Un sistema ampliado de desplegado/plegado para la síntesis de programas funcionales

Miguel Arcas-Guijarro,[*] Cristóbal Pareja-Flores[*] y J. Ángel Velázquez-Iturbide[**]

[*] Depto. Sistemas Informáticos y Programación, Univ. Complutense de Madrid.
Avda. Puerta de Hierro s/n, 28040-Madrid, Spain.
E-mail: arcas@eucmos.sim.ucm.es, cpareja@sip.ucm.es
[**] Escuela Superior de Ciencias Experimentales y Tecnología, Univ. Rey Juan Carlos.
Camino de Humanes 63, 28936-Móstoles, Madrid, Spain.
E-mail: a.velazquez@escet.urjc.es

Resumen

Proponemos una extensión, para la síntesis de funciones, del sistema de transformación de desplegado/plegado. Se amplían tanto el lenguaje de programación funcional como el sistema de transformación. El lenguaje ampliado permite relaciones (es decir, restricciones) que expresan precondiciones y poscondiciones, pero que no pueden usarse para construir funciones ejecutables, sino que su uso está limitado al proceso transformador. El sistema de transformación también se amplía para tratar restricciones e igualdades generales, resultando un sistema de cinco reglas, potente pero sencillo. Hemos estudiado principalmente su aplicación a la síntesis de funciones inversas de otras existentes. El sistema resulta especialmente útil y natural cuando se combina con una forma poderosa de especificar inversas de funciones, llamada patrones generalizados. En el papel ilustramos estas ideas con la aplicación del sistema a varios problemas, destacando la síntesis de una función que reconstruye un árbol binario a partir de sus recorridos en preorden y en inorden.

Palabras claves: Programación funcional, transformación de programas, síntesis de programas, sistema de desplegado/plegado, patrones, restricciones.

1 Introducción

El modelo de transformaciones para el desarrollo de programas [Par90] permite obtener programas correctos por construcción: básicamente, se empieza con una especificación y se aplica una secuencia de reglas que conservan la corrección, hasta que se obtiene un programa satisfactorio. Este modelo de desarrollo de programas es especialmente adecuado para programación declarativa, donde el razonamiento sobre los programas es más sencillo debido a la ausencia de estado (transparencia referencial).

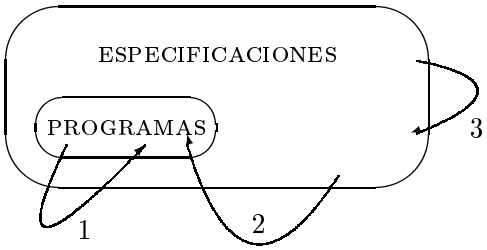
Este sencillo modelo de transformaciones puede resultar más complejo cuando se refina. Por ejemplo, los lenguajes de especificación y transformación pueden ser distintos. También es usual (y más fácil) proceder en varios pasos: primero, sintetizar un programa a partir de una especificación, y después mejorarla mediante transfomaciones sucesivas. Concretamente, hay diferentes actividades que se dan cita en el cálculo de programas no triviales mediante este enfoque:

1 Transformación: Un programa ejecutable se convierte en otro, equivalente pero en general más eficiente.

2 Síntesis: A partir de una especificación no ejecutable, se obtiene un programa ejecutable que la verifica.

3 Verificación de propiedades: El razonamiento sobre programas es un proceso complejo en general, donde a menudo hace falta demostrar propiedades. Este proceso de verificación se puede hacer para dos actividades: la verificación misma de un programa, o la obtención de una propiedad cierta (un teorema) que resulta útil o necesaria para la síntesis o la transformación.

Estas actividades pueden ilustrarse informalmente mediante la siguiente figura, donde se muestra el dominio relativo a cada una de estas actividades:



El sistema de transformaciones más conocido es el de desplegado/plegado [BD77]. Dos de sus principales ventajas son la sencillez y la generalidad. Sin embargo, también cuenta con limitaciones, que nos han llevado a extenderlo. Una primera y trivial modificación del sistema consistiría en adaptarlo a un lenguaje funcional más moderno y complejo (como Haskell) que el lenguaje para el que este sistema fue concebido (Hope); por ejemplo, debe ser expresable en el mismo la distinción por casos para poder sintetizar funciones con guardas. Una limitación más importante del sistema de desplegado/plegado es su incapacidad para sintetizar funciones y para transformar ciertas clases de algoritmos. Estos aspectos exigen ampliar el lenguaje (además del sistema), dando lugar a un lenguaje de especificación que es un superconjunto del de programación.

Nuestro lenguaje de especificación extiende el lenguaje funcional puro en dos aspectos: la definición de igualdades generales y la inclusión de predicados sobre igualdades (i.e. restricciones). El sistema de desplegado/plegado también se extiende para manejar el nuevo lenguaje de especificación.

Existen algunos trabajos que también proponen extensiones (p.ej. [Sch81, Chi90, PV97a, PV97b]) del sistema de desplegado/plegado para evitar los inconvenientes citados, pero son más restrictivos que el nuestro. Scherlis adopta [Sch80], al igual que nosotros, el enfoque de la síntesis mediante transformaciones. Se diferencia del nuestro en que las restricciones son asertos (*calificadores*) que no sólo afectan a ecuaciones, sino a expresiones cualesquiera. Chin, siguiendo también este enfoque [Chi90], permite expresar precondiciones, lo que hace posible ciertas transformaciones (como la factorización). Estos trabajos, a diferencia del nuestro, no abordan la síntesis, ya que parten de especificaciones ejecutables (programas).

Existe una corriente diferente, basada en la demostración constructiva de especificaciones [MW92, PW93]). Una especificación se expresa mediante una sentencia de lógica de predicados, donde los predicados permiten expresar relaciones cualesquiera. La demostración de una especificación produce como producto lateral un programa que la satisface.

El contenido del artículo es el siguiente. En los apartados 2 y 3, describimos el lenguaje de especificación y el sistema de transformación, respectivamente. El apartado

4 trata una importante aplicación de la síntesis: la de la función inversa de una función dada. En el apartado 5, introducimos un nuevo concepto, el de patrón generalizado, que es la base de una clase de síntesis particular, que surge en muchos problemas. Finalmente, resumimos nuestras conclusiones e incluimos un apéndice con el problema no trivial de reconstruir un árbol a partir de dos de sus recorridos.

2 El lenguaje de especificación

La primera extensión del lenguaje consiste en permitir el uso de igualdades sin restricción alguna sobre su miembro izquierdo. Como en las ecuaciones funcionales, las variables libres de ambos miembros están cuantificadas universalmente, y son de los tipos apropiados. Esta característica es necesaria tanto para demostrar propiedades como para sintetizar funciones a partir de igualdades generales. Por ejemplo, la igualdad

```
length (list1 ++ list2) = length list1 + length list2
```

es una propiedad útil que significa que

```
∀ list1, list2 ::  [α] .
   length (list1 ++ list2) = length list1 + length list2
```

Una segunda deficiencia del sistema de despliegue/pliege es la imposibilidad de tratar con predicados que establecen asertos sobre igualdades, tales como pre- y poscondiciones. En nuestra extensión, una igualdad `E1 = E2` puede ser válida bajo ciertas condiciones `Ps`, lo que se expresa como sigue:[1]

```
E1 = E2 st Ps
```

En general, llamamos restricciones a esos predicados, o a secuencias de ellos. Los predicados de una restricción se separan con punto y coma, que indica su conjunción. En términos lógicos, esto significa la siguiente clausura, mediante cuantificación universal:

```
∀(Ps ⇒ E1 = E2)
```

Naturalmente, consideramos que una igualdad sin restricción alguna está trivialmente calificada por la restricción `True`.

Tal notación nos permite especificar funciones en términos de sus pre- y poscondiciones. Por ejemplo, una función `f` especificada mediante

```
∀ x::α .  Pre[x] ⇒ ∀ y::β .  Post[x,y] ⇒ f x = y
```

puede describirse como sigue en nuestra notación

```
f x = y st Pre[x]; Post[x,y]
```

omitiendo los cuantificadores universales y los tipos de datos. Nótese que ambas, pre- y poscondiciones, están al mismo nivel de una especificación, aunque se distinguen por las variables a las que hacen referencia; es decir, la precondición `Pre` es la parte de la restricción que sólo menciona variables de entrada, mientras que `Post` relaciona variables de entrada y de salida.

Como ejemplo, una función que calcula el cociente y el resto de dos enteros, cuya especificación lógica es

```
∀ dnd,dsor,quot,rem ::  Int .
   dnd>=0; dsor>0; rem>=0; rem<dsor; dnd=dsor*quot+rem
   ⇒ divMod (dnd,dsor) = (quot,rem)
```

se expresa en nuestra notación así:

[1] `st` viene de "such that" (tal que).

```
divMod (dnd,dsor) = (quot,rem) st
   dnd>=0; dsor>0; rem>=0; rem<dsor; dnd=dsor*quot+rem
```

donde los dos primeros predicados en la restricción forman la precondición y el resto la poscondición.

3 El sistema de síntesis

Usamos el término síntesis de funciones para referirnos al proceso en el cual se transforman igualdades generales en ecuaciones. Para que la síntesis sea posible, admitimos el uso de restricciones expresadas como cláusulas `st`, pero su eliminación de las igualdades finales debe producir definiciones de functiones. Por ejemplo, la síntesis para el esquema de especificación dado en el apartado anterior consiste en producir un patrón (y probablemente guardas) en el miembro izquierdo de la igualdad, y en eliminar `y` de la relación `Post[x,y]`, de forma que se obtenga una expresión functionalmente computable.

Como ejemplo, podemos transformar la especificación `divMod` del apartado anterior, llegando a la siguiente definición de función, después de suprimir la pre- y poscondición:

```
divMod (dnd,dsor) | dnd< dsor = (0, dnd)
                  | dnd>=dsor = (u+1,v) where (u,v) = divMod (dnd-dsor,dsor)
```

Las expresiones de nuestro lenguaje ampliado serán transformadas con el sistema de desplegado/plegado [BD77], extendido para manejar especificaciones. Primero, necesitamos reglas de transformación capaces de operar sobre igualdades generales en vez de sólo sobre definiciones de función. Además, necesitamos reglas de transformación para manejar restricciones. El sistema de desplegado/plegado resultante consta de las siguientes reglas:

Definición. Se introduce una nueva igualdad `lhs = rhs`, ya sea con un ámbito global o como un nuevo predicado en una restricción. En el primer caso, puede estar condicionada por una restricción.

Una igualdad nueva puede introducir un nuevo objeto o una relación entre otros ya existentes. En el primer caso, la regla es idéntica a la del desplegado/plegado; en el segundo, se trata de una propiedad que sólo puede considerarse correcta cuando se demuestra dentro del sistema. Una definición de una propiedad sin demostrar puede usarse durante el proceso de transformación y síntesis, pero la corrección del programa resultante con respecto a la especificación está condicionada por la corrección de la propiedad:

$$\frac{\text{Propiedades no demostradas}}{\text{Especificación} \sqsupseteq \text{Programa}}$$

Instanciación. Se introduce una igualdad `lhs = rhs st P` que es un caso particular de otra preexistente, ya sea sustituyendo algunas variables libres por expresiones funcionales cualesquiera, o bien añadiendo algún predicado a las restricciones.

Esta regla es más general que la correspondiente del sistema desplegado/plegado, donde sólo se instancia con patrones.

Implicación. Dadas dos ecuaciones `E1 st P` y `E2 st Q;P';S`, donde `P'` es un caso particular de `P`, podemos transformar la última en `E2 st Q;P';S;E1'`, siendo `E1'` el ejemplar correspondiente de `E1`.

En realidad, esta regla constituye un caso particular de la anterior, en que la restricción nueva (`E1'`) es cierta en las restricciones (`Q;P';S`) a las que se añade, y por tanto no restringe en nada la ecuación inicial.

Sustitución. Dada una igualdad `lhs = rhs`, se sustituye una subexpresión `e` que es un ejemplar de un miembro de esa igualdad por la expresión correspondiente del otro miembro de la igualdad.

Esta regla sólo puede aplicarse cuando la subexpresión está en el ámbito de la igualdad, esto es, cuando la igualdad es global o se trata de una restricción local que afecta a la subexpresión. Si la igualdad `lhs = rhs` está afectada por una restricción `P`, la subexpresión reemplazada `e` también debe estar en el ámbito del ejemplar correspondiente de `P`.

Obsérvese que la regla de sustitución incluye las reglas de desplegado, plegado y leyes del sistema de desplegado/plegado, ya que todas son casos particulares de la sustitución por igualdad. Sin embargo, frecuentemente nos referiremos a estos casos particulares de la regla de sustitución por sus nombres antiguos.

Conversión. Los predicados de las restricciones se pueden suprimir en ciertas condiciones, apareciendo a cambio construcciones ejecutables (y viceversa):

- Un predicado computable que sólo afecta a variables de entrada, puede convertirse en una guarda, en el miembro izquierdo de la igualdad.
- Un predicado `P[y] = E[x]`, donde `P` es un patrón, se puede transformar en una definición local `where`.

La conversión de especificaciones en programas suele hacerse al final del proceso de síntesis. Aquellos predicados que son necesarios para el programa se convierten en guardas o en cláusulas `where` mediante la regla de conversión, y aquellos otros que son innecesarios simplemente se eliminan; éste es el caso de las pre- y poscondiciones.

Para facilitar la lectura de los ejemplos incluidos en el artículo, conviene hacer dos aclaraciones. En primer lugar, la regla de abstracción del sistema de desplegado/plegado ha desaparecido, ya que puede simularse trivialmente mediante la aplicación de otras reglas. Cuando se emplea esta regla, nosotros la indicamos con su nombre habitual por brevedad. Otra observación es que hemos omitido las restricciones innecesarias por simplicidad.

Ejemplo

Como ejemplo del sistema presentado, sintetizamos seguidamente el algoritmo especificado antes para la función `divMod`:

```
  divMod (dnd,dsor) =  (quot,rem) st
                          dnd>=0; dsor>0; dnd=dsor*quot+rem; 0<=rem; rem<dsor
≡ {- instanciación por casos:  dnd<dsor y dnd>=dsor -}
  divMod (dnd,dsor) =  (quot,rem) st
                          dnd>=0; dsor>0; dnd=dsor*quot+rem; 0<=rem; rem<dsor
                          dnd<dsor
                    ≡ {- leyes aritméticas -}
                       (0,dnd)
  divMod (dnd,dsor) =  (quot,rem) st
                          dnd>=0; dsor>0; dnd=dsor*quot+rem; 0<=rem; rem<dsor
```

```
      dnd>=dsor
≡ {- sustitución (leyes aritméticas) -}
   ((quot-1)+1,rem) st
      dnd-dsor=dsor*(quot-1)+rem; 0<=rem; rem<dsor
      dnd>=dsor
≡ {- abstracción:  (u,v)=(quot-1,rem) -}
   (u+1,v) st dnd-dsor=dsor*u+v; 0<=v; v<dsor; dnd>=dsor
≡ {- implicación (plegar divMod) -}
   (u+1,v) st divMod (dnd-dsor,dsor) = (u,v); dnd>=dsor
≡ {- conversión -}
   (u+1,v) where (u,v)=divMod (dnd-dsor,dsor)
```

Reproducimos el algoritmo final, tras la regla de conversión, de donde surgen las guardas; también se han suprimido el resto de los predicados:

```
divMod (dnd,dsor) | dnd< dsor = (0,dnd)
                  | dnd>=dsor = (u+1,v) where (u,v) = divMod (dnd-dsor,dsor)
```

4 Inversión de funciones

La inversión de una función es una clase particular de síntesis de funciones, donde se parte de la definición de una función `f` como `f x = E[x]`, y se desea derivar otra función $\texttt{f}^{-1}$ especificada como $\texttt{f}^{-1}$ `E[x] = x`. Para llevar a cabo la síntesis de inversas, se puede derivar una *ley de inversión*:

```
   f⁻¹ E[x] = x
≡ {- abstraer E[x] con y -}
   f⁻¹ y = x st y = E[x]
≡ {- sustitución (plegar f) -}
   f⁻¹ y = x st y = f x
```

Una clase particular de inversión es la de funciones parcialmente aplicadas:

```
(f k)⁻¹ y = x st f k x = y
```

Como `f k` depende en general de `k`, la función $\texttt{(f k)}^{-1}$, inversa de `f k`, también depende de `k`. Por tanto, lo que debemos sintetizar es una función `unF` tal que `unF k` = $\texttt{(f k)}^{-1}$. (Obsérvese que `unF` no es en general la inversa de `f`.) Dada una función `f x y` definida como

```
f x y = E[x,y]
```

donde `x` representa los parámetros de la aplicación parcial que deseamos invertir, transformaremos la especificación de su inversa $\texttt{(f x)}^{-1}$:

```
(f x)⁻¹ E[x,y] = y
```

El objetivo de la transformación de $\texttt{(f x)}^{-1}$ es doble: obtener a partir de `E[x,y]` un patrón en el miembro izquierdo de la igualdad, y una expresión computable dependiente de `y` en el miembro derecho.

Ejemplo

Supongamos que tenemos una función que concatena dos listas, conociendo además la longitud de la primera:

```
appLen n (l1,l2) = l1++l2 st n = length l1
```

y se desea sintetizar la inversa (`unAppLen n`), especificada como:

```
unAppLen n (l1++l2) = (l1,l2) st n = length l1
```

Para ello, transformamos esta definición. Primero, formamos los ejemplares correspondientes al primer parámetro de acuerdo con los casos de la definición usual de (`++`)):

- Caso base (`l1 = []`)

```
   unAppLen n ([]++l2) = ([],l2) st n = length []
≡ {- sustitución (desplegar (++) y length) -}
   unAppLen n l2 = ([],l2) st n = 0
≡ {- sustitución (n) -}
   unAppLen 0 l2 = ([],l2)
```

  Así pues,

```
unAppLen 0 l2 = ([],l2)
```

- Caso recursivo (`l1 = x:xs`)

```
   unAppLen n ((x:xs)++l2)     = (x:xs,l2) st n = length (x:xs)
≡ {- sustitución (desplegar (++) y length) -}
   unAppLen n (x:(xs++l2))     = (x:xs,l2) st n = length xs + 1
≡ {- abstracción (k) -}
   unAppLen n (x:(xs++l2))     = (x:xs,l2) st n = k+1; k = length xs
≡ {- sustitución (n) -}
   unAppLen (k+1) (x:(xs++l2)) = (x:xs,l2) st k = length xs
≡ {- abstracción (list) -}
   unAppLen (k+1) (x:list)     = (x:xs,l2) st k = length xs; list = xs++l2
≡ {- sustitución (plegar appLen) -}
   unAppLen (k+1) (x:list)     = (x:xs,l2) st k = length xs
                                                 list = appLen k (xs,l2)
≡ {- ley de inversión para unAppLen, implicación -}
   unAppLen (k+1) (x:list)     = (x:xs,l2) st (xs,l2) = unAppLen k list
```

  que da lugar a lo siguiente:

```
unAppLen (k+1) (x:list) = (x:xs,l2) where (xs,l2) = unAppLen k list
```

El programa resultante es

```
unAppLen 0 list = ([],list)
unAppLen (k+1) (x:list) = (x:xs,l2) where (xs,l2) = unAppLen k list
```

5 Inversión de patrones generalizados

Un patrón representa la aplicación de una función invertible para descomponer un cierto dato en varias componentes. Por ejemplo, la ecuación

```
f (x:xs) = E[x,xs]
```

contiene un patrón que representa la lista resultante de aplicar el constructor `(:)` a `x` y `xs`. Por conveniencia, usamos aquí la forma `Cons(x,y)` sin currificar, como equivalente a `x:y`. Esto es:

```
f (Cons(x,xs)) = E[x,xs]
```

El uso del patrón extrae las componentes originales de la aplicación de `Cons` que forman tal lista. Esto se puede expresar como sigue:

$$\texttt{f list = E[x,xs] where (x,xs) = Cons}^{-1}\ \texttt{list}$$
$$\equiv \texttt{(head list, tail list)}$$

En general, en una ecuación con un patrón `Pat`:

$$\texttt{f (Pat (...,}x_i\texttt{,...))} \ = \texttt{E[...,}x_i\texttt{,...]}$$

los identificadores x_i no son parámetros en realidad, sino sólo los resultados del encaje de patrones, lo cual equivale a computar la función inversa del patrón:

$$\texttt{f x} \ = \ \texttt{E[...,}x_i\texttt{,...]} \ \ \texttt{st Pat (...,}x_i\texttt{,...)} \ = \texttt{x}$$
$$\equiv \ \texttt{E[...,}x_i\texttt{,...]} \ \ \texttt{st (...,}x_i\texttt{,...)} \ = \texttt{Pat}^{-1}\ \texttt{x}$$

Esta simple idea puede generalizarse más si admitimos que los patrones estén formados por cualquier expresión funcional que sea invertible, y no sólo aquéllos definidos mediante las constructoras de un tipo de datos algebraico. Esos *patrones generalizados* nos proporcionan otro modo de especificar funciones. Por ejemplo, sea la siguiente definición de función:

```
f (list1++[0]++list2) = E[list1,list2]
```

donde `f` se aplica a una lista de enteros que contiene exactamente un cero. Ciertamente, `list1++[0]++list2` no es un patrón, pero esta expresión puede ser entendida como tal, por medio de un proceso de inversión:

$$\texttt{f list} \ = \ \texttt{E[list1,list2] st list1++[0]++list2 = list}$$
$$\equiv \ \texttt{E[list1,list2] st (list1,list2) = Pat}^{-1}\ \texttt{list}$$

Obsérvese que esta forma de especificar funciones nos lleva a un caso particular de síntesis: la de la inversa de un patrón. En este ejemplo, de la síntesis de $\texttt{Pat}^{-1}$, se derivará una función equivalente a `span (/=0)` en Haskell [HF92, HJW*95].

Ejemplo

Sea el siguiente patrón generalizado:

```
concatBy a (l1,l2) = l1++[a]++l2
```

Deseamos sintetizar su inversa, cuya especificación es:

```
splitBy a (l1++[a]++l2) = (l1,l2) st not (member a (l1++l2))
```

Siguiendo la definición usual de `(++)`, instanciamos `l1`:

- Caso base (`l1 = []`):

```
   splitBy a ([]++[a]++l2) = ([],l2) st not (member a ([]++l2))
≡ {- sustitución (desplegar (++) tres veces) -}
   splitBy a (a:l2)        = ([],l2) st not (member a l2)
≡ {- abstracción (x) -}
   splitBy a (x:l2)        = ([],l2) st x = a; not (member a l2)
```

- Caso recursivo (`l1 = x:xs`):

```
   splitBy a ((x:xs)++[a]++l2) = (x:xs,l2)
     st not (member a ((x:xs)++l2))
≡ {- sustitución (desplegar (++) tres veces)
   splitBy a (x:(xs++[a]++l2)) = (x:xs,l2)
     st not (member a (x:(xs++l2)))
≡ {- sustitución (desplegar (member)) -}
   splitBy a (x:(xs++[a]++l2)) = (x:xs,l2)
     st not (a=x || member a (xs++l2))
≡ {- sustitución (leyes de not y (||)) -}
   splitBy a (x:(xs++[a]++l2)) = (x:xs,l2)
     st not (a=x); not (member a (xs++l2))
≡ {- abstracción (list) -}
   splitBy a (x:list) = (x:xs,l2)
     st list = xs++[a]++l2; not (a=x); not (member a (xs++l2))
≡ {- sustitución (plegar (concatBy)) -}
   splitBy a (x:list) = (x:xs,l2)
     st list = concatBy a (xs,l2); not (a=x); not (member a (xs++l2))
≡ {- ley de inversión (splitBy), implicación -}
   splitBy a (x:list) = (x:xs,l2)
     st (xs,l2) = (splitBy a list); not (a=x); not (member a (xs++l2))
```

En resumen, el programa sintetizado con una sintaxis estándar es:

```
splitBy a (x:list) | a == x = ([],list)
splitBy a (x:list) | a /= x = (x:xs,l2) where (xs,l2) = splitBy a list
```

6 Conclusiones

El sistema de desplegado/plegado es aplicable a una amplia clase de transformaciones, pero no puede usarse para la síntesis de funciones. En este trabajo hemos propuesto una extensión de dicho sistema que permite la síntesis de funciones.

Para que la síntesis sea posible, el lenguaje funcional se ha extendido para permitir igualdades generales y restricciones. Sin embargo, esas extensiones sólo se usan durante el proceso de transformación, de forma que, cuando se suprimen finalmente, se obtienen programas funcionales estándar. Una técnica particularmente útil de especificación se basa en el uso de patrones generalizados.

También hemos extendido el sistema de transformación de desplegado/plegado para manejar el lenguaje de especificación. El nuevo sistema sólo consta de cinco reglas: tres de ellas engloban a las del sistema de desplegado/plegado, una cuarta permite convertir restricciones en términos funcionales, y una última se usa para razonar con la implicación lógica.

El nuevo sistema de transformación se ha aplicado a la síntesis de funciones inversas. Se ha dado un esquema para esta clase particular de inversión, siendo posible también invertir funciones parcialmente aplicadas. En particular, la inversión de patrones generalizados permite derivar funciones fácilmente especificadas.

Hemos aplicado el lenguaje y el sistema de transformación extendidos a la síntesis de diversos algoritmos. Hemos incluido uno no trivial en el apéndice, pero hemos sintetizado asimismo otros importantes problemas, tales como diversos problemas numéricos o el algoritmo de Huffman [PV98]. Este último es especialmente llamativo, pues en él se dan cita diversos aspectos de la derivación de programas, como son la síntesis, la transformación y la demostración de propiedades, así como la integración de todo el proceso con el uso de TADs.

En relación con este trabajo, existen todavía algunos aspectos pendientes de un estudio más detallado. En primer lugar, las extensiones se han propuesto con la corrección en mente. Por eso es necesario definir su semántica y demostrar la corrección del sistema con respecto a esa semántica. Un segundo trabajo consiste en sistematizar el proceso de síntesis, a ser posible mediante tácticas. Finalmente, un problema interesante es el relativo al manejo de subespecificaciones.

A Inversión del recorrido de un árbol

Este ejemplo muestra el uso de patrones generalizados para la síntesis de un algoritmo no trivial. Consideramos el siguiente tipo `Tree`, para árboles binarios:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

Es sencillo definir dos funciones que recorren un árbol en inorden y preorden respectivamente:

```
travIn, travPr ::  Tree a -> [a]

travIn Empty = []
travIn (Node t1 a t2) = travIn t1 ++ [a] ++ travIn t2

travPr Empty = []
travPr (Node t1 a t2) = [a] ++ travIn t1 ++ travIn t2
```

Y también es fácil definir una función para efectuar ambos recorridos:

```
travInPr ::  Tree a -> ([a],[a])
travInPr t = (travIn t, travPr t)
```

La definición de `travInPr` es ineficiente, porque se atraviesa el árbol dos veces, siendo posible hallar los dos recorridos en una sola pasada. La transformación de `travInPr` usando las funciones `travIn` y `travPr` es fácil y nos lleva a lo siguiente:

```
travInPr Empty = ([],[])
travInPr (Node t1 a t2) = (u1++[a]++u2,[a]++v1++v2)
    where (u1,v1) = travInPr t1
          (u2,v2) = travInPr t2
```

El problema que se plantea [Knu73, Sne91] consiste en escribir la función inversa de `travInPr`, es decir, dados dos recorridos de un árbol, reconstruir éste. Especificamos tal función invirtiendo las dos ecuaciones de `travInPr`:

```
unTravInPr ([],[]) = Empty
unTravInPr (u1++[a]++u2,[a]++v1++v2) = Node t1 a t2
    st t1 = unTravInPr (u1,v1); length u1 = length v1
       t2 = unTravInPr (u2,v2); length u2 = length v2
       not (member a (u1++u2))
```

Las restricciones de esta especificación son necesarias para definir una única función, y provienen de las poscondiciones de la función `travInPr`, que no se han demostrado aquí por brevedad.

El miembro izquierdo de la segunda ecuación no contiene un patrón, pero sí un patrón generalizado, el cual, apropiadamente invertido, puede ser usado en una ecuación válida. Definimos una función `twoTraversals` para construir ese patrón generalizado:

```
twoTraversals (a,u1,u2,v1,v2) = (u1++[a]++u2,[a]++v1++v2)
```

Esta función produce dos listas con la misma longitud, de tal forma que el primer elemento de la segunda aparece una sola vez en la primera lista, las listas `u1` y `v1` tienen la misma longitud, y lo mismo ocurre con las listas `u2` y `v2`. La inversión de esta función produce:

```
unTwoTraversals (u1++[a]++u2,[a]++v1++v2) = (a,u1,u2,v1,v2)
      st length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
```

La síntesis de la función `unTravInPr` puede hacerse en dos pasos:

1. La función original `unTravInPr` se puede expresar en términos de la nueva función `unTwoTraversals`:

```
   unTravInPr (u1++[a]++u2,[a]++v1++v2) = Node t1 a t2
       st t1 = unTravInPr (u1,v1); length u1 = length v1
          t2 = unTravInPr (u2,v2); length u2 = length v2
          not (member a (u1++u2))
≡ {- sustitución (desplegar (++)) -}
   unTravInPr (u1++[a]++u2,a:(v1++v2)) = Node t1 a t2
       st t1 = unTravInPr (u1,v1); length u1 = length v1
          t2 = unTravInPr (u2,v2); length u2 = length v2
          not (member a (u1++u2))
≡ {- abstracción (list1 y tail2) -}
   unTravInPr (list1,a:tail2) = Node t1 a t2
       st (list1,a:tail2) = (u1++[a]++u2,a:(v1++v2))
          t1 = unTravInPr (u1,v1); length u1 = length v1
          t2 = unTravInPr (u2,v2); length u2 = length v2
          not (member a (u1++u2))
≡ {- sustitución (plegar twoTraversals) -}
   unTravInPr (list1,a:tail2) = Node t1 a t2
```

```
      st (list1, a:tail2) = twoTraversals (a,u1,u2,v1,v2)
         t1 = unTravInPr (u1,v1); length u1 = length v1
         t2 = unTravInPr (u2,v2); length u2 = length v2
         not (member a (u1++u2))
≡ {- implicación y ley de inversión (unTwoTraversals) -}
  unTravInPr (list1,a:tail2) = Node t1 a t2
      st (a,u1,u2,v1,v2) = unTwoTraversals (list1,a:tail2)
         t1 = unTravInPr (u1,v1); length u1 = length v1
         t2 = unTravInPr (u2,v2); length u2 = length v2
         not (member a (u1++u2))
```

2. Se sintetiza la función auxiliar `unTwoTraversals`:

```
  unTwoTraversals (u1++[a]++u2,[a]++v1++v2) = (a,u1,u2,v1,v2)
      st length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
≡ {- sustitución (leyes de (:)  y (++)), abstracción (list1 y list2) -}
  unTwoTraversals (list1, a:list2) = (a,u1,u2,v1,v2)
      st list1 = u1++[a]++u2
         list2 = v1++v2
         length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
≡ {- sustitución (plegar appLen y concatBy) -}
  unTwoTraversals (list1,a:list2) = (a,u1,u2,v1,v2)
      st list1 = concatBy a (u1,u2)
         list2 = appLen (length v1) (v1,v2)
         length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
≡ {- sustitución (length v1)-}
  unTwoTraversals (list1,a:list2) = (a,u1,u2,v1,v2)
      st list1 = concatBy a (u1,u2)
         list2 = appLen (length u1) (v1,v2)
         length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
≡ {- implicación, ley de inversión (splitBy,unAppLen) -}
  unTwoTraversals (list1,a:list2) = (a,u1,u2,v1,v2)
      st (u1,u2) = splitBy a list1
         (v1,v2) = unAppLen (length u1) list2
         length u1 = length v1
         length u2 = length v2
         not (member a (u1++u2))
```

El programa resultante es:

```
unTravInPr ([],[]) = Empty
unTravInPr (list1,a:tail2) = Node t1 a t2
   where (a,u1,u2,v1,v2) = unTwoTraversals (list1,a:tail2)
```

```
            t1 = unTravInPr (u1,v1)
            t2 = unTravInPr (u2,v2)
unTwoTraversals (list1, a:list2) = (a,u1,u2,v1,v2)
    where (u1,u2) = splitBy a list1
            (v1,v2) = unAppLen (length u1) list2
```

References

[DFP87] J. Darlington, A. J. Field y H. Pull. The unification of functional and logic languages. In G. Linstrom and D. de Groot (eds.). *Logic Programming: Functions, Relations and Equations*. Prentice-Hall, 1987.

[BD77] R.M. Burstall y J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), 1977.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College of Science, Technology and Medicine, Londres, 1990.

[HF92] P. Hudak y J. H. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Notices*, 27(5), Mayo 1992.

[HJW*95] P. Hudak, S. P. Jones, P. Wadler, J. Fairbairn, B. Boutel, J. Fasel, K. Hammond, M. M. Guzmán, J. Hughes, T. Johnsson, R. Nikhil D. Kieburtz, W. Partain y J. Peterson. *Report on the programming language Haskell*. Available by anonymous FTP in `ftp.dcs.gla.ac.uk//pub/haskell`, junio 1995.

[HPW92] P. Hudak, S. Peyton-Jones y P. Wadler. Report on the functional programming language Haskell. version 1.2. *SIGPLAN Notices*, 27(5), mayo 1992.

[Knu73] D. E. Knuth. *The Art of Computer Programming, vol.3: Sorting and Searching*. Addison-Wesley, 1973.

[MW92] Z. Manna y R. Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on software engineering*, 18(8), Agosto 1992.

[Par90] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[PV97a] C. Pareja y J. Á. Velázquez, *Constraint-Based Synthesis of Functions: Rules, Applications and Examples*, Informe técnico DIA-55.97, Depto. Informática y Automática, Univ. Complutense de Madrid, España, abril 1997.

[PV97b] C. Pareja y J. Á. Velázquez, Synthesis of functions by transformations and constraints, *International Conference on Functional Programming (ICFP'97)*, pág. 317, 1997. (Se puede encontrar un resumen extendido en la dirección `http://www.cse.ogi.edu/~walidt/icfp-poster/crist-1.ps`).

[PV98] C. Pareja y J. Á. Velázquez, *An exercise in the synthesis and transformation of functional programs with ADTs: encoding and decoding with prefix codes*, Informe técnico 98-11, Universidad Rey Juan Carlos, mayo 1998.

[PW93] C. Paulin-Mohring y B. Werner. Synthesis of ML programs in the system Coq. *J. Symbolic Computation*, 15, 1993, pág. 607–640.

[Sch80] W. L. Scherlis. *Expression Procedures and Program Derivation*. Ph. D. Thesis, Stanford Computer Science, informe STAN-CS-80-818, agosto, 1980.

[Sch81] W. L. Scherlis. Program improvement by internal specialization. *Technical 8th ACM Symposium on Principles of Programming Languages*, 1981, pág. 41–49.

[Sne91] J. L. A. Snepscheut. *Inversion of a recursive tree traversal*. Technical report JAN–171a-0, University of Pasadena, California, Department of Computing, mayo 1991.