Eliminación de Variables Extra en Programación Lógico Funcional

TRABAJO DE INVESTIGACIÓN

Autor: Javier de Dios Castro
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid. (España)

Tutor: Francisco Javier López Fraguas

Junio 2005

Índice general

1. Introducción 5
1.1. Objetivo del trabajo.. 6
1.2. Organización del trabajo 8

2. La programación lógico funcional 9
2.1. Programación lógica.. 10
2.2. Programación funcional.. 14

3. CRWL 17
3.1. Conceptos básicos 19
3.2. El cálculo de pruebas 22

4. Variables extra en programación lógico funcional................................. 26

5. Eliminación de variables extra en la práctica 36
5.1. Preliminares.. 36
5.2. El problema de los generadores 37
5.3. Programación de generadores equitativos 40
5.3.1. Programación de generadores basados en profundidad creciente........... 41
5.3.2. Programación de generadores basados en número de símbolos............. 43
5.3.3. Programación de generadores basados en número de constructores...... 45
5.3.4. Programación de generadores basados en pesos de los constructores... 47
5.4. Ejemplos de generadores 51
5.5. Comparación de tiempos 68

6. Otras alternativas... 72
6.1. Generadores indeterministas 72
6.1.1. Obtención del generador indeterminista.. 73
6.1.2. Restricciones en la definición de los generadores indeterministas 76
6.1.3. Transformación de la regla condicional 77
6.1.4. Propiedades de las reglas transformadas sin variables extra 88
6.2. Transformación fold-unfold.. 90

7. Revisión de otros trabajos.. 95
7.1. Variables extra en programación lógico ecuacional... 95
7.1.1. Eliminación de variables extra en programas lógicos puros...................... 95
7.1.2. Eliminación de variables extra en programas lógico ecuacionales 96
7.2. Eliminación de variables locales en programas lógicos definidos 98
7.3. Transformación de sistemas de reescritura condicionales con variables extra en sistemas no condicionales 101

8. Conclusiones 104

Bibliografía 106

1. Introducción

En el paradigma de la programación lógica, donde se utilizan cláusulas de la forma L <= B, las variables extra, también denominadas variables existenciales ([28], variables locales [7] o variables frescas [27], son aquellas variables que aparecen en el cuerpo de la cláusula, B, pero no en la cabeza, L. En el caso de que el paradigma se extienda a la programación lógica con restricciones, las variables extra también pueden aparecer en la restricción de la cláusula.

El principal cometido de las variables extra consiste en almacenar resultados intermedios que se propagan a través de los átomos del cuerpo de la cláusula.

Ahora bien, la utilización de variables extra puede provocar ciertos problemas de incompletitud e ineficiencia. En el caso de la programación lógica con negación la variables extra están cuantificadas existencialmente implícitamente en el ámbito de las cláusulas, al introducir la negación la cuantificación existencial se transforma en universal, esto produce que la búsqueda de soluciones se complica considerablemente. En los programas lógicos ecuacionales, como se indica en [21], utilizados en ocasiones como base para la integración de la programación lógica y funcional, es bien conocido que las variables extra pueden provocar problemas debido a la semántica operacional estándar para la programación lógico funcional que puede ser incompleta en presencia de variables extra.

EJEMPLO 1.1
Sea el siguiente programa lógico ecuacional

$$a \rightarrow b \qquad b \rightarrow c \Leftarrow f(X,b) = f(c,X)$$
$$a \rightarrow c$$

En principio, este sistema tiene todas las propiedades requeridas para su confluencia y terminación. Sin embargo, la semántica operacional no puede inferir la validez de la ecuación $b = c$ debido a que produce la siguiente derivación infinita (el subtérmino al cual se aplica la regla en cada paso está en negrita):

$$\boldsymbol{b} = c \rightarrow f(X,\boldsymbol{b}) = f(c,X), c = c \rightarrow f(X1,\boldsymbol{b}) = f(c,X1), f(X,c) = f(c,X), c = c \quad \rightarrow \ldots$$

Para demostrar la condición de la última regla, la variable extra X, debe instanciarse a a y las ocurrencias instanciadas deben derivarse a c y b respectivamente. Pero esto no lo proporciona el cálculo de la semántica operacional. Por tanto, queda demostrado que la semántica operacional puede ser incompleta en presencia de variables extra.

Como se indica en [4], en programación lógica es bien conocido que las variables extra son la principal causa de ineficiencia y en algunos casos,

también de incompletitud, en la computación de objetivos negativos. Esto es debido a que la cuantificación universal no es posible eliminarla en la computación de la negación del cuerpo de una cláusula que contenga variables extra y no es posible computar la cuantificación universal de manera eficiente.

En algunos trabajos, como en [15], se ha indicado que las variables extra son necesarias y contribuyen a la potencia expresiva de los lenguajes lógicos. Esto no siempre es cierto, como demostró Hanus [21], puesto que las cláusulas que contienen variables extra pueden transformarse en otras cláusulas sin variables extra. Esta transformación consiste en añadir las variables extra como un nuevo argumento al predicado de cabeza.

EJEMPLO 1.2
A continuación se muestra un ejemplo de variable extra.

```
last Xs X = true <== append(Ys,X,Xs)
```

La variable Ys se encuentra en la restricción pero no en la cabeza de la cláusula, por tanto, Ys es una variable extra.

1.1. Objetivo del trabajo

Este trabajo tiene como objetivo principal identificar una técnica generalizable para la eliminación de variables extra en la concepción más moderna de los lenguajes lógico funcionales, tal como se lleva a cabo en sistemas como TOY[1] y Curry[20]. La motivación fundamental consiste en que muchos trabajos sobre programación lógico funcional sólo consideran sistemas de reescritura sin variables extra. Esto sucede, en particular, en algunos trabajos sobre fallo finito en programación lógico funcional[31], cuyas técnicas son sólo aplicables a programas libres de variables extra.

Para ello, proponemos un nuevo método de eliminación de variables extra en la programación lógico funcional dentro del marco de la lógica de reescritura basada en constructores (Constructor-based ReWriting Logic, CRWL), fundamento semántico adecuado para tales lenguajes.

CRWL descrito en [19], es un sistema de demostración que combina los paradigmas lógico y funcional mediante funciones perezosas no deterministas. Mediante este sistema de demostración es posible modelar funciones no deterministas en sistemas de reescritura de términos basados en constructores no confluentes, y aporta un cálculo de estrechamiento perezoso completo y correcto que utiliza compartición sin reescritura de grafos.

El método para la eliminación de las variables extra consiste en incluir, en el programa lógico funcional, una función indeterminista que representa una función generadora universal de datos y realizar una transformación del programa original con el fin de obtener un programa transformado sin variables extra.
Como hemos indicado, la función generadora universal es una función

indeterminista, pues bien, una función es indeterminista si y solo si existe alguna invocación a dicha función con todos sus argumentos evaluados que puede devolver más de un resultado. Las implementaciones habituales de los sistemas mencionados llevan a cabo los cómputos con funciones indeterministas mediante búsqueda secuencial y backtracking. Para poder representar dichas funciones indeterministas, los lenguajes lógico funcionales, como TOY[1], permiten definir funciones indeterministas mediante la definición de reglas que solapen sus partes izquierdas.

EJEMPLO 1.3

La siguiente función es una función indeterminista

```
parienteDe :: persona -> persona
parienteDe X --> padreDe X
parienteDe X --> madreDe X

padreDe pedro --> antonio
madreDe pedro --> olga
```

Como puede observarse, una misma invocación a la función parienteDe pedro, puede obtener varios resultados, antonio y olga.

Dado que la base del método de eliminación de las variables extra consiste en definir una función generadora universal de datos, aclaramos el concepto de dicha función. Si suponemos un dominio concreto de datos, por ejemplo, los números enteros, una función generadora universal de números enteros consiste en una función indeterminista cuyo resultado será todos los valores que representa el dominio, es decir, los números enteros.

EJEMPLO 1.4
Función generadora universal de números enteros

```
generaEnteros :: int
generaEnteros = 0
generaEnteros = generaEnteros + 1
```

Esta función es indeterminista, puesto que para una invocación de la función generaEnteros, podemos obtener varios resultados. En este caso, y utilizando una implementación que simula el indeterminismo, esta función permite obtener como resultado todos los números enteros.

Utilizando una función indeterminista generadora universal de datos junto con una simple transformación del programa lógico funcional obtenemos un programa transformado que no contiene variables extra, y mostramos que existe una fuerte correspondencia entre el programa original y el programa transformado.

1.2. Organización del trabajo

El trabajo está organizado en las siguientes secciones. En la Sección 2, se introducen las principales características de la programación lógico funcional. En la Sección 3, se recapitula sobre los conceptos básicos del marco de la lógica de reescritura basada en constructores, CRWL, contenidos en el trabajo [19].

En la Sección 4 se realiza un estudio teórico donde se establecen las bases formales para la eliminación de variables extra dentro del marco CRWL, mediante la utilización de la función indeterminista generadora universal de datos y la transformación del programa original con variables extra, mostrándose la fuerte correspondencia entre el programa original y el programa transformado. En la Sección 5 se indica el problema existente con los generadores definidos en la sección anterior y se muestran diferentes implementaciones de los generadores que solucionan este problema así como un estudio comparativo del comportamiento del programa original con los programas transformados utilizando diferentes implementaciones de los generadores. En la Sección 6 se realiza un estudio de otras alternativas tenidas en cuenta a la hora de realizar el trabajo.

La Sección 7 contiene una revisión de otros trabajos relacionados con la eliminación de variables extra y su comparación con la técnica aportada en este trabajo. Por último se indican algunas conclusiones sobre el trabajo.

2. La programación lógico funcional

La programación lógico funcional, que forma parte de la denominada programación declarativa, consiste en un estilo de programación en el cual se especifica qué debe computarse en lugar de cómo deben realizarse los cómputos. Su característica principal consiste en combinar las principales ventajas de los dos paradigmas fundamentales de la programación declarativa, la programación funcional, que permite utilizar estrategias de reducción eficientes, orden superior, etc, y la programación lógica con restricciones que permite la resolución de objetivos, la computación con información parcial mediante la utilización de variables lógicas, etc.

En programación declarativa, la lógica es una especificación de las restricciones de un determinado problema y el control se compone de una máquina de evaluación (es decir, un interprete o un compilador) sobre la cual se evaluarán la satisfacción de las restricciones. Por tanto, en este paradigma de programación, la lógica es la herramienta principal para que el programador realice sus programas.

Ahora bien, para que la lógica pueda ser utilizada debe disponer de las siguientes características:

a.- Un lenguaje con la suficiente capacidad expresiva, permitiendo cubrir un campo de aplicación.

b.- Una semántica operacional. La semántica operacional es un mecanismo de cómputo que permite la ejecución de los programas lógicos.

c.- Una semántica declarativa. La semántica declarativa permite dotar de significado a los programas lógicos de forma independiente a su computación.

d.- Resultados de corrección y completitud. Estos resultados deben asegurar que las computaciones coinciden con lo definido como verdadero en la semántica declarativa.

Las principales ventajas de la programación declarativa son: su mayor nivel de abstracción; separación entre lógica (QUÉ); y el control (CÓMO) y su mejor aplicabilidad a métodos formales.

Los principales campos de aplicación de la programación declarativa se encuentran en el prototipado rápido, la computación simbólica, la inteligencia artificial y los sistemas basados en el conocimiento.

Centrándonos en la programación lógico-funcional, desde que Robinson y Sibert en [29] iniciarán la integración de las características de los programas lógicos y las de los programas funcionales se han desarrollado diferentes lenguajes lógico-funcionales como K-LEAF[9] o BABEL[25]. Actualmente, existen dos lenguajes lógico-funcionales muy desarrollados, Curry[20] y TOY[1]. Estos dos lenguajes se basan en la parte funcional en el lenguaje Haskell[8] y la parte lógica en Prolog[12].

Curry, además de proporcionar las características de integración de los lenguajes puros, incorpora, desde el punto de vista funcional: búsqueda, computación con información parcial, y desde el punto de vista lógico: evaluación más eficiente de funciones deterministas.

Por su parte, el lenguaje TOY introduce características como las funciones no deterministas combinadas con la evaluación perezosa, que mejoran la eficiencia en problemas de búsqueda, y los patrones de orden superior, que permiten una representación de funciones intensional muy útil en computaciones lógicas y funcionales, sin introducir problemas de unificación indecidibles.

La semántica operacional de los lenguajes lógico-funcionales se basa en la combinación de la semántica operacional de los lenguajes funcionales, basado en la reescritura, y la semántica operacional de los lenguajes lógicos, como son la unificación y la resolución. Como resultado de esta combinación surge el estrechamiento. Los lenguajes lógicos-funcionales Curry y TOY utilizan un tipo de estrechamiento perezoso denominado estrechamiento necesario. El estrechamiento necesario consiste en “congelar” la evaluación de los argumentos que intervienen en las funciones a procesar hasta que son necesarios para poder calcular el resultado de la invocación a la función.

Al introducir funciones indeterministas junto con la evaluación perezosa se plantea una semántica de elección en la invocación de las funciones indeterministas, surgiendo la lógica de reescritura basada en constructores [19] (CRWL, Constructor-based ReWriting Logic). Además existen otros trabajos que incluyen entre otros: el orden superior, tipos polimórficos, tipos algebraicos, etc (Véase [30] para un resumen de todos ellos).

A continuación se describe, brevemente, las principales características de la programación lógica y de la programación funcional.

2.1. Programación lógica

Basado en los conceptos propuestos por Kowalski ([23]) la programación lógica se basa en la lógica de predicados. La lógica de predicados para la programación lógica más utilizada es la lógica de cláusulas de Horn, que se utiliza como base para los lenguaje de programación lógica pues posee una semántica operacional que permite una implementación eficiente, mediante la resolución SLD.

Se denominan programas lógicos a los conjuntos de cláusulas de Horn definidas y cómputos lógicos a las refutaciones SLD. Los cómputos son refutaciones de las cláusulas de Horn negativas, denominadas objetivos, en base a las cláusulas del programa.

Las cláusulas de Horn ser caracterizan por que poseen a lo más un literal positivo, o equivalentemente, si al escribir la cláusula en notación Kowalski tiene una de las siguientes 4 formas

Hecho : $P \leftarrow$
Regla : $P \leftarrow Q_1,..., Q_n$
Objetivo : $\leftarrow R_1,..., R_n$
Éxito : $\leftarrow$

Kowalski, a partir de la notación anterior de las cláusulas de Horn, define una doble lectura, desde un punto de vista declarativo, que se corresponde con el sentido de tomar las cláusulas de Horn como sentencias de primer orden, y desde un punto de vista procedural, donde los símbolos de predicado se corresponden con nombres de procedimientos, y los argumentos como parámetros.

En el siguiente ejemplo muestra los diferentes tipos de cláusulas de Horn, según la notación de Kowalski.

EJEMPLO 2.1

El siguiente ejemplo muestra las diferentes formas que pueden presentarse las cláusulas de Horn mediante la notación Kowalski

Hechos: Genealogía bíblica

madre(Milka,Betuel) ←
padre(Teráj,Abraham) ←

Reglas: Relaciones de parentesco

progenitor(x,y) ← madre(x,y)
progenitor(x,y) ← padre(x,y)

hijo(x,y) ← progenitor(y,x),hombre(x)

Objetivos

← progenitor(Milka, x)

Como puede observarse, en ningún caso se está realizando una referencia explícita a cómo se debe de representar en memoria los nombres de los individuos. En la programación lógica no existe una referencia explícita a los

tipo de representación en memoria de la estructura de datos. Los lenguajes de programación lógica proporcionan una gestión automática de la memoria, con las facilidades que conllevan para el programador de no preocuparse de este aspecto.

También se puede observar que las llamadas no necesitan realizarse secuencialmente, pueden ejecutarse secuencialmente en cualquier orden. Esto produce que los programas lógicos sean indeterministas y presentan dos tipos de indeterminismo. El indeterminismo conjuntivo, en el cual la elección del siguiente átomo a procesar del objetivo se realiza de una forma indeterminista, y el indeterminismo disyuntivo, en el cual la elección de la cláusula del programa también es indeterminista seleccionándose entre aquellas cláusulas del programa cuya cabeza sea unificable con el átomo seleccionado.

Dado este indeterminismo, los programas lógicos podrían ejecutarse incluso en paralelo, con tal de que todas las variables compartidas queden instanciadas del mismo modo. La explotación del procesamiento paralelo en programación lógica es aún un objetivo de investigación.

Los programas lógicos son de uso múltiple, es decir, un mismo procedimiento puede ser empleado de diferentes formas según el patrón del objetivo inicial, además no existe una distinción entre los parámetros de entrada y de salida, y permiten multiplicidad de respuestas que son obtenidas de diferentes cómputos. Veámoslo a través del siguiente ejemplo.

EJEMPLO 2.2

Comprobar si es existe una relación de padre e hijo entre los individuos y además si el hijo es varón

← hijo(Lot,Haran)
Respuesta: Sí (ÉXITO)

Calcular los progenitores de un individuo dado

← hijo(Esaú,x)
Respuestas: x=Rebeca, x=Isaac

Calcular los hijos varones de un individuo dado

← hijo(x,Teráj)
Respuestas: x=Abraham, x=Najor, x=Haran

Otra característica utilizada con mucha frecuencia en programación lógica consiste en utilizar cláusulas recursivas. Las cláusulas recursivas son aquellas en las cuales el cuerpo incluye el símbolo del predicado de cabeza. Desde un punto de vista procedural, se corresponden con los procedimientos recursivos.

La semántica declarativa corresponde al sentido que se da a las cláusulas de Horn como sentencias de primer orden. La semántica declarativa se define mediante una teoría de modelos cuyo dominio de interpretación es un universo puramente sintáctico, denominado Universo de Herbrand.

El Universo de Herbrand de un programa lógico puede entenderse como un tipo de datos recursivo cuyos términos son las únicas estructuras de datos que un programa lógico puro puede procesar.

La semántica operacional propone que los símbolos de predicado se entiendan nombres de procedimiento y los argumentos como parámetros. Un átomo $p(t_1,\ldots,t_n)$ es una llamada a un procedimiento y su tarea consiste en instanciar las variables $t_1,\ldots,t_n$ a los valores que satisfagan $p(t_1,\ldots,t_n)$. Los hechos y las reglas se entienden como declaraciones de procedimientos.

La resolución SLD es el proceso que ejecuta tales llamadas, refutando el objetivo $\leftarrow p(t_1,\ldots,t_n)$. La resolución SLD es un método de prueba por refutación que emplea el algoritmo de unificación como mecanismo base y permite la extracción de respuestas, enlazando valores a las variables lógicas existentes.

Si la respuesta obtenida mediante la resolución SLD contiene variables, estas variables se interpretan como cuantificadas universalmente.

Formalmente, la resolución SLD esta formada por un conjunto de cláusulas definidas R, un objetivo $\mathrm{G} \leftarrow \mathrm{R}_1,\ldots,\mathrm{R}_\mathrm{m}$ y una función de selección φ. Esta función de selección asocia a cada objetivo, no vacío, un índice $\varphi(G) = i \in 1,\ldots,n$, entonces se dice que R_i es el átomo seleccionado por la función de selección φ.

Al considerar un conjunto de cláusulas y un determinado objetivo, si fijamos una función de selección, habrá, en general, varias cláusulas del conjunto cuya cabeza será unificable con el átomo del objetivo seleccionado por la función de selección, sucediendo lo mismo para cada uno de los nuevos objetivos resultantes de aplicar estas cláusulas. De esta forma, se crea un árbol cuyos nodos representan objetivos y cuyas ramas representan todas las posibles derivaciones SLD basadas en el conjunto de cláusulas. A este árbol se le denomina espacio de búsqueda SLD.

Dependiendo de la función de selección utilizada, se determinará un espacio de búsqueda diferente, correspondiéndose cada uno de ellos con una manera de abordar el problema. Ahora bien, estos espacios de búsqueda van a contener el mismo número de éxitos, que se corresponden con las respuestas, pero posiblemente van a contener un número distinto de fallos. Por tanto, una misma respuesta puede ser calculada empleando diferentes funciones de selección.

La resolución SLD es un método de prueba correcto y completo para la lógica de cláusulas de Horn.

En los cómputos de los programas lógicos, la unificación y la sustitución se encargan de varias tareas: invocación de procedimientos, selección de casos, acceso a los componentes de los datos, construcción de los datos, etc. Los términos con variables juegan, a este respecto, un papel importante como patrones incompletos de datos.

Es posible definir nociones de completitud y terminación para programas lógicos, basándose en la finitud o infinitud de los espacios de búsqueda y en el tamaño de los objetivos y las derivaciones en él incluidos.

2.2. Programación funcional

Los lenguajes funcionales se basan en el concepto de función, según el concepto de función matemática, donde un programa es una colección de definiciones de funciones. La definición de estas funciones se realiza mediante ecuaciones que, normalmente son recursivas. Desde un punto de vista computacional, el usuario se dedica a evaluar expresiones para obtener un resultado, que son independientes del orden de evaluación.

A través del siguiente ejemplo, veremos las características fundamentales de la programación funcional:

EJEMPLO 2.3

Este ejemplo, típico en programación funcional, muestra una función denominada append con dos argumentos (tipo de datos listas) y devuelve un resultado (también de tipo de datos de listas) que expresa la concatenación de las listas argumento.

La primera línea consiste en la declaración del tipo de datos de la función, con sus variables de tipo de datos para los argumentos, es decir su dominio, y su variable de tipo de datos listas para el resultado, que representa su codominio.

Las siguientes dos líneas representan la declaración de la función desde el punto de vista declarativo.

```
append :: [t] -> [t] -> [t]
append [] Ys = Ys
append [X:Xs]  Ys = [X: append Xs Ys]
```

Como puede observarse, las funciones en los lenguajes funcionales se basan en la idea de función matemática, disponiendo de un dominio, asociado al tipo del argumento, y un codominio asociado al tipo del resultado. Las definiciones de las funciones se realizan utilizando ecuaciones, una o varias, y posibles condiciones, denominadas guardas.

Los lenguajes funcionales, a través de la noción de dominio y codomino, incluyen los tipos de datos, permitiendo detectar errores y ayudar a definir estructuras de datos. Normalmente, los lenguajes funcionales se encuentran entre los lenguajes fuertemente tipados, realizando una comprobación de tipos de todas las expresiones del programa antes de la ejecución.

Las principales características de los lenguajes funcionales con disciplina de tipos son las siguientes:

1.- Toda expresión tiene un tipo.

2.- Los tipos se infieren, es decir se comprueban, de forma estática, en tiempo de compilación.

3.- Los tipos utilizados pueden ser básicos o construidos.

Como puede observarse en el ejemplo, se ha definido un tipo de datos lista, mediante una declaración de datos, en la cual aparecen los símbolos”[ ]" y “:" que son denominados los constructores del tipo y aparece también el símbolo t, que se denomina variable de tipo. Esto implica que es posible construir listas de diferentes con diferentes tipos de datos.

En programación funcional, las expresiones utilizadas denotan valores, y el valor de una expresión depende de un contexto. En el caso de utilizar una función, cada valor de su domino está asociado a un único valor en el codomino. Esta propiedad se denomina transparencia referencial y consiste en que el valor de una expresión es independiente del orden de evaluación y del modo de uso de esta expresión como subexpresión de otra.

Una característica muy importante de las funciones consiste en la operación de composición. La composición de funciones es una de las principales técnicas utilizadas en la programación funcional, debido a que una de las formas más simples de estructurar un programa consiste en realizar cierto número de operaciones una después de otra, esto se consigue mediante la composición de funciones en la cual el resultado de una función es la entrada de otra.

EJEMPLO 2.4

El siguiente ejemplo muestra la composición de funciones. Supongamos la siguiente declaración de las funciones not y even:

```
not :: bool -> bool
not true = false
not false = true

even :: int -> bool
even X     = (X `mod` 2) == 0
```

mediante la utilización de la composición de funciones, es posible definir la función odd, mediante la composición de las funciones not y even. En este caso, el operador de composición es el símbolo (.).

```
odd :: int -> bool
odd       = not . even
```

Otra característica fundamental en los lenguajes funcionales consiste en tratar las funciones como "ciudadanos de primera clase", es decir, que las funciones puede ser pasadas, a su vez, como argumentos en otras funciones, pueden devolverse como resultados y pueden aparecer dentro de estructuras de datos. Si una función contiene alguna de estas características se denomina función de orden superior.

EJEMPLO 2.5

Este ejemplo declara la función de orden superior más típica de la programación funcional, map, que tiene como primer argumento una función f y como segundo argumento una lista. El resultado es una lista compuesta por los elementos resultantes de aplicar la función f a cada uno de los elementos de la lista.

```
map:: (t1 -> t2) -> [t1] -> [t2]
map F [] = []
map F [X:Xs] = [(F X):(map F Xs)]
```

En los lenguajes funcionales, la ejecución de un programa consiste en la evaluación de una expresión inicial utilizando la definición de funciones del programa y un proceso de reducción. Este proceso, que se ejecuta paso a paso dependiendo de la estrategia de reducción empleada, reduce o simplifica una parte de la expresión inicial, de tal forma que esta expresión inicial compuesta por símbolos de función y constructores de datos se transforma, finalmente, en un valor que sólo está compuesto por ocurrencias de símbolos constructores. Este último valor es el resultado de la evaluación.

Como se ha indicado anteriormente, la estrategia de reducción define la secuencia de reducción en la evaluación de una expresión. Las principales estrategias de reducción son impaciente y la perezosa.

La estrategia de reducción perezosa, evalúa los argumentos sólo si su valor es necesario para computar el resultado. En el caso de la estrategia de reducción impaciente, evalúa siempre primero todos los argumentos. Por tanto la estrategia de reducción perezosa tiene claras ventajas como son: evitar cómputos innecesarios de forma automática; y conseguir cómputos que terminen aunque se estén procesando una estructura de datos, potencialmente, infinita. Aunque la implementación de la estrategia impaciente es más sencilla, las secuencias de reducción que pueden terminar al utilizar una estrategia de evaluación perezosa, pueden no terminar en esta estrategia.

3. CRWL

La lógica de reescritura basada en constructores (Constructor-based ReWriting Logic, CRWL) descrita en [19] es una aproximación de los lenguajes declarativos, que combina los paradigmas lógico y funcional mediante funciones perezosas no deterministas. Gracias a ello, es posible modelar funciones no deterministas en sistemas de reescritura de términos basados en constructores no confluentes, donde un término puede rescribirse en términos construidos (posiblemente con variables) de más de una forma. Además, CRWL aporta un cálculo de reducción perezosa que es completo, correcto y además soporta compartición sin necesidad de utilizar la reescritura de grafos.

Un sistema de reescritura de términos basados en constructores clasifica las operaciones de símbolos en dos categorías: las funciones definidas, que se expresan mediante reglas de reescritura y los constructores, que se utilizan para representar los valores computados como términos construidos.

Una característica de los lenguajes lógico funcionales perezosos es la distinción entre la igualdad común, $e = e'$, y la igualdad estricta $e == e'$. La igualdad estricta se entiende como que e y e' pueden reducirse al mismo término construido, que es un valor totalmente definido y finito, mientras que la igualdad común permite la posibilidad de que el valor de e y e' puede ser infinito y/o parcialmente definido. Normalmente, la igualdad estricta se utiliza para construir objetivos y para las condiciones de las reglas de reescritura condicionales.

Otra característica fundamental, consiste en la utilización de funciones no deterministas. Esto se consigue manteniendo constructores deterministas y definiendo funciones con una serie de términos construidos fijados como argumentos y que devuelven más de un término construido como resultado.

A continuación se muestra un ejemplo de reglas de reescritura de términos basados en constructor donde se define la función merge que mezcla dos listas obtenidas de forma no determinista.

EJEMPLO 3.1

```
merge([],Ys) -> Ys
merge([X|Xs],[]) -> [X|Xs]
merge([X|Xs],[Y|Ys]) -> [X| merge(Xs,[Y|Ys])]
merge([X|Xs],[Y|Ys]) -> [Y| merge([X|Xs],Ys)]
```

En este caso la invocación con merge([1],[2,3]) obtendremos los resultados: [1,2,3], [2,1,3] y [2,3,1].

Como vimos anteriormente, normalmente, para expresar objetivos y condiciones en reglas de reescritura condicionales se utilizaba la igualdad

estricta. En el caso de CRWL, la igualdad estricta se reemplaza por el concepto más general de confluencia, la confluencia se define como: dos términos a y b son confluentes (representado como $a \bowtie b$) si y solo si pueden rescribirse en un término construido común, permitiendo computar soluciones generales para un objetivo.

La presencia de indeterminismo en los lenguajes de programación produce problemas semánticos debido a las diferentes semánticas existentes. CRWL se basa en la siguiente opción semántica: no determinismo angélico con elección en tiempo de ejecución en funciones no estrictas.

El no determinismo angélico significa que los resultados de todas las posibles computaciones, debido a las diferentes elecciones no deterministas, son recogidas por la semánticas aunque no se excluya la posibilidad de computaciones infinitas.

La elección en tiempo de ejecución significa que dada una llamada de función $f'(e_1,\ldots,e_n)$, se elige un valor (posiblemente parcial) de los parámetros actuales e_i antes de aplicar la regla de reescritura que define f.

Por último, señalar que, como la reescritura innermost es correcta con respecto a la elección en tiempo de ejecución pero no es completa con respecto a las funciones no estrictas, se determina que todas las variables que tengan más de una ocurrencia en la parte derecha de la misma regla de reescritura, debe ser compartida. Esto está incluido en el cálculo de reescritura, por tanto no es necesario utilizar grafos de términos.

A través del siguiente ejemplo se muestra la ventaja que supone utilizar las funciones no deterministas y la evaluación perezosa frente a un programa lógico típico de “generación y prueba” para mejorar su eficiencia.

EJEMPLO 3.2

Un ejemplo de programa lógico ineficiente es la ordenación por permutación, donde la lista Ys es el resultado de realizar la ordenación sobre la lista Xs si Ys es una permutación de Xs y Ys está ordenada

```
permutation_sort(Xs,Ys) :- permute(Xs,Ys), sorted(Ys)
```

Este programa es muy ineficiente cuando se ejecuta mediante reglas estándar de computación de Prolog pues cada solución candidata Ys se genera completamente antes de probarse.

Mediante la utilización de CRWL se pueden utilizar funciones no deterministas que describen la generación de los candidatos de forma declarativa. Debido a que los candidatos se generan de uno en uno, es posible eliminar las computaciones de estructuras más grandes, como se haría en programación funcional, sin perder la completitud. Además, se elimina la ineficiencia del

programa inicial, debido a que la evaluación perezosa asegura que la generación de cada permutación se interrumpirá tan pronto como el testeador, en este caso sorted, reconozca que no produce una lista ordenada.

El generador permute se define como:

```
permute([]) -> []
permute([X|Xs]) -> insert(X,permute(Xs))

insert(X,Ys) -> [X|Ys]
insert(X,[Y|Ys]) -> [Y|insert(X,Ys)]
```

donde permute es una función no determinista debido a la función auxiliar insert. Por cada permutación generada por permute, Ys, se aplicará la prueba sorted(Ys) y devolverá Ys como resultado si Ys está ordenada. Si falla, se intentará con otro valor de permute(Xs).

El programa queda de la siguiente forma:

```
sort(Xs) -> check(permute(Xs))
check(Xs) -> Ys <== sorted(Ys) ⋈ true
```

con la siguiente definición de sorted, suponiendo que las listas se componen de números naturales representados por los constructores zero y suc

```
sorted([]) -> true
sorted([X]) -> true
sorted([X|X'|Xs]) -> true <== leq(X,X') ⋈ true, sorted([X'|Xs]) ⋈ true

leq(zero,X) -> true
leq(suc(X),zero) -> false
leq(suc(X),suc(Y)) -> leq (X,Y)
```

Se asume que el lector está familiarizado con los conceptos básicos de programación lógica [5, 22] y la reescritura de términos [15]. A continuación se indican los conceptos básicos de notación y terminología que se utilizarán a lo largo del artículo.

3.1. Conceptos básicos

Posets y CPOs

Un conjunto parcialmente ordenado, denominado poset, con bottom $\perp$, es un conjunto S que comprende un orden parcial $\sqsubseteq$ y un elemento mínimo $\perp$, con respecto a $\sqsubseteq$.

Un elemento esta totalmente definido si y solo si es maximal con respecto a $\sqsubseteq$. El conjunto de todos los elementos definidos de S se denota mediante $Def(S)$. $D \subseteq S$ es un conjunto dirigido si y solo si para todo $x, y \in D$ existe $z \in D$ con $x \sqsubseteq z$ y $y \sqsubseteq z$.

Un poset con bottom C es un orden parcial completo, denominado cpo si y solo si D tiene un supremo $\cup D$, denominado también límite, para cada conjunto dirigido $D \subseteq C$. En particular $\cup\varnothing = \perp$. Un elemento $u \in C$ se denomina elemento finito si y solo si siempre que $u \sqsubseteq \cup D$ para un conjunto D dirigido no vacío, existe $x \in D$ con $u \sqsubseteq x$.

El orden parcial se interpreta como un orden de aproximación entre valores definidos parciales, es decir $x \sqsubseteq y$ expresa que y está más definido que x.

Signaturas, términos y C-términos

Una *signatura con constructores* es un conjunto numerable $\sum = DC_{\sum} \bigcup FS_{\sum}$, donde $DC_{\sum} = \bigcup_{n \in N} DC^n_{\sum}$ y $FS_{\sum} = \bigcup_{n \in N} FS^n_{\sum}$ son, respectivamente, conjuntos disjuntos de *constructores* y *símbolos de función definidos*, cada uno de ellos con una *aridad* asociada.

Dada una signatura (con constructores) $\sum$ y un conjunto de símbolos de variable V, disjuntos de todos los conjuntos de $DC^n_{\sum}$ y $FS^n_{\sum}$, se definen los $\sum$*-términos* de la siguiente manera: cada símbolo de V y cada símbolo de $DC^0_{\sum}$ y $FS^0_{\sum}$ es un $\sum$-término, y para cada $h \in DC^n_{\sum} \bigcup FS^n_{\sum}$ y términos $t_1,...,t_n$, $h(t_1,...,t_n)$ es un término. El conjunto $Term_{\sum}$ esta compuesto por todos los $\sum$-términos.

Además, se establece el subconjunto $CTerm_{\sum}$ compuesto por los términos de $Term_{\sum}$ (denominados *términos construidos* o *C-términos*) formados únicamente con símbolos de $Term_{\sum}$ y V. También, es necesario añadir un constructor con aridad 0, $\perp$, obteniendo una nueva signatura $\sum_{\perp}$ cuyos términos se denominan $\sum$*-términos parciales*. Se escribe $Term_{\perp}$ y $CTerm_{\perp}$ para indicar los correspondientes conjuntos de términos de esta signatura extendida.

Un *orden de aproximación* $\sqsubseteq$ para términos parciales puede definirse como el menor orden parcial sobre $Term_{\perp}$ que satisface las siguientes propiedades:
* $\perp \sqsubseteq e$ para todo $e \in Term_{\perp}$,

* $e_1 \sqsubseteq e'_1, ..., e_n \sqsubseteq e'_n \Rightarrow h(e_1,...,e_n) \sqsubseteq h(e'_1,...,e'_n)$ para todo $h \in DC^n \bigcup FS^n$, $e_1,...,e_n \in Term_\perp$.

Sustituciones

Las *C-Sustituciones* son aplicaciones $\theta : V \to CTerm$ que tiene $\hat{\theta} : Term \to Term$ como extensiones naturales, también denotadas como θ. Así, $t\theta$ denota el resultado de aplicar θ al término t. De manera análoga, se definen las *C-Sustituciones parciales* como aplicaciones $\theta : V \to CTerm_\perp$. El conjunto de todas las C-Sustituciones (C-Sustituciones parciales) se denota mediante $CSubst$ ($CSubst_\perp$).

Como se ha visto anteriormente, la semántica utilizada comprende indeterminismo angélico, elección en tiempo de ejecución y funciones no estrictas. Por ello se utiliza un sistema especial de demostración denominado lógica de reescritura basada en constructores (CRWL). A continuación se formalizan aspectos básicos de que dispone CRWL.

Si suponemos una signatura con constructores $\sum = DC \bigcup FS$, las teorías-CRWL, comúnmente denominadas programas, se definen como reglas de reescritura de la forma:

$$f(\bar{t}) \to r$$

donde $f(\bar{t})$ es la parte izquierda de la regla, con f símbolo de función con aridad $n \geq 0$, Cuando $n > 0$, $\bar{t}$ es una n-tupla lineal de C-términos, donde lineal significa que cada variable aparece una única vez. Cuando $n = 0$, la regla tiene la forma $f \to r$. r es la parte derecha de la regla.

En este sentido, se utiliza la siguiente notación para las C-instancias parciales de las reglas de reescritura:

$$[\Re]_\perp = \{(l \to r)\sigma \mid (l \to r) \in \Re, \sigma \in CSubst_\perp\}$$

Aunque en los trabajos originales de CRWL se permite utilizar reglas condicionales, a lo largo del presente trabajo únicamente se utilizarán reglas no condicionales, puesto que la parte condicional de toda regla puede sustituirse mediante otras construcciones, como por ejemplo, *if ...then* en el lenguaje lógico funcional TOY (véase [31]). Por otra parte, y a semejanza de otros trabajos, en éste no consideramos a la igualdad estricta, $\bowtie$, como primitiva con reglas especiales, sino que la suponemos definida como cualquier otra función[1].

[1] En programas concretos utilizamos = y == en lugar de $\to$ y $\bowtie$ para ajustarnos a la sintaxis de TOY

Veamos estas consideraciones a través del siguiente ejemplo.

EJEMPLO 3.3

Supongamos el siguiente programa, que utiliza reglas condicionales:

```
insert X []     = [X]
insert X [Y|Ys] = [Y|Zs]   <== X > Y, insert X Ys  == Zs
insert X [Y|Ys] = [X,Y|Ys] <== X <= Y
```

En programación al estilo TOY este programa con parte condicional puede sustituirse mediante construcciones if ... then, como sigue:

```
% if ... then
if true then X = X

% And
true  /\ X = X

insert' X []     = [X]
insert' X [Y|Ys] = if ((X > Y) /\ (insert' X Ys  == Zs))  then [Y|Zs]
insert' X [Y|Ys] = if (X <= Y) then [X,Y|Ys]
```

Suponemos una definición adecuada de <, <= y ==. Por ejemplo, si el programa maneja números naturales representados mediante las constructoras 0 y s (sucesor), la igualdad, ==, vendría definida por las reglas:

```
0 == 0 = true
0 == s(X) = false
s(X) == 0 = false
s(X) == s(Y) = X == Y
```

3.2. El cálculo de pruebas

Un programa CRWL $\Re$ es capaz de derivar sentencias de reducción de la forma $a \to b$ donde “reducción” incluye la posibilidad de aplicar reglas de reescritura de $\Re$ o reemplazamiento de algunos subtérminos de a por $\perp$.

La especificación formal de las CRWL-derivaciones a partir de un programa $\Re$ viene dada por el siguiente cálculo de pruebas, denominado Cálculo de Reescritura Básico (BRC)

B Bottom: $e \rightarrow \perp$

MN Monotonía:

$$\frac{e_1 \rightarrow e'_1 \ldots e_n \rightarrow e'_n}{h(e_1,\ldots,e_n) \rightarrow h(e'_1,\ldots,e'_n)} \quad \text{para } h \in DC^n \cup FS^n$$

RF Reflexividad: $e \rightarrow e$

R Reducción

$$\overline{l \rightarrow r} \quad \text{para cualquier instancia } (l \rightarrow r) \in [\Re]_\perp$$

TR Transitividad

$$\frac{e \rightarrow e' \quad e' \rightarrow e''}{e \rightarrow e''}$$

Para poder utilizar CRWL como base lógica para la programación declarativa, se introduce un segundo calculo de reescritura denominado Cálculo de Reescritura orientado a Objetivos (Goal-Oriented Proof Calculus, GORC) el cual permite construir también demostraciones de sentencias de aproximación y de confluencia. Se prueba en [19] que una sentencia $e \rightarrow t$, con $t \in CTerm_\perp$, es derivable utilizando BRC si y sólo si es derivable utilizando GORC.

Las demostraciones orientadas a objetivos tiene la propiedad de que la estructura sintáctica más externa de la declaración que va a ser demostrada determina la regla de inferencia que debe ser aplicada en el paso de demostración, por tanto, la estructura de la demostración viene determinada por la estructura del objetivo.

La representación formal de GORC es la siguiente:

BT Bottom: $$\frac{}{e \to \perp}$$

RR Reflexividad restringida: $$\frac{}{X \to X}$$ para $X \in V$

DC Descomposición

$$\frac{e_1 \to t_1, \ldots, e_n \to t_n}{c(e_1, \ldots, e_n) \to c(t_1, \ldots, t_n)}$$ para $c \in DC^n, t_i \in CTerm_\perp$

OR Reducción más externa

$$\frac{e_1 \to t_1, \ldots, e_n \to t_n, r \to t}{f(e_1 \ldots e_n) \to t}$$ si $t \neq \perp, (f(t_1, \ldots, t_n) \to r) \in [\Re]_\perp$

Una derivación-CRWL puede presentarse como un árbol invertido de sentencias φ_i , $0 \le i \le n$, de tal forma que cada sentencia φ_i , $0 \le i \le n$ genera nuevas sentencias φ_j , $i < j \le n$, al aplicar alguna regla de inferencia CRWL, pasando a ser estas sentencias, φ_j, los hijos en el árbol de la sentencia inicial, φ_i.

Una sentencia φ se denomina CRWL-probable para un programa $\Re$ (y se escribe como $\Re \vdash_{CRWL} \varphi$) si y solo si existe alguna derivación-CRWL de tal forma que φ_0 es φ utilizando únicamente instancias de reglas del conjunto de $\Re$ y pasos de reducción OR.

EJEMPLO 3.4

A continuación se muestra un ejemplo de sentencia CRWL-probable, asumiendo que el programa $\Re$ incluye las siguientes reglas:

```
coin::int
coin --> 0
coin --> 1

coins::[int]
coins --> [coin|coins]
```

entonces se muestra que $\Re \vdash_{CRWL} coins \to 0:1:\bot$ con la siguiente derivación CRWL:

$$OR\;\dfrac{DC\;\dfrac{OR\;\dfrac{DC\;\dfrac{}{0 \to 0}}{coin \to 0} \qquad OR\;\dfrac{DC\;\dfrac{OR\;\dfrac{DC\;\dfrac{}{1 \to 1}}{coin \to 1} \qquad B\;\dfrac{}{coin \to \bot}}{coin : coins \to [1|\bot]}}{coins \to [1|\bot]}}{coin : coins \to [0,1|\bot]}}{coins \to [0,1|\bot]}$$

4. Variables Extra en Programación Lógico Funcional

Como hemos visto, en la sección anterior, los programas-CRWL se componen de un conjunto de reglas de reescritura que tienen la forma:

$$f(\bar{t}) \rightarrow r$$

Las variables extra (también denominadas variables existenciales [28], variables locales[7], o variables frescas [27]) son aquellas variables x, tal que $x \in Var(r) - Var(\bar{t})$.

Uno de los principales usos de las variables extra consiste en almacenar resultados intermedios que se propagan a través de los átomos del cuerpo de la cláusula. Ahora bien, la utilización de variables extra puede provocar ciertos problemas de incompletitud e ineficiencia.

En esta sección se establecen las bases formales para la eliminación de variables extra mediante una función indeterminista generadora y la transformación del programa original, obteniendo un programa transformado sin variables extra.

A lo largo del trabajo no se tendrán en cuenta dos aspectos: la parte condicional C (como vimos anteriormente, siempre es posible eliminarla ya que en programación al estilo TOY puede sustituirse mediante construcciones $if \ldots then$); los tipos polimórficos, debido al propio concepto de generador universal de datos y la transformación del programa original. Este problema será descrito más adelante, cuanto se defina el generador y la transformación del programa original.

Algunos de los resultados que siguen, como los Lemas 1 y 2, y salvo algunas variaciones ya han sido probados (en [32], [19]). Los incluimos aquí por completitud del trabajo.

Lema 1. Sea $\Re$ un programa, $e \in Term_\perp$, y $t \in CTerm_\perp$. Si $\Re \vdash_{CRWL} e \rightarrow t$, entonces $\Re \vdash_{CRWL} e\theta \rightarrow t\theta$, para todo $\theta \in CSubst_\perp$. Además, puede conseguirse que el número de pasos OR en ambas derivaciones sea el mismo.

Demostración. La demostración se realiza por inducción sobre la derivación de $\Re \vdash_{CRWL} e \rightarrow t$.

- Caso base: La derivación tiene que tener una de las dos formas siguientes:
  - Regla B: $e \rightarrow \perp$ es trivial.
  - Regla RR: $X \rightarrow X$, $\Re \vdash_{CRWL} X\theta \rightarrow X\theta$, para todo $\theta \in CSubst_\perp$, lo que se prueba fácilmente usando, por ejemplo, la regla de reflexivilidad de BRC.

- Casos inductivos: Se distinguen dos subcasos, de acuerdo a la regla de reducción CRWL utilizada en el último paso de inferencia.

  - Regla DC: Si la derivación es

$$\frac{e_1 \to t_1 ... e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$$

    donde entonces podemos construir

$$\frac{e_1\theta \to t_1\theta ... e_n\theta \to t_n\theta}{c(e_1\theta,...,e_n\theta) \to c(t_1\theta,...,t_n\theta)}$$

    donde $\Re \vdash_{CRWL} e_i\theta \to t_i\theta \ (1 \le i \le n)$ se cumple por hipótesis de inducción.

  - Regla OR: Si la derivación es

$$\frac{e_1 \to t_1\sigma, ...e_n \to t_n\sigma \ r\sigma \to t}{f(e_1,...,e_n) \to t}$$

    siendo $f(t_1,...,t_n) \to r \in \Re$, donde $\Re \vdash_{CRWL} e_i \to t_i\sigma \ (1 \le i \le n)$ y $\Re \vdash_{CRWL} r\sigma \to t$. Entonces, por hipótesis de inducción tenemos $\Re \vdash_{CRWL} e_i\theta \to t_i\sigma\theta \ (1 \le i \le n)$ y $\Re \vdash_{CRWL} r\sigma\theta \to t\theta$

$$\frac{e_1\theta \to t_1\sigma\theta ... e_n\theta \to t_n\sigma\theta \ r\sigma\theta \to t\theta}{f(e_1\theta,...,e_n\theta) \to t\theta}$$

Se debe observar que este es un resultado conocido en el marco CRWL. Pero hay que tener en cuenta que si se consideran reglas condicionales con condiciones de la forma $e \bowtie e'$ (en este trabajo no lo estamos considerando), entonces el resultado anterior sólo es válido para $\sigma \in CSubst$.

A continuación se muestra un ejemplo donde no es válido para $\sigma \in CSubst_\perp$.

EJEMPLO 4.1

Sea $\Re \equiv f(x) \to 0 \Leftarrow x \bowtie y$

Se tiene $\Re \vdash_{CRWL} f(x) \to 0$,

Pero no $\Re' \vdash_{CRWL} f(\perp) \to 0$

Lema 2. (Lema de Monotonía). Sea $\Re$ un programa, $e, e' \in Term_\perp$, y $t \in CTerm_\perp$. Si $\Re \vdash_{CRWL} e \to t$ y $e \sqsubseteq e'$, entonces $\Re \vdash_{CRWL} e' \to t$.

Demostración. La demostración se realiza por inducción sobre la derivación de $\Re \vdash_{CRWL} e \to t$.

- Caso base: La derivación tiene que tener una de las dos formas siguientes:

  - Regla B: $e \to \perp$, si $e \sqsubseteq e'$ entonces $\Re \vdash_{CRWL} e' \to \perp$

  - Regla RR: $X \to X$, si $e \sqsubseteq e'$ supone que $e \equiv e'$ luego $\Re \vdash_{CRWL} e' \to t$

- Casos inductivos: Se distinguen dos subcasos, de acuerdo a la regla de reducción CRWL utilizada en el último paso de inferencia.

  - Regla DC: Si la derivación es

$$\frac{e_1 \to t_1 ... e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$$

    y $c(e_1,...,e_n) \sqsubseteq e'$ debe ser $e' = c(e'_1,...,e'_n)$ con $e_i \sqsubseteq e'_i$ $(1 \le i \le n)$ y entonces podemos construir,

$$\frac{e'_1 \to t_1 ... e'_n \to t_n}{c(e'_1,...,e'_n) \to c(t_1,...,t_n)}$$

    donde $\Re \vdash_{CRWL} e'_i \to t_i$ $(1 \le i \le n)$ se cumple por hipótesis de inducción.

  - Regla OR: Si la derivación es

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; r\sigma \to t}{f(e_1,...,e_n) \to t}$$

    siendo $f(t_1,...,t_n) \to r \in \Re$ y $\sigma \in CSubst_\perp$, y si $f(e_1,...,e_n) \sqsubseteq e'$ será $e' = f(e'_1,...,e'_n)$ con $e_i \sqsubseteq e'_i$ $(1 \le i \le n)$ y entonces podemos construir

$$\frac{e'_1 \to t_1\sigma ... e'_n \to t_n\sigma \; r\sigma \to t}{f(e'_1,...,e'_n) \to t}$$

    donde $\Re \vdash_{CRWL} e'_i \to t_i\sigma$ $(1 \le i \le n)$ se cumple por hipótesis de inducción.

Lema 3. Sea $\Re$ un programa, $e \in Term_\perp$, y $t \in CTerm_\perp$. Si $\Re \vdash_{CRWL} e \to t$ y $Var(e) = Var(t) = \emptyset$, entonces $e \to t$ tiene una derivación sin variables.

Demostración. La demostración se realiza por inducción sobre una medida de la complejidad de la derivación de $\Re \vdash_{CRWL} e \to t$, definida como $|e \to t| = (n,k)$, siendo n el número de pasos OR en la derivación, y k el número de pasos totales. $|e \to t|$ se ordena lexicográficamente.

- Caso base: Corresponde a la complejidad mínima (0,1). La derivación tiene que tener una de las dos formas siguientes:

  - Regla B: $e \to \perp$ es trivial
  - Regla RR: $c \to c$ es trivial

  Nótese que el caso $X \to X$ está excluido por ser $Var(e \to t) = \emptyset$.

- Casos inductivos: Se distinguen dos subcasos, de acuerdo a la regla de reducción CRWL utilizada en el último paso de inferencia.

  - Regla DC: Si la derivación es

    $$\frac{e_1 \to t_1 ... e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$$

    el resultado es inmediato por la hipótesis de inducción, ya que $|e_i \to t_i|$ $(1 \le i \le n)$ es claramente menor que $|c(e_1,...,e_n) \to c(t_1,...,t_n)|$.

  - Regla OR: Si la derivación es

    $$\frac{e_1 \to t_1, ... e_n \to t_n \; r \to t}{f(e_1,...,e_n) \to t}$$

    siendo $f(t_1,...,t_n) \to r \in [\Re]_\perp$.

    Sea $\sigma \in CSubst_\perp$ cualquiera (con lo que podemos, incluso, elegir $\sigma \in CSubst$) que sea cerrada para $Var(\{t_1,...,t_n,r\})$, es decir $Var(X\sigma) = \emptyset$ para todo $X \in Var(\{t_1,...,t_n,r\})$.

    Se tiene, por el Lema 1, que $\Re \vdash_{CRWL} e_i\sigma \to t_i\sigma \; (1 \le i \le n)$ y $r\sigma \to t\sigma$, donde además el número de pasos OR de $e_i\sigma \to t_i\sigma \; (1 \le i \le n)$ y $r\sigma \to t\sigma$ no varía respecto a $e_i \to t_i \; (1 \le i \le n)$ y $r \to t$. Así pues, $|e_i\sigma \to t_i\sigma| \; (1 \le i \le n)$ y $|r\sigma \to t\sigma|$ son menores que $|f(e_1,...,e_n) \to t|$, ya que ésta tiene un paso OR adicional.

Además, como $Var(e_i) = Var(t) = \emptyset$, podemos decir que $e_i\sigma = e_i (1 \le i \le n)$ y $t\sigma = t$, luego tenemos que $\Re \vdash_{CRWL} e_i \to t_i\sigma \; (1 \le i \le n)$ y $r\sigma \to t$. Como todas las expresiones anteriores son cerradas, $Var(e_i \to t_i\sigma) = \emptyset \; (1 \le i \le n)$. Así pues podemos aplicar hipótesis de inducción, y suponer que todas ellas son derivaciones sin variables.

Como además, $f(t_1\sigma,...,t_n\sigma) \to r\sigma \in [\Re]_\perp$, podemos construir una derivación sin variables de $f(e_1,...,e_n) \to t$ mediante el paso de reducción

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; r\sigma \to t}{f(e_1,...,e_n) \to t}$$

Aunque no nos resultará necesario para lo que sigue, el lema anterior puede generalizarse al siguiente:

Lema 4. Sea $\Re$ un programa, $e \in Term_\perp$, y $t \in CTerm_\perp$. Si $\Re \vdash_{CRWL} e \to t$, entonces existe una derivación de $e \to t$ sin variables distintas de las de $e \to t$.

Demostración (esbozo). Se pueden cambiar las variables de $e \to t$ por nuevas constantes y aplicar el Lema 3, obteniendo una derivación sin variables. Bastaría, en esta derivación, volver a cambiar las constantes introducidas por las variables originales.

A continuación, pasamos a realizar la definición de la función generadora y se indica cómo se debe realizar la transformación del programa original con variables extra para obtener el programa transformado sin variables extra.

Definición. (Función gen y programa transformado). Dado un programa $\Re$, su transformado resulta de reemplazar cada regla:

$f(t_1,...,t_n) \to r$ con $Var(r) - Var(\overline{t}) = \{x_1,...,x_m\} \ne \emptyset$ por

$f(t_1,...,t_n) \to f'(t_1,...,t_n, gen,..., gen)$
$f'(t_1,...,t_n, x_1,..., x_m) \to r$

junto con las reglas para la función gen de aridad 0 gen

$gen \to a$ %para cada constructor $a \in CS^0$
$gen \to c(gen,..., gen)$ %para cada constructor $c \in CS^n$ con $n \ge 1$

EJEMPLO 4.2

Supongamos el siguiente programa $\Re$:

```
plus :: nat -> nat -> nat -> bool
plus c     X  X    = true
plus (s X) Y (s Z) = true <== plus X Y Z

times :: nat -> nat -> nat -> bool
times  c    X c = true
times (s X) Y Z = true <== times X Y W, plus W Y Z
```

y el tipo de datos:

```
data nat = c | s nat
```

su programa transformado es:

```
times  c    X c = true
times (s X) Y Z = times’ (s X) Y Z gen
times’ (s X) Y Z W = true <== times X Y W, plus W Y Z
```

y la función gen es:

```
gen = c
gen = s (gen)
```

Es importante señalar que la variable extra se sustituye por una función generadora, gen, correspondiente a un tipo de datos concreto. En el caso de que la función times fuera polimórfica, el problema que se plantea es el siguiente: el tipo de datos se determina en tiempo de ejecución y, por tanto, deberían existir tantas funciones generadoras como posibles tipos de datos pudiera utilizar esta función polimórfica.

El siguiente lema muestra que la función generadora permite obtener todos los términos, cerrados, del universo de datos.

Lema 5. Sea $\Re$ un programa que tenga definida la función gen, y $t \in CTerm_{\perp}$ cerrado, entonces $\Re \vdash_{CRWL} gen \to t$

Demostración. Sea $E_{\sum}$ el conjunto de constructores con aridad 0, $DC_{\sum}$ el conjunto de constructores con aridad ≥ 1 y sea $CTerm$ el siguiente conjunto:

$$CTerm ::= a_m \in E_{\sum}, 0 \leq m \leq n \,|\, c(t_1,...,t_n), c \in DC_{\sum}, 0 \leq m < n$$

y sea $\Re$ el programa que consta de la siguiente definición de reglas de gen:

$gen \rightarrow a_m$, con $a_m \in E_{\Sigma}, 0 \leq m \leq n$

$gen \rightarrow c(gen,...,gen)$, con $c \in DC_{\Sigma}, 0 \leq m < n$

La demostración se realiza por inducción sobre el tamaño del término t.

- Si $t = a_m$, con $a_m \in E_{\Sigma}, 0 \leq m \leq n$. Trivial.

- Si $t = c(t_1,...,t_n)$, la derivación, aplicando la regla OR es $c(gen,...,gen) \rightarrow c(t_1,...,t_n)$ y su derivación, aplicando DC es $gen \rightarrow t_i$ $0 \leq i \leq n$, que es cierto por hipótesis de inducción.

En el siguiente lema mostramos que si existen variables extra, al realizar la transformación del programa original, la semántica del programa original es equivalente a la semántica del programa transformado.

Lema 6. Sea $\Re$ un programa, $\Re'$ el programa transformado, $e \in Term_{\perp}$, y $t \in CTerm_{\perp}$. Si $Var(e \rightarrow t) = \emptyset$ entonces $\Re \vdash_{CRWL} e \rightarrow t \Leftrightarrow \Re' \vdash_{CRWL} e \rightarrow t$.

Demostración. (a) $\Rightarrow$ La demostración se realiza por inducción sobre la derivación de $\Re \vdash_{CRWL} e \rightarrow t$. Como e y t no tienen variables, podemos suponer de acuerdo al Lema 3, que las derivaciones son sin variables.

- Caso base: La derivación es de una de las formas siguientes:

  - Regla B: $e \rightarrow \perp$ trivial.

  - Regla RR: $e \rightarrow e$ trivial.

- Casos inductivos: Se distinguen dos subcasos, de acuerdo a la regla de reducción CRWL utilizada en el último paso de inferencia.

  - Regla DC: Si la derivación es

$$\frac{e_1 \rightarrow t_1 ... e_n \rightarrow t_n}{c(e_1,...,e_n) \rightarrow c(t_1,...,t_n)}$$

    es inmediato por la hipótesis de inducción.

- Regla OR: Si la derivación es

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; r\sigma \to t}{f(e_1,...,e_n) \to t}$$

siendo $f(t_1,...,t_n) \to r \in \Re$, $\sigma \in CSubst_\perp$, y $t_i\sigma$ $(1 \le i \le n)$, $r\sigma$ cerrados.

Sean $x_1 ... x_m$ las variables extra de r. Por el Lema anterior, se tiene $\Re' \vdash_{CRWL} gen \to x_i\sigma$ $(1 \le i \le n)$.
Por hipótesis de inducción tenemos $\Re' \vdash_{CRWL} r\sigma \to t$, y además $\Re' \vdash_{CRWL} t_i\sigma \to t_i\sigma$ $(1 \le i \le n)$ tenemos la siguiente derivación (con $\Re'$)

$$\frac{t_1\sigma \to t_1\sigma ... t_n\sigma \to t_n\sigma \; gen \to x_1\sigma ... gen \to x_m\sigma \; r\sigma \to t}{f'(t_1,...,t_n, gen, ... gen) \to t}$$

Por el resto de hipótesis de inducción, $\Re' \vdash_{CRWL} e_i \to t_i\sigma$ $(1 \le i \le n)$, y tenemos que

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; f'(t_1\sigma,...,t_n\sigma, gen, ... gen) \to t}{f(e_1,...,e_n) \to t}$$

ya que $f(t_1\sigma,...,t_n\sigma) \to f'(t_1\sigma,...,t_n\sigma, gen, ... gen) \in \Re'$.

(b) $\Leftarrow$ La demostración se realiza por inducción sobre la derivación de $\Re' \vdash_{CRWL} e \to t$. Como e y t no tienen variables, podemos suponer de acuerdo al Lema 3, que las derivaciones son sin variables.

- Caso base: La derivación tiene que tener una de las dos formas siguientes:

  - Regla B: $e \to \perp$ trivial.

  - Regla RR: $e \to e$ trivial.

- Casos inductivos: Se distinguen dos subcasos, de acuerdo a la regla de reducción CRWL utilizada en el último paso de inferencia.

  - Regla DC: Si la derivación es

$$\frac{e_1 \to t_1 ... e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$$

es inmediato por la hipótesis de inducción.

- Regla OR: Si la derivación es

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; r\sigma \to t}{f(e_1,...,e_n) \to t}$$

siendo $f(t_1,...,t_n) \to r \in \Re'$, $\sigma \in CSubst_\perp$, y $t_i\sigma$ $(1 \le i \le n)$, $r\sigma$ cerrados.

Puesto que $f(t_1\sigma,...,t_n\sigma) \to f'(t_1\sigma,...,t_n\sigma, gen,...gen) \in \Re'$ tenemos la siguiente derivación.

$$\frac{e_1 \to t_1\sigma ... e_n \to t_n\sigma \; f'(t_1\sigma,...,t_n\sigma, gen,...gen) \to t}{f(e_1,...,e_n) \to t}$$

Por hipótesis de inducción tenemos que $\Re \vdash_{CRWL} e \to t_i\sigma$ $(1 \le i \le n)$.

Para $f'(t_1\sigma,...,t_n\sigma, gen,...gen) \to t$ podemos obtener la siguiente derivación

$$\frac{t_1\sigma \to t_1\sigma ... t_n\sigma \to t_n\sigma \; gen \to x_1\sigma ... gen \to x_m\sigma \; r\sigma \to t}{f'(t_1,...,t_n, gen,...gen) \to t}$$

siendo $f(t_1,...,t_n, x_1,...,x_m) \to r \in \Re'$ y $x_1 ... x_m$ las variables extra de r.

Por hipótesis de inducción tenemos $\Re \vdash_{CRWL} r\sigma \to t$ y además $\Re \vdash_{CRWL} t_i\sigma \to t_i\sigma$ $(1 \le i \le n)$. Para finalizar, por el Lema Anterior, se tiene $\Re' \vdash_{CRWL} gen \to x_i\sigma$ $(1 \le i \le n)$ y por hipótesis de inducción se tiene también que $\Re \vdash_{CRWL} gen \to x_i\sigma$ $(1 \le i \le n)$.

Por último, y como principal resultado, se muestra que para derivaciones $e \to t$ con $Var(t) = \emptyset$, el programa original y el programa transformado son semánticamente equivalentes.

Teorema. Sea $\Re$ un programa, $\Re'$ el programa transformado, $e \in Term_\perp$, y $t \in CTerm_\perp$. Si $\Re \vdash_{CRWL} e \to t$ y $Var(t) = \emptyset$ entonces $\Re' \vdash_{CRWL} e \to t$.

Demostración. Sea $Var(e) = \{x_1,...,x_n\}$ y consideremos $\sigma = \{x_1/\perp,...,x_n/\perp\}$.

- Por el Lema 1, $\Re \vdash_{CRWL} e\sigma \to t\sigma$, es decir, $\Re \vdash_{CRWL} e\sigma \to t$, pues t es cerrado.
- Por el Lema 6, $\Re' \vdash_{CRWL} e\sigma \to t$, pues $e\sigma$ es cerrado.
- Por monotonía, Lema 2, $\Re' \vdash_{CRWL} e \to t$ ya que $e\sigma \sqsubseteq e$

Observemos que este teorema no es válido sin la condición $Var(t) = \emptyset$, como muestra el siguiente ejemplo:

EJEMPLO 4.3

Supongamos que la signatura consta sólo de la constante 0 y la constructora s, y que tenemos el programa

$\Re \equiv f \to X$

El programa transformado será:

$$
\begin{aligned}
\Re' \equiv\ & f \to gen \\
& gen \to 0 \\
& gen \to s(gen)
\end{aligned}
$$

Se tiene que $\Re \vdash_{CRWL} f \to X$, pero no $\Re' \vdash_{CRWL} f \to X$, sino que $\Re' \vdash_{CRWL} f \to 0$, $\Re' \vdash_{CRWL} f \to s(0)$, $\Re' \vdash_{CRWL} f \to s(s(0))$, ...

Observemos también que el anterior teorema tampoco es válido con $\bowtie$ y reglas condicionales, porque como vimos anteriormente el Lema 1 no es correcto. A continuación se muestra un ejemplo.

EJEMPLO 4.4

Sea $\Re \equiv f(X) \to 0 \Leftarrow X \bowtie Y$

Su programa transformado es:

$$
\begin{aligned}
\Re' \equiv\ & f(X) \to f'(X, gen) \\
& f'(X,Y) \to 0 \Leftarrow X \bowtie y
\end{aligned}
$$

Por tanto se tiene que $\Re \vdash_{CRWL} f(X) \to 0$, pero no $\Re' \vdash_{CRWL} f(X) \to 0$

5. Eliminación de variables extra en la práctica

5.1. Preliminares

Como hemos visto, la estrategia que se ha definido para eliminar las variables extra consiste en sustituir cada aparición de variable extra por una función generadora indeterminista que devuelva todos los valores (cerrados) posibles para cada variable extra.

En las secciones anteriores se ha estado utilizado sintaxis de primer orden, puesto que las bases formales estudiadas están limitadas a primer orden. Sin embargo, en esta sección los ejemplos utilizados son programas reales ejecutables para el lenguaje de programación lógico funcional TOY, para ello introducimos unos breves conceptos de la sintaxis de orden superior definida por este lenguaje. Para mayor detalle de la sintaxis de TOY véase [11]

La sintaxis abstracta de TOY para las expresiones es la siguiente:

$$e ::= X \mid c \mid f \mid (e\,e_1) \mid (e_1, \ldots, e_n)$$

En general las letras mayúsculas, X, representan variables. Se utiliza la notación $c \in DC^n$ para indicar que c es un constructor con n argumentos, $f \in FS^n$ para indicar que f es un símbolo de función con n parámetros formales. Las expresiones $(e_1, \ldots, e_n)\ n \geq 0$ representan tuplas y $(e\,e_1)$ representa la aplicación de la función e al argumento e_1.

Existen dos subclases de expresiones importantes, la primera, denominada términos de datos, se corresponde con la noción de patrón definida para los lenguajes funcionales como Haskell [8] y está definida mediante:

$$t ::= X \mid (t_1, \ldots, t_n) \mid c\,t_1 \ldots t_n\ (c \in DC^n)$$

La segunda subclase se denomina patrones, o también patrones funcionales o patrones de orden superior, y permiten representar valores funcionales como patrones. Se definen como sigue:

$$t ::= X \mid (t_1, \ldots, t_n) \mid c\,t_1 \ldots t_m\ (c \in DC^n, 0 \leq m \leq n) \mid f\ t_1 \ldots t_m\ (f \in FS^n, 0 \leq m < n)$$

La definición de los tipos de datos, denominados en TOY tipos construidos se realiza mediante constructores de datos y sus valores van precedidos por la palabra clave **data** y tienen el siguiente esquema:

$$\text{data}\ \delta\ \overline{A} = c_0 \overline{\tau}_0 \mid \ldots \mid c_{m-1} \overline{\tau}_{m-1}$$

donde δ es el constructor de tipo, $\overline{A}$ (que abrevia $A_1 \ldots A_n\ (n \geq 0)$) es una secuencia de variables de tipo diferentes que actúan como parámetros formales del tipo. $c_i\ (0 \leq i < m)$ son los constructores de datos para los valores

del tipo $\delta\ \overline{A}$ y $\overline{\tau_i}$, abreviatura de $\overline{\tau}_{i,1} \ldots \overline{\tau}_{i,n_i}$, es una serie de tipos correspondientes a los argumentos de c_i.

La definición de funciones en TOY consiste en una declaración de tipos opcional y una o más reglas definitorias, que son reglas de reescritura posiblemente condicionales. El esquema de una regla definitoria es el siguiente:

$$f \quad :: \tau_1 -> \ldots \tau_n -> \tau$$
$$\ldots$$
$$f\ t_1 \ldots t_n = r <== C_1, \ldots, C_m$$
$$\ldots$$

Para cada regla, la parte izquierda de la regla, $f\ t_1 \ldots t_n$, debe ser lineal, es decir, no puede existir variables repetidas, los parámetros formales, t_i, deben ser patrones y la parte derecha de la regla, r, puede ser cualquier expresión formada por variables, operaciones predefinidas, constructores de datos y funciones que existan en el programa. C_i son las condiciones, que en la parte teórica hemos eliminado para evitar complicaciones innecesarias pero que en la práctica podemos mantener.

5.2. El problema de los generadores

A continuación veremos que la definición de la función gen anteriormente mostrada, aunque es "teóricamente" correcta, en la práctica, cuando se realizan programas para implementaciones de lenguajes lógico funcionales, como TOY o Curry, el funcionamiento del generador no es el esperado, es decir, que la función no nos devolverá todos los valores posibles.

Supongamos el siguiente tipo de datos genérico:

data t = a_1| ...|a_n | c_1 t_{11} ... t_{1k1} | c_m t_{m1} ...t_{mkm}

siguiendo la definición de gen, la función generadora para este tipo de datos es:

gen = a_1
...
gen = a_n

gen = c_1 gen …… gen
....
gen = c_m gen ... ... gen

En la práctica, como veremos más adelante, este generador no siempre es completo. Existen ciertos casos en los cuales este generador devuelve todos los valores posibles: el primero se basa en que únicamente existen

constructores de aridad 0 en el universo de datos, y el segundo caso se basa en que el universo es infinito pero el tipo de datos tiene una única constructora de aridad 1. A continuación se muestran ejemplos de estos dos casos.

EJEMPLO 5.1

Se define el siguiente tipo de datos:

data t = a_1 | .... | a_n

aplicando la definición de gen, su función generadora es:

gen = a_1
....
gen = a_n

y mediante la ejecución de gen en una implementación lógico funcional se obtiene: a_1, ..., a_n

EJEMPLO 5.2

Se define el siguiente tipo de datos:

data t1 = a | c t1

aplicando la definición de gen, su función generadora es:

gen = a
gen = c gen

y su ejecución daría como resultado los siguientes términos:

a, (c a), (c (c a)), (c (c (c a))), ...

En el caso de que no ocurra ninguna de estas dos particularidades aunque es “teóricamente” completo, al ejecutarlo mediante una implementación de un lenguaje lógico funcional, como TOY o Curry, el generador no devuelve todos los valores, debido a cómo se recorre el espacio de búsqueda. El recorrido estándar de este tipo de lenguajes se basa en una estrategia de búsqueda primero en profundidad, pudiendo encontrarse realizando el recorrido en una rama infinita.

Si suponemos que tenemos más de un constructor en el tipo de datos, la definición de gen determina una función generadora que no devuelve todos los valores. Veámoslo a través de un ejemplo.

EJEMPLO 5.3

Sea la siguiente definición de tipo de datos:

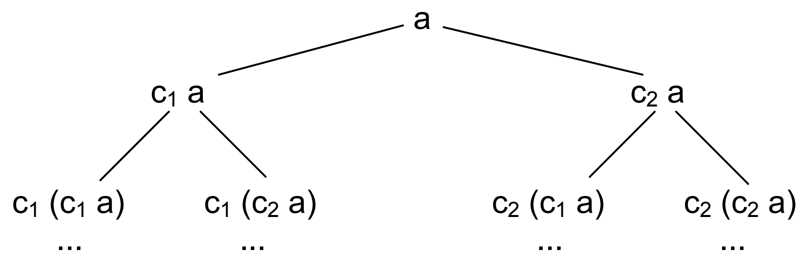
data t1 = a | c_1 t1 | c_2 t1

aplicando la definición de gen, su función generadora es:

gent1 = a

gent1 = c_1 gent1
gent1 = c_2 gent1

y su espacio de búsqueda se representa como:



en este caso su ejecución en TOY sería:

gent1 $\rightarrow$ a	por gent1, 0. (backtracking)
gent1 $\rightarrow$ c_1 gent1	por gent1, 1.
c_1 gent1 $\rightarrow$ c_1 a	por gent1, 0. (backtracking)
c_1 gent1 $\rightarrow$ c_1 (c_1 gent1)	por gent1, 1.

…

Es decir, los términos generados son: a, c_1 a, c_1 (c_1 a), c_1 (c_1 (c_1 a)), ...

Como puede verse, se recorre la rama más externa y más a la izquierda del espacio de búsqueda, que es infinita, y nunca se recorrerán las demás ramas del espacio de búsqueda. En este caso, los términos que se deberían obtener con el constructor c_2 nunca se obtendrán.

Esto también se produce si el tipo de datos tiene un constructor de aridad > 1.

EJEMPLO 5.4

Sean las siguientes definiciones de tipo de datos:

data t2 = a | c_1 t2 | c_2 t2
data t3 = b | c_3 t2 t3

aplicando la definición de la función gen, su función generadora sería:

gent2 = a

gent2 = c_1 gent2
gent2 = c_2 gent2

gent3 = b
gent3 = c_3 gent2 gent3

en este caso su ejecución en TOY sería:

gent3 $\rightarrow$ b	por gent3, 0. (backtracking)
gent3 $\rightarrow$ c_3 gent2 gent3	por gent3, 1.
c_3 gent2 gent3 $\rightarrow$ c_3 a gent3	por gent2, 0.
c_3 a gent3 $\rightarrow$ c_3 a b	por gent3, 0. (backtracking)
c_3 a gent3 $\rightarrow$ c_3 a (c3 gent2 gent3)	por gent3, 1.
c_3 a (c_3 gent2 gent3) $\rightarrow$ c_3 a (c_3 a gent3)	por gent2, 0.
c_3 a (c_3 a gent3) $\rightarrow$ c_3 a (c_3 a b)	por gent3, 0. (backtracking)
c_3 a (c_3 a gent3) $\rightarrow$ c_3 a (c_3 a (c_3 gent1 gent2))	
....	

Los términos generados son: b, c_3 a b, c_3 a (c_3 a b), c_3 a (c_3 a (c_3 a b)), …..

Nuevamente, el sistema recorre la rama más externa y a la izquierda del espacio de búsqueda, que es infinita.

Por tanto, ninguna de las definiciones anteriores de la función gen podría utilizarse en un programa para eliminar las variables extra puesto que no nos devolverá todos los posibles valores del universo.

Para que esto sea posible es necesario programar de otra forma los generadores para que, de forma "equitativa", produzcan todos los valores. A continuación veremos diversas estrategias de programar estos generadores.

5.3. Programación de generadores equitativos

Como hemos vistos anteriormente, la definición de la función gen nos permitía de forma "teórica" obtener todos los valores posibles del tipo de datos. Ahora bien, en la práctica, a la hora de utilizar estos generadores en programas que se ejecutan en implementaciones de lenguajes lógico funcionales, puede ser que no se obtengan realmente todos los valores, debido a cómo se recorre el espacio de búsqueda.

A continuación se muestran varias alternativas de implementaciones de generadores siguiendo distintos criterios.

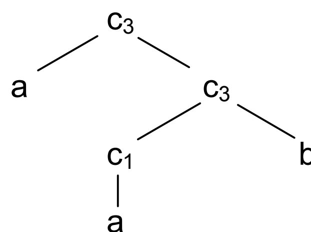
5.3.1. Programación de generadores basados en profundidad creciente

Esta alternativa consiste en programar un generador que produce términos de profundidad creciente, donde la profundidad de un término es la profundidad del árbol que lo representa.

Por ejemplo, supongamos los siguientes tipos de datos:

data t1 = a | c_1 t1 | c_2 t1
data t2 = b | c_3 t1 t2

Todo término se puede representar como un árbol. Por ejemplo, en el caso del término c_3 a (c_3 (c_1 a) b) del tipo t2 se representaría por el siguiente árbol:



y su profundidad sería 3.

Los términos para el tipo t2 quedaría jerarquizada por profundidad como sigue:

Profundidad 0: b
Profundidad 1: (c_3 a b)
Profundidad 2: (c_3 a (c_3 a b)), (c_3 (c_1 a) b), (c_3 (c_2 a) b), (c_3 (c_1 a) (c_3 a b)),
(c_3 (c_2 a) (c_3 a b))
....

Nótese que sólo hay un número finito de términos en cada profundidad k.

La función generadora para el tipo t1 puede definirse a través de las siguientes funciones:

- gen_t1 : genera los términos del tipo t1
- gen_t1' N : genera los términos del tipo t1 de profundidad >= N
- gen_t1'' N : genera los términos del tipo t1 de profundidad N
- gen_t1''' N : genera los términos del tipo t1 de profundidad =< N
- gen_t1'''' N : genera los términos del tipo t1 de profundidad < N

% Utilizamos nat para fijar las profundidades
dat nat = z | s nat

% Genera todos los términos de tipo t1
gen_t1 = gen_t1' z

```
% Genera los términos de tipo t1 de profundidad >= N
gen_t1' N = gen_t1'' N // gen_t1' (s N)


% Genera los términos de tipo t1 de profundidad N
gen_t1'' z = a
gen_t1'' (s N) = c1 (gen_t1'' N) // c2 (gen_t1'' N)

% También se podría utilizar la siguiente variante
gen_t1''bis (s N) = (c1 // c2) (gen_t1''bis N)

% Genera los términos de tipo t1 de profundidad <=N
gen_t1''' N = a
gen_t1''' (s N) = c1 (gen_t1''' N) // c2 (gen_t1''' N)

% Genera los términos de tipo t1 de profundidad <N
gen_t1'''' (s N) = gen_t1''' N
```

donde:

```
% Función de elección indeterminista
X // Y = X
X // Y = Y
```

La función generadora para el tipo t2 es similar a la función generadora del tipo t1, y puede definirse a través de las siguientes funciones:

- gen_t2 : genera los términos del tipo t2
- gen_t2' N : genera los términos del tipo t2 de profundidad >= N
- gen_t2'' N : genera los términos del tipo t2 de profundidad N
- gen_t2''' N : genera los términos del tipo t2 de profundidad =< N
- gen_t2'''' N : genera los términos del tipo t2 de profundidad < N

```
% Genera todos los términos de tipo t2
gen_t2 = gen_t2' z

% Genera los términos de tipo t2 de profundidad >= N
gen_t2' N = gen_t2'' N // gen_t2' (s N)


% Genera los términos de tipo t2 de profundidad N
gen_t2'' z = b
gen_t2'' (s N) = c3 (gen_t1'' N) (gen_t2''' N)
                  //
                 c3 (gen_t1''' N) (gen_t2'' N)
```

```
% Genera los términos de tipo t2 de profundidad <=N
gen_t2''' N = b
gen_t2''' (s N) = c3 (gen_t1''' N) (gen_t2''' N)
```

Esta función generadora produce algunas repeticiones, que pueden evitarse si definimos el generador de términos de profundidad N como:

```
% Genera los términos de tipo t2 de profundidad N (sin repeticiones)
gen_t2''bis z = b
gen_t2''bis (s N) = c3 (gen_t1'' N) (gen_t2''' N)
                    //
                   c3 (gen_t1'''' N) (gen_t2''bis N)

gen_t2bis = gen_t2'bis z
gen_t2'bis N = gen_t2''bis N // gen_t2'bis (s N)
```

También es posible realizar una variante de gen_t2''', realizándola de una manera más generalizable, basándonos en gen_t2''

```
var_gen_t2''' z = b
var_gen_t2''' (s N) = gen_t2''bis (s N) // var_gen_t2''' N
```

Como puede observarse, este esquema de función generadora nos aporta una propiedad muy importante. Esta propiedad radica en que es generalizable para cualquier tipo de datos, ya que únicamente es necesario sustituir las funciones generadoras ('') y ('''), es decir, las funciones generadoras de los términos de profundidad N y profundidad <=N para que generen los términos del tipo de datos elegido. Por desgracia, lo que no es posible es realizar un generador polimórfico.

5.3.2. Programación de generadores basados en número de símbolos

Esta programación consiste en definir un generador en el cual el número de símbolos del término nos permite generar en cada paso un número finito de términos.

Nuevamente, supongamos los siguientes tipos de datos:

data t1 = a | c_1 t1 | c_2 t1
data t2 = b | c_3 t1 t2

Según el número de símbolos utilizados tenemos:

1 símbolo: b
3 símbolos: (c_3 a b)
4 símbolos: (c_3 (c_1 a) b), (c_3 (c_2 a) b)

5 símbolos: (c_3 a (c_3 a b))
6 símbolos: (c_3 a (c_3 (c_1 a) b)), (c_3 a (c_3 (c_2 a) b)), (c_3 (c_1 a) (c_3 a b)),
(c_3 (c_2 a) (c_3 a b))
….

La idea del generador basado en el número de símbolos consiste en utilizar la cardinalidad de los símbolos del término como criterio de tamaño e ir incrementando este valor, con lo cual obtendremos un número de términos finitos.

En este caso, la función generadora para el tipo t1 puede definirse como:

- s_gen_t1 : genera los términos del tipo t1
- s_gen_t1' N : genera los términos del tipo t1 con >= N símbolos
- s_gen_t1" N : genera los términos del tipo t1 con N símbolos
- s_gen_t1'" N : genera los términos del tipo t1 con =< N símbolos
- s_gen_t1"" N : genera los términos del tipo t1 con < N símbolos

```
% Genera todos los términos de tipo t1
s_gen_t1 = s_gen_t1' (s z)


% Genera los términos de tipo t1 con >= N símbolos
s_gen_t1' N = s_gen_t1" N // s_gen_t1' (s N)


% Genera los términos de tipo t1 con N símbolos
s_gen_t1" (s z) = a
s_gen_t1" (s (s N)) = c1 (s_gen_t1" (s N)) // c2 (s_gen_t1" (s N))

% Genera los términos de tipo t1 con <=N símbolos
s_gen_t1'" (s z) = a
s_gen_t1'" (s (s N)) = s_gen_t1" (s (s N)) // s_gen_t1"" (s (s N))

% Genera los términos de tipo t1 con <N símbolos
s_gen_t1"" (s N) = s_gen_t1'" N
```

donde:

```
% Función de elección indeterminista
X // Y = X
X // Y = Y
```

La función generadora para el tipo t2, se define como:

- s_gen_t2: genera términos del tipo t2
- s_gen_t2' N: genera términos del tipo t2 con >= N símbolos
- s_gen_t2" N: genera términos del tipo t2 con N símbolos
- s_gen_t2'" N M: genera términos del tipo t2 con N + M símbolos

```
% Genera todos los términos de tipo t2
s_gen_t2 = s_gen_t2' (s z)


% Genera los términos de tipo t2 con >= N símbolos
s_gen_t2' N = s_gen_t2'' N // s_gen_t2' (s N)

% Genera los términos de tipo t2 con N símbolos
s_gen_t2'' (s z) = b
s_gen_t2'' (s (s (s N))) = s_gen_t2''' (s z) (s (s N))

% Genera los términos de tipo t2 con N + (s (s M)) símbolos
s_gen_t2''' N (s (s M)) = c3 (s_gen_t1'' N) (s_gen_t2'' (s M))
                          //
                          s_gen_t2''' (s N) (s M)
```

Nuevamente, este esquema de función generadora permite generalizarse para cualquier tipo de datos, sustituyendo únicamente la función generadora ('''), para que generen los términos del tipo de datos elegido.

5.3.3. Programación de generadores basados en número de constructores

Esta alternativa es una variante de la anterior, en este caso, en lugar de utilizar como criterio el número de símbolos del término, se utiliza el número de constructores utilizados en el término.

Otra vez, supongamos los siguientes tipos de datos:

data t1 = a | c_1 t1 | c_2 t1
data t2 = b | c_3 t1 t2

Según el número de constructores del término tenemos:

0 constructores: a
1 constructor: (c_3 a b)
2 constructores: (c_3 a (c_3 a b)), (c_3 (c_1 a) b), (c_3 (c_2 a) b)
3 constructores: (c_3 a (c_3 a (c_3 a b))), (c_3 a (c_3 (c_1 a) b)), (c_3 a (c_3 (c_2 a) b)),
(c_3 (c_1 a) (c_3 a b)), (c_3 (c_2 a) (c_3 a b)), (c_3 (c_1 (c_1 a)) b),
(c_3 (c_1 (c_2 a)) b), (c_3 (c_2 (c_1 a)) b), (c_3 (c_2 (c_2 a)) b)
….

La idea del generador basado en el número de constructores es similar a la basada en el número de símbolos y consiste en utilizar el número de constructores del término como criterio de tamaño e ir incrementando este valor, con lo cual obtendremos un número de términos finitos.

Igual que en la alternativa anterior, la función generadora para el tipo t1 puede definirse como:

- c_gen_t1 : genera los términos del tipo t1
- c_gen_t1' N : genera los términos del tipo t1 con >= N constructores
- c_gen_t1'' N : genera los términos del tipo t1 con N constructores
- c_gen_t1''' N : genera los términos del tipo t1 con =< N constructores
- c_gen_t1'''' N: genera los términos del tipo t1 con < N constructores

```
% Genera todos los términos de tipo t1
c_gen_t1 = c_gen_t1' z


% Genera los términos de tipo t1 con >= N constructores
c_gen_t1' N = c_gen_t1'' N // c_gen_t1' (s N)


% Genera los términos de tipo t1 con N constructores
c_gen_t1'' z = a
c_gen_t1'' (s N) = c1 (c_gen_t1'' N) // c2 (c_gen_t1'' N)



% Genera los términos de tipo t1 con <=N constructores
c_gen_t1''' N = a
c_gen_t1''' (s N) = c1 (c_gen_t1''' N) // c2 (c_gen_t1''' N)


% Genera los términos de tipo t1 con <N constructores
c_gen_t1'''' (s N) = c_gen_t1''' N
```

donde:

```
% Función de elección indeterminista
X // Y = X
X // Y = Y
```

La función generadora para el tipo t2, vendría definida como:

- c_gen_t2: genera términos del tipo t2
- c_gen_t2' N: genera términos del tipo t2 con >= N constructores
- c_gen_t2'' N: genera términos del tipo t2 con N constructores
- c_gen_t2''' N M: genera términos del tipo t2 con N + M constructores

```
% Genera todos los términos de tipo t2
c_gen_t2 = c_gen_t2' (s z)

% Genera los términos de tipo t2 con >= N constructores
```

```
c_gen_t2' N = c_gen_t2'' N // c_gen_t2' (s N)


% Genera los términos de tipo t2 con N constructores
c_gen_t2'' z = b
c_gen_t2'' (s N) = c_gen_t2''' z (s N)


% Genera los términos de tipo t2 con N + (s M) constructores
c_gen_t2''' N (s M) = c3 (c_gen_t1'' N) (c_gen_t2'' M)
                      //
                      c_gen_t2''' (s N) M
```

5.3.4. Programación de generadores basados en pesos de los constructores

Esta programación consiste en definir un generador basándonos en unos pesos asignados a cada uno de los constructores de los cuales se compone el tipo de datos a generar. La suma de los pesos de los constructores que contiene el término nos permite generar en cada paso un número finito de términos.

Supongamos los siguientes tipos de datos:

data t1 = a | c_1 t1
data t2 = b | c_2 t1 t2 | c_3 t1 t2
data t3 = m | c_4 t2 t3

En este caso, asignamos los siguientes pesos a los constructores:

constructor c_1 , peso 1
constructor c_2 , peso 2
constructor c_3 , peso 3
constructor c_4 , peso 4

Según la suma de pesos de los constructores que existen en el término tenemos:

Suma de Pesos 0 : m
Suma de Pesos 4 : c_4 b m
Suma de Pesos 6 : c_4 (c_2 a b) m
Suma de Pesos 7 : c_4 (c_3 a b) m, c_4 (c_2 (c_1 a) b) m
Suma de Pesos 8 : c_4 b (c_4 b m), c_4 (c_3 (c_1 a) b) m, c_4 (c_2 a (c_2 a b)) m,
c_4(c_2 (c_1 (c_1 a)) b) m
….
En este caso, se observa que el peso del constructor coincide con el subíndice

de dicho constructor, por tanto, se puede comprobar fácilmente que para los términos generados para un valor de suma de pesos, la suma de los pesos de los constructores que forman parte del término coincide con el valor de la suma de pesos.

La idea del generador basado en pesos de los constructores consiste en utilizar ciertos pesos asignados a los constructores como criterio de tamaño e ir incrementando este valor, con lo cual obtendremos un número de términos finitos.

En principio, la asignación de los pesos de los constructores es irrelevante, pero como veremos en los ejemplos, para que la creación de las funciones generadoras sea más fácil, la asignación de pesos de los constructores se realizará asignando los pesos de forma creciente según el grado de anidamiento de los constructores del tipos de datos, es decir, como hemos visto anteriormente, para el tipo de datos más interno, t1, el constructor c_1 se le asigna el peso 1, para el siguiente tipo de datos t2, el constructor c_2 se le asigna el peso 2, y así sucesivamente.

A continuación veremos como se definen las funciones generadoras para estos tipos de datos

En este caso, la función generadora para el tipo t1 puede definirse como:

- p_gen_t1 : genera los términos del tipo t1
- p_gen_t1' N : genera los términos del tipo t1 con suma de pesos >= N
- p_gen_t1'' N : genera los términos del tipo t1 con suma de pesos N
- p_gen_t1''' N : genera los términos del tipo t1 con suma de pesos =< N
- p_gen_t1'''' N: genera los términos del tipo t1 con suma de pesos < N

```
% Genera todos los términos de tipo t1
p_gen_t1 = p_gen_t1' z

% Genera los términos de tipo t1 con suma de pesos >= N
p_gen_t1' N = p_gen_t1'' N // p_gen_t1' (s N)

% Genera los términos de tipo t1 con suma de pesos N
p_gen_t1'' z = a
p_gen_t1'' (s N) = c1 (p_gen_t1'' N)

% Genera los términos de tipo t1 con suma de pesos <=N
p_gen_t1''' N = a
p_gen_t1''' (s N) = c1 (p_gen_t1''' N)

% Genera los términos de tipo t1 con suma de pesos <N
p_gen_t1'''' (s N) = p_gen_t1''' N
donde:
```

```
% Función de elección indeterminista
X // Y = X
X // Y = Y
```

La función generadora para el tipo t2, se define como:

- p_gen_t2: genera términos del tipo t2
- p_gen_t2' N: genera términos del tipo t2 con suma de pesos >= N
- p_gen_t2'' N: genera términos del tipo t2 con suma de pesos N
- p_gen_t2_c2''' N M: genera términos del tipo t2 con suma de pesos N+M
- p_gen_t2_c3''' N M: genera términos del tipo t2 con suma de pesos N+M

```
% Genera todos los términos de tipo t2
p_gen_t2 = p_gen_t2' (s z)

% Genera los términos de tipo t2 con suma de pesos >= N
p_gen_t2' N = p_gen_t2'' N // p_gen_t2' (s N)

% Genera los términos de tipo t2 con suma de pesos N
p_gen_t2'' z = b
p_gen_t2'' (s (s N)) = p_gen_t2_c2'''  z N
p_gen_t2'' (s (s (s N))) = p_gen_t2_c3'''  z N


% Genera los términos de tipo t2 con suma de pesos N + M
p_gen_t2_c2''' N z = c2 (p_gen_t1'' N) b
p_gen_t2_c2''' N (s M) = c2 (p_gen_t1'' N) (p_gen_t2'' (s M))
                         //
                         p_gen_t2_c2''' (s N) M

% Genera los términos de tipo t2 con suma de pesos N + M
p_gen_t2_c3''' N z = c3 (p_gen_t1'' N) b
p_gen_t2_c3''' N (s M) = c3 (p_gen_t1'' N) (p_gen_t2'' (s M))
                         //
                         p_gen_t2_c3''' (s N) M
```

La función generadora para el tipo t3, se define como:

- p_gen_t3: genera términos del tipo t3
- p_gen_t3' N: genera términos del tipo t3 con suma de pesos >= N
- p_gen_t3'' N: genera términos del tipo t3 con suma de pesos N
- p_gen_t3''' N M: genera términos del tipo t3 con suma de pesos N + M

donde:

```
% Genera todos los términos de tipo t3
p_gen_t3 = p_gen_t3' z
```

```
% Genera los términos de tipo t3 con suma de pesos >= N
p_gen_t3' N = p_gen_t3'' N // p_gen_t3' (s N)

% Genera los términos de tipo t3 con suma de pesos N
p_gen_t3'' z = m
p_gen_t3'' (s (s (s (s N)))) = p_gen_t3''' z (s N)

% Genera los términos de tipo t3 con suma de pesos N + M
p_gen_t3''' N (s M) = c4 (p_gen_t2'' N) (p_gen_t3'' M) // p_gen_t3''' (s N) M
```

Si nos detenemos en las funciones generadoras que determinan términos con peso N de cada tipo, por ejemplo para el tipo de datos

data t2 = b | c_2 t1 t2 | c_3 t1 t2

```
% Genera los términos de tipo t2 con suma de pesos N
p_gen_t2'' z = b
p_gen_t2'' (s (s N)) = p_gen_t2_c2'''  z N
p_gen_t2'' (s (s (s N))) = p_gen_t2_c3'''  z N
```

Podemos observar que en el tipo de datos t2 el constructor c_2 tiene asignado un peso 2 y el constructor c_3 tiene asignado un peso 3, el argumento de la función generadora para este tipo de datos viene determinado por dos patrones. Estos patrones se corresponden con los pesos de los correspondientes constructores. Los patrones de datos son los siguientes:

- (s (s N)): Genera todos los términos de tipo t2 formados por el constructor c_2 y donde sus términos argumento t1 y t2 son términos cuya suma de pesos es N.

- (s (s (s N))): Genera todos los términos de tipo t2 formados por el constructor c_3 y donde sus términos argumento t1 y t2 son términos cuya suma de pesos es N.

Para el tipo de datos t3, el constructor c_4 tiene un peso 4, de tal forma que su función generadora es:

data t3 = m | c_4 t2 t3

```
% Genera los términos de tipo t3 con suma de pesos N
p_gen_t3'' z = m
p_gen_t3'' (s (s (s (s N)))) = p_gen_t3''' z (s N)
```

en este caso, existe un único constructor c_4 que tiene asignado un peso 4, por tanto, solo es necesario un patrón de datos en la función generadora correspondiente al peso 4 del constructor:

- (s (s (s (s N)))): Genera todos los términos de tipo t3 formados por el constructor c_4 y donde sus términos argumento t2 y t3 son términos cuya suma de pesos es N.

5.4. Ejemplos de generadores

A continuación se muestran varios ejemplos utilizando funciones generadoras indeterministas.

Ejemplo 1.- Tipo de datos sólo con constructores de aridad 0 en el universo de datos.

En este ejemplo, la función generadora se realiza siguiendo la definición de gen.

Sea el siguiente programa:

```
father ::  [ char ] ->  [ char ] -> bool
father "abraham" "isaac"   = true
father "haran"   "lot"     = true
father "haran"   "milcah"  = true
father "haran"   "yiscah"  = true

grandfather ::  [ char ] ->  [ char ] -> bool
grandfather X Z = true <== father X Y, father Y Z
```

En este caso, la variable extra es Y, que aparece en la declaración grandfather.

La función generadora para el tipo de datos [char] es:

```
gen_grandfather ::  [ char ]
gen_grandfather = "abraham"
gen_grandfather = "isaac"
gen_grandfather = "haran"
gen_grandfather = "lot"
gen_grandfather = "haran"
gen_grandfather = "milcah"
gen_grandfather = "haran"
gen_grandfather = "yiscah"
```

y el programa transformado es:

```
grandfather' ::  [ char ] ->  [ char ] -> bool
grandfather' X Z = true <== grandfather'' X Z gen_grandfather

grandfather'' ::  [ char ] ->  [ char ] ->  [ char ] -> bool
grandfather'' X Z Y  = true <== father X Y, father Y Z
```

Ejemplo 2.- Tipo de datos con un único constructor y con aridad = 1

Sea el tipo de datos:

```
data nat = c | s nat
```

y la función times definida como:

```
plus :: nat -> nat -> nat -> bool
plus c     X  Y    = true <== X == Y
plus (s X) Y (s Z) = true <== plus X Y Z

times :: nat -> nat -> nat -> bool
times  c    X c = true
times (s X) Y Z = true <== times X Y W, plus W Y Z
```

la variable extra es W que se encuentra en la declaración de times, para eliminarla, creamos una función generadora del tipo de datos nat:

```
gen_nat :: nat
gen_nat = c
gen_nat = s (gen_nat)
```

y aplicamos la transformación obteniendo una nueva función times':

```
times' :: nat -> nat -> nat -> bool
times'  c    X c    = true
times' (s X) Y Z    = true <== times'' (s X) Y Z gen_nat

times'' :: nat -> nat -> nat -> nat -> bool
times''(s X) Y Z W  = true <== times' X Y W, plus W Y Z
```

Ejemplo 3.- Tipo de datos con varios constructores

Supongamos el siguiente tipo de datos, con dos constructores:

```
data either = void | left either | right either
```

y el tipo de datos nat:

```
data nat = z | s nat
```

y la función de elección indeterminista

```
// :: _A -> _A -> _A
X // Y = X
X // Y = Y
```

Como hemos visto anteriormente, simplemente con la definición de gen no es posible obtener una función generadora completa. En este caso, se utilizará la estrategia de la definir la función generadora por niveles de profundidad.

La función generadora para el tipo de datos either es:

```
% Genera todos los términos del tipo either
gen_either :: either
gen_either = gen_either' z

% Genera los términos del tipo either de profundidad >= N
gen_either' :: nat -> either
gen_either' N = gen_either'' N // gen_either' (s N)

% Genera los términos del tipo either de profundidad N
gen_either'' :: nat -> either
gen_either'' z = void
gen_either'' (s N) = left (gen_either'' N) // right (gen_either'' N)

% Genera los términos del tipo either de profundidad < N
gen_either''' :: nat -> either
gen_either''' N = void
gen_either''' (s N) = left (gen_either''' N)  //  right (gen_either''' N)

% Genera los términos del tipo either de profundidad <= N
gen_either'''' :: nat -> either
gen_either'''' (s N) = gen_either''' N
```

Ejemplo 4.- Función generadora “equitativa”

Sean los tipos de datos:

```
data either = void | left either | right either

data nat = z | s nat
```

y la función de elección indeterminista

```
// :: _A -> _A -> _A
X // Y = X
X // Y = Y
```

La función auxiliar append definida como:

```
append ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
append [] Ys Zs = true <== Ys == Zs
append [X|Xs] Ys [X|Zs] = true <== append Xs Ys Zs
```

y la función last donde se encuentra la variable extra Ys

```
last :: _A ->  [ _A ] -> bool
last  X Xs     = true <== append Ys [X] Xs
```

A continuación se detallarán los generadores de los tipos de datos para either y las listas y los programas transformados utilizando las distintas estrategias de profundidad, número de símbolos y número de constructores.

Generadores por nivel de profundidad

Para el generador por nivel de profundidad del tipo de datos either véase el Ejemplo 3.

El generador para el tipo de datos de las listas es:

```
% Genera todos los términos del tipo listas de elementos de tipo either
gen_listas ::  [ either ]
gen_listas = gen_listas' z

% Genera los términos del tipo listas de profundidad >= N
gen_listas' :: nat ->  [ either ]
gen_listas' N = gen_listas'' N  // gen_listas' (s N)

% Genera los términos del tipo listas de profundidad N
gen_listas'' :: nat ->  [ either ]
gen_listas'' z = []
gen_listas'' (s N) = [(gen_either'' N)|(gen_listas''' N)]
                     //
                     [(gen_either'''' N)|(gen_listas'' N)]
```

```
% Genera los términos del tipo listas de profundidad <=N
gen_listas''' :: nat ->  [ either ]
gen_listas''' z = []
gen_listas''' (s N) = gen_listas'' (s N)  //  gen_listas''' N
```

y la transformación de last utilizando este generador es:

```
last' :: either ->  [ either ] -> bool
last'  X Xs     = true <== last'' X  Xs gen_listas

last'' :: _A ->  [ _A ] ->  [ _A ] -> bool
last'' X Xs Ys  = true <== append Ys [X] Xs
```

Como puede observarse, la función gen_listas’’ que es la encargada de generar los términos del tipo de listas de profundidad N es la única que necesitar utilizar información del tipo de datos para el cual se generarán los términos. Las restantes funciones, son funciones auxiliares que nos permitirán generar todos los términos.

Generadores por número de símbolos

El generador para el tipo de datos either:

```
% Genera todos los términos del tipo either
s_gen_either :: either
s_gen_either = s_gen_either' (s z)

% Genera los términos del tipo either con >= N símbolos
s_gen_either' :: nat -> either
s_gen_either' N = s_gen_either'' N //  s_gen_either' (s N)

% Genera los términos del tipo either con N símbolos
s_gen_either'' :: nat -> either
s_gen_either'' (s z) = void
s_gen_either'' (s (s N)) = left (s_gen_either'' (s N))  // right (s_gen_either'' (s N))
```

El generador para el tipo de datos de las listas es:

```
% Genera todos los términos del tipo listas
s_gen_listas ::  [ either ]
s_gen_listas = s_gen_listas' (s z)


% Genera los términos del tipo listas con >= N símbolos
s_gen_listas' :: nat ->  [ either ]
s_gen_listas' N = s_gen_listas'' N // s_gen_listas' (s N)
```

```
% Genera los términos del tipo listas con N símbolos
s_gen_listas'' :: nat ->  [ either ]
s_gen_listas'' (s z) = []
s_gen_listas'' (s (s (s N))) = s_gen_listas''' (s z) (s N)


% Genera los términos del tipo listas, con N símbolos del tipo either
% y M símbolos del tipo listas
s_gen_listas''' :: nat -> nat ->  [ either ]
s_gen_listas''' N (s M) = [(s_gen_either'' N)|(s_gen_listas'' (s M))]
                          //
                          s_gen_listas''' (s N) M
```

y la transformación de last utilizando este generador es:

```
s_last' :: either ->  [ either ] -> bool
s_last'  X Xs     = true <== s_last'' X  Xs s_gen_listas

s_last'' :: _A ->  [ _A ] ->  [ _A ] -> bool
s_last'' X Xs Ys  = true <== append Ys [X] Xs
```

Generadores por número de constructores

El generador para el tipo de datos either es:

```
% Genera todos los términos del tipo either
c_gen_either :: either
c_gen_either = c_gen_either' z

% Genera los términos del tipo either con >= N constructores
c_gen_either' :: nat -> either
c_gen_either' N = c_gen_either'' N // c_gen_either' (s N)

% Genera los términos del tipo either con N constructores
c_gen_either'' :: nat -> either
c_gen_either'' z = void
c_gen_either'' (s N) = left (c_gen_either'' N)  //  right (c_gen_either'' N)
```

El generador para el tipo de datos de las listas es:

```
% Genera todos los términos del tipo listas
c_gen_listas ::  [ either ]
c_gen_listas = c_gen_listas'  z

% Genera los términos del tipo listas con >= N constructores
c_gen_listas' :: nat ->  [ either ]
c_gen_listas' N = c_gen_listas'' N // c_gen_listas' (s N)
```

```
% Genera los términos del tipo listas con N constructores
c_gen_listas'' :: nat ->  [ either ]
c_gen_listas'' z = []
c_gen_listas'' (s N) = c_gen_listas''' z (s N)

% Genera los términos del tipo listas, con N constructores del tipo either
% y M constructores del tipo listas
c_gen_listas''' :: nat -> nat ->  [ either ]
c_gen_listas''' N (s M) = [(c_gen_either'' N)|(c_gen_listas'' M)]
                          //
                          c_gen_listas''' (s N) M
```

y la transformación de last utilizando este generador es:

```
c_last' :: either ->  [ either ] -> bool
c_last'  X Xs     = true <== c_last'' X  Xs c_gen_listas
c_last'' :: _A ->  [ _A ] ->  [ _A ] -> bool
c_last'' X Xs Ys  = true <== append Ys [X] Xs
```

Generadores por pesos en constructores

Supongamos los siguientes pesos para los constructores de los tipos de datos:

Tipo de datos either : Constructor left. Peso 1. Constructor right Peso 2
Tipo de datos listas: Constructor listas (:).   Peso 3

El generador para el tipo de datos either es:

```
% Genera todos los términos del tipo either
p_gen_either :: either
p_gen_either = p_gen_either' z

% Genera los términos del tipo either con suma de pesos >= N
p_gen_either' :: nat -> either
p_gen_either' N = p_gen_either'' N // p_gen_either' (s N)

% Genera los términos del tipo either con suma de pesos N
p_gen_either'' :: nat -> either
p_gen_either'' z = void
p_gen_either'' (s N) = left (p_gen_either'' N)
p_gen_either'' (s (s N)) = right (p_gen_either'' N)
```

El generador para el tipo de datos de las listas es:

```
% Genera todos los términos del tipo listas
p_gen_listas ::  [ either ]
p_gen_listas = p_gen_listas' z
```

```
% Genera los términos del tipo listas con suma de pesos >= N
p_gen_listas' :: nat ->  [ either ]
p_gen_listas' N = p_gen_listas'' N // p_gen_listas' (s N)

% Genera los términos del tipo listas con suma de pesos N
p_gen_listas'' :: nat ->  [ either ]
p_gen_listas'' z = []
p_gen_listas'' (s (s (s N))) = p_gen_listas''' z (s N)

% Genera los términos del tipo listas, con suma de pesos N del tipo either
% y suma de pesos M del tipo listas
p_gen_listas''' :: nat -> nat ->  [ either ]
p_gen_listas''' N (s M) = [(p_gen_either'' N)|(p_gen_listas'' M)]
                          //
                          p_gen_listas''' (s N) M
```

y la transformación de last utilizando este generador es:

```
p_last' :: either ->  [ either ] -> bool
p_last'  X Xs     = true <== p_last'' X  Xs p_gen_listas
p_last'' :: _A ->  [ _A ] ->  [ _A ] -> bool
p_last'' X Xs Ys  = true <== append Ys [X] Xs
```

Ejemplo 5.- Función generadora "equitativa"

Sea el tipo de datos:

```
data nat = z | s nat
```

y la función de elección indeterminista

```
// :: _A -> _A -> _A
X // Y = X
X // Y = Y
```

Las funciones auxiliares

```
preffix ::  [ _A ] ->  [ _A ] -> bool
preffix [] Ys         = true
preffix [X|Xs] [Y|Ys]  = preffix Xs Ys <== X == Y

suffix ::  [ _A ] ->  [ _A ] -> bool
suffix Xs Ys          = true <== Xs == Ys
suffix Xs [Y|Ys]      = suffix Xs Ys
```

y la función sublist donde se encuentra la variable extra Ps

```
sublist ::  [ _A ] ->  [ _A ] -> bool
sublist  Xs Ys   :- preffix Ps Ys, suffix Xs Ps
```

A continuación se detallarán los generadores de los tipos de datos para nat y las listas y los programas transformados utilizando las distintas estrategias de profundidad, número de símbolos y número de constructores.

Generadores por nivel de profundidad

El generador para el tipo de datos nat:

```
% Genera todos los términos del tipo nat
gen_nat :: nat
gen_nat = gen_nat' z

% Genera los términos del tipo nat de profundidad >= N
gen_nat' :: nat -> nat
gen_nat' N = gen_nat'' N // gen_nat' (s N)

% Genera los términos del tipo nat de profundidad N
gen_nat'' :: nat -> nat
gen_nat'' z = z
gen_nat'' (s N) = s (gen_nat'' N)
```

```
% Genera los términos del tipo nat de profundidad < N
gen_nat''' :: nat -> nat
gen_nat''' N = z
gen_nat''' (s N) = s (gen_nat''' N)

% Genera los términos del tipo either de profundidad <= N
gen_nat'''' :: nat -> nat
gen_nat'''' (s N) = gen_nat''' N
```

Para el generador por nivel de profundidad del tipo de datos listas véase el Ejemplo 4.

La transformación de sublist utilizando este generador es:

```
sublist' ::  [ nat ] ->  [ nat ] -> bool
sublist'  Xs Ys     :- sublist'' Xs Ys gen_listas

sublist'' ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps
```

Generadores por número de símbolos

El generador para el tipo de datos nat:

```
% Genera todos los términos del tipo nat
s_gen_nat :: nat
s_gen_nat = s_gen_nat' (s z)

% Genera los términos del tipo nat con >= N símbolos
s_gen_nat' :: nat -> nat
s_gen_nat' N = s_gen_nat'' N // s_gen_nat' (s N)

% Genera los términos del tipo nat con N símbolos
s_gen_nat'' :: nat -> nat
s_gen_nat'' (s z) = (s z)
s_gen_nat'' (s (s N)) = s (s_gen_nat'' (s N))
```

Para el generador por número de símbolos del tipo de datos listas véase el Ejemplo 4.

Y la transformación de sublist utilizando este generador es:

```
s_sublist' ::  [ nat ] ->  [ nat ] -> bool
s_sublist'  Xs Ys     :- s_sublist'' Xs Ys s_gen_listas

s_sublist'' ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
s_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps
```

Generadores por número de constructores

El generador para el tipo de datos nat es:

```
% Genera todos los términos del tipo nat
c_gen_nat :: nat
c_gen_nat = c_gen_nat' z

% Genera los términos del tipo nat con número de constructores >= N
c_gen_nat' :: nat -> nat
c_gen_nat' N = c_gen_nat'' N // c_gen_nat' (s N)

% Genera los términos del tipo nat con número de constructores N
c_gen_nat'' :: nat -> nat
c_gen_nat'' z = z
c_gen_nat'' (s N) = s (c_gen_nat'' N)

% Genera los términos del tipo nat con número de constructores  < N
c_gen_nat''' :: nat -> nat
c_gen_nat''' N = z
c_gen_nat''' (s N) = s (c_gen_nat''' N)
```

Para el generador por número de constructores del tipo de datos listas véase el Ejemplo 4.

La transformación de sublist utilizando este generador es:

```
c_sublist' ::  [ nat ] ->  [ nat ] -> bool
c_sublist'  Xs Ys     :- c_sublist'' Xs Ys c_gen_listas

c_sublist'' ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
c_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps
```

Generadores por pesos en constructores

Supongamos los siguientes pesos para los constructores de los tipos de datos:

Tipo de datos nat: Constructor s. Peso 1.
Tipo de datos listas: Constructor listas (:). Peso 2

El generador para el tipo de datos nat es:

```
% Genera todos los términos del tipo nat
p_gen_nat :: nat
p_gen_nat = p_gen_nat' z

% Genera los términos del tipo nat con suma de pesos >= N
p_gen_nat' :: nat -> nat
p_gen_nat' N = p_gen_nat'' N // p_gen_nat' (s N)
```

```
% Genera los términos del tipo nat con suma de pesos N
p_gen_nat'' :: nat -> nat
p_gen_nat'' z = z
p_gen_nat'' (s N) = s (p_gen_nat'' N)
```

Para el generador por pesos en constructores para el tipo de datos listas véase el Ejemplo 4.

La transformación de sublist utilizando este generador es:

```
p_sublist' ::  [ nat ] ->  [ nat ] -> bool
p_sublist'  Xs Ys     :- p_sublist'' Xs Ys p_gen_listas

p_sublist'' ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
p_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps
```

Ejemplo 6.- Función generadora "equitativa"

Sean los tipos de datos:

```
data nat = z | s nat
data either = void | left nat either | right nat either
data trees = nil | tree either trees
```

y la función de elección indeterminista

```
// :: _A -> _A -> _A
X // Y = X
X // Y = Y
```

Las funciones auxiliares

```
member :: _A ->  [ _A ] -> bool
member X [Y|Xs] = true <== X == Y
member X [Y|Xs] = true <== member X Xs
```

y la función intersecan donde se encuentra la variable extra Z

```
intersecan ::  [ _A ] ->  [ _A ] -> bool
intersecan Xs Ys  = true <== member Z Xs , member Z Ys
```

A continuación se detallarán los generadores de los tipos de datos para nat , el tipo de datos either y el tipo de datos trees y los programas transformados utilizando las distintas estrategias de profundidad, número de símbolos y número de constructores.

Generadores por nivel de profundidad

Para el generador por nivel de profundidad del tipo de datos nat véase el Ejemplo 5.

Para el generador por nivel de profundidad del tipo de datos either véase el Ejemplo 4.

El generador para el tipo de datos de los árboles es:

```
% Genera todos los términos del tipo trees de elementos de tipo either
gen_tree :: trees
gen_tree = gen_tree' z

% Genera los términos del tipo trees de profundidad >= N
gen_tree' :: nat -> trees
gen_tree' N = gen_tree'' N //  gen_tree' (s N)
```

```
% Genera los términos del tipo trees de profundidad N
gen_tree'' :: nat -> trees
gen_tree'' z = nil
gen_tree'' (s N) = tree (gen_either'' N) (gen_tree''' N)

% Genera los términos del tipo trees de profundidad <=N
gen_tree''' :: nat -> trees
gen_tree''' z = nil
gen_tree''' (s N) = gen_tree'' (s N) // gen_tree''' N
```

y la transformación de intersecan utilizando este generador es:

```
intersecan' ::  [ trees ] ->  [ trees ] -> bool
intersecan' Xs Ys = intersecan'' Xs Ys gen_tree

intersecan'' ::  [ _A ] ->  [ _A ] -> _A -> bool
intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys
```

Generadores por número de símbolos

Para el generador por número de símbolos del tipo de datos nat véase el Ejemplo 5.

Para el generador por número de símbolos del tipo de datos either véase el Ejemplo 4.

El generador para el tipo de datos de los árboles es:

```
% Genera todos los términos del tipo trees con elementos de tipo either
s_gen_tree :: trees
s_gen_tree = s_gen_tree' (s z)

% Genera los términos del tipo trees con >= N símbolos
s_gen_tree' :: nat -> trees
s_gen_tree' N = s_gen_tree'' N // s_gen_tree' (s N)

% Genera los términos del tipo trees con N símbolos
s_gen_tree'' :: nat -> trees
s_gen_tree'' (s z) = nil
s_gen_tree'' (s (s (s N))) = s_gen_tree''' (s z) (s N)

% Genera los términos del tipo listas, con N símbolos del tipo either
% y M símbolos del tipo trees
s_gen_tree''' :: nat -> nat -> trees
s_gen_tree''' N (s M) = tree (s_gen_either'' N) (s_gen_tree'' (s M))
                        //
                       s_gen_tree''' (s N) M
```

y la transformación de intersecan utilizando este generador es:

```
s_intersecan' ::  [ trees ] ->  [ trees ] -> bool
s_intersecan' Xs Ys = s_intersecan'' Xs Ys s_gen_tree

s_intersecan'' ::  [ _A ] ->  [ _A ] -> _A -> bool
s_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys
```

Generadores por número de constructores

Para el generador por número de constructores del tipo de datos nat véase el Ejemplo 5.

Para el generador por número de constructores del tipo de datos either véase el Ejemplo 4.

El generador para el tipo de datos de los árboles es:

```
% Genera todos los términos del tipo trees con elementos de tipo either
c_gen_tree :: trees
c_gen_tree = c_gen_tree' (s z)

% Genera los términos del tipo trees con >= N constructores
c_gen_tree' :: nat -> trees
c_gen_tree' N = c_gen_tree'' N // c_gen_tree' (s N)

% Genera los términos del tipo trees con N constructores
c_gen_tree'' :: nat -> trees
c_gen_tree'' z = nil
c_gen_tree'' (s N) = c_gen_tree''' z (s N)

% Genera los términos del tipo trees, con N constructores del tipo either
% y M constructores del tipo trees
c_gen_tree''' :: nat -> nat -> trees
c_gen_tree''' N (s M) = tree (c_gen_either'' N) (c_gen_tree'' M)
                          //
                         c_gen_tree''' (s N) M
```

y la transformación de sublist utilizando este generador es:

```
c_intersecan' ::  [ trees ] ->  [ trees ] -> bool
c_intersecan' Xs Ys = s_intersecan'' Xs Ys c_gen_tree

c_intersecan'' ::  [ _A ] ->  [ _A ] -> _A -> bool
c_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys
```

Generadores por pesos en constructores

Supongamos los siguientes pesos para los constructores de los tipos de datos:

Tipo de datos nat: Constructor s. Peso 1.
Tipo de datos either: Constructor left Peso 2. Constructor right Peso 3
Tipo de datos trees: Constructor tree. Peso 4

El generador para el tipo de datos nat es:

```
% Genera todos los términos del tipo nat
p_gen_nat :: nat
p_gen_nat = p_gen_nat' z

% Genera los términos del tipo nat con suma de pesos >= N
p_gen_nat' :: nat -> nat
p_gen_nat' N = p_gen_nat'' N // p_gen_nat' (s N)

% Genera los términos del tipo nat con suma de pesos N
p_gen_nat'' :: nat -> nat
p_gen_nat'' z = z
p_gen_nat'' (s N) = s (p_gen_nat'' N)
```

El generador para el tipo de datos either es:

```
% Genera todos los términos del tipo either
p_gen_either :: either
p_gen_either = p_gen_either' z

% Genera los términos del tipo either con suma de pesos >= N
p_gen_either' :: nat -> either
p_gen_either' N = p_gen_either'' N // p_gen_either' (s N)

% Genera los términos del tipo either con suma de pesos N
p_gen_either'' :: nat -> either
p_gen_either'' z = void
p_gen_either'' (s (s N)) = p_gen_either_left''' z (s N)
p_gen_either'' (s (s (s N))) = p_gen_either_right''' z (s N)

% Genera los términos del tipo either con suma de pesos N del tipo nat
% y suma de pesos M del tipo either
p_gen_either_left''' N (s M) = left (p_gen_nat'' N) (p_gen_either'' M)
                                // p_gen_either_left''' (s N) M
p_gen_either_right''' N (s M) = right (p_gen_nat'' N) (p_gen_either'' M)
                                // p_gen_either_right''' (s N) M
```

El generador para el tipo de datos de los árboles es:

```
% Genera todos los términos del tipo trees
p_gen_tree ::  trees
p_gen_tree = p_gen_tree' z

% Genera los términos del tipo trees con suma de pesos >= N
p_gen_tree' :: nat ->  trees
p_gen_tree' N = p_gen_tree'' N // p_gen_tree' (s N)

% Genera los términos del tipo trees con suma de pesos N
p_gen_tree'' :: nat ->  trees
p_gen_tree'' z = nil
p_gen_tree'' (s (s (s (s N)))) = p_gen_tree''' z (s N)

% Genera los términos del tipo trees, con suma de pesos N del tipo either
% y suma de pesos M del tipo trees
p_gen_tree''' :: nat -> nat ->  trees
p_gen_tree''' N (s M) = tree (p_gen_either'' N) (p_gen_tree'' M)
                              // p_gen_tree''' (s N) M
```

y la transformación de sublist utilizando este generador es:

```
p_intersecan' ::  [ trees ] ->  [ trees ] -> bool
p_intersecan' Xs Ys = p_intersecan'' Xs Ys p_gen_tree

p_intersecan'' ::  [ _A ] ->  [ _A ] -> _A -> bool
p_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys
```

5.5. Comparación de tiempos

En este apartado se realiza un estudio detallado de las estimaciones de tiempos de los programas transformados anteriormente en los ejemplos. A continuación se indican los principales recursos utilizados para realizar esta comparativa.

- Lenguaje lógico-funcional : TOY (con Sicstus Prolog 3.9.0.)
- Entorno de desarrollo: IDE TOY System [13]
- Sistema Operativo: Windows Me
- Procesador y memoria: Pentium II (133 Mhz), 64 Mb de RAM

Comparación de tiempos del Ejemplo 4.

```
append [] Ys Zs = true <== Ys == Zs
append [X|Xs] Ys [Y|Zs] = append Xs Ys Zs <== X == Y

last  X Xs     = true <== append Ys [X] Xs

last'  X Xs     = true <== last'' X  Xs gen_listas
last'' X Xs Ys  = true <== append Ys [X] Xs

s_last'  X Xs     = true <== s_last'' X  Xs s_gen_listas
s_last'' X Xs Ys  = true <== append Ys [X] Xs

p_last'  X Xs     = true <== c_last'' X  Xs p_gen_listas
p_last'' X Xs Ys  = true <== append Ys [X] Xs

c_last'  X Xs     = true <== c_last'' X  Xs c_gen_listas
c_last'' X Xs Ys  = true <== append Ys [X] Xs

lista = [ (left (left (left (left (left (left (left (left (left void))))))))), (left (left (left (left
(left (left (left (left void)))))))), (left (left (left (left (left (left (left void))))))), (left (left
(left (left (left (left void)))))), (left (left (left (left (left void))))), (left (left (left (left
void)))), (left (right (left void))), (left (left void)), (left void), void, void ]
```

Objetivo original:

```
last    X lista   == true                 31 ms
```

Objetivo transformado last’. Estrategia nivel de profundidad:

```
last'   X lista   == true                 94 ms
```

Objetivo transformado s_last’. Estrategia número de símbolos:

```
s_last' X lista   == true                 9172 ms
```

Objetivo transformado c_last’. Estrategia número de constructores

```
c_last' X lista   == true                 6671 ms
```

Objetivo transformado p_last’. Estrategia peso en constructores

```
p_last' X lista   == true                 6703 ms
```

Comparación de tiempos del Ejemplo 5.

```
preffix [] Ys          = true
preffix [X|Xs] [Y|Ys]  = preffix Xs Ys <== X == Y

suffix Xs Ys           = true <== Xs == Ys
suffix Xs [Y|Ys]       = suffix Xs Ys

sublist  Xs Ys   :- preffix Ps Ys, suffix Xs Ps

sublist'  Xs Ys     :- sublist'' Xs Ys gen_listas
sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps

s_sublist'  Xs Ys     :- sublist'' Xs Ys s_gen_listas
s_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps

c_sublist'  Xs Ys     :- sublist'' Xs Ys c_gen_listas
c_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps

p_sublist'  Xs Ys     :- sublist'' Xs Ys p_gen_listas
p_sublist'' Xs Ys Ps  :- preffix Ps Ys, suffix Xs Ps

lista =  [ (s (s (s (s (s (s (s (s (s z))))))))), (s (s (s (s (s (s (s (s z)))))))), (s (s (s (s
(s (s (s z))))))), (s (s (s (s (s (s z)))))), (s (s (s (s (s z))))), (s (s (s (s z)))), (s (s (s
z))), (s (s z)), (s z) ]
```

Objetivo original:

```
sublist    lista lista   == true          15 ms
```

Objetivo transformado sublist’. Estrategia nivel de profundidad:

```
sublist'   lista lista   == true          15 ms
```

Objetivo transformado s_ sublist’. Estrategia número de símbolos:

```
s_sublist' lista lista   == true          594 ms
```

Objetivo transformado c_ sublist’. Estrategia número de constructores

```
c_sublist' lista lista   == true          610 ms
```

Objetivo transformado p_sublist’. Estrategia peso en constructores

```
p_sublist' lista lista   == true          797 ms
```

Comparación de tiempos del Ejemplo 6.

```
member X [Y|Xs] = true <== X == Y
member X [Y|Xs] = true <== member X Xs

intersecan Xs Ys  = true <== member Z Xs , member Z Ys

intersecan' Xs Ys = intersecan'' Xs Ys gen_tree
intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys

s_intersecan' Xs Ys = s_intersecan'' Xs Ys s_gen_tree
s_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys

c_intersecan' Xs Ys = s_intersecan'' Xs Ys c_gen_tree
c_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys

p_intersecan' Xs Ys = p_intersecan'' Xs Ys p_gen_tree
p_intersecan'' Xs Ys Z = true <== member Z Xs , member Z Ys

arbol =  [ (tree (right (s (s (s z))) (left (s (s z)) (left (s z) (left z void)))) (tree (left (s
(s z)) (left (s z) (right z void))) (tree (left (s z) (right z void)) (tree void nil)))) ]
```

Objetivo original:

intersecan arbol arbol == true 31 ms

Objetivo transformado intersecan'. Estrategia nivel de profundidad:

intersecan' arbol arbol == true 140 ms

Objetivo transformado s_ intersecan'. Estrategia número de símbolos:

s_intersecan' arbol arbol == true 13218 ms

Objetivo transformado c_ intersecan'. Estrategia número de constructores

c_intersecan' arbol arbol == true 1718 ms

Objetivo transformado p_intersecan'. Estrategia peso en constructores

p_intersecan' arbol arbol == true 20718 ms

Como puede observarse a través de las diferentes comparaciones de tiempos realizadas anteriormente, las transformaciones realizadas con las técnicas propuestas para la eliminación de variables extra no mejoran los tiempos de los programas originales.

Los mejores tiempos son los obtenidos mediante la estrategia de nivel de profundidad, que son bastante cercanos a los tiempos obtenidos en los programas originales. En el caso de las estrategias de número de símbolos y número de constructores, aunque lejos de los tiempos de los programas originales, mantienen una semejanza entre ellos debido a su similitud para obtener los términos. Por último, la estrategia de pesos en constructores refleja los peores tiempos.

Ahora bien, estas técnicas de transformación no tienen como objetivo obtener programas más eficientes. Su principal interés se centra en aquellos casos en que las variables extra deben ser eliminadas, por ejemplo, cuando las variables extra plantean dificultades en el ámbito de la programación lógica al utilizar la negación. En programación lógico funcional también es necesaria su eliminación en determinados casos, como por ejemplo, al considerar programas con fallo finito constructivo[31].

6. Otras alternativas

En esta sección se presentan otras alternativas consideradas al realizar el estudio de la eliminación de variables extra en programas lógico funcionales.

A diferencia de la técnica de la eliminación de variables extra mediante funciones generadoras universales de datos, vista en las secciones anteriores, las diferentes alternativas aquí comentadas no producen técnicas totalmente generales y mecanizables y se indicará aquellos casos en los que no es posible aplicar la solución de forma general. Otra característica importante de estas técnicas consiste en que se aplican únicamente a predicados donde se encuentran las variables extra, que para lenguajes lógico funcionales como TOY, son un caso particular de funciones.

La primera alternativa se basa en utilizar la idea de funciones indeterministas generadoras, pero con otra estrategia para la definición de dichas funciones generadoras.

Como hemos visto en secciones anteriores, con el fin de eliminar las variables extra, definimos una función generadora universal de datos. Esta función generadora consiste en una función indeterminista cuyo resultado son todos los valores que representa el dominio de los posibles valores de las variables extra.

En este caso, la función indeterminista generadora no se obtiene a través de los valores del dominio de la variable extra. Consiste en crear una función indeterminista a partir del predicado donde se encuentre la variable extra, de tal forma, que, intuitivamente, este generador "genera los posibles valores de la variable extra". Para realizar esto, cada aparición de una variable extra dará lugar a un generador indeterminista. La función generadora se obtendrá a partir de la definición del predicado donde se encuentre la variable extra, es decir, es necesario realizar una conversión de la definición del predicado de tal forma que obtengamos un generador que devuelva los posibles valores para la variable extra. En las siguientes secciones se mostrará mediante ejemplos cómo se realiza esta conversión y las restricciones que existen para realizarla.

En algunos casos, se verá que esta alternativa produce unos programas transformados que tienen mejores propiedades de terminación que los programas originales.

La segunda alternativa considerada consiste en realizar transformaciones fold-unfold en aquellas reglas donde aparecen las variables extra.

6.1. Generadores Indeterministas

Los programas lógico funcionales permiten la definición de predicados mediante cláusulas de Horn y la definición de funciones mediante ecuaciones condicionales. Los predicados pueden representarse como funciones booleanas, y asumimos que estas cláusulas, también denominadas reglas

condicionales, tienen la forma:

$$l \to r \Leftarrow s_1 == u_1, \ldots, s_k == u_k$$

Atendiendo a la clasificación de reglas de acuerdo a la aparición de variables extra, este método permite eliminar variables extra en sistemas 3-CTRS, es decir, en sistemas donde las variables extra pueden encontrarse tanto en la parte derecha de la regla como en la condición, con algunas restricciones que se detallarán más adelante.

La eliminación de variables extra mediante funciones generadoras indeterministas consta de dos pasos:

1.- Obtención de la función generadora indeterminista
2.- Transformación de la regla donde se encuentra la variable extra, eliminándola

A través de los siguientes puntos, y mediante una serie de ejemplos, se mostrará como es posible obtener la función generadora indeterminista y la transformación necesaria para eliminar las variables extra de la regla.

6.1.1. Obtención del generador indeterminista.

Como hemos indicado anteriormente, la obtención de un generador indeterminista consiste en convertir el predicado donde se encuentra la variable extra en un generador de elementos, es decir, la aparición de una variable extra dará lugar a una función indeterminista que proporcionará los valores para esta variable.

El ejemplo más sencillo de obtención de una función generadora indeterminista se corresponde con aquellas definiciones de predicados que son átomos. El siguiente ejemplo muestra cómo se obtiene el generador para este caso.

EJEMPLO 6.1

Supongamos la siguiente definición del predicado grandfather y del predicado father

```
grandfather :: [ char ] -> [ char ] -> bool
grandfather X Z = true <== father X Y, father Y Z

father :: [ char ] -> [ char ] -> bool
father "abraham" "isaac" = true
father "haran" "lot"     = true
father "haran" "milcah" = true
father "haran" "yiscah" = true
```

como puede observarse, el predicado grandfather tiene una variable extra, Y. La función generadora se obtiene a partir del predicado donde se encuentra la

variable extra, en este caso existen dos predicados father en la condición. Ahora bien, como la variable extra está situada en diferentes argumentos de los dos predicados father vamos a crear dos generadores, uno para uno de los predicados.

En el caso del primer predicado father, father X Y, el generador se obtiene convirtiendo el predicado father de forma que obtengamos una función indeterminista que representa, de manera informal, “un generador de personas Y que son hijos de X”.

```
gen_father1 ::  [ char ] ->  [ char ]
gen_father1 "abraham" = "isaac"
gen_father1 "haran" = "lot"
gen_father1 "haran" = "milcah"
gen_father1 "haran" = "yiscah"
```

Para el segundo predicado father, father Y Z, necesitamos un generador que representa, de manera informal, “un generador de personas Y que son padres de Z”. En este caso el generador viene definido por:

```
gen_father2 ::  [ char ] ->  [ char ]
gen_father2 "isaac" = "abraham"
gen_father2 "lot"     = "haran"
gen_father2 "milcah" = "haran"
gen_father2 "yiscah" = "haran"
```

El siguiente ejemplo muestra la obtención de un generador indeterminista a partir de un predicado que no está definido mediante átomos.

EJEMPLO 6.2

Supongamos la siguiente definición del predicado member y del predicado intersecan

```
member :: _A ->  [ _A ] -> bool
member X [Y|Xs] = true <== X == Y
member X [Y|Xs] = true <== member X Xs
```

```
intersecan ::  [ _A ] ->  [ _A ] -> bool
intersecan  Xs Ys = true <== member Z Xs == true, member Z Ys == true
```

En este caso, en el predicado intersecan existe una variable extra, Z. La función generadora se obtiene a partir del predicado donde se encuentra la variable extra, en este caso member. La creación de este generador consiste en convertir el predicado member de forma que obtengamos una función indeterminista que representa, de manera informal, “un generador de miembros de Xs”, es decir, los posibles valores X de Xs que cumplen la condición “X es miembro de Xs”.

En este caso, la definición de la función generadora es:

```
gen_member ::  [ _A ] -> _A
gen_member [X|Xs] = X
gen_member [X|Xs] = gen_member Xs
```

En algunos casos es posible que, en la parte condicional de la regla, la variable extra no se encuentre en un predicado si no en una función y por tanto no es posible utilizar la estrategia de las funciones generadoras directamente. La solución propuesta consiste en realizar una transformación de la función convirtiéndola en un predicado de tal forma que a partir de este predicado, se pudiese definir el generador y utilizar la técnica indicada anteriormente.

A continuación se muestra un ejemplo donde la condición esta formada por una función. Mediante una simple transformación, se convierte en un predicado y, a partir de ella, obtendremos el generador indeterminista.

EJEMPLO 6.3

Supongamos la siguiente definición de last.

```
append ::  [ _A ] ->  [ _A ] ->  [ _A ]
append [] Ys = Ys
append [X|Xs] Ys = [X|append Xs Ys]

last ::  [ _A ] -> _A -> bool
last Xs X = true <== append Ys [X] == Xs
```

En este caso, la variable extra, Ys, se encuentra en una función, la función append, y por tanto no es posible obtener directamente el generador a partir de ella.

Para ello, debemos realizar la transformación de la función append en un predicado. Esta transformación es sencilla:

```
predicado_append ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
predicado_append [] Ys Zs  = true <== Ys == Zs
predicado_append [X|Xs] Ys [Y|Zs] = predicado_append Xs Ys Zs <== X == Y
```

Una vez que hemos realizado esta transformación, ya podemos obtener el generador indeterminista.

Nuevamente, de forma intuitiva, este generador “obtiene aquellas listas Ys, tales que añadidas a [X] obtenemos las listas Xs”. Observar que la transformación se realiza sobre el primer argumento de predicado_append, donde se encuentran los posibles valores que se desean obtener.

```
gen_append :: _A ->  [ _B ] ->  [ _B ]
gen_append Ys [X|Zs] = []
gen_append Ys [X|Zs] = [X|gen_append Ys Zs]
```

6.1.2. Restricciones en la definición de los generadores indeterministas

La creación de funciones generadoras indeterministas puede aplicarse en un gran número de casos, aunque no siempre es posible utilizarlo y por tanto esta técnica no es totalmente general.

Una restricción a la hora de poder crear los generadores indeterministas consiste en disponer de una regla condicional donde la condición contiene varias variables extra diferentes. Estas variables extra deben aparecer, como mínimo, en otra ecuación de la condición, pues en caso contrario no es posible obtener una función generadora debido a que si aparece únicamente en una ecuación no es posible definir un generador que devuelva los términos para cada una de las variables extra. Veámoslo a través del siguiente ejemplo.

EJEMPLO 6.4

Supongamos el predicado append, definido como:

```
append ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
append [] Ys Zs = true <== Ys == Zs
append [X|Xs] Ys [Y|Zs] = append Xs Ys Zs <== X == Y
```

y la cláusula last, definida como:

```
last ::  [ _A ] -> _A -> bool
last Xs = X <== append Ys [X] Xs
```

En este caso, el predicado append contiene dos variables extra, Ys y X. Estas variables extra no aparecen en ninguna otra ecuación de la condición de la regla y por tanto, no es posible obtener un generador que nos devuelva los posibles valores para Ys y X.

Otra restricción se produce cuando los generadores obtenidos pueden, a su vez, tener variables extra. A continuación veremos un ejemplo donde se pone de manifiesto esta particularidad

EJEMPLO 6.5

Supongamos el siguiente programa:

p X = true <== q X Y, r Y

q c (s X) = true
q (s X) (s Y) = true <== q X Y

r (s Y) = true

donde el predicado p tiene la variable extra tanto en el predicado q como en el predicado r de la condición.

Si obtenemos la función generadora para el predicado q:

gen_q c = (s X)
gen_q (s X) = (s Y) <== q X Y

y la función generadora para r es:

gen_r = (s Y)

Como puede verse, los generadores para la variable extra Y tanto si lo obtenemos del predicado q como si lo obtenemos del predicado r tienen a su vez variables extra que ya no es posible eliminarlas con esta técnica.

Hasta ahora hemos visto como es posible obtener las funciones generadoras indeterministas cuando existen reglas con variables extra. Ahora bien, una vez que tenemos la función generadora, es necesario transformar las reglas donde aparecen las variables extra, con el fin de obtener la regla condicional sin variables extra.

En el punto siguiente veremos una serie de estrategias para realizar estas transformaciones y se mostrará, dependiendo de la forma de la regla y del número de apariciones de la variable extra, cómo llevar a cabo esta transformación.

6.1.3. Transformación de la regla condicional.

Con el fin de eliminar las variables extra, en la realización de la transformación de la regla condicional se pueden utilizan dos estrategias: la igualdad de generadores y la aplicación de generadores.

La igualdad de generadores consiste en que, cada aparición de la variable extra da lugar a un generador indeterminista, si dicha variable extra contiene

varias apariciones dentro de la regla condicional, será posible realizar la igualdad de estos generadores. En el caso de que la variable extra se encuentre en predicados análogos en diferentes partes de la condición, bastará con un solo generador.

La aplicación de generadores consiste en eliminar la primera aparición de la variable extra, que da lugar a un generador. Este generador será reemplazado en todas las demás apariciones de la variable extra.

Estas estrategias de transformación no siempre es posible utilizarlas. A continuación se detallan varios ejemplos donde se incluyen la utilización de estas estrategias, indicando aquellos casos en los cuales no es posible aplicar alguna de ellas.

Los principales casos donde se pueden realizar la transformación de la regla condicional son los siguientes:

a.- Una única aparición de la variable extra en la condición de la regla.

Supongamos la siguiente definición de regla condicional:

$$l(t) \to r \Leftarrow s_1(t_{11},\ldots,X,\ldots,t_{1n}) == true$$

donde X es una variable extra en la ecuación s_1.Para realizar la transformación de la regla se utiliza la estrategia de aplicación de generadores, sustituyendo la variable extra X, por la función indeterminista generadora, obteniendo:

$$l(t) \to r \Leftarrow s_1(t_{11},\ldots,gen_s_1(t_{11},\ldots,t_{1n}),\ldots,t_{1n}) == true$$

donde: gen_s_1: es la función indeterminista generadora de términos obtenida mediante el predicado s_1

En este caso no tiene sentido realizar la transformación utilizando la estrategia de igualdad de generadores puesto que solamente existe una única aparición de la variable extra

EJEMPLO 6.6

Partiendo de las definiciones del EJEMPO 6.3, en este ejemplo se muestra la transformación de una regla condicional donde la condición contiene una variable extra y existe una única aparición.

Supongamos la función:

```
append ::  [ _A ] ->  [ _A ] ->  [ _A ] -> bool
append [] Ys Ys = true
append [X|Xs] Ys [X|Zs] = true <== append Xs Ys Zs == true
```

y la regla condicional last, definida como:

```
last ::  [ _A ] -> _A -> bool
last Xs X = true <== append Ys [X] Xs == true
```

en este caso, el predicado append sólo tiene la variable extra Ys y además sólo aparece en una ecuación.

El generador de append, definido en el EJEMPLO 6.3, es:

```
gen_append :: _A ->  [ _B ] ->  [ _B ]
gen_append Ys [X|Zs] = []
gen_append Ys [X|Zs] = [X|gen_append Ys Zs]
```

Aplicando la estrategia de transformación de aplicación de funciones, este generador se sustituirá por la única aparición de la variable extra, quedando la regla transformada como sigue:

```
last' ::  [ _A ] -> _A -> bool
last’ Xs X = true <== append (gen_append [X] Xs) [X] Xs
```

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado, utilizando los mismos recursos de la Sección 5, apartado Comparación de Tiempos:

genera_lista X es una función que genera una lista de X elementos

```
genera_lista:: real -> [real]
genera_lista 0 = []
genera_lista (N+1) = [N+1 | genera_lista N]
```

Los tiempos de generación de la lista son los siguientes:

genera_lista 50000 == X	375 ms
genera_lista 100000 == X	733 ms
genera_lista 250000 == X	1906 ms
genera_lista 500000 == X	3844 ms

Tiempos para el programa original:

last (genera_lista 50000) X == true	1376 ms
last (genera_lista 100000) X == true	2797 ms
last (genera_lista 250000) X == true	7016 ms
last (genera_lista 500000) X == true	14015 ms

Tiempos para el programa transformado:

last' (genera_lista 50000) X == true	1468 ms
last' (genera_lista 100000) X == true	2937 ms
last' (genera_lista 250000) X == true	7375 ms
last' (genera_lista 500000) X == true	14686 ms

b.- Varias apariciones de la variable extra en la condición de la regla.

Supongamos la siguiente definición de regla condicional:

$$l(t) \to r \Leftarrow s_1(t_{11},\ldots,X,\ldots,t_{1n}) == true, s_2(t_{21},\ldots,X,\ldots,t_{2m}) == true$$

donde X es una variable extra que aparece en las ecuaciones s_1 y s_2.

En este caso es posible realizar la transformación de la regla utilizando ambas estrategias.

La estrategia de aplicación de generadores, consiste en eliminar el predicado donde se encuentra la primera aparición de la variable extra X, obteniendo un generador indeterminista, y este generador será reemplazado en todas las demás apariciones de la variable extra, obteniendo:

$$l(t) \to r \Leftarrow s_2(t_{21},\ldots,gen_s_1(t_{11},\ldots,t_{1n}),\ldots,t_{2m}) == true$$

donde gen_s_1: es la función indeterminista generadora de términos obtenida mediante el predicado s_1

La estrategia de igualdad de generadores consiste, en este caso, en que una vez obtenido el generador para la variable extra X, y dado que la variable extra tiene varias apariciones en la condición de la regla, se realiza la igualdad de estos generadores.

$$l(t) \to r \Leftarrow gen_s_1(t_{11},\ldots,t_{1n}) == gen_s_2(t_{21},\ldots,t_{2m})$$

donde: gen_s_1 es la función indeterminista generadora de términos obtenida mediante el predicado s_1 y gen_s_2 es la función indeterminista generadora de términos obtenida mediante el predicado s_2. En el caso de que los predicados s_1 y s_2 fueran el mismo bastaría obtener un único generador.

En el caso de que s_1 y s_2 fueran predicados distintos, mediante la estrategia de igualdad entre generadores, sería necesario definir dos generadores para cada una de las ecuaciones y luego pedir la igualdad entre los generadores.

Utilizando la estrategia de transformación de aplicación de funciones, sólo es necesario definir un único generador, ya que en las restantes apariciones de la variable extra se sustituirán por dicho generador.

EJEMPLO 6.7

A partir de las definiciones realizadas en el EJEMPLO 6.2, en este ejemplo se muestra la transformación de una regla condicional donde la condición contiene una variable extra y existen varias apariciones de la variable extra.

Sea el predicado member definido como:

```
member :: _A ->  [ _A ] -> bool
member X [Y|Xs] = true <== X == Y
member X [Y|Xs] = true <== member X Xs
```

y el predicado intersecan definido como:

```
intersecan ::  [ _A ] ->  [ _A ] -> bool
intersecan  Xs Ys = true <== member Z Xs == true, member Z Ys == true
```

Como habíamos visto en el EJEMPLO 6.2, el generador se define convirtiendo el predicado member en una función indeterminista:

```
gen_member ::  [ _A ] -> _A
gen_member [X|Xs] = X
gen_member [X|Xs] = gen_member Xs
```

En primer lugar, se realiza una transformación siguiendo la estrategia de aplicación de funciones, eliminando el predicado donde se encuentra la primera aparición de la variable extra Z, member Z Xs, y el generador será reemplazado en todas las demás apariciones de la variable extra, obteniendo:

```
intersecan' ::  [ _A ] ->  [ _A ] -> bool
intersecan'  Xs Ys = true <== member (gen_member XS) Ys == true
```

Ahora, realizaremos una transformación mediante la estrategia de igualdad entre generadores, una vez obtenido el generador para la variable extra Z, gen_member, y dado que la variable extra tiene varias apariciones en la condición de la regla, se realiza la igualdad de estos generadores.

```
intersecan'' ::  [ _A ] ->  [ _A ] -> bool
intersecan''  Xs Ys  = true <== gen_member Xs == gen_member Ys
```

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado. Para realizar las estimaciones de tiempo se utilizarán dos listas cuyo último elemento es el único que se encuentra en las dos listas.

genera_lista_1 X y genera_lista_2 X es una función que genera una lista de X elementos 1 y 2

```
genera_lista’:: real -> real -> [real]
genera_lista’ 0 X = [3]
genera_lista’ (N+1) X = [X| genera_lista_9_1 N X]
```

Los tiempos de generación de la lista son los siguientes:

```
genera_lista' 500 1 == X           0 ms
genera_lista' 1000 1 == X         15 ms
genera_lista' 1500 1 == X         16 ms
```

Tiempos para el programa original:

```
intersecan (genera_lista' 500 1)  (genera_lista' 500 2) == true          1453 ms
intersecan (genera_lista' 1000 1) (genera_lista' 1000 2) == true        5828 ms
intersecan (genera_lista' 1500 1) (genera_lista' 1500 2) == true       13110 ms
```

Tiempos para el programa transformado intersecan':

```
intersecan' (genera_lista' 500 1)  (genera_lista' 500 2) == true         1235 ms
intersecan' (genera_lista' 1000 1) (genera_lista' 1000 2) == true       4938 ms
intersecan' (genera_lista' 1500 1) (genera_lista' 1500 2) == true      11922 ms
```

Tiempos para el programa transformado intersecan'':

```
intersecan'' (genera_lista' 500 1)  (genera_lista' 500 2) == true        1281 ms
intersecan'' (genera_lista' 1000 1) (genera_lista' 1000 2) == true      5156 ms
intersecan'' (genera_lista' 1500 1) (genera_lista' 1500 2) == true     11750 ms
```

Estos tiempos se han obtenido utilizando los mismos recursos descritos en la Sección 5, apartado Comparación de Tiempos.

A continuación se muestra un ejemplo donde la estrategia de aplicación de funciones es más útil que la estrategia de igualdad de generadores ya que en el primer caso, únicamente es necesario definir una función generadora.

EJEMPLO 6.8

En este ejemplo se muestra la transformación de una regla condicional donde la condición contiene una variable extra y existen varias apariciones de la variable extra. Además, las apariciones se encuentran en predicados distintos.

Sean los predicados preffix y suffix definidos como:

```
preffix ::  [ _A ] ->  [ _A ] -> bool
preffix [] Ys        = true
preffix [X|Xs] [Y|Ys] = preffix Xs Ys <== X == Y

suffix ::  [ _A ] ->  [ _A ] -> bool
suffix Xs Zs        = true <== Xs == Zs
suffix Xs [Y|Ys]     = suffix Xs Ys
```

y el predicado sublist definido como:

```
sublist ::  [ _A ] ->  [ _A ] -> bool
sublist Xs Ys = true <== preffix Ps Ys == true, suffix Xs Ps == true
```

En este caso, la variable extra Ps se encuentra en dos predicados de la parte condicional de la regla, los predicados son: preffix y suffix.

Para la creación del generador para el predicado preffix, se convierte dicho predicado de tal forma que obtengamos una función indeterminista que representa, de manera informal, "un generador de elementos listas Ps que son prefijos de la lista Ys".

```
gen_preffix ::  [ _A ] ->  [ _A ]
gen_preffix Ys     = []
gen_preffix [X|Ys] = [X|gen_preffix Ys]
```

Nuevamente, para la creación del generador para el predicado suffix se convierte este predicado de tal forma que obtengamos una función indeterminista que representa, de manera informal, "un generador de elementos listas Ps que son sufijos de la lista Xs".

```
gen_suffix ::  [ _A ] ->  [ _A ]
gen_suffix Xs      = Xs
gen_suffix Xs  = [Y|gen_suffix Xs]
```

Este ejemplo muestra un punto importante. En este caso no es posible utilizar la transformación mediante la estrategia de igualdad entre generadores, puesto que gen_suffix contiene variables extra.

La alternativa consiste en realizar una transformación siguiendo la estrategia de aplicación de funciones, eliminando el predicado donde se encuentra la primera aparición de la variable extra Ps, preffix Ps Ys, y el generador será reemplazado en todas las demás apariciones de la variable extra, obteniendo:

```
sublist' ::  [ _A ] ->  [ _A ] -> bool
sublist' Xs Ys = true <== suffix Xs (gen_preffix Ys) == true
```

Puede observarse en este caso que únicamente es necesario utilizar un generador y este generador no contiene variables extra.

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado. Para realizar las estimaciones de tiempo se utilizarán dos listas cuyo que contienen un único elemento. La función genera_lista es la función descrita anteriormente.

Tiempos para el programa original:

```
sublist (genera_lista 100) (genera_lista 100)   == true            156 ms
sublist (genera_lista 500) (genera_lista 500)   == true           4172 ms
sublist (genera_lista 1000) (genera_lista 1000)   == true        16906 ms
```

Tiempos para el programa transformado sublist’:

```
sublist' (genera_lista 100) (genera_lista 100)   == true          0 ms
sublist' (genera_lista 500) (genera_lista 500)   == true         16 ms
sublist' (genera_lista 1000) (genera_lista 1000)   == true       15 ms
```

Estos tiempos se han obtenido utilizando los mismos recursos descritos en la Sección 5, apartado Comparación de Tiempos.

c.- Apariciones de la variable extra en la parte derecha de la regla y en condición.

Supongamos la siguiente definición de regla condicional:

$$l(t) \to r(t_{r1},\ldots,X,\ldots,t_{rk}) \Leftarrow s_1(t_{11},\ldots,X,\ldots,t_{1n}) == true, s_2(t_{21},\ldots,t_{2m}) == true$$

donde X es una variable extra que aparece en la parte derecha de la regla, r, y en el predicado s_1 de la condición. Para realizar la transformación de la regla se utiliza la estrategia de aplicación de generadores. Para ello, elegimos el predicado de la condición que contiene la variable extra, s_1, la eliminación de esta variable produce un generador que reemplazá la variable extra de la parte derecha de la regla, obteniendo:

$$l(t) \to r(t_{r1},\ldots,gen_s_1(t_{11},\ldots,t_{1n}),\ldots,t_{rk}) \Leftarrow s_2(t_{21},\ldots,t_{2m}) == true$$

En el caso de que no existiera ningún predicado más en la condición que aquél donde se encuentra la variable extra, al realizar la transformación se obtendría una regla no condicional, de la forma:

$$l(t) \to r(t_{r1},\ldots,gen_s_1(t_{11},\ldots,t_{1n}),\ldots,t_{rk})$$

EJEMPLO 6.9

En este ejemplo se muestra la transformación de una regla condicional donde la parte derecha y las condiciones también contienen apariciones de variables extra.

Sean los predicados preffix y suffix definidos como:

```
preffix ::  [ _A ] ->  [ _A ] -> bool
preffix [] Ys        = true
preffix [X|Xs] [Y|Ys] = preffix Xs Ys <== X == Y

suffix ::  [ _A ] ->  [ _A ] -> bool
suffix Xs Zs         = true <== Xs == Zs
suffix Xs [Y|Ys]      = suffix Xs Ys
```

y la función sublist definida como:

```
sublist ::  [ _A ] ->  [ _A ]
sublist Ys = Xs <== preffix Ps Ys == true, suffix Xs Ps == true
```

En este caso, existen dos variables extra, la variable extra Xs que se encuentra en la parte derecha de la regla y en el predicado preffix Ps Ys == true, y la variable extra Ps que se encuentra en dos predicados de la parte condicional de la regla, los predicados son: preffix y suffix.

Basándonos en la creación de generadores obtenidos anteriormente, los generadores para los predicados preffix y suffix son los siguientes:

```
gen_preffix ::  [ _A ] ->  [ _A ]
gen_preffix Ys     = []
gen_preffix [X|Ys] = [X|gen_preffix Ys]
```

```
gen_suffix ::  [ _A ] ->  [ _A ]
gen_suffix Xs      = Xs
gen_suffix [X|Xs]  = gen_suffix Xs
```

En un primer paso, eliminaremos la variable extra Ps, tal y como vimos en el ejemplo anterior, de tal forma que solamente nos quede como variable extra Xs en la parte derecha de la regla y en la condición.
Para ello utilizaremos el generador gen_suffix. Una vez realizada la transformación, obtenemos:

```
sublist' ::  [ _A ] ->  [ _A ]
sublist' Ys = Xs <== suffix Xs (gen_preffix Ys) == true
```

En este momento, la función sublist' únicamente contiene la variable extra Xs, que se encuentra en la parte derecha de la regla y en la condición.

Nuevamente, para eliminar la variable extra Xs, transformaremos esta función mediante la estrategia de aplicación de funciones, de tal forma que eliminaremos la variable extra Xs, y el generador obtenido será reemplazado en la aparición de la variable extra que se encuentra en la parte derecha de la regla, de esta forma:

```
sublist'' ::  [ _A ] ->  [ _A ]
sublist'' Ys = gen_suffix (gen_preffix Ys)
```

Puede observarse que, partiendo de una regla condicional, al realizar la transformación para eliminar las variable extra, hemos obtenido una regla transformada sin parte condicional.

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado. Para realizar las estimaciones de tiempo se utilizarán dos listas cuyo que contienen un único elemento. La función genera_lista es la función descrita anteriormente.

Tiempos para el programa original:

```
sublist (genera_lista 100) == (genera_lista 100)      547 ms
sublist (genera_lista 200) == (genera_lista 200)     3500 ms
sublist (genera_lista 300) == (genera_lista 300)    10938 ms
```

Tiempos para el programa transformado sublist’:

```
sublist' (genera_lista 100) == (genera_lista 100)     172 ms
sublist' (genera_lista 200) == (genera_lista 200)     797 ms
sublist' (genera_lista 300) == (genera_lista 300)    2031 ms
```

Tiempos para el programa transformado sublist’’:

```
sublist'' (genera_lista 100) == (genera_lista 100)     0 ms
sublist'' (genera_lista 200) == (genera_lista 200)    16 ms
sublist'' (genera_lista 300) == (genera_lista 300)    16 ms
```

Estos tiempos se han obtenido utilizando los mismos recursos descritos en la Sección 5, apartado Comparación de Tiempos.

A continuación se muestra un ejemplo interesante, en el cual existe una definición recursiva de una regla condicional que contiene variables extra.

EJEMPLO 6.10

El ejemplo muestra la transformación de una regla condicional donde las condiciones contienen apariciones de variables extra.

Sean los predicados plus y fib definidos como:

```
plus :: nat -> nat -> nat -> bool
plus X  c    Y     = true <== X == Y
plus X (s Y) (s Z) = true <== plus X Y Z

fib :: nat -> nat -> bool
fib  c        c    = true
fib (s c)    (s c) = true
fib (s (s N)) F    = true <== fib N D, fib (s N) E, plus D E F
```

En este caso, existen dos variables extra, la variable extra D que se encuentra en los predicados fib N D y plus D E F, y la variable extra E que se encuentra

en los predicados fib (s N) E y nuevamente en plus D E F. La idea para eliminar las variables extra consiste en crear un generador para el predicado fib de tal forma, que intuitivamente, “genera el número de fibonacci F para un numero dado N”.

```
gen_fib :: nat -> nat
gen_fib  c = c
gen_fib (s c) = (s c)
gen_fib (s (s N)) = F <== plus (gen_fib N) (gen_fib (s N)) F
```

Como puede observarse, al intentar obtener el generador, nuevamente encontramos una variable extra, F. Para eliminar la variable extra del generador, aplicamos los pasos vistos en el punto anterior, ya que la variable extra aparece en la parte derecha y en un predicado de la condición.

El primer paso consiste en obtener un generador para el predicado plus, que tiene la siguiente declaración:

```
gen_plus :: nat -> nat -> nat
gen_plus X c = X
gen_plus X (s Y) = s (gen_plus X Y)
```

Una vez que disponemos del generador gen_plus, ya podemos completar la declaración del generador para fib. Su declaración final es la siguiente:

```
gen_fib :: nat -> nat
gen_fib  c        = c
gen_fib (s c)     = (s c)
gen_fib (s (s N)) = gen_plus (gen_fib N) (gen_fib (s N))
```

Obviamente, en este caso, no es posible utilizar la estrategia de transformación de igualdad entre generadores, debido que el predicado plus D E F contiene las dos variables extra. Por tanto, se utiliza la estrategia de transformación de aplicación de funciones, de tal forma que eliminaremos la variable extra D y E, y el generador será reemplazado en cada aparición de las variables extra que se encuentra en los predicados de la condición de la regla, obteniendo:

```
fib' :: nat -> nat -> bool
fib'  c  c  = true
fib' (s c) (s c) = true
fib' (s (s N)) F = true <== plus (gen_fib N) (gen_fib (s N)) F
```

En este caso, ha sido posible realizar la eliminación de variables extra hasta encontrar un programa, o un generador, sin variables extra. Pero como vimos anteriormente, esto puede no ser siempre posible.

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado.

La función gen_nat X devuelve en valor de X representado como un término del tipo nat

```
data nat = c | s nat


gen_nat:: real -> nat
gen_nat 0 = c
gen_nat (N+1) = s (gen_nat N)
```

Tiempos para el programa original:

```
fib (gen_nat 10) X == true          0 ms
fib (gen_nat 20) X == true        171 ms
fib (gen_nat 25) X == true       2344 ms
```

Tiempos para el programa transformado:

```
fib' (gen_nat 10) X == true         0 ms
fib' (gen_nat 20) X == true       109 ms
fib' (gen_nat 25) X == true      1547 ms
```

6.1.4. Propiedades de las reglas transformadas sin variables extra

Propiedad de terminación

En algunos casos, la versión transformada sin variables extra tiene mejores propiedades de terminación que la versión original.

El siguiente ejemplo ilustra que la versión transformada de la regla intersecan tiene mejores propiedades de terminación que la versión original.

EJEMPLO 6.11

Supongamos el siguiente tipo de datos nat y la función loop:

```
data nat = c | s nat

loop = loop
```

Veremos que la ejecución del programa original intersecan y la transformación intersecan’ tienen un comportamiento muy distinto, cuando se evalúan los siguientes objetivos: intersecan [s loop] [c] e intersecan’ [s loop] [c]

La ejecución de TOY para intersecan [s loop] [c] sería:

Por loop, 0.
intersecan [s loop] [c] $\rightarrow$ intersecan [s loop] [c]

Por loop, 0.
intersecan [s loop] [c] $\rightarrow$ intersecan [s loop] [c]

En este caso, la función intersecan no termina nunca.

La ejecución de TOY para intersecan’ [s loop] [c] sería:

Por intersecan’, 0.
intersecan’ [s loop] [c] $\rightarrow$ gen_member [s loop] == gen_member [c]

Por gen_member, 0.
(s loop) == gen_member [c]

Por gen_member, 0.
(s loop) == c. **Falla**

Como puede observarse, en el caso de la función transformada, si termina, es más, falla infinitamente.

6.2. Transformación fold-unfold

La última alternativa se basa en realizar una transformación fold-unfold de las reglas que contengan variables extra de tal forma que obtengamos una versión donde no existan estas variables.

Originariamente, la técnica de plegado/desplegado fue desarrollada para realizar la optimización de programas funcionales. Esta técnica se basa generalmente en la construcción, por medio de estrategias (heurísticas), una secuencia de programas (semánticamente equivalentes) en la que cada uno de ellos se obtiene a partir de los precedentes aplicando una regla de transformación elemental. Las reglas básicas propuestas por Burstall y Darlington [10] de este tipo de sistemas son :

Definición. Consiste en introducir una nueva ecuación recursiva cuya expresión del lado izquierdo de la ecuación no sea una instancia de la expresión del lado izquierdo de ninguna otra ecuación previa.

Instanciación. Consiste en incluir una instancia de sustitución en una ecuación existente.

Desplegado. Si se dispone de las ecuaciones E <= E’ y F <= F’ y existe una ocurrencia en F’ de una instancia de E, se reemplaza F’ por la correspondiente instancia de E’, obteniendo F’’, y se añade la ecuación F <= F’’.

Plegado. Si se dispone de las ecuaciones E <= E’ y F <= F’ y existe alguna ocurrencia en F’ de una instancia de E’, se reemplaza E’ por la correspondiente instancia de E, obteniendo F’’, y se añade la ecuación F <= F’’

Abstracción. Consiste en introducir una cláusula *where* mediante la derivación de una ecuación previa, E <= E’, a una nueva ecuación:

E <= E’[u1/F1,...,un/Fn] where <u1,...,un> = <F1,...,Fn>

Leyes. Es posible transformar una ecuación utilizando en la parte derecha cualquiera de las leyes que tengan sus primitivas (asociatividad, conmutatividad, etc), obteniendo una nueva ecuación.

Burstall y Darlington presentan una estrategia simple para aplicar estas reglas, que se basa en los siguientes pasos:

1.- Aplicar la regla “Definición” tantas veces como sea necesario.

2.- Aplicar la regla “Instanciación”

3.- Para cada instanciación realizada, aplicar la regla “Desplegado” repetidamente. Además, en cada etapa del desplegado:
  3.1.- Intentar aplicar las reglas de “Leyes” y “Abstracción”.
  3.2.- Aplicar la regla “Plegado” repetidamente.

La estrategia propuesta no es mecanizable puesto que existen ciertos pasos no obvios en el sistema que deben ser indicados por el usuario, estos pasos Burstall y Darlington los denominan “eureka”. En principio, este conjunto de reglas, tampoco pueden constituir un algoritmo debido a que pueden existir numerosas formas diferentes de aplicar estas reglas. A continuación se describen las principales estrategias desarrolladas para realizar este tipo de transformaciones.

a.- Composición o deforestación

Su objetivo principal consiste en eliminar múltiples ocurrencias de las mismas estructuras de datos. Es un método automático en el cual se obtiene la eliminación de las estructuras de datos utilizadas para transferir resultados intermedios entre llamadas. Entre las estrategias de deforestación más importantes se encuentran la propuesta por Walder [34] y la estrategia de aplicación en programación funcional propuesta por Moreno y Alpuente [24], [3] y [2].

b.- Tupling

La idea básica de esta estrategia consiste en eliminar aquellas computaciones múltiples que utilizan los mismos valores. Utilizando esta estrategia semi-automática es posible eliminar múltiples accesos a las mismas estructuras de datos o a computaciones comunes, de forma similar a la idea de compartición (sharing) utilizada en reescritura de grafos. [6]

c.- Evaluación parcial

La estrategia consiste en obtener programas especializados con respecto a partes de su entrada de datos. Para ello, se obtiene un evaluador parcial, que es un mapping que toma un programa P y una llamada C y deriva un programa especializado Pc el cual produce los mismos valores y respuestas para C. Esta estrategia se utiliza en diferentes campos de aplicación, como supercompilación [33], computación parcial generalizada [17] o deducción parcial conjuntiva [14].

Como veremos, esta estrategia no siempre es posible utilizarla ya que simplemente con fold-unfold puede darse el caso de que no sea posible aplicar ninguna reducción sobre la regla y no obtengamos una transformación final del programa.

A continuación se muestran varios ejemplos de cómo se realiza esta transformación.

EJEMPLO 6.12

Sea el siguiente programa:

```
++ ::  [ _A ] ->  [ _A ] ->  [ _A ]
[] ++ Xs = Xs
[Y|Ys] ++ Xs = [Y|Ys++Xs]


last ::  [ _A ] -> _A
last Xs = X <== Ys ++  [X] == Xs
```

* Caso base: [] ++ Xs’ = Xs’.

last Xs = X <== Ys ++ [X] == Xs — Por ++, 0. Sustitución: (*)
last [X] = X

(*) Ys/[], Xs’/[X]

* Caso inductivo: [Y’|Ys’] ++ Xs’ = [Y’|Ys’++Xs’]

last Xs = X <== Ys ++ [X] == Xs — Por ++, 1. Sustitución: (*)
last [Y’|Xs’’] = X <== [Y’|Ys’] ++ [X] == [Y’|Xs’’] — Eliminación de Y’ en ++
last [Y’|Xs’’] = X <== Ys ++ [X] == Xs’’ — fold last
last [Y’|Xs’’] = last Xs’’

(*) Ys/[Y’|Ys’], Xs’/[X], Xs/[Y’|Ys’++Xs’], si Xs’’/[Ys’++Xs’]

El resultado de la transformación es:

```
last’ ::  [ _A ] -> _A
last’ [X] = X
last’ [X|Xs] = last’ Xs
```

A continuación se muestran algunas estimaciones de tiempo, tanto del programa original como del programa transformado. La función genera_lista se indicó anteriormente.

Tiempos para el programa transformado, utilizando los mismos recursos de la Sección 5, apartado Comparación de Tiempos:

last' (genera_lista 50000) == X	1468 ms
last' (genera_lista 100000) == X	2937 ms
last' (genera_lista 250000) == X	7375 ms
last' (genera_lista 500000) == X	14686 ms

EJEMPLO 6.13

Sea el siguiente programa:

P0. preffix [] Ys = true
P1. preffix [X|Xs] [Y|Ys] = preffix Xs Ys <== X == Y

S0. suffix Xs Zs = true <== Xs == Zs
S1. suffix Xs [Y|Ys] = suffix Xs Ys

sublist Xs Ys = true <== preffix Ps Ys, suffix Xs Ps

* Caso base: preffix [] Ys' = true

sublist Xs Ys = true <== preffix Ps Ys, suffix Xs Ps   Por P0. Ps/[], Ys/Ys'
sublist Xs Ys' = true <== suffix Xs []   Por S1. Xs/[]
sublist [] Ys' = true

* Caso base: suffix Xs' Xs' = true

sublist Xs Ys = true <== preffix Ps Ys, suffix Xs Ps   Por S0. Xs/Xs', Ps/Xs'

sublist Xs' Ys = true <== preffix Xs' Ys   Por P0. Xs'/[], Ys/Ys'
sublist [] Ys' = true

sublist Xs' Ys = true <== preffix Xs' Ys   Por P1. Xs'/[X''|Xs''], Ys/[X''|Ys'']
sublist [X''|Xs''] [X''|Ys''] = true <== preffix Xs'' Ys''

* Caso inductivo: preffix [X'|Xs'] [X'|Ys'] = preffix Xs' Ys'

sublist Xs Ys = true <== preffix Ps Ys, suffix Xs Ps
Por P1. Ps/[X'|Xs'], Ys/[X'|Ys']

sublist Xs [X'|Ys'] = true <== preffix Xs' Ys', suffix Xs [X'|Xs']
Por S1. Xs/Xs'', X'/Y'', Xs'/Ys''

sublist Xs'' [Y''|Ys'] = true <== preffix Ys'' Ys', suffix Xs'' Ys''
Por fold sublist
sublist Xs'' [Y''|Ys'] = true <== sublist Xs'' Ys'

El resultado de la transformación es:

sublist' :: [_A] -> _A
sublist' [] Ys = true
sublist' [X|Xs] [X|Ys] = true <== preffix Xs Ys
sublist' Xs [Y|Ys] = true <== sublist' Xs Ys

EJEMPLO 6.14

En este ejemplo se verá que esta alternativa no siempre puede utilizarse, debido a que el proceso de transformación es infinito:

```
splitAt :: nat ->  [ _A ] -> ( [ _A ],  [ _A ])
splitAt c Zs = ([],Zs)
splitAt (s N) [] = ([],[])
splitAt (s N) [Z|Zs] = ([Z|Xs],Ys) <== splitAt N Zs == (Xs,Ys)
```

La transformación consiste en:

* Caso base: SplitAt c Zs' = ([],Zs')

Sustitución: N/c, Zs/Zs', Xs/[], Ys/Zs'

splitAt (s c) [Z|Zs'] = ([Z],Zs')

* Caso base: splitAt (s N') [] = ([],[])

Sustitución: N/(s N'), Zs/[], Xs/[], Ys/[]

splitAt (s N) [Z] = ([Z],[])

* Caso inductivo: splitAt (s N') [Z'|Zs'] = ([Z'|Xs'],Ys') <== splitAt N' Zs' == (Xs',Ys')

splitAt (s N) [Z|Zs] = ([Z|Xs],Ys) <== splitAt N Zs == (Xs,Ys) Por splitAt, 2. (*)
splitAt (s (s N')) [Z,Z'|Zs']] = ([Z,Z'|Xs']],Ys) <== splitAt (s N') [Z'|Zs'] == ([Z'|Xs'],Ys')

(*) Sustitución: N/(s N'), Zs/[Z'|Zs'], Xs/[Z'|Xs'], Ys/Ys'

En este caso, no se puede aplicar ni splitAt, 0, ni splitAt, 1 y tampoco fold, por tanto no es posible realizar ninguna operación sobre:

splitAt (s N') [Z'|Zs'] == ([Z'|Xs'],Ys')

donde sigue existiendo la variable extra Xs'

7. Revisión de otros trabajos

7.1. Variables extra en programación lógico ecuacional.

En la primera parte del artículo *Variables extra en programación lógica (ecuacional)* [21] Hanus propone una transformación sintáctica de los programas lógicos, obteniendo un programa transformado que no contiene variables extra y muestra que existe una fuerte correspondencia entre el programa original y el programa transformado.

En la segunda parte, se presenta una transformación sobre programas lógicos ecuacionales para la eliminación de las variables extra. El propósito de esta transformación consiste en proporcionar un método general para derivar resultados de completitud en presencia de variables extra.

Al realizar esta última transformación se comprueba que la eliminación de variables extra en programas lógico funcionales es muy similar a los programas lógicos puros, pero existe una diferencia esencial: La transformación en programación lógica pura no cambia el significado pero si lo hace en el caso de la programación lógico ecuacional.

Esto sucede cuando es necesario reescribir una instancia de una variable en diferentes términos del programa original. Por tanto, se puede asegurar que el programa original y el transformado tienen el mismo significado si todas las ocurrencias de la misma variable se reducen al mismo término, consiguiéndose mediante la noción de compartición (sharing)

7.1.1. Eliminación de variables extra en programas lógicos puros

En los programas lógico puros, el método elegido para eliminar las variables extra consiste en realizar una transformación sintáctica de los programas lógicos en otros programas sin variables extra. La transformación propuesta por Hanus consiste en lo siguiente.

Sea la cláusula:

$$C : p(\bar{t}) \Leftarrow q_1(t_1), ..., q_k(t_k)$$

Para eliminar todas las variables extra se aplica la transformación eev a la cláusula, esta transformación se define como:

$$eev(C) : p(\bar{t}, v_{n+k}(x_1, ..., x_n, y_1, ..., y_k)) \Leftarrow q_1(\overline{t_1}, y_1), ..., q_k(\overline{t_k}, y_k)$$

donde $x_1, ..., x_n$ son las variables extra de C y $y_1, ... y_k$ son nuevas variables que no se encuentran en C, donde el orden de las variables en $v_{n+k}(x_1, ..., x_n, y_1, ..., y_k)$ es irrelevante.

Además, $v_0, v_1, v_2, \ldots$ es una familia de nuevos símbolos de función que no pertenecen al programa original. Es posible extender la transformación eev a programas aplicando dicha transformación a cada cláusula del programa.

A continuación se muestra un ejemplo de esta transformación.

EJEMPLO 7.1

$$\begin{array}{l} \text{append}([], L, L) \\ \text{append}([E \mid R], L, [E \mid RL]) \Leftarrow \text{append}(R, L, RL) \\ \text{last}(L, E) \Leftarrow \text{append}(R, [E], L) \end{array}$$

el programa transformado contiene las siguientes cláusulas:

$$\begin{array}{l} \text{append}([], L, L, v_0) \\ \text{append}([E \mid R], L, [E \mid RL], v_1(Y)) \Leftarrow \text{append}(R, L, RL, Y) \\ \text{last}(L, E, v_2(R, Y)) \Leftarrow \text{append}(R, [E], L, Y) \end{array}$$

7.1.2. Eliminación de variables extra en programación lógico ecuacional

Desde el punto de vista de la programación lógico ecuacional, Hanus considera un programa lógico ecuacional como un sistema de reescritura de términos condicional (CTRS). Como en los CTRS normales casi ortogonales su relación de reescritura es confluente, restringe los programas lógico ecuacionales a CTRS normales.

Hanus propone un método sistemático para la eliminación de variables extra en programas lógico ecuacionales. Esta transformación consiste en proporcionar un método general para derivar resultados de completitud cuando existen variables extra. Este método consiste en los siguiente pasos:

1. Transformar el programa lógico ecuacional en un nuevo programa sin variables extra.

2. Aplicar una estrategia de reducción completa al programa transformado (muchas de estas estrategias son conocidas si no contiene el programa variables extra).

3. Comprobar la correspondencia de las derivaciones de reducción entre el programa original y el programa transformado.

La transformación propuesta en el paso 1, consiste en transformar cada regla de reescritura añadiendo nuevos argumentos a cada función existente de la regla. Debido a que las funciones pueden estar anidadas, se tienen que añadir nuevos argumentos en cada subtérmino. Para este propósito, se denota

mediante $\hat{t}$ el termino obtenido a partir de t añadiendo nuevos argumentos variables en cada función existente en t, es decir, t se define como:

$\hat{x} = x$ para todas las variables x

$\hat{t} = f(\hat{t}_1, ..., \hat{t}_n, y)$ si $t = f(t_1, ..., t_n)$ y y es una nueva variable.

Cada regla condicional $\Re : f(\bar{t}) \rightarrow r \Leftarrow C$ se transforma en una regla $eev(\Re)$ aplicando la transformación eev a $\bar{t}$, r y C, y añadiendo variables extra en la parte izquierda, es decir:

$$eev(\Re) : f(\hat{\bar{t}}, v_n(x_1, ..., x_n)) \rightarrow \hat{r} \Leftarrow \widehat{C}$$

donde $\{x_1, ..., x_n\} = (Var(\hat{r}) \cup Var(\widehat{C})) \setminus Var(\hat{\bar{t}})$.

Las principales aplicaciones de eliminación de variables extra son:

Sistemas secuenciales inductivos.

Son sistemas particulares de los sistemas de reescritura ortogonales no condicionales basados en constructor. En estos sistemas todas las reglas que definen una función pueden organizarse en una estructura jerárquica, denominada árbol definicional, que representa una selección única de una regla mediante una distinción de casos sobre los argumentos para cada invocación a la función. Un CTRS se denomina secuencial inductivo si es un CTRS normal basado en constructor y su parte no condicional es secuencial inductiva. Debido a que los sistemas secuenciales inductivos son ortogonales, es posible transformar un CTRS normal secuencial inductivo en un sistema no condicional. Para ello, en cada regla condicional de la forma $R : l \rightarrow r \Leftarrow s = u$ se introduce un nuevo símbolo de función cond_R y se reemplaza R por las siguientes reglas incondicionales:

$$l \rightarrow \mathrm{cond}_R(s, r)$$
$$\mathrm{cond}_R(u, x) \rightarrow x$$

EJEMPLO 7.2

Sea el siguiente sistema secuencial inductivo que define la función member mediante la función append

$\mathrm{append}([], L) \rightarrow L$

$\mathrm{append}([E \mid R], L) \rightarrow [E \mid \mathrm{append}(R, L)]$

$\mathrm{member}(E,L) \rightarrow \mathrm{true} \Leftarrow \mathrm{append}(L1, [E|L2]) \equiv L$

el programa transformado $uc(eev(R))$ contiene las siguientes cláusulas:

$$\mathrm{append}([\,],\mathrm{L},v_0) \to \mathrm{L}$$
$$\mathrm{append}([\mathrm{E}\,|\,\mathrm{R}],\mathrm{L},v_1(\mathrm{X})) \to [\mathrm{E}|\mathrm{append}(\mathrm{R},\mathrm{L},\mathrm{X})]$$
$$\mathrm{member}(\mathrm{E},\mathrm{L},v_3(\mathrm{L1},\mathrm{L2},\mathrm{X})) \to \mathrm{cond}(\mathrm{append}(\mathrm{L1},[\mathrm{E}|\mathrm{L2}],\mathrm{X}) \equiv \mathrm{L},\mathrm{true})$$
$$\mathrm{cond}(\mathrm{true},\mathrm{X}) \to \mathrm{X}$$

***Variables extra parte derecha de las reglas condicionales*.**

La restricción para la confluencia, en este caso, viene dada por los CTRS funcionales. Un $\Re$ CTRS es funcional si cumple las siguientes condiciones:

1. $\Re$ es un CTRS normal.
2. La parte no condicional $\Re_u$ es casi-ortogonal.
3. La relación de reescritura $\to_{\Re}$ es confluente.

7.2. Eliminación de variables locales en programas lógicos definidos

Es bien conocido que las variables extra, o locales, son la causa principal de ineficiencia y algunas veces también de incompletitud en computaciones de objetivos negativos. Esto es debido a que la cuantificación universal es necesaria en el cálculo de la negación del cuerpo de la cláusula cuando existen variables extra, lo que puede provocar que su computación sea poco eficiente. Existen mecanismos de computación que tratan sobre objetivos universalmente cuantificados, como los propuestos en [7, 16, 32], que en general no tienen comportamientos muy eficientes.

En este trabajo [4] se presenta un algoritmo que permite realizar la transformación de programas lógicos definidos con variables extra en un programa lógico equivalente sin variables extra. Su principal objetivo consiste en mejorar el rendimiento de la implementación cuando se realizan consultas negativas con respecto a un programa lógico definido. Esta eficiencia se consigue mediante dos aspectos: el primero se basa en realizar la consulta negativa sobre el programa transformado que no contiene variables extra, el segundo consiste en que el programa transformado se puede obtener en tiempo de compilación. Hay que tener en cuenta que el programa transformado sólo se utilizará cuando se realicen consultas negativas, utilizando el programa original en el caso de que las consultas sean positivas.

La transformación se basa en la técnica de fold-unfold puesto que la corrección de la transformación viene dada por la secuencia de transformación fold-unfold. También, se realiza una partición preliminar de los argumentos de los átomos denominada especificación de modo. La especificación de modo asocia un modo (input/output) para cada posición de los argumentos y depende de las

variables extras a eliminar. Esta especificación se utiliza únicamente para la eliminación de variables extra y por tanto no es necesario indicarse; ni en las restricciones de los objetivos de usuario, ni con el flujo de datos asumido por el programador.

A continuación se describen brevemente algunos conceptos que se utilizan a lo largo del trabajo para poder obtener el algoritmo de eliminación de variables extra.

Sea un programa definido P. Para un predicado p, denotamos $\mathrm{Def}_{\mathrm{P}}(p)$ el conjunto de todas las cláusulas de P con cabeza p. Dado p y q y un programa P, decimos que p depende directamente de q si q se encuentra en alguna cláusula de $\mathrm{Def}_{\mathrm{P}}(p)$. Por el cierre reflexivo-transitivo de esta relación, se obtiene el conjunto $\mathrm{Dpd}_{\mathrm{P}}(p)$ tal que, con respecto al programa P, p depende de todos los predicados de $\mathrm{Dpd}_{\mathrm{P}}(q)$. $\mathrm{MR}_{\mathrm{P}}(p) \equiv \bigcup \{q \mid q \in \mathrm{Dpd}_{\mathrm{P}}(p) \text{ y } p \in \mathrm{Dpd}_{\mathrm{P}}(q)\}$ es el conjunto de todos los predicados que son mutuamente recursivos con p.

La especificación de modo de una cláusula $C \equiv B :- B_1, \ldots, B_m$, es una $m+1$ tupla $(MS^0(B), MS^1(B_1), \ldots, MS^m(B_m))$, donde $MS^i(B_i) \subseteq \{in, out\}$

La extensión de una especificación de modo $MS(p)$ a $MS(MRDef_{\mathrm{P}}(p))$ se realiza como sigue:

1. Asignar $MS(p)$ a cada átomo del predicado p en cualquier cláusula $C \in MRDef_{\mathrm{P}}(p)$.

2. Para cada cláusula $C \equiv B :- B_1, \ldots, B_m \in MRDef_{\mathrm{P}}(p)$ donde $MS(C)$ es parcial y $MS^0(B)$ está definida, extender $MS(C)$ a una especificación de modo total. Además, para $B_i = q$ tal que $q \in \mathrm{Dpd}_{\mathrm{P}}(p)$, el $MS^i(B_i)$ obtenido es asignado a cada átomo q en cualquier cláusula en $MRDef_{\mathrm{P}}(p)$.

La definición recursiva de cola $TailRDef_{\mathrm{P}}(p)$ de un predicado p con respecto a una especificación de modo $MS(MRDef_{\mathrm{P}}(p))$ es el conjunto de cláusulas que permiten realizar una invocación recursiva mediante el literal más a la derecha y no es posible realizar más computaciones después.

El método propuesto para realizar la transformación de un programa lógico definido se basa en 3 partes principales:

1.- Normalización de variables extra

Esta normalización se basa en normalizar las cláusulas del programa con respecto a las variables extra. Para que una cláusula esté normalizada es necesario que cada variable extra se encuentre exactamente en dos átomos

consecutivos de la cláusula y no se encuentre en ningún átomo más de la cláusula.

2.- Transformación recursiva de cola

Se dice que un predicado es recursivo de cola si cuando se realiza la invocación recursiva utilizando el literal más a la derecha del predicado no es posible realizar más computaciones después. En este caso, para la transformación que es necesaria para la eliminación de las variables extra, la definición recursiva de cola de un predicado se realiza con respecto a una especificación de modo.

3.- Eliminación de variables extra

La eliminación de las variables extra se realiza tomando cada vez el par de átomos consecutivos situados más a la izquierda de la cláusula que contenga alguna variable extra. Para ello es necesario asignar una especificación de modo a los dos átomos consecutivos de tal forma que, el primer átomo se le asigna el modo out en aquellas posiciones de los argumentos cuyos términos contienen variables auxiliares y el modo in en el resto. Se realiza el proceso contrario en el segundo átomo. En este momento se realiza la transformación de la cláusulas del programa obteniendo un conjunto de cláusulas, denominado $AVF(C)$. Para ver cómo se realiza esta transformación véase [4].

Por tanto, con los apartados comentados anteriormente, el algoritmo de eliminación de variables extra propuesto se resume como sigue.

Si suponemos el conjunto de cláusulas C, primero se realiza la normalización de las cláusulas que contengan variables extra. Ahora, para cada cláusula del programa, con cabeza h, se selecciona el par de átomos más a la izquierda que contengan variables auxiliares, p_j y p_{j+1}, y se les asigna una especificación de modo, como se ha descrito en el punto 2. Si el primer átomo $p_j \notin \mathrm{MR}_{\mathrm{P}}(h)$, entonces se transforma $\mathrm{Def}_{\mathrm{P}}(p_j)$ en una definición recursiva de cola con respecto a $MS(p_j)$ siempre que sea necesario, y se sustituye la transformación $AVF(C)$ por C. En caso contrario, transformamos $\mathrm{Def}_{\mathrm{P}}(h)$ en una definición recursiva de cola con respecto a $MS(p_j)$.

Formalmente, las variables extra contenidas en este par de átomos son eliminadas. Ahora bien, puesto que no es posible directamente eliminar estos átomos, primero transformamos $\mathrm{Def}_{\mathrm{P}}(h)$ en $TailRDef_{\mathrm{P}}(h)$ con respecto a $MS(p_j)$, que hace que $MR(h) \equiv \{h\}$. Esta transformación elimina alguna de las variables extra. Las demás cláusulas que se encuentran en $TailRDef_{\mathrm{P}}(h)$ nuevamente repetirán el proceso, y como su primer átomo no pertenece al conjunto de todos los predicados que son mutuamente recursivos con la cabeza h, eliminándose las variables extra. En otro caso, cuando la cláusula no pertenece a $TailRDef_{\mathrm{P}}(h)$, $TailRDef_{\mathrm{P}}(h)$ no se ve afectado por la eliminación de las variables extra.

En ningún paso realizado a lo largo de la transformación se introducen nuevas variables extra, por tanto, el procedimiento termina cuando no quedan variables extra.

7.3. Transformación de sistemas de reescritura condicionales con variables extra en sistemas no condicionales

La programación lógico funcional se basa en los sistemas de reescritura de términos condicionales, abreviadamente CTRSs. Ahora bien, a menudo en los CTRSs se prohíbe la utilización de la variables que se encuentren en el lado derecho de una regla de reescritura pero que no se encuentren en el lado izquierdo puesto que, en general, no está muy claro como instanciar estas variables. Sin embargo, la utilización de variables extra en el lado derecho de las reglas condicionales permiten escribir programas lógico funcionales de una forma más natural y eficiente.

Los sistemas que permiten utilizar variables extra en el lado derecho de las reglas condicionales inducen relaciones de reescritura que permiten que tengan una terminación efectiva, es decir que son computables y terminan. Estos sistemas se denominan CTRSs deterministas casi-reducibles [18].

Ohlebusch demuestra que los sistemas CTRS deterministas $\Re$ pueden ser transformados en sistemas no condicionales TRS $U(\Re)$ de tal forma que la terminación de $U(\Re)$ implica la casi-reducibilidad de $\Re$ y que la casi-reducibilidad de $\Re$ implica la terminación más interna (innermost) de $U(\Re)$.

En particular, sobre el marco de las variables extra, si la terminación de $U(\Re)$ implica la casi-reducibilidad de $\Re$, entonces $\Re$ puede contener variables extra en el lado derecho de las reglas condicionales. Al realizar la transformación en un sistema no condicional TRS $U(\Re)$ se han eliminado las condiciones de las reglas y por tanto también se eliminan las variables extra. Esta transformación permite obtener un sistema no condicional sin variables extra.

Estos resultados pueden aplicarse a la modularidad en la reescritura de términos y en las demostraciones de terminación en los denominados programas lógicos bien-modelados.

La transformación de un 3-CTRS determinista en un TRS no condicional se define como:

Sea $\Re$ un 3-CTRS determinista sobre la signatura F. Para cada regla de reescritura $\rho : l \to r \Leftarrow c \in \Re$, $|\rho|$ denota el número de condiciones en ρ. En la transformación, se necesitan $|\rho|$ símbolos de función frescos $U_1^{\rho},...,U_{|\rho|}^{\rho}$ para cada regla condicional $\rho \in \Re$. Por otra parte, Var (respectivamente ξVar)

denota una función que asigna la secuencia de variables (en una cierto orden fijo) en el conjunto $Var(t)$ (respectivamente $\xi Var(t)$) a un término t. Se transforma: $\rho : l \rightarrow r \Leftarrow s_1 \leftarrow t_1, ..., s_{|\rho|} \leftarrow t_{|\rho|}$ en un conjunto $U(\rho)$ de $|\rho|+1$ reglas de reescritura no condicionales como sigue:

$$l \rightarrow U_1^{\rho}(s_1, Var(l))$$
$$U_1^{\rho}(t_1, Var(l)) \rightarrow U_2^{\rho}(s_2, Var(l), \xi Var(t_1))$$
$$U_2^{\rho}(t_2, Var(l), \xi Var(t_1)) \rightarrow U_3^{\rho}(s_3, Var(l), \xi Var(t_1), \xi Var(t_2))$$
$$...$$
$$U_{|\rho|}^{\rho}(t_{|\rho|}, Var(l), \xi Var(t_1), ..., \xi Var(t_{|\rho|-1})) \rightarrow r$$

Puesto que $\Re$ es determinista, el sistema $U(R) = \cup_{\rho \in \Re} U(\rho)$ es un TRS no condicional sobre la signatura extendida $F' = F \cup \bigcup_{\rho \in \Re, 1 \leq i \leq |\rho|} \{U_i^{\rho}\}$ (es decir, $Var(r') \subseteq Var(l')$ se satisface para cada regla de reescritura $l' \rightarrow r' \in U(\Re)$.
A partir de la transformación de un sistema $\Re$ 3-CTRS determinista en un TRS no condicional $U(\Re)$, Ohlebusch llega a las siguientes conclusiones:
Si $U(\Re)$ termina $\Rightarrow$ $\Re$ es casi-reducible $\Rightarrow$ $\Re$ es casi-decreciente $\Rightarrow$ $U(\Re)$ tiene una terminación más interna (innermost).

Finalmente, el teorema principal del artículo muestra que si $\Re$ es un 3-CTRS determinista casi-decreciente, entonces $U(\Re)$ tiene una terminación más interna (innermost).

Las principales aplicaciones de estos resultados son:

Modularidad. El concepto de modularidad se describe de la siguiente forma: Una propiedad P es modular para cierta clase de CTRSs si, para todos los CTRSs $(F_1, \Re_1)$ y $(F_2, \Re_2)$ que pertenece a esa clase y tienen la propiedad P, su unión $(F_1 \cup F_2, \Re_1 \cup \Re_2)$ también pertenece a esa clase y tiene la propiedad P.

El artículo demuestra que la casi-reducibilidad es modular para un 3-CTRS sintácticamente determinista sin solapamiento, donde un 3-CTRS determinista $\Re$ se denomina sintácticamente determinista si, para cada $l \rightarrow r \Leftarrow s_1 \rightarrow t_1, ..., s_k \rightarrow t_k$ en $\Re$, cada término t_i es un término construido o una forma normal $\Re_u$, donde $\Re_u = \{l \rightarrow r \mid l \rightarrow r \Leftarrow c \in \Re\}$.

Programas lógicos bien-modelados. Mediante los resultados anteriores, también se puede demostrar la terminación de programas lógicos bien-modelados. Los programas bien modelados se definen como:

1. Sea $C = A \leftarrow B_1, ..., B_m$ una cláusula y $x \in Var(C)$. La cabeza A de C se denomina un productor (consumidor) de x, si x se encuentra en una

posición de entrada (salida) de A. El átomo B_j del cuerpo se llama un productor (consumidor) de x, si x se encuentra en una posición de salida (entrada) de B_j.

2. La cláusula $B_0 \leftarrow B_1,...,B_m$ se llama LR bien-modelada, si cada variable x en la cláusula tiene un productor B_i $(0 \le i \le m)$ y $i < j$ para cada consumidor B_j $(1 \le j \le m)$ de x en el cuerpo de la cláusula. Un programa lógico P es LR bien-modelado si cada cláusula en P es LR bien-modelada.

3. Una pregunta $\leftarrow B_1,...,B_m$ es LR bien-modelado si cada variable x en la pregunta tiene un productor B_i tal que para cada consumidor B_j de x tenemos $i < j$.

La transformación de un programa lógico P bien-modelado en un 3-CTRS determinista $\Re_P$ se define como:

Para cada átomo $A = P(t_1,...,t_n)$ con posiciones de entrada $i_1,...,i_k$ y posiciones de salida $i_{k+1},...,i_n$ existen dos nuevos símbolos de función P_{in} y P_{out}, y se define $\rho_{in}(A) = P_{in}(t_{i_1},...,t_{i_k})$, $\rho_{out}(A) = P_{out}(t_{i_{k+1}},...,t_{i_n})$ y $\rho(A) = \rho_{in} \to \rho_{out}$. La transformación $\rho(C)$ de una cláusula $C = A \leftarrow B_1,...,B_m$ se define mediante la regla:

$$\rho(A \Leftarrow \rho(B_1),...\rho(B_m)$$

y a cada programa lógico P se le asocia $\Re_P = \{\rho(C) \mid C\, en\, P\}$.

Finalmente, se llega a las siguientes conclusiones: Si $U(\Re_P)$ tiene terminación $\Rightarrow$ $\Re_P$ es casi-reducible $\Rightarrow$ $\Re_P$ es casi-decreciente $\Rightarrow$ $U(\Re_P)$ tiene terminación más interna (innermost) $\Rightarrow U(\Re_P)$ tiene terminación single redex

8. Conclusiones

En este trabajo se plantea una nueva técnica para la eliminación de variables extra en programas lógico funcionales al estilo de los lenguajes lógico funcionales, como Curry[20] y TOY[1]. Esta técnica consiste en la definición de una función generadora universal y la posterior transformación del programa original con variables extra en un programa sin ellas.

Basándonos en el trabajo teórico realizado, es posible definir una función generadora universal que es "teóricamente" completa. No obstante, al intentar ejecutar dicha función mediante una implementación de un lenguaje lógico funcional no devuelve todos los valores, debido a que el recorrido estándar de este tipo de lenguajes se basa en una estrategia de búsqueda primero en profundidad, y es posible que pueda intentar realizar el recorrido en una rama infinita.

Para solventar este problema, se plantean varias alternativas para la implementación de los generadores siguiendo distintos criterios, como son: la programación de generadores basados en profundidad creciente, que consiste en programar un generador en el cual la profundidad de los términos generados se basa en la profundidad del árbol que lo representa; la programación de generadores basados en número de símbolos, que consiste en definir un generador en el cual la cardinalidad de los símbolos del término se utiliza como criterio de tamaño obteniendo un número de términos finitos; la programación de generadores basados en número de constructores es una variante de la anterior, utilizando el número de constructores del término y por último, la programación de generadores basados en pesos de los constructores que consiste en definir un generador basándonos en unos pesos asignados a cada uno de los constructores de los cuales se compone el tipo de datos a generar y la suma de los pesos de los constructores que contiene el término se utiliza como crtierio de tamaño. Todas estas alternativas permiten generar en cada paso un número finito de términos. Mediante una serie de ejemplos, se muestra la transformación realizada y se comprueba el comportamiento de todas ellas, con respecto al programa original.

Por tanto, como punto principal del trabajo, se presenta una técnica totalmente general que permite la eliminación de las variables extras en programas lógico funcionales. En futuros trabajos e investigaciones se pueden realizar otras implementaciones de funciones generadoras universales más eficientes, de tal forma que obtengamos unos programas transformados también más eficientes.

Además, se plantean otras técnicas para la eliminación de variables extra, como son: la utilización de funciones indeterministas obtenidas a partir de los predicados donde se encuentran las variables extra y la posterior transformación de las reglas, sustituyendo las variables extra por las funciones indeterministas y otra técnica basada en la utilización de fold-unfold. Es preciso indicar que, aunque en algunos casos las transformaciones obtenidas con estas alternativas tienen mejores propiedades que los programas originales una vez eliminadas las variables extra, ninguna de estas técnicas conducen a estrategias totalmente generales.

En este trabajo se recoge las principales características de otros trabajos relacionados con la eliminación de variables extra. En [21], Hanus, propone realizar transformaciones en programas lógicos y programas lógico ecuacionales para eliminar las variables extra.

En [26], Ohlebusch realiza una transformación de CTRS deterministas con variables extra en sistemas no condicionales TRS sin variables extra, de tal forma que la terminación del sistema transformado implica la casi-reducibilidad del sistema original y la casi-reducibilidad del sistema original implica la terminación más interna del sistema transformado.

En [4], destaca que las variables extra son la causa principal de ineficiencia y en algunos casos también de incompletitud en computaciones de objetivos negativos. El trabajo obtiene un algoritmo que permite eliminar las variables extra en un programa lógico definido transformándolo en otro sin variables extra. Su principal objetivo consiste en mejorar el rendimiento de la implementación cuando se realizan consultas negativas con respecto a un programa lógico definido.

Bibliografía

[1] M. Abengózar Carneros, P. Arenas Sánchez, J.C. González Moreno, R. Caballero Roldán, A. Gil Luezas, J. Leach Albert, F.J. López Fraguas, N. Martí Oliet, J.M. Molina Bravo, E. Pimentel Sánchez, M. Rodríguez Artalejo, M.M. Roldán García, J.J. Ruz Ortiz, y J. Sánchez Hernández. TOY a multiparadigm declarative language, version 2.0. Technical report, UCM, Madrid, February 2002.

[2] M. Alpuente, M. Falaschi, G. Moreno, y G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlavác, K.G. Jeffery, and J. Wie-dermann, editors, Sofsem 2000—Theory and Practice of Informatics, volume 1963 of LNCS, Pp. 289–297. Springer-Verlag, 2000.

[3] M. Alpuente, M. Falaschi, G. Moreno, y G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, Proc. of the 4th Fuji International Symp. on Functional and Logic Programming, FLOPS’99, pp. 147–162. Springer LNCS 1722, 1999.

[4] J. Álvez, P. Lucio. Elimination of Local Variables from Definite Logic Programs. PROLE 2004, 2004

[5] K. R. Apt, Logic Programming, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol B. ch. 10, Elsevier, Amsterdam; MIT Press, Cambridge, MA, 1990 , pp. 493-574

[6] F. Baader F. y T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.

[7] R. Barbuti, P. Mancarella, D. Pedreschi y F. Turini. A Transformational Approach to Negation in Logic Programming. Journal of Logic Programming (8), pp. 201-228, 1990.

[8] R. Bird. Introduction to Functional Programming using Haskell. Prentice Hall Press, second edition, 1998.

[9] P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, y C. Palamidessi. A complete semantic characterization of K-LEAF, a logic language with partial functions. In Proc. International Logic Programming Symposium (ILPS'87), pp 1-27. The MIT Press, 1987.

[10] R. M. Burstall y J. Darlington. A transformation system for developing recursive programs. Journal of the Association for Computing Machines, 24(1), pp. 44-67, 1977.

[11] R. Caballero Roldán, F.J. López Fraguas, y J. Sánchez Hernández. User's manual for TOY. Technical Report 97/57, Depto. SIP, UCM Madrid, 1997.

[12] W.F. Clocksin and C.S. Mellish. Programming in Prolog. Springer, third revised and extended edition, 1987.

[13] J. de Dios Castro, J.C. González Moreno. A Graphical Development Environment for Functional Logic Languages. Technical Report, SIP-105.00.UCM, 2000.

[14] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, y M. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. Journal of Logic Programming, 41(2&3): pp. 231–277, 1999.

[15] N. Dershowitz, J.P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol B. ch. 6, Elsevier, Amsterdam; MIT Press, Cambridge, MA, 1990 , pp. 243-320

[16] W. Drabent. What is failure? An approach to constructive negation. Acta Informática, 32. pp. 27-59, 1995

[17] Y. Futamura y K. Nogi. Generalized Partial Computation. In D. Bjørner, A. Ershov, and N. Jones, editors, Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation, pp. 133–151. North-Holland, Amsterdam, 1988.

[18] H. Ganzinger. Order-sorted completion: The many-sorted way. Theoretical Computer Science, 89, pp. 3-32, 1991

[19] J.C.González Moreno, M.T.Hortalá González,F.J. López Fraguas y M. Rodríguez Artalejo.An Approach to Declarative Programming Based on a Rewriting Logic. Journal of Logic Programming , 40(1), pp. 47 –87,1999.

[20] M.Hanus (ed.).Curry: an Integrated Functional Logic Language,version 0.7.1.Technical report,Universität Kiel,Junio 2000.Available at http://www.informatik.uni-kiel.de/~mh/curry/.

[21] M. Hanus. On extra variables in (equational) logic programming. In Proc. Twelfth International Conference on Logic Programming, pages 665679. MIT Press, 1995.

[22] J.W. Klop, Foundations of Logic Programming, 2nd ed., Springer, Berlin, 1987

[23] Kowalski, R. Predicate Logic as a Programming Language. North-Holland, 1974.

[24] G. Moreno. Rules and Strategies for Transforming Functional Logic Programs. Ph.D. thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia. In spanish, 2000.

[25] J.J. Moreno-Navarro. Diseño, semántica e implementación de Babel: un lenguaje que integra la programación funcional y lógica. PhD thesis, Fac. de Informática, UPM, Madrid, 1989.

[26] E. Ohlebusch. Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems. LPAR 1999, pp. 111-130, 1999

[27] P. Padawitz. Generic Induction Proofs. In Proc. of the 3rd International. Workshop on Conditional Term Rewriting Systems, pp. 175-197. Springer LNCS 656, 1992.

[28] M. Prioietti y A. Pettorossi. Completeness of Some Transformation Strategies for Avoiding Unnecessary Logical Variables. In Proc. Eleventh International Conference of Logic Programming, pp. 714-729. MIT Press, 1994.

[29] J. A. Robinson y E. E. Sibert. The LOGLISP user's manual. Technical Report 12/81, Syracuse University, New York, 1981.

[30] M. Rodríguez Artalejo. Functional and constraint logic programming. In Revised Lectures of the International Summer School CCL'99, H. Common, C. Marché and R. Treinen (eds.), Constraints in Computational Logics, Theory and Applications, chapter 5, pp 202-270. Springer LNCS 2002, 2001.

[31] J. Sánchez Hernández. Una aproximación al fallo en programación declarativa multiparadigma. . PhD thesis, SIP, UCM. Junio 2004.

[32] P. J. Stuckey. Negation and constraint logic programming. Information and Computation, 118(1). pp. 12-33, 1995.

[33] V. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N. Jones, editors, Programs as Data Objects, pp. 257–281. Springer LNCS 217, 1985.

[34] P. Wadler. Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science, 73, pp. 231–248, 1990