(s) ) ==>
       change(s,i,x)=append(change(rest(s),i-1,x),top(s))
end of type.
```

Sean

1.

```
funct cuadrado=(sequ real L)sequ real:
if L=@  then   @
        else top(L)*top(L) & cuadrado(rest(L))
fi.
```

2.

```
funct suma=(sequ real L)real:
if  L=@    then  0
           else  top(L) + suma(rest(L))
fi.
```

3. Definición- Estrategia de composición

```
funct sumcuad=(sequ real L)real:
suma(cuadrado(L)).
```

4. Unfolding en 3. con 2.

```
funct sumcuad=(sequ real L)real:
if  L=@    then   0
           else   top(cuadrado(L)) + suma(rest(cuadrado(L)))
fi.
```

5. Unfolding en 4. con 1.

```
funct sumcuad=(sequ real L)real:
if  L=@  then  0
         else
        5'. top(if L=@  then  @
                else top(L)*top(L)&cuadrado(rest(L))
                fi)  +
        5''. suma(rest(if L=@  then  @
                       else  top(L)*top(L)&cuadrado(rest(L))
                       fi)
fi.
```

6.Regla del condicional 16 en 5'.y 5''. :

```
funct sumcuad=(sequ real L)real:
if L=@  then 0
        else  (if  L=@  then top(@)
                        else  top(top(L)*top(L)&cuadrado(rest(L)))
               fi)  +
        (if  L=@  then suma(rest(@))
                  else suma(rest(top(L)*top(L)&cuadrado(rest(L))))
         fi)
 fi.
```

7. Por reglas generales, axiomas de sequ 7 y 8

```
funct sumcuad=(sequ real L)real:
if L=@  then  0
        else
            (if L=@  then top(@)
                     else top(L)*top(L)
             fi) +
            (if L=@  then suma(rest(@))
                     else suma(cuadrado(rest(L))
             fi)
fi.
```

8. Regla del condicional 16

```
funct sumcuad=(sequ real L)real:
if  L=@  then  if  L=@  then  0
                       else top(@) +
                       (if L=@  then suma(rest(@))
                                else  suma(cuadrado(rest(L))
                        fi)
            fi
       else  if L=@  then  0
                     else top(L)*top(L) +
                          (if L=@  then suma(rest(@))
                                   else suma(cuadrado(rest(L))
                           fi)
                fi
 fi.
```

9. Por regla del condicional 2:

```
funct sumcuad=(sequ real L)real:
if L=@ and L=@       then  0
#  L=@ and not(L=@)  then  top(@) + (if L=@  then  suma(rest(@))
                                     else suma(cuadrado(rest(L)) fi)
#  not(L=@) and L=@ then  0
#  not(L=@) and not(L=@) then  top(L) *top(L) +
                               (if L=@ then suma(rest(@))
                                       else suma(cuadrado(rest(L)) fi)
fi.
```

10.
L = @ and L=@ --> L = @ por regla 21 de expresiones lógicas,

L = @ and not(L = @) --> false por regla 45 de expresiones lógicas,

not(L=@) and L = @ ---> L=@ and not(L=@) --> false por regla 16 y 45 de expresiones lógicas,

not(L=@) and not(L=@) ---> not(L=@) por regla 21. de expresiones lógicas

```
funct sumcuad=(sequ real L)real:
if  L=@     then  0
#  false    then  top(@) +
                  (if L=@  then suma(rest(@))
                           else suma(cuadrado(rest(L))
                   fi)
# false     then  0
# not(L=@)  then  top(L)*top(L) + (if L=@ then suma(rest(@))
                                   else suma(cuadrado(rest(L))
                                   fi)
fi.
```

11. Por regla 19. del condicional

```
funct sumcuad=(sequ real L)real:
if  L=@     then  0
# not(L=@)  then  top(L)*top(L) + (if L=@ then suma(rest(@))
                                   else suma(cuadrado(rest(L))
                                   fi)
fi.
```

12. Regla 7. del condicional

```
funct sumcuad=(sequ real L)real:
if  L=@     then  0
# not(L=@)  then  if  L=@ then top(L)*top(L) + suma(rest(@))
                  else top(L)*top(L) + suma(cuadrado(rest(L))
                  fi
fi.
```

13. Regla 14 del condicional

```
funct sumcuad=(sequ real L)real:
if L=@      then  0
# not(L=@)  then  top(L)*top(L) + suma(cuadrado(rest(L))
fi.
```

14. Folding de 13. con 3.

```
funct sumcuad=(sequ real L)real:
if  L=@      then  0
# not (L=@) then  top(L)*top(L) + sumcuad(rest(L))
fi.
```

Esta última versión evita generar una secuencia intermedia de cuadrados como la versión inicial.

Ejemplo 3

El problema de las curvas de Hilbert

Sean
movimentos=(N,S,E,O) un conjunto de posibles movimientos (hacia el norte, hacia el sur, hacia el este y oeste respectivamente)

```
mode movi=atomic (N,S,E,O)
mode secuencia=(sequ movi w,sequ movi x,sequ movi y,sequ movi z)
```

y el siguiente sistema de rutinas

1.
```
funct Hilbert(nat n)sequ movi:
A(n).
```

2.
```
funct A=(nat n) sequ movi:
if  n=0  then @
         else D(n-1) & O & A(n-1) & S & A(n-1) & E & B(n-1)
fi.
```

3.
```
funct B=(nat n) sequ movi:
if  n=0  then @
         else C(n-1) & N & B(n-1) & E & B(n-1) & S & A(n-1)
fi.
```

4.
```
funct C=(nat n) sequ movi:
if  n=0  then @
         else B(n-1) & E & C(n-1) & N & C(n-1) & O & D(n-1)
fi.
```

5.
```
funct D=(nat n) sequ movi:
if  n=0  then @
         else A(n-1) & S & D(n-1) & O & D(n-1) & N & C(n-1)
fi.
```

Las funciones A,B,C,D muestran claramente evaluaciones repetidas. Esto sugiere aplicar la estrategia tupling. A fin de encontrar una tupla EUREKA se construye el DAG de evaluación simbólica surgiendo la siguiente DEFINICION EUREKA- ESTRATEGIA TUPLING

6.
```
funct T=(nat n)secuencia:
if n= 0  then  <@,@,@,@>
else  < A(n),B(n),C(n),D(n) >
fi.
```

7. Unfolding de 6. con 2.,3.,4.,5.
```
funct T=(nat n)secuencia:
if  n=0  then <@,@,@,@>
         else <D(n-1) & 0 & A(n-1) & S & A(n-1) & E & B(n-1),
               C(n-1) & N & B(n-1) & E & B(n-1) & S & A(n-1),
               B(n-1) & E & C(n-1) & N & C(n-1) & 0 & D(n-1),
               A(n-1) & S & D(n-1) & 0 & D(n-1) & N & C(n-1)>
fi.
```

8. Por abstracción de 7. con 6.
```
funct T=(nat n)secuencia:
if  n=0   then   <@,@,@,@>
          else   f1(T(n-1))
fi where
funct f1=(secuencia u)secuencia:
<u.z & 0 & u.w & S & u.w & E & u.x,
 u.y & N & u.x & E & u.x & S & u.w,
 u.x & E & u.y & N & u.y & 0 & u.z,
 u.w & S & u.z & 0 & u.z & N & u.y>.
```

9. De 1. y 6.
```
funct Hilbert=(nat n)sequ movi:
T(n).w.
```

10. Abstracción en 9. con 6.
```
funct Hilbert=(nat n)sequ movi:
T(n).w where
funct T=(nat k)secuencia:
if  k=0  then  <@,@,@,@>
         else  f1(T(k-1))
fi. where
funct f1=(secuencia u)secuencia:
<u.z & 0 & u.w & S & u.w & E & u.x,
 u.y & N & u.x & E & u.x & S & u.w,
 u.x & E & u.y & N & u.y & 0 & u.z,
 u.w & S & u.z & 0 & u.z & N & u.y>
```

La complejidad temporal de Hilbert es O(n), considerando constantes a las operaciones asociadas a la estructura computacional primitiva sequ.

4. Conclusiones

Los ejemplos derivados muestran que programas eficientes pueden ser derivados en una forma simple, transparente y correcta mediante transformaciones. Por ejemplo, se conocen soluciones recursivas exponenciales en tiempo e intrincadas versiones iterativas sin prueba de correctitud para el problema de las curvas de Hilbert [SPA71]. La versión aplicativa derivada en este trabajo puede transformarse directamente por esquemas en una versión iterativa eficiente de complejidad temporal lineal.

A fin de profundizar el análisis de transformaciones en un nivel aplicativo se requiere de un sistema implementado que soporte la metodología planteada. Se continúa en esta línea de trabajo.

5. Referencias bibliográficas

[BAU82] "Algorithmic Language and Program Development" Bauer, F., Wossner, H. Springer Verlag, 1982.

[BAU85] "The Munich Project CIP, vol I : The Wide Spectrum Language CIP-L". Bauer, F. Lecture Notes in Computer Science 183, Springer, Berlin, 1985.

[BAU87] "The Munich Project CIP, vol II : The Program Transformation System CIP-S" Bauer, F. Lecture Notes in Computer Science 292, Springer, Berlin, 1987.

[BAU89] "Formal program construction by Transformations--Computer-Aided, Intuition Guided Programming", Bauer, F.L., Moller, B., Partsch, H., Pepper, P. IEEE Transactions On Software Engineering, Vol 15. NO. 2, Febraury 1989.

[BOI90] "USTOPIA Requirements Thoughts on a User-friendly System for Transformation of Programs In Abstracto". Boiten, E.A., van der Brand, M.G.J., van Diepen, N.W.F.,Koster, C.H.A. Technical Report NO. 90-12,June 1990. University of Niejmegen. Department of Informatics.

[BUR77] "Program Development by Transformations : an overview". Burstall R. Feather, M. Les Fondements de la Programation(eds. M.Amiochaby e D.Neel) Touluse, Dec 1977

[BUR77-1]"Recursive programs:proof, transformation and synthesis" Burstall,R. Rivista di Informatica, vol VII, suplemento al, n. 1, gennaio-marzo 1977.

[BUR77-2]" A Transformation System for Developing Recursive Programs". Burstall, R.;Darlington, J. ACM, vol 24,N 1, January 1977.

[DAR74] " A System which automatically improves programs." Proc. Sixth Annual ACM Symp. Theory of Comtg.,1974, pp 13-26.

[DIJ76] " A Discipline of Programming". Dijkstra, E. W. Englewood Cliffs, N.J. :Prentice Hall 1976.

[MAN75] "Knowledge and reasoning in program synthesis. Manna, Z. and Waldinger, R. Artif. Intel. J.6.2(1975), 175-208.

[PAR83] "Program Transformation systems".Partsch, H.,Steinbruggen, R. Computing Surveys, Vol 15, No. 3,September 1983.

[PAR86] " Program transformations expressed by algebraic type manipulations". Technique et Science Informatiques, vol 5, No. 3. 1986.

[PAR88] " A formal derivation of Boyer and Moore's pattern matching algorithm". Partsch, H., Stomp, F. Internal Report no. 88-12, 1988 department of Informatics. Faculty of Science. University of Nijmegen.

[PAR89] " From informal requirements to a running program: A case study in algebraic specification and transformational programming". Partsch, H.Science of Computer Programming 11(1988/1989) 263-297. North-Holland.

[PAR90] " Specification and Transformation of Programs. A Formal Approach to Software Development". Springer-Verlag. 1990.

[PET82]" Deriving very Efficient Algorithms for Evaluation Linear Recurrence Relations Using the Program Transformation Technique". Pettorossi,A.Burstall,R.Acta Informatica 18,181-206(1982).

[PET84]"A Powerful Strategy for Deriving Efficient Programs by Transformation". Pettorossi,A. Papers presented at the Symposium 1984 ACM Symposium on Lisp and Functional Programming.pp 273-281.

[SPA71] " Space Filling Curves, or How to Waste time on a Plotter". Softw. Pract. And Exper. Vol 1, No. 4(1971)403-410

[WAL73] "Characterizations of Flowchartable Recursions". Walker,S.A. and Strong, H.R. JCSS Vol. 7 no. 4 (Aug 73) pag 404-447.