Restricciones sobre dominios finitos en un lenguaje lógico funcional

Autor: Sonia Estévez Martín
Directora: Teresa Hortalá

Trabajo de Tercer Ciclo
Departamento: Sistemas Informáticos y Programación
Facultad de Informática
Univ. Complutense de Madrid

Septiembre 2004

Índice general

Índice de figuras

Capítulo 1

Introducción

Este trabajo ha sido motivado a partir de la realización de un curso de doctorado sobre la programación con restricciones.

1.1. Estado del arte.

En 1994 nació la programación lógica con restriciones ($\mathcal{CLP}$), véase [17] y [18], en un intento de unir la declarativid de la programación lógica ($\mathcal{LP}$) y la eficiencia de la programación con restricciones ($\mathcal{CP}$). El componente fundamental de la programación $\mathcal{CLP}$ es que puede ser parametrizada por un dominio computacional, de tal manera que diferentes tipos de dominios determinen diferentes instancias del esquema.

La importacia de la programación con restricciones sobre dominios finitos ($\mathcal{CP}(\mathcal{FD})$) ([24] y [16]) radica en:

- está basada en fuertes fundamentos teóricos [29], [8] y [1] haciendo que sea un paradigma sólido.
- su posibilidad de resolver problemas de campos tan diversos que van desde problemas matemáticos hasta aplicaciones reales de la industria.
- está basado sobre restricciones que son básicamente relaciones entre objetos, por lo tanto quien programa debe definir las restricciones y las relaciones entre ellas.

Por este último motivo, los leguajes declarativos parecen ser más apropiados que los lenguajes imperativos para enunciar problemas con restricciones de dominios finitos ($\mathcal{FD}$). De hecho una de la más exitosa instancia de $\mathcal{CP}(\mathcal{FD})$ es $\mathcal{CLP}(\mathcal{FD})$.

Los dos paradigmas que representan a la programación declarativa son el paradigma lógico y el paradigma funcional. La principal aportación del *paradigma lógico* es el uso de variables lógicas, el indeterminismo y la notación relacional, por otra parte, la aportación del *paradigma funcional* es el orden superior, polimorfismo, evaluación perezosa, tipos, composición de funciones y sintáxis funcional. El uso de las funciones da una gran flexibilidad al lenguaje, ya que estas son *ciudadanos de primera clase*, es decir, pueden ser utilizadas como otro objeto en el lenguaje (resultados, argumentos, elementos de estructuras de datos...). La unión de todas estas caracteristicas permiten escribir programas más breves que por consiguiente incrementan el nivel de expresividad. Más aún, que el lenguaje sea tipado permite que se pueda aplicar un inferidor de tipos y la comprobación

de tipos hace que se detecten antes los errores con lo que el desarrollo y mantenimiento de los programas se simplifica.

La programación lógico funcional ($\mathcal{FLP}$) integra las técnicas declarativas usadas en la programación lógica ($\mathcal{LP}$) y en la programación funcional ($\mathcal{FP}$) dando lugar a nuevas características no existentes en estos dos paradigmas por separado ([14] y [26]), como poder utilizar funciones de forma reversible como si fuesen predicados *Prolog*. Aunque $\mathcal{TOY}$ tiene notación currificada, admite como azúcar sintáctico predicados al estilo *Prolog*. $\mathcal{FLP}$ no tiene las limitaciones de $\mathcal{FP}$ y por lo tanto es un marco adecuado para la integración de funciones y restricciones.

El primer intento de combinar la programación lógica con restricciones y la programación lógico funcional fue el esquema $\mathcal{CFLP}(\mathcal{FD})$ propuesto por [5]. La idea que hay detrás de esta aproximación puede ser descrita por la ecuación $\mathcal{CFLP}(\mathcal{D}) = \mathcal{CLP}(\mathcal{FP}(\mathcal{D}))$, donde un lenguaje $\mathcal{CFLP}$ sobre el dominio de restricciones (D) es visto como una extensión de dominios finitos $\mathcal{FP}(\mathcal{D})$ cuyas restricciones incluyen ecuaciones entre expresiones que involucran funciones definidas por el usuario, las cuales son resueltas por medio del estrechamiento.

El esquema $\mathcal{CFLP}$ propuesto por [20] intenta dar una semántica declarativa en la que los programas $\mathcal{CLP}(\mathcal{D})$ puedan ser entendidos como un caso particular de los programas $\mathcal{CFLP}(\mathcal{D})$. En la aproximación clásica a la semántica $\mathcal{CLP}$, un dominio de restricciones es visto como una estructura de primer orden $\mathcal{D}$. En [20], los programas son construidos como un conjunto de reglas de reescritura con restricciones. Como debe soportar la semántica perezosa para las funciones definidas por el usuario, el dominio de restricciones $\mathcal{D}$ se formaliza mediante estructuras continuas, con un dominio de Scott [13] y una interpretación continua de símbolos de función y de predicado. La semántica resultante tenía buenas propiedades.

En un trabajo más reciente [22], se ha propuesto un nuevo esquema genérico $\mathcal{CFLP}(\mathcal{D})$, como un marco lógico y semántico para la programación lógico funcional perezosa con restricciones sobre un dominio de restricciones $\mathcal{D}$ parametrizable, lo cual supone una semántica declarativa rigurosa para lenguajes $\mathcal{CFLP}(\mathcal{D})$ al igual que en el esquema $\mathcal{CLP}(\mathcal{D})$, superando las limitaciones del antiguo esquema $\mathcal{CFLP}(\mathcal{D})$ [20]. Los programas $\mathcal{CFLP}(\mathcal{D})$ son un conjunto de reglas de reescritura con restricciones que definen el comportamiento del orden superior y las funciones perezosas indeterministas sobre $\mathcal{D}$. La principal novedad de [22] es una nueva formalización de los dominios de restricciones para $\mathcal{CFLP}$, una nueva noción de interpretación para programas $\mathcal{CFLP}(\mathcal{D})$, una nueva lógica de reescritura de restricciones $\mathcal{CRWL}(\mathcal{D})$ parametrizada por un dominio de reestricciones, lo cual da una caracterización lógica de la semántica de los programas. En [23] se presenta una semántica operacional para el nuevo esquema genérico $\mathcal{CFLP}(\mathcal{D})$ propuesto en [22], que presenta un cálculo de estrechamiento con restricciones $\mathcal{CLNC}(\mathcal{D})$ que resuelve objetivos para programas $\mathcal{CFLP}(\mathcal{D})$, que es correcto y fuertemente completo con respecto a la semántica $\mathcal{CRWL}(\mathcal{D})$. Estas propiedades hacen a $\mathcal{CLNC}(\mathcal{D})$ un mecanismo de computación conveniente para los lenguajes declarativos con restricciones .

Lo que se propone es la integración de las restricciones $\mathcal{FD}$ de la forma que aparecen en $\mathcal{CLP}(\mathcal{FD})$ dentro de un lenguaje $\mathcal{FLP}$ dando lugar a una instancia particular sobre $\mathcal{FD}$ del esquema $\mathcal{CFLP}(\mathcal{D})$, que denotaremos como $\mathcal{CFLP}(\mathcal{FD})$. Este trabajo expone la incorporación de los dominios finitos sobre el lenguaje lógico funcional perezoso de orden superior $\mathcal{TOY}$ ([21] y [26]), que denotaremos como $\mathcal{TOY}(\mathcal{FD})$ y que se encuentra enmarcado en el paradigma $\mathcal{CFLP}(\mathcal{FD})$. Como resolutor de restricciones se toma el resolutor eficiente de SICStus [3]. La semántica operacional de $\mathcal{TOY}(\mathcal{FD})$ es una instancia de

esquema general definido por [23].

En este nuevo marco las restricciones $\mathcal{FD}$ son definidas como funciones, lo que hace que $\mathcal{CFLP}(\mathcal{FD})$ tenga mejores herramientas respecto a $\mathcal{CLP}(\mathcal{FD})$. Por lo tanto la expresividad de $\mathcal{TOY}(\mathcal{FD})$ es mayor. Además en $\mathcal{CFLP}(\mathcal{FD})$ se pueden resolver todas las aplicaciones de $\mathcal{CLP}(\mathcal{FD})$ así como problemas de la programación funcional.

$\mathcal{TOY}$ se traduce a *Prolog*, el cual dispone de un resolutor de restricciones que se activa sobre el dominio de los booleanos 2.2, o sobre el dominio de los números reales 2.3 o sobre dominios finitos 2.4. En $\mathcal{TOY}$ están implementadas las retricciones aritméticas sobre reales [27], pero no están implementadas las restricciones sobre dominios finitos, siendo este tipo de restricciones de interés porque resuelven muchos problemas de la "vida real", como por ejemplo los problemas de planificación de tareas o diseño de circuitos, algunos de estos problemas son dificiles de expresar y resolver usando los lenguajes tradicionales de programación.

1.2. Organización de este trabajo

Este trabajo está organizado en los siguientes capitulos:

- Capítulo 2, Restricciones: En este capítulo se explican las ventajas de incorporar las restricciones a un lenguaje de programación. Se muestra cómo se cargan los resolutores de booleanos, reales y dominio finitos de Sicstus Prolog, ya que $\mathcal{TOY}(\mathcal{FD})$ se traduce a este sistema. Se definen los predicados y operadores de dichos dominios y se dan ejemplos. Se resuelven mediante *Prolog*, con el resolutor de los dominios finitos activado, problemas de la programación $\mathcal{CLP}(\mathcal{FD})$ y se muestran las técnicas mediante las cuales se reducen los dominios, éstas son la consistencia, tanto por arcos como por caminos y la propagación. Posteriormente se explica el etiquetado, el cual asigna valores a variables para buscar soluciones. Estas búsquedas de soluciones se realizan mediante backtracking simple o con comprobación previa y como opción se puede tener en cuenta cual debe ser la siguiente variable a evaluar para comprobar si forma parte de la solución. Una vez elegida la variable a evaluar es posible también, como opción, elegir cúal debe ser el orden de los valores para evaluar.

- Capítulo 3, $\mathcal{TOY}$: Después de haber expuesto qué entendemos por restricciones se exponen las principales caracteristicas de $\mathcal{TOY}$, estas son la evaluación perezosa, estructuras de datos infinitas, funciones indeterministas o multivaluadas, funciones de orden superior, restricciones de igualdad, desigualdad y aritméticas sobre los números reales. Previamente es necesario definir cómo son las expresiones, tipos, funciones y objetivos que se manejan en $\mathcal{TOY}$.

- Capítulo 4, $\mathcal{TOY}(\mathcal{FD})$: Con ayuda de los dos capítulos anteriores (Restricciones y $\mathcal{TOY}$), establecemos la base para extender el lenguaje lógico funcional $\mathcal{TOY}$ con las restricciones sobre dominios finitos $\mathcal{FD}$ obteniendo $\mathcal{TOY}(\mathcal{FD})$. Como conceptos fundamentales de la sintáxis de $\mathcal{TOY}(\mathcal{FD})$ y de la disciplina de tipos se definen signatura, tipos, patrones y expresiones. Las sustituciones pueden ser de tipos o de variables. Se define cómo es una expresión bien tipada y por último los objetivos.

  Pasamos a la implementacion del lenguaje toyfd, definiendo qué es un dominio finito y una restricción sobre un dominio finito. Para cada restricción de dominio finito se da su definición, tipo y ejemplos que en la mayoría de los casos están relacionandos

con sus correspondientes primitivas de *Prolog* a las cuales van a ser traducidas y resueltas por el resolutor de Sicstus *Prolog*. También se dan ejemplos de aplicaciones de $\mathcal{TOY}(\mathcal{FD})$, los primeros corresponden a los ejemplos que se mostraron en el capítulo 2, puesto que todos los problemas que se pueden resolver en $\mathcal{CLP}(\mathcal{FD})$ también se pueden resolver en $\mathcal{CFLP}(\mathcal{FD})$. Después se muestra un ejemplo que se resuelve en $\mathcal{TOY}(\mathcal{FD})$ aplicando evaluación perezosa porque trata con lista infinita, este ejemplo es interesante porque la programación $\mathcal{CLP}(\mathcal{FD})$ no puede tratar listas infinitas, mostrando así una buena propiedad de $\mathcal{TOY}(\mathcal{FD})$. Se exponen las modificaciones que se han hecho sobre el código para poder implementar en $\mathcal{TOY}$ las restricciones sobre dominios finitos, se han modificando módulos existentes en el sistema y se han creado ficheros nuevos.

- Capítulo 5, Conclusiones y trabajo futuro: Se presentan las conclusiones y trabajo futuro.

- Apéndice A `cflpfd.toy`: En este apéndice se muestra el fichero *basic.toy* el cual contiene las declaraciones de las funciones y tipos predefinidos de $\mathcal{TOY}$.

- Apéndice B `cflpfd.toy`: En este apéndice se muestra el fichero *cflpfd.toy* el cual contiene las declaraciones de las funciones y tipos predefinidos de la extensión sobre dominios finitos de $\mathcal{TOY}$, es decir, $\mathcal{TOY}(\mathcal{FD})$. Además están declaradas funciones y tipos que se utilizan a bajo nivel.

- Apéndice C `cflpfdfile.pl`: En este apéndice se muestra el fichero *cflpfdfile.toy* el cual contiene la llamada al resolutor de restricciones sobre dominios finitos de SISCtus, el código que traduce cada restricción $\mathcal{TOY}(\mathcal{FD})$ a su correspondiente restricción *Prolog*, y el código de las funciones definidas a bajo nivel que son necesarias.

Capítulo 2

Restricciones

En este capítulo se introduce el concepto de restricción y se muestran algunas restricciones del ámbito $\mathcal{CLP}(\mathcal{FD})$ porque este es el ámbito que se toma como modelo para hacer la extensión de dominios finitos sobre $\mathcal{TOY}$. Concretamente se toman las restricciones de Sicstus Prolog porque el lenguaje $\mathcal{TOY}(\mathcal{FD})$ envía sus restricciones al resolutor eficiente de Sisctus.

Los ejemplos que se muestran y se resuelven en *Prolog* ($\mathcal{CLP}(\mathcal{FD})$) estan pensados para ser posteriormente resueltos en $\mathcal{TOY}(\mathcal{FD})$ ($\mathcal{CFLP}(\mathcal{FD})$). Las caracteristicas de $\mathcal{CLP}(\mathcal{FD})$ las posee $\mathcal{CFLP}(\mathcal{FD})$, por lo tanto algunos ejemplos que se resuelven es $\mathcal{CLP}(\mathcal{FD})$ de forma análoga se pueden resolver en $\mathcal{CFLP}(\mathcal{FD})$. Pero además es posible que un mismo problema sea más sencillo en $\mathcal{CFLP}(\mathcal{FD})$ que en $\mathcal{CLP}(\mathcal{FD})$ si se emplean características no existentes en $\mathcal{CLP}(\mathcal{FD})$ como por ejmplo el orden superior, la composición de funciones, etc. y eso se refleja en los ejemplos.

Existen mecanismos que con ayuda de las restricciones podan los dominios de las variables todo lo que pueden excluyendo valores que no verifican las restricciones. Las técnicas mediante las cuales se reducen los dominios son la consistencia 2.4.2 y la propagación 2.4.3. Después de eliminar todos los valores que no forman parte de la solución se puede lanzar un proceso que asigna valores a las variables y comprueba si se verifican las restricciones, este proceso es el etiquetado 2.4.4.

2.1. Introducción

Una de las ventajas de incorporar las restricciones a un lenguaje de programación es que se simplifica el código que expresa las relaciones existentes entre variables. En los lenguajes de programación tradicionales expresar dichas relaciones o restricciones entre variables puede llegar a ser extenso y complicado, lo que hace que mantener el código sea más tedioso.

Un ejemplo simple puede ser la ley de *Ohm*, (`V = I x R`) En los lenguajes de programación tradicionales se evalúan los datos de entrada y se calculan los valores de las variables mediante las siguientes sentencias

```
V := I x R; I := V / R; R := V / I;
```

Si el lenguaje tiene las restricciones incorporadas entonces no es necesario mantener una sentencia por cada variable, lo cual simplifica el código.

Otra ventaja de la incorporación de las restricciones al lenguaje de programación es la sencillez al mostrar restricciones como datos de salida.

En *Prolog* el código correspondiente a la ley de *Ohm* utiliza la biblioteca de las restricciones sobre números reales (clpr)

```
:- use_module(library(clpr)).
 ohm(V,I,R):-{ V = I * R }.
```

Con las reglas que se acaban de mostrar se puede resolver los siguientes objetivos:

Objetivo	Solución
`|?- ohm(220,100,I).`	I = 2.2 ? ;
`|?- ohm(220,R,2.2).`	R = 99.99999999999998 ? ;
`|?- ohm(V,100,2.2).`	V = 220.0 ? ;
`|?- ohm(V,R,2.2).`	R=0.4545454545454545*V ? ;
`|?- ohm(V,100,I).`	I=0.01*V ? ;
`|?- ohm(220,R,I).`	clpr:220.0-I*R=0.0 ? ;

El código que resuelve estos objetivos en la programación tradicional, cuyas respuestas son valores o restricciones, es más complicado y extenso pues tiene que tener en cuenta cada uno de los casos de los datos de entrada y salida. Por otra parte, la programación con restricciones nos permite especificar de una forma más natural.

Un *problema con restricciones* es un conjunto de variables X={x_1,... x_n} tal que $x_i \in D_i$ (Dominio de la variable x_i) $\forall i \in [1,..,n]$ y además existe un conjunto de restricciones que determinan los valores que las variables pueden tomar simultáneamente. El dominio de las variables puede ser un conjunto infinito, por ejemplo, el conjunto de los números reales o puede ser un conjunto finito, por ejemplo, el dominio de la variable moneda es {cara,cruz}

Una *restricción* entre varias variables determina un subconjunto del conjunto formado por las combinaciones de los valores de las variables. Por ejemplo si $Dom(x) = \{1, 2, 3\}$ y $Dom(y) = \{1, 2\}$ entonces algún subconjunto del conjunto {(1,1), (1, 2), (2, 1),(2, 2), (3, 1), (3, 2) } satisface una restricción entre x e y. De hecho, la restricción $x = y$ es representada por el subconjunto {(1, 1), (2, 2)}.

Aunque normalmente las restricciones no son representadas de esta forma, la definición pone énfasis en que las restricciones no siempre necesitan corresponderse con ecuaciones o inecuaciones.

Las restricciones se clasifican respecto al dominio de las variables que implican. El sistema SICStus Prolog provee resolutores de restricciones dependiendo del dominio de estas, hay resolutores para restricciones booleanas (sección 2.2), racionales-reales (sección 2.3) y dominios finitos (sección 2.4).

Si las restricciones son sobre dominios finitos entonces se pueden representar mediante grafos. Las restricciones binarias (aquellas que tiene dos variables afectadas) se pueden representar como un grafo donde los nodos representan las variables y las aristas representan las restricciones. Las restricciones unarias (aquellas que tiene una variable afectada) se representan con una arista sobre un mismo nodo, un ejemplo de una restricción de este tipo puede ser excluir un valor del dominio ($x \neq 1$). Las restricciones de aridad mayor de dos se pueden representar como multigrafos.

Un problema clásico de grafos puede ser el problema de colorear un mapa con un número finito de colores, de tal forma que no existan dos regiones adyacentes con el mismo color. Los nodos serían las regiones y las aristas representan las fronteras entre regiones. Una aplicación concreta sería el segundo ejemplo de 2.4.1

Una *solución*, si existe, a un problema con restricciones es el resultado de asignar satisfactoriamente valores del dominio a las variables. En un problema con restricciones se puede buscar una solución cualquiera, todas las soluciones o la solución optima. Las soluciones a los problemas de restricciones se pueden encontrar asignando valores a las variables sistemáticamente.

2.2. Retricciones Sobre Booleanos

En esta sección se muestra cómo se carga el resolutor, que operaciones y predicados admite y un ejemplo.

Para cargar el resolutor de booleanos en Sicstus Prolog se debe ejecutar:

```
|?- use_module(library(clpb)).
```

Las expresiones booleanas están compuestas por los operandos False (0)y True (1) y variables lógicas. Si P y Q son expresiones booleanas las operaciones que permite el lenguaje son

```
~ P es True si P es False.
P * Q es True si P y Q son True.
P + Q es True si al menos P o Q son True.
P # Q es True si exactamente una de la dos expresiones es True.
X ^ P es True si existe una variable lógica X tal que P es true.
P =:= Q lo mismo que ~P # Q.
P =\= Q lo mismo que  P # Q.
P =< Q lo mismo que  ~P + Q.
P >= Q lo mismo que  P + ~Q.
P < Q lo mismo que ~P * Q.
P > Q lo mismo que  P * ~Q.
card(Is, Es) es True si el numero de expresiones True en la lista de
expresiones booleanas Es está contenido en la lista de enteros Is.
```

Además tiene definidos los siguientes predicados

sat(+Expression): comprueba la consistencia de la expresión booleana que se pasa por parámetro.

taut(+Expression, ?Truth): comprueba si la expresión es una tautología.

Ejemplo: mostramos algunas ejecuciones sobre la línea de comandos. Se ha de tener cargada la biblioteca del resolutor de restricciones sobre booleanos, por lo tanto se ha de ejecutar previamente `use_module(library(clpb)).` y una vez cargado correctamente podemos lanzar los siguientes objetivos.

```
|?- taut(A =< C, T).
no
```

```
A=<C no es una tautología porque puede ser verdadero o falso

     |?- sat(A =< B), sat(B =< C), taut(A =< C, T).
     T = 1
     sat(A=:=_A*_B*C),
     sat(B=:=_B*C) ?;
     no

Si se satisface que A =<B y B =<C entonces A =<C es una tautología.

     |?- taut(a, T).
     T = 0 ? ;
     no

Siempre es falsa porque a es un átomo.
```

2.3. Retricciones Sobre los Números Racionales y Reales

De forma análoga a las restricciones sobre booleanos, en esta sección se muestra cómo se carga el resolutor, que operaciones y predicados admite y ejemplo.

Para cargar el resolutor de los números racionales se debe especificar

```
|?- use_module(library(clpq)).
```

y para cargar el resolutor de los números reales se debe especificar

```
|?- use_module(library(clpr)).
```

La gramática para este tipo de restricciones es

```
Constraint --> C | C , C     ( conjunction )

C --> Expr =:= Expr | Expr = Expr | Expr < Expr   | Expr > Expr
    | Expr =<  Expr | Expr >=Expr | Expr =\= Expr

Expr --> variable        | number          | + Expr
        | - Expr         | Expr + Expr     | Expr - Expr
        | Expr * Expr    | Expr / Expr     | abs(Expr)
        | sin(Expr)      | cos(Expr)       | tan(Expr)
        | pow(Expr,Expr) | exp(Expr,Expr)  | min(Expr,Expr)
        | max(Expr,Expr) | #(Const)
```

Un **ejemplo** introductorio a las restricciones sobre los reales puede ser modelar la aplicación que calcula las distintas opciones que existen para amortizar un préstamo. Sea P la variable que representa la cantidad a amortizar, I el tipo de interés, R es el capital amortizado y NP es la nueva base para el cálculo, cuyo comportamiento se encapsula en la siguiente restricción.

$$\text{NP} = \text{P} + \text{P} * \text{I} - \text{R}.$$

Esta restricción se aplica a un periodo de tiempo T que normalmente se mide en años. Lo amortizado al final de este tiempo T es el balance (B).

El programa \ sicstus3\library\clpqr\mg.pl' captura este comportamiento. Se muestra su código a continuación

```
:- use_module(library(clpr)).
mg(P,T,I,B,MP):-
{  T = 1,
    B + MP = P * (1 + I)  }.
mg(P,T,I,B,MP):-
{   T > 1,
   P1 = P * (1 + I) - MP,
   T1 = T - 1  },
mg(P1, T1, I, B, MP).
```

Con las reglas que se acaban de mostrar se puede resolver los siguientes objetivos:

```
|?- mg(1000,12,0.01,B,MP).
{B=1126.82503013197-12.68250301319697*MP} ? ;
no
|?- mg(1000,12,0.01,150,MP).
MP = 77.02147037659044 ? ;
no
|?- mg(P,T,I,R,B).
T = 1.0,
clpr:{-(P)-I*P+R+B=0.0} ? ;
T = 2.0,
clpr:{-(P)-I*P+B+_A=0.0},
clpr:{R-_A*I+B-_A=0.0} ?
```

2.4. Retricciones Sobre Dominios Finitos

Al igual que en las dos secciones anteriores, en esta sección se muestra cómo se carga el resolutor, operaciones y predicados que admite. Además se muestran ejemplos introductorios de cada predicado pues esta sección es el preámbulo de la implementación de las restricciones sobre dominios finitos en el lenguaje lógico funcional $\mathcal{TOY}$.

Para cargar el resolutor de restricciones sobre dominios finitos se ha de especificar `|?- use_module(library(clpfd)).`

Los predicados y operaciones que el resolutor provee se dividen en los siguientes bloques:

- *Aritméticas*; estas restricciones están definidas por: `?Expr RelOp ?Expr` donde la sintaxis de las expresiones aritméticas es definida por [28] como a continuación se muestra

```
X --> variable
N --> integer
LinExpr --> N | X | N * X | N * N | LinExpr + LinExpr | LinExpr - LinExpr
Expr --> LinExpr | Expr + Expr | Expr - Expr | Expr * Expr | Expr / Expr
       | Expr mod Expr | min(Expr,Expr) | max(Expr,Expr) | abs(Expr)
RelOp --> #= | #\= | #< | #=< | #> | #>=
```

Ejemplo

```
|?- X #>0, X #<Y, Y #<Z, Z #<10, X + Y #<Z.
 X in 1..6,
 Y in 2..7,
 Z in 4..9 ?;
 no
```

Además existen dos predicados eficientes para computar la suma y el producto escalar:

sum(+Xs, +RelOp, ?Value): donde `Xs` es una lista de variables, `RelOp` como se acaba de definir y `Value` es un entero.

Ejemplo

```
| ?- sum([2,3,4],#=,X).
 X = 9 ? ;
 no
 | ?- sum([2,3,4],#<,X).
 X in 10..sup ? ;
 no
```

scalar_product(+Coeffs, +Xs, +RelOp, ?Value): donde `Coeffs` es una lista de longitud n de enteros, `Xs` es una lista de longitud n de variables y `Value` es un numero entero.

Ejemplo

```
| ?- scalar_product([1,2,3], [4,5,6],#=,X).
 X = 32 ? ;
 no
 | ?- scalar_product([1,2,3], [4,5,6],#<,X).
 X in 33..sup ? ;
 no
```

- *dominios sobre variables*:

domain(+Variables, +Min, +Max): `Variables` es una lista de variables, `Min` es un entero o el átomo `inf` que determina la cota inferior de las variables de la lista `Variables` y `Max` es un entero o el átomo `sup` que denota la cota superior.

Ejemplo

```
|?- domain([Y,Z],2,8).
 Y in 2..8,
 Z in 2..8 ? ;
 no
```

- *Restricciones proposicionales*; se permiten las siguientes combinaciones proposicionales

```
#\ Q True si la restricción Q es falsa.
P #/\ Q True si la restricciones P y Q son ambas ciertas.
P #\ Q True si exactamente una de las restricciones P y Q son true.
P #\/ Q True si al menos una de las restricciones P y Q son true.
```

```
P \#=> Q
Q \#<= P True si la restricción Q es true o la restricción P es false.
P \#<=> Q True si las restricciones P y Q son ambas true o son ambas false.
```

Ejemplo

```
|?- (4 #>2) #=>(4 #>0).
yes
```

- *Restricciones combinatorias*

count(+Val,+List,+RelOp,?Count): `Val` es un entero, `List` es una lista de variables, `Count` es un entero o una variable y `RelOp` es un operador relacional definido en las restricciones aritméticas. Este predicado devuelve True if `Count` es el número de veces que se cumple `val` en la lista `List` dependiendo de `RelOp`

Ejemplo:

```
|?- count(10,[2,3,4],#=,A).
A = 0 ? ;
no
|?- count(2,[3,2,4,2,6,1,2],#=,A).
A = 3 ? ;
no
```

element(?X,+CList,?Y): es true si el elemento `X`-ésimo de la lista `CList` coincide con `Y`

Ejemplo:

```
|?- element(2,[1,2,3,4],Y).
Y = 2 ? ;
no
|?- element(5,[1,2,3,4],Y).
no
|?- element(X,[1,2,3,4],2).
X = 2 ? ;
no
```

all_different(+Variables), all_distinct(+Variables): este predicado hace que los valores que toman las variables de la lista `Variables` sean todos distintos

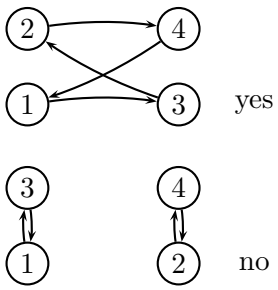
Ejemplo:

```
|?- all_different([2,3,4,5,6,7]).
yes
|?- all_different([2,3,4,5,6,7,2]).
no
```

circuit(+Succ), circuit(+Succ, +Pred): Circuito hamiltoniano. `Succ` (`Pred`) es una lista de N enteros o variables de dominio en la cual el elemento i-ésimo es el sucesor (o predecesor en orden inverso), del vértice i del grafo.

Ejemplo:

```
|?- circuit([3,4,2,1]).
yes
|?- circuit([3,4,1,2]).
no
|?- circuit([3,4,A,B]).
A = 2, B = 1 ? ;
no
| ?- circuit([3,4,2,1],[4,3,1,2]).
yes
| ?- circuit([3,4,2,1],[4,3,2,1]).
no
```



assignment(+Xs, +Ys): es aplicada sobre dos listas de longitud N donde cada variable toma un valor entre 1 y N que es único.

Ejemplo:

```
|?- assignment([A,B],[C,D]),labeling([ ],[A,B,C,D]).
A = 1,
B = 2,
C = 1,
D = 2?;
A = 2,
B = 1,
C = 2,
D = 1?;
no
```

serialized(+Starts,+Durations): Fabricar en serie. `Starts` es una lista de enteros que marca el comienzo de la tarea i-ésima y `Durations` es también una lista de enteros que representa el tiempo que se tarda en realizar la tarea i-ésima

Ejemplo:

```
|?- serialized([1,20,53], [12,24,34]).
yes
|?- serialized([1,20,53], [12,88,2]).
no
|?- domain([A,B],20,50),serialized([1,20,53],[12,A,B]),
labeling([],[A,B]).
A = 20, B = 20 ? ;
A = 20, B = 21 ? ;
A = 20, B = 22 ?
yes
```

cumulative(+Starts,+Durations,+Resources,?Limit): `Starts, Durations y Resources` son tres listas de enteros de la misma longitud N, `Starts y Durations`

representan lo mismo que en `Serializaed` el comienzo y la duración de las N tareas y `Resources` son los recursos, el total de los recursos consumidos no puede exceder de `Limit`. Serialized es un caso particular de cumulative

Ejemplo:

```
|?- cumulative([2,4,6],[4,6,8],[3,2,1],40).
yes
|?- cumulative([2,4,6],[4,6,8],[3,2,1],4).
no
```

Una vez visto los predicados y operadores, se va a mostrar a en la siguiente sección problemas que se pueden resolver con restricciones sobre dominios finitos.

2.4.1. Problemas Introductorios

Puzzles aritméticos

Hay clásicos puzzles aritméticos como puede ser el típico *send more money* o el puzzle que a continuación se muestra.

Cada letra de la siguiente suma expresa un dígito. Se trata de averiguar qué dígito corresponde a cada letra.

```
  D O N A L D
+ G E R A L D
--------------
= R O B E R T
```

Las Variables D, O, N, A, L, G, E, R, B, T tienen dominio finito $Dom(D, G, R) = \{1, .., 9\}$, $Dom(O, N, A, L, E, B, T) = \{0, .., 9\}$

Del enunciado se obtiene como retricción implícita que las 10 variables deben ser asignadas a valores diferentes. Es decir, hay una restricción de 'no igualdad' entre cada par de variables, dándonos 45 restricciones binarias.

Además está la restricción explícita de la suma

```
      100000*D + 10000*O + 1000*N + 100*A + 10*L + D
+     100000*G + 10000*E + 1000*R + 100*A + 10*L + D,
=     100000*R + 10000*O + 1000*B + 100*E + 10*R + T;
```

A continuación se muestra su código *Prolog*

```
:- use_module(library(clpfd)).
donald(Lab,LD):-
    LD=[D,O,N,A,L,G,E,R,B,T],
    domain(LD,0,9),
    domain([D,G],1,9),
    all_different(LD),
       100000*D+10000*O+1000*N+100*A+10*L+D +
       100000*G+10000*E+1000*R+100*A+10*L+D
    #= 100000*R+10000*O+1000*B+100*E+10*R+T,
    labeling(Lab,LD).
```

y su ejecución desde la línea de comandos.

```
|?- donald([],L).
L = [5,2,6,4,8,1,9,7,3,0] ? ;
no
```

Colorear un mapa

Este problema trata de colorear un mapa con uno ciertos colores, de tal forma que no existan dos regiones adyacentes del mismo color. Un ejemplo concreto es colorear el mapa de Australia. (figura 2.1)

Figura 2.1: Mapa de Australia

El dominio de las siete variables (WA, NT, SA, O, NSW, V, T) es {rojo, amarillo, azul}

Las restricciones que capturan el hecho de que no puede haber dos regiones adjuntas con el mismo color

$$WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q,$$
$$SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V$$

El grafo que representa este problema (figura: 2.2) tiene por nodos las regiones y las aristas representan las fronteras entre regiones.

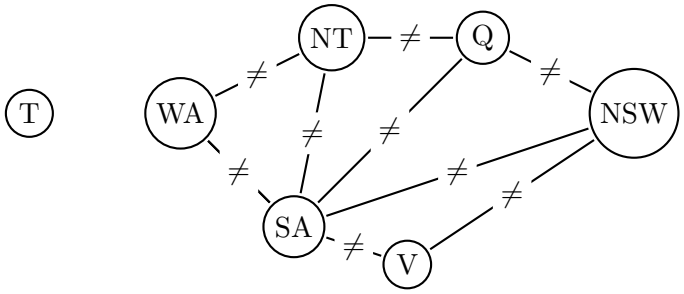


Figura 2.2: Grafo del mapa de Australia

Para codificar este problema en *Prolog* con restricciones sobre dominios finitos se ha de trabajar con los números 1,2 y 3 en vez de trabajar con los colores rojo, amarillo y azul.

```
:- use_module(library(clpfd)).
mapa_australia([T,WA,NT,SA,Q,V,NSW]) :- mapa([T,WA,NT,SA,Q,V,NSW]).
mapa([T,WA,NT,SA,Q,V,NSW]) :- domain([T,WA,NT,SA,Q,V,NSW], 1, 3),
    WA #\= NT, WA #\= SA, NT #\= SA, NT #\= Q, Q #\= SA, Q #\= NSW,
    NSW #\= SA, NSW #\= V, SA #\= V,
    labeling([],[T,WA,NT,SA,Q,V,NSW]).
```

El sistema da 18 soluciones al objetivo que a continuación se muestra, aquí sólo se muestran dos.

```
|?- mapa_australia([T,WA,NT,SA,Q,V,NSW],3).
Q = 1,
T = 1,
V = 1,
NT = 2,
SA = 3,
WA = 1,
NSW = 2 ? ;
Q = 1,
T = 1,
V = 1,
NT = 3,
SA = 2,
WA = 1,
NSW = 3 ? ;
```

Problema de las n reinas

En este problema se dispone de un tablero de dimensiones n_xn y de n reinas, se deben colocar las n reinas sobre el tablero, de tal forma que ninguna reina esté amenazada por otra reina.

Se define las variables R_i y C_i como la fila y la columna que ocupa la reina i.

Si se considera n=4, el dominio de las variables R_1 .. R_4, C_1 .. $_4$ es { 1, 2, 3, 4 }

La restricción de no amenaza entre dos las reinas, se descompone en las siguientes restricciones

- no puede haber dos reinas en la misma fila

$$R_1 \neq R_2, R_1 \neq R_3, R_1 \neq R_4, R_2 \neq R_3, R_2 \neq R_4, R_3 \neq R_4$$

- ni dos reinas en la misma columna

$$C_1 \neq C_2, C_1 \neq C_3, C_1 \neq C_4, C_2 \neq C_3, C_2 \neq C_4, C_3 \neq C_4$$

- ni dos reinas en la misma diagonal

$$C_1 - R_1 \neq C_2 - R_2, C_1 - R_1 \neq C_3 - R_3, C_1 - R_1 \neq C_4 - R_4,$$
$$C_2 - R_2 \neq C_3 - R_3, C_2 - R_2 \neq C_4 - R_4, C_3 - R_3 \neq C_4 - R_4$$
$$C_1 + R_1 \neq C_2 + R_2, C_1 + R_1 \neq C_3 + R_3, C_1 + R_1 \neq C_4 + R_4,$$
$$C_2 + R_2 \neq C_3 + R_3, C_2 + R_2 \neq C_4 + R_4, C_3 + R_3 \neq C_4 + R_4$$

El código correspondiente en *Prolog* es:

```
:- use_module(library(clpfd)).

queens(Lab, L, N) :-
    length(L,N),
    domain(L, 1, N),
    constrain_all(L),
    labeling(Lab, L).s
constrain_all([]).
constrain_all([X|Xs]):-
    constrain_between(X,Xs,1),
    constrain_all(Xs).
constrain_between(_X,[],_N).
constrain_between(X,[Y|Ys],N) :-
    no_threat(X,Y,N),
    N1 is N+1,
    constrain_between(X,Ys,N1).
no_threat(X,Y,I) +:
    X in \({Y} \/ {Y+I} \/ {Y-I}),
    Y in \({X} \/ {X+I} \/ {X-I}).
```

La solución a los objetivos con 4 y 5 reinas es

```
|?- queens([],L,4).
L = [2,4,1,3] ? ;
L = [3,1,4,2] ? ;
no
```

```
|?-queens([],L,5).
L = [1,3,5,2,4] ? ;
L = [1,4,2,5,3] ? ;
L = [2,4,1,3,5] ? ;
L = [2,5,3,1,4] ? ;
L = [3,1,4,2,5] ? ;
L = [3,5,2,4,1] ? ;
L = [4,1,3,5,2] ? ;
L = [4,2,5,3,1] ? ;
L = [5,2,4,1,3] ? ;
L = [5,3,1,4,2] ? ;
no
```

Emparejamientos

En este problema se tiene un conjunto de mujeres { María, Sara, Ana } y un conjunto de hombres { Pedro, Pablo, Juan}. Se trata de emparejar hombres y mujeres.

Se definen las variables X_{Maria}, X_{Sara} , X_{Ana}, de dominio { Pedro, Pablo, Juan}

Las restricciones implícitas son

$X_{Maria} \neq X_{Sara}$, $X_{Maria} \neq X_{Ana}$, $X_{Ana} \neq X_{Sara}$

Pero además existen unas restricciones explícitas que son reflejadas en el grafo de la figura 2.3, en el cual, las aristas representan las preferencias.

Para programar este ejercicio en *Prolog* con dominios finitos hay que representar las variables JUAN, PEDRO y PABLO como números enteros comprendidos entre 1 y 3. Hay que tratar de asignar a las variables MARIA, SARA, ANA números entre 1 y 3 con las

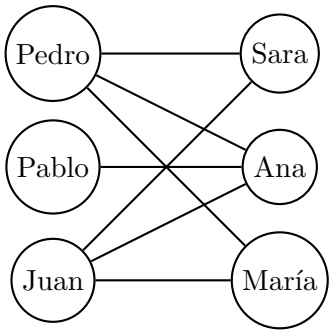


Figura 2.3: Grafo correspondiente al problema de emparejar

restricciones explícitas dadas y la retricción implícita de asignar valores distintos, para evitar emparejar a una mujer con dos hombres. El correspondiente código *Prolog* es el siguiente

```
:- use_module(library(clpfd)). parejas([MARIA, SARA, ANA],N) :-
domain([MARIA, SARA, ANA], 1, N),
    SARA #\= 2,
    MARIA #\= 2,
    all_different([MARIA,SARA,ANA]),
    labeling([],[MARIA, SARA, ANA]).
```

y su ejecución

```
|?- parejas([MARIA, SARA, ANA],3).
ANA = 2,
SARA = 3,
MARIA = 1 ? ;
ANA = 2,
SARA = 1,
MARIA = 3 ?;
no
```

Estas dos soluciones reflejan

X_{Ana} := Pablo, X_{Sara} := Juan y X_{Maria} := Pedro
X_{Ana} := Pablo, X_{Sara} := Pedro y X_{Maria} := Juan

2.4.2. Consistencia

Antes de empezar una búsqueda de la solución, conviene quitar de los dominios aquellos valores que nunca van a formar parte de la solución. Esto se consigue aplicando técnicas de consistencia.

Consistencia por arcos

Supongamos que existe una restricción binaria C_{ij} entre las variables x_i y x_j, entonces arc(x_i, x_j) es un *arco con consistencia* si por cada valor *a* perteneciente al dominio de la

variable x_i existe valor b perteneciente al dominio de la variable x_j, tal que la asignación $x_i := a$ y $x_j := b$ satisface la restricción C_{ij}

Es interesante quitar del dominio todos los valores que hacen que el arco no sea consistente. Reducir los dominios debería hacer que el problema se resuelva más fácilmente. Hay resolutores como por ejemplo ILOG Solver que realizan automáticamente la reducción del dominio haciendo que los arcos sean consistentes.

Ejemplo, [**?**]

$\{1..5\}$ —————— x < y -2 ——————- y$\{1..5\}$
Aplicamos consistencia al arco (x,y)
$\{1..2\}$ —————— x < y -2 ——————- y$\{1..5\}$
Aplicamos consistencia al arco (y,x)
$\{1..2\}$ —————— x < y -2 ——————- y$\{4..5\}$

Una aplicación real en la cual se pueden aplicar consistencia a los arcos es el típico problema de planificación.

Sean la actividades A, B, C, D con duración y precedencias descritas a continuación

Tarea	Duración	Precedencias	Restricciones
A	3	B	inicio(A)+Duración(A) $\leq$ inicio(B)
-	-	C	inicio(A)+Duración(A) $\leq$ inicio(C)
B	2	D	inicio(B)+Duración(B) $\leq$ inicio(D)
C	4	D	inicio(C)+Duración(C) $\leq$ inicio(D)
D	2	-	-

Podemos expresar la planificación como un problema con restricciones binarias introduciendo variables que representan el inicio y fin del proyecto y otras variables que representan el inicio y fin de cada actividad.

Variable	Dominio Inicial
inicio	0
inicioA, inicio B, inicioC, inicioD	0..11
final	0..11

Reducimos el dominio de estas variables haciendo consistencia de arcos, se muestra gráficamente en la figura 2.4

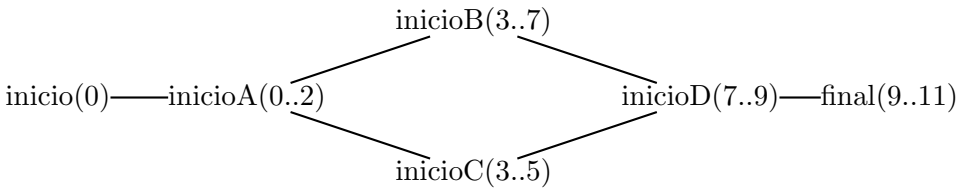


Figura 2.4: Consistencia por arcos

Si el problema es encontrar la forma de minimizar el tiempo, entonces tomaríamos la solución que nos muestra el valor mínimo de todos los posibles valores que toma la variable "final".

Por lo tanto en un problema con restricciones hacer consistencia por arcos es suficiente para quitar algunos valores pertenecientes al dominio que no forman parte de la solución.

Consistencia por caminos

En el problema que muestra la figura 2.5 se han aplicado consistencia por arcos para reducir los dominios de las variables, pero se puede observar que la variable x no puede tener el valor 2, porque si la variable x vale 2 entonces la variable y ha de tomar el valor 5 ($x < y$ - 2) y la variable z toma el valor 10 ($z \geq 5 * x$), w ha de valer 10 ($w \geq 5 * x$), y la restricción $w + z < 20$ no se cumple, por lo tanto x no puede tomar el valor 2. Sin embargo la consistencia por arcos no lo han detectado.

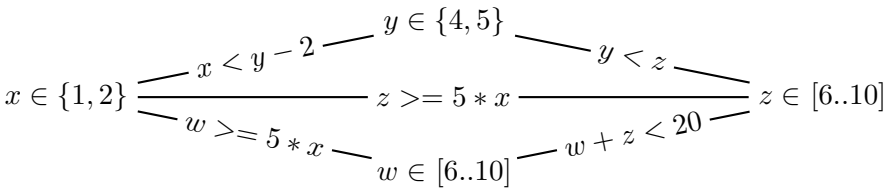


Figura 2.5: Consistencia por arcos con un valor que no forma parte de la solución

Hacer consistencia por caminos es un paso más para reducir el dominio de las variables. Sin embargo, hacer camino consistente puede tener un tiempo de complejidad alto. Por ejemplo, se consideran tres variables, dos pares de variables están relacionadas por una restricción no trivial.

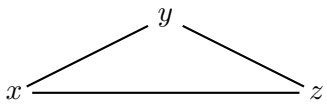


Figura 2.6: Disposición de las variables para hacer consistencia por camino

El camino (x,y,z) es un *camino consistente* si para cada par de valores:

a perteneciente al dominio de x

c perteneciente al dominio de z

hay un valor b perteneciente al dominio de y tal que (a, b) satisface la restricción que une x e y, y (b, c) satisface la restricción que une y y z. Si no se puede hallar el valor de b entonces a y c se quitan.

En el ejemplo anterior (figura: 2.5) si hacemos (x, w, z) un camino consistente, comprobamos que la x no puede ser el valor 2, ya que si x=2 tengo y = 5, z =10, w=10, entonces 10+10 < 20, y esto es una contradicción. Por lo tanto se debe eliminar el valor 2 del dominio de las x

Este algoritmo no se utiliza mucho porque tiene un tiempo de complejidad alto. En la práctica, se aplica la técnica del camino consistente en los casos semejantes a la figura 2.7 donde la consistencia por caminos muestra que la restricción entre x y z debe ser la restricción $x <> z$.

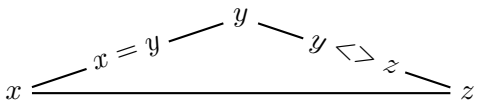


Figura 2.7: Caso en el cual se aplica consistencia por camino.

2.4.3. Propagación

Cada variable implicada en restricciones tiene un dominio asociado que se puede hacer menor razonando sobre los valores mínimos y máximos manteniendo la consistencia con dichas restricciones, a este método se le llama propagación. Por ejemplo, la restricción

$$x = y + z$$

con dominios `dom(x)={4..8}, dom(y)={0..3}, dom(z)={2..2}`
se puede escribir de tres formas `x=y+z, y=x-z, z=x-y` y razonando sobre el mínimo y el máximo de las partes derecha de los signos de igualdad tenemos

$$x \geq min(y) + min(z),\ y \geq min(x) - max(z),\ z \geq min(x) - max(y)$$
$$x \leq max(y) + max(z),\ y \leq max(x) - min(z),\ z \geq max(x) - min(y)$$

Aplicando consistencia a estas reglas y sustituyendo los valores máximos y mínimos obtenemos

$$2 \leq x \leq 5, 2 \leq y \leq 6, 1 \leq z \leq 8$$

con estas nuevas restricciones los dominios de reducen a

$$dom(x) = \{4, 5\}, dom(y) = \{2, 3\}, dom(z) = \{2\}$$

También se puede aplicar la propagación sobre restricciones con inecuaciones como por ejemplo

Problema de la mochila del contrabandista

Un contrabandista tiene una mochila limitada por la capacidad, 9 unidades. Los productos de contrabando son botellas de whiskey con una capacidad de 4 unidades , botellas de perfume de 3 unidades y cartones de cigarros de 2 unidades. Y el precio son 15, 10 y 7 $ respectivamente. El contrabandista sólamente se arriesga si gana 30 $ o más. ¿qué productos debe introducir en la mochila?

Para resolver este problema se toman las variables W, P y C para denotar el wishkey, perfume y cigarros respectivamente con $dom(W, P, C) =$[0 .. 9].
La restricción de la capacidad es: $4 * W + 3 * P + 2 * C \leq 9$
La restricción de las ganancias es: $15 * W + 10 * P + 7 * C \geq 30$
Aplicando propagación sobre la capacidad

$$W \leq 9/4 - 3/4 * min(P) - 2/4 * min(C)$$
$$P \leq 9/3 - 4/3 * min(W) - 2/3 * min(C)$$
$$C \leq 9/2 - 4/2 * min(W) - 3/2 * min(P)$$

$$W \leq 9/4 - 3/4 * 0 - 2/4 * 0$$
$$P \leq 9/3 - 4/3 * 0 - 2/3 * 0$$
$$C \leq 9/2 - 4/2 * 0 - 3/2 * 0$$

$$W \leq 2, P \leq 3C \leq 4$$

se reduce el dominio a: $dom(W)$ =[0 .. 2], $dom(P)$ =[0 .. 3] y $dom(C)$ =[0 .. 4].

aplicando propagación sobre las ganancias no se reduce el dominio

$$W \geq 30/15 - 10/15 * max(P) - 7/15 * max(C)$$
$$P \geq 30/10 - 15/10 * max(W) - 7/10 * max(C)$$
$$C \geq 30/7 - 15/7 * max(W) - 10/7 * max(P)$$

$$W \geq 30/15 - 10/15 * 3 - 7/15 * 5$$
$$P \geq 30/10 - 15/10 * 2 - 7/10 * 5$$
$$C \geq 30/7 - 15/7 * 2 - 10/7 * 3$$

$$W \geq -7/15 * 5, P \geq -7/10 * 5, C \geq -10/7 * 3$$

el dominio sigue siendo $dom(W)$ =[0 .. 2], $dom(P)$ =[0 .. 3] y $dom(C)$ =[0 .. 4].

Otro caso interesante de propagación es la *desigualdad*. Sea la restricción $x \neq y$ con $dom(x) = \{2, 3\}$ y $dom(y) = \{2\}$. Aplicado propagación se obtiene $x \neq min(y)$ reduciendo el dominio $dom(x) = \{3\}$ y $dom(y) = \{2\}$

Aplicar propagación sobre otros tipos de restricciones puede ser muy costoso.

2.4.4. Etiquetado

El ejemplo visto en la sección 2.4.3 del problema de la mochila del contrabandista los dominios se reducen pero no se obtiene una respuesta concreta, esto es debido a que el resolutor de dominios finitos es incompleto. La razón por la cual los sistemas proporcionan resolutores incompletos es porque proporcionar un resolutor completo puede ser muy costoso.

Por otra parte, es habitual que se requiera una solución si existe, la optima o todas las soluciones, por este motivo se puede disponer de un resolutor completo para los dominios finitos que utiliza el backtracking o vuelta atrás. Estos algoritmos buscan la soluciones *sistemáticamente* mediante la asignación de valores a variables y garantizan una solución si existe, la optima o todas, pero si no existe solución pueden tardar mucho tiempo en mostrar que no hay solución.

Los siguientes predicados dan variantes de la búsqueda sistemática.

indomain(?X) asigna valores factibles del dominio a la variable `X` vía backtracking.

labeling(:Options, +Variables) donde `Variables` es una lista de variables y `Options` es una lista de opciones de búsqueda. Es true si exite una asignación de valores del dominio a las variables que hagan satisfactible las restricciones. Las opciones (`Options`) controlan la naturaleza de la búsqueda y están divididas en cuatro grupos.

El primer grupo de `Opciones` controla el orden en el cual las variables son seleccionadas para asignarles valores del dominio 2.4.6

- `leftmost`, toma las variables en el orden en el cual se definieron, este es el valor por defecto.
- `ff` usa el principio first-fail, selecciona la variable de menor dominio.
- `ffc` extiende la opción `ff` seleccionando la variable envuelta en el mayor número de restricciones.
- `min` la variable con el valor menor.
- `max` la variable con el valor mayor.

El segundo grupo de `Opciones` controla el orden en el cual los valores de las variables son seleccionados 2.4.7

- `enum`, elige los valores desde el menor hasta el mayor.
- `step`, selecciona el mayor o el menor valor.
- `bisect`, divide al dominio en dos y toma el valor medio
- `up`, controla que el dominio sea explorado en orden ascendente. Este valor se toma por defecto.
- `down`, controla que el dominio sea explorado en orden descendente.

El tercer grupo de `Opciones` controla cuántas soluciones se desean o cuales.

- `all`, se desean todas las soluciones, este valor es por defecto.
- `minimize(X), maximize(X)`, utiliza un algoritmo "branch-and-bound"para encontrar una solución que minimice (maximice) la variable `X`

El cuarto grupo de `Opciones` controla el número se asunciones

- `assumptions(K)`, K es el número de asunciones hechas durante la búsqueda.

2.4.5. Algoritmos de Búsqueda

En esta sección se van a mostrar dos algoritmos de búsqueda sistemática de la solución

- algoritmo de backtracking simple
- algoritmo de backtracking con comprobación previa (forward checking)

Ambos algoritmos instancian una variable en cada paso, y así van haciendo una construcción parcial de la solución, cuando la solución parcial no es parte de la solución final entonces se hace vuelta atrás y se prueba con otros valores pertenecientes a los dominios de las variables que forman la solución parcial.

Backtracking simple

En cada paso del algoritmo de búsqueda backtracking simple se toma una variable y se asigna un valor del dominio a dicha variable, se comprueba si forma parte de la solución parcial, es decir, si este valor junto con los valores de la solución parcial ya construida, verifican las restricciones. Si no verifican todas las restricciones entonces se desecha ese valor y se toma otro valor perteneciente al dominio, este proceso se repite hasta encontrar un valor que verifique las restricciones o hasta probar con todos los valores del dominio. Si se verifican las restricciones entonces este valor pasa a formar parte de la solución parcial, se toma otra variable para instanciar y se repite el proceso con la nueva variable instanciada. Si se intentan todos los posibles valores de una variable y ningún valor verifica las restricciones, entonces la variable no forma parte de la solución, por lo tanto se pasa a la variable anteriormente evaluada (backtracking) y se toma un nuevo valor de esta variable. Este proceso se repite hasta que se encuentre una solución, es decir, todas las variables están instanciadas, o hasta que se hayan probado todos los posibles casos.

Este algoritmo sólo evalúa si se satisfacen las restricciones entre la última variable tomada y las variables evaluadas anteriormente.

Un ejemplo de backtracking simple es: dadas las variables x, y y z de dominios $\{1, 2\}$ y las restricciones $x < y$ y $y < z$. Calcular los valores que verifican las restricciones.

Se elige una variable: x
Se elige un valor del dom(x) (1)
se sustituye en $x < y, y < z$; $1 < y, y < z$ es satisfactible
Se elige una variable: y
Se elige un valor del dom(y) (1)
se sustituye en $1 < y, y < z$; $1 < 1, 1 < z$ no es satisfactible
Se elige otro valor del dom(y) (2)
se sustituye dicho valor en $1 < y, y < z$; $1 < 2, 2 < z$ es satisfactible
Se elige una variable: z
Se elige un valor del dom(z) (1)
se sustituye en $1 < 2, 2 < z$; $1 < 2, 2 < 1$ no es satisfactible
Se elige otro valor del dom(z) (2)
se sustituye en $1 < 2, 2 < z$; $1 < 2, 2 < 2$ no es satisfactible

Al no haber más valores para z significa que $1 < 2, 2 < z$ no es satisfactible. Tampoco hay más valores para y, por lo tanto $1 < y, y < z$ no es satisfactible.

Se elige otro valor del dom(x) (2)
se sustituye en $x < y, y < z$; $2 < y, y < z$ es satisfactible
Se elige una variable: y
Se elige un valor del dom(y) (1)
se sustituye en $2 < y, y < z$; $2 < 1, 1 < z$ no es satisfactible
Se elige otro valor del dom(y) (2)
se sustituye en $2 < y, y < z$; $2 < 2, 2 < z$ no es satisfactible

El algoritmo de backtracking devuelve que este problema no es satisfactible.

El árbol de búsqueda para este ejemplo de backtracking es el que se muestra en la figura 2.8

Otro ejemplo de backtracking simple es el problema de las reinas visto en la sección 2.4.1. La figura 2.9 es la representación del árbol de búsqueda de este problema con 4 reinas, en esta figura sólo se muestra dos ramas del árbol ya que las otras dos son simétricas.

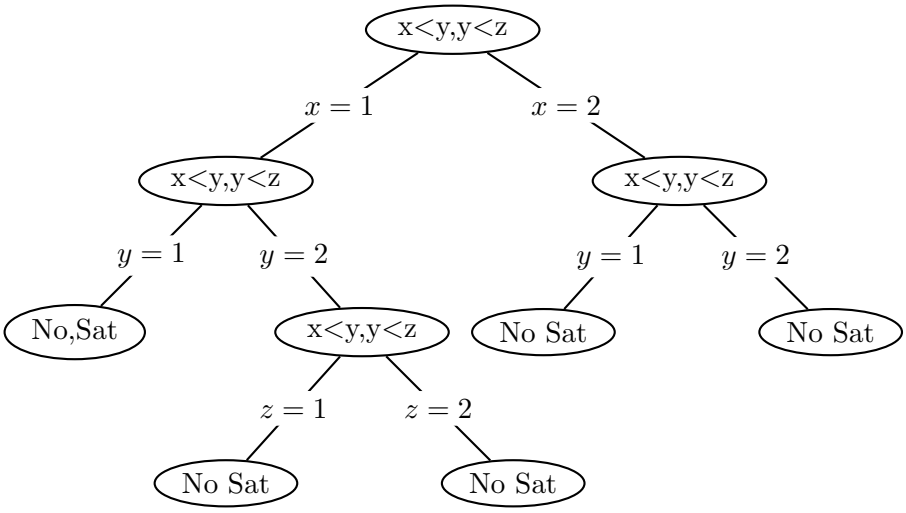


Figura 2.8: Árbol de búsqueda

Este resolutor es completo pero en el peor de los casos tiene un tiempo de ejecución exponencial.

Backtracking con comprobación previa

En el algoritmo de búsqueda backtracking simple se evaluan las restricciones que implican a las variables que forman parte de la solución parcial con la variable que se va a instanciar en ese momento. En el algoritmo de búsqueda backtracking con chequeo previo se evalúan las restricciones que implican a la variable actual con las variables anteriormente evaluadas y las variables que se evaluaran en el futuro, es decir, con todas las variables.

Cuando un valor es asignado a la variable en evaluación, y algún valor de una futura asignación tiene conflicto con esta asignación entonces se quita el valor de la futura asignación temporalmente del dominio.

La ventaja de este algoritmo es que si el dominio de una futura variable llega a ser vacío, esto significa que la solución parcial es inconsistente, con lo cual se prueba con otro valor de la variable activa o se hace vuelta atrás,según corresponda, y el dominio de las futuras variables es restaurado. Con el backtracking simple, no se habría detectado la inconsistencia hasta que no se hubiesen evaluado las futuras variables. El algoritmo de backtracking con comprobación previa es más eficiente respecto al algoritmo de backtracking simple ya que las ramas de los árboles que van a dar una inconsistencia son podadas con anterioridad.

Tenemos 4 un vector de 4 posiciones dónde la posición indica la fila y el valor la columna. El dominio de estas variables es 1..4

Colocamos la primera reina en la primera columna, la solución parcial será (1, , ,), por lo tanto las demás reinas no se pueden colocar en la primera columna, esto implica que su dominio se reduce 2..4 y además no se pueden colocar en diagonal, por lo tanto que excluyen las casillas (2,2), (3,3), (4,4).

En el árbol se marcan como x las casillas que han sido borradas del dominio. El árbol de búsqueda ha sido podado.

En general, si todas las restricciones son binarias, entonces solamente las restricciones entre la variable activa y las variables por evaluar necesitan ser comprobadas, con las

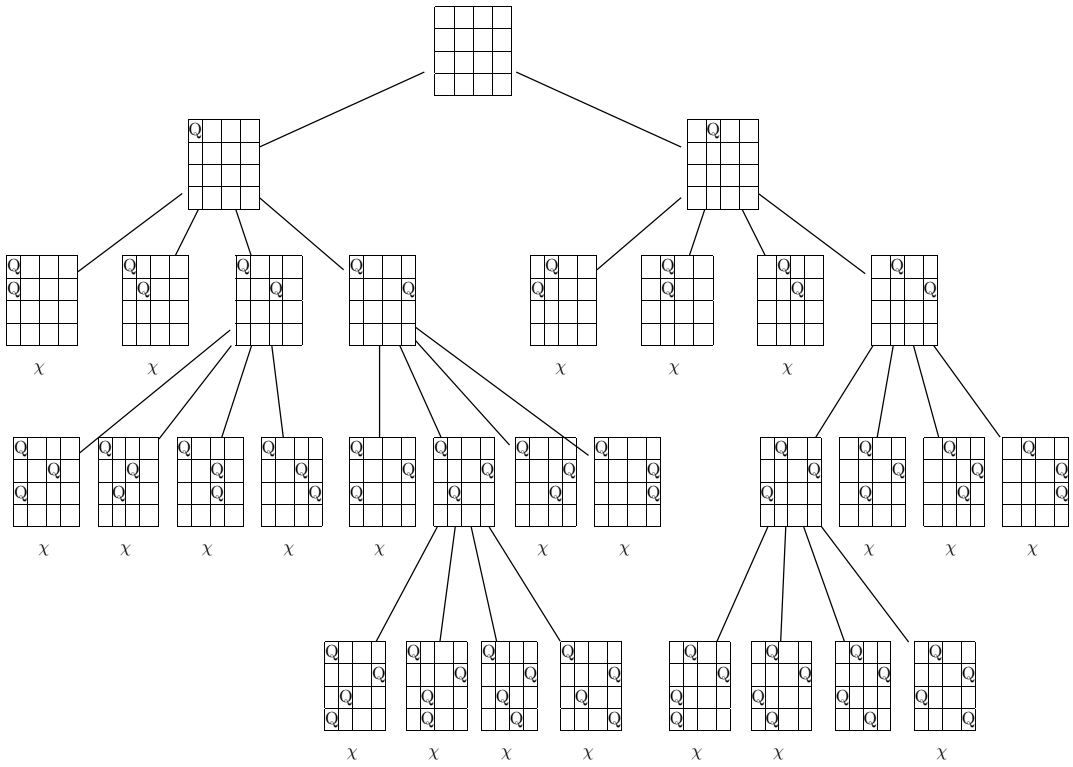


Figura 2.9: Problema de las 4 reinas con backtracking simple

variables anteriores no es necesario comprobar, ya se habían comprobado anteriormente.

En el ejemplo de las 4 reinas, el backtracking con comprobación previa realiza más o menos el mismo trabajo para evaluar la consistencia de las restricciones que el backtracking simple. Sin embargo vamos a ver un ejemplo dónde backtracking con comprobación previa ahorra mucho trabajo con respecto a el backtracking simple.

Ejemplo: sean x_1,x_2,....,x_n variables tal que, dom(x_1,x_2,....,x_n) = {1,2} y sea la restricción "todas deben tener diferente valor". ¿qué valores pueden tomar x_1,x_2,....,x_n?

Si se aplica backtracking simple, se asigna x_1=1, x_2=2, y continua intentando asignar x_3,... ,x_n y cuando la asignación de x_n sea inconsistente entonces se hará backtracking a x_{n-1} y se intentará la otra alternativa para esta variable, esto se repite hasta agotar todas las posibilidades. Esta técnica no es muy eficiente porque no es capaz de detectar la inconsitencia hasta que prueba con todos los casos.

Si se aplica backtracking con chequeo previo, se asigna x_1=1, entonces dom(x_2,....,x_n) = {2}, en el siguiente paso se asigna x_2=2, haciendo dom(x_3,....,x_n) = {}, al llegar a ser el conjunto vacío el dominio de las futuras variables se hace vuelta atrás y prueba con el valor que nos queda x_1=2 entonces dom(x_2,....,x_n) = {1}, en el siguiente paso se asigna x_2=1, haciendo dom(x_3,....,x_n) = {}, ya no se puede hacer vuelta atrás porque ya se han probado todos los posibles valores de la variable x_1 y ningún valor lleva a la solución, por

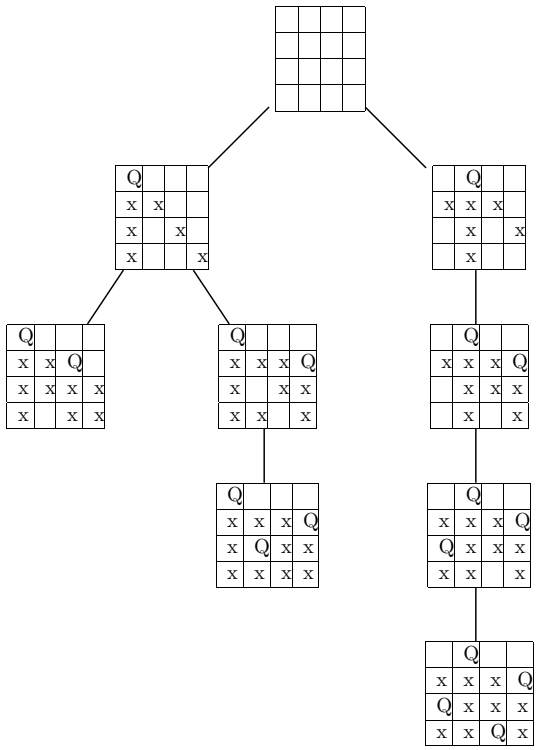


Figura 2.10: Problema de las 4 reinas con backtracking con comprobación previa

lo tanto no hay más comprobaciones haciendo que este algoritmo sea más eficiente.

Se pueden mejorar los algoritmos de backtracking combinándolos con los algoritmos de consistencia introducidos en 2.4.2. Inicialmente se aplica consistencia para reducir el dominio de las variables y cada vez que el algoritmo de backtracking elija un valor para una variable se vuelve a aplicar consistencia para restaurar los dominios de las variables. Si se llega a dominio vacio, quiere decir que la solución parcial es insatisfactible, se hace vuelta atrás y se aplica el algoritmo de nuevo.

2.4.6. Ordenación de Variables

Un algoritmo de búsqueda en un árbol que satisface ciertas restricciones requiere que las variables que van a ser consideradas estén ordenadas. Debe de existir un orden para poder decidir cual es la siguiente variable a evaluar en el paso de vuelta atrás, aunque sea el orden en el cual las variables fueron definidas.

El orden puede ser:

- estático, si se ha de especificar antes de comenzar la búsqueda
- dinámico, si la próxima variable a considerar depende del estado de la búsqueda. Y no siempre es factible.

En el backtracking simple no hay suficiente información durante la búsqueda para hacer una elección distinta de la ordenación inicial. Sin embargo en el estado activo del

backtracking con comprobación previa se encuentra la información en los dominios de las variables y esto es una base para elegir la próxima variable a evaluar.

Una heuristica para la ordenación de las variables está basada en el principio "fail-first"de *Haralick and Elliott*, [15], que explicaron como "para tener éxito, hay que manejar primero aquello que nos lleva más fácilmente al fracaso". En el backtracking con comprobación previa este principio está implementado ya que elige la variable de *menor dominio*.

Si la solución parcial no nos dirige a la solución, entonces la rama será una rama muerta, cuanto antes se descubra esto mejor. Se intentará crear la solución con la solución parcialmente construida y las variables que quedan por instanciar. Tomar la variable de menor dominio hará que se necesiten menos comprobaciones para detectar si la rama es muerta.

Por otra parte, si la solución parcial puede ser expandida a una solución completa, entonces cada variable debe ser instanciada. Si se toma primero la variable con mayor dominio lo más probable será tener que hacer más instancias. Si el problema tiene solución, para cada variable existira al menos un valor que forme parte de alguna solución, ese valor será más sencillo encontrarlo a partir de un dominio menor. Cuando la solución parcial permita una solución completa, interesa tomar la solución directamente sin fomentar backtracking.

Esta heurística podría reducir la media de la profundidad de las ramas de los árboles provocando fallos tempranamente. Por lo tanto, aún no habiendo solución, o pidiendo todas las soluciones, el tamaño del árbol que se explora es menor que si se usa una ordenación estática.

Cuando todas las variables tienen el mismo número de valores en sus dominios, se debería elegir la variable que participa en más restricciones.

2.4.7. Ordenación de Valores

Habiendo seleccionado la próxima variable para asignar valor, un algoritmo de búsqueda tiene que seleccionar un valor para asignarlo.

Los valores pueden ser asignados en el orden en el cual aparecen en el dominio. O podríamos decidir cual es el orden en el cual los valores podrían ser asignados.

Una ordenación distinta, cambiaría las ramas del árbol. Esto es bueno si sólo se quiere una solución. Pero si el árbol hay que recorrerlo entero porque se desean todas las soluciones o la solución optima (ya que se deben conocer todas las soluciones para saber cual es la mejor), o en el caso que no exista solución, entonces el orden de los valores no tiene importancia.

Para problemas de generación aleatoria y de probabilidad, en general no es rentable tener una heurística para elegir el valor que me llevará a la solución, debido al coste de la comprobación previa de cada posible valor.

En ciertos problemas, puede haber información disponible que permita ordenar los valores de acuerdo al principio de elegir primero aquellos valores con mas probabilidad de tener éxito.

2.4.8. Simetrías

En muchos problemas, si existe una solución, existe una clase de soluciones equivalentes. Si se explora una rama de un árbol que no lleva a solución, sus ramas simétricas tampoco tendrán solución. Análogamente si una rama lleva a una solución sus ramas simétricas llevan a solución.

La simetría ha de ser quitada, si es posible, añadiendo restricciones en la formulación del problema permitiendo sólo una solución por cada clase de soluciones equivalentes.

Es difícil dar una pauta general de cómo hacer esto, porque depende de cada problema en particular.

Suponganse dos variables x e y con dominios [1..10]. La restricción $x + y = 6$ es representada por la siguiente tabla

x	1	2	3	4	5
y	5	4	3	2	1

Las soluciones (1,5) y (5,1) son simétricas. Si se añade la restricción $x \leq y$ la tabla se reduce a:

x	1	2	3
y	5	4	3

2.4.9. Optimización

Normalmente, todas las herramientas de programación con restricciones, adoptan la misma estrategia para encontrar la solución óptima. Toman una solución, (solución inicial), continúa el proceso, cuando se encuentra otra solución se deja la solución mejor, y así sucesivamente. La ultima solución es entonces la solución optima.

Tener una buena heurística que guíe a la solución optima es mejor que comprobar todas las soluciones, pero esto no siempre es posible.

Capítulo 3

El Lenguaje $\mathcal{TOY}$

En este capítulo se definen los conceptos básicos de $\mathcal{TOY}$: las expresiones, los tipos, las funciones y los objetivos. Además se exponen las principales caracteristicas de $\mathcal{TOY}$, estas son la evaluación perezosa, estructuras de datos infinitas, funciones indeterministas o multivaluadas, funciones de orden superior, restricciones de igualdad, desigualdad y aritméticas sobre los números reales.

3.1. Introducción

$\mathcal{TOY}$ es un lenguaje y sistema multiparadigma diseñado para soportar los principales estilos de programación declarativa [21], en el cual se combina la notación relacional mediante definiciones de predicados al estilo *Prolog* y la notación funcional mediante funciones cuya sintáxis está basada en *Haskell*. Los programas $\mathcal{TOY}$ pueden usar restricciones al estilo CLP sobre predicados y funciones. Actualmente, el sistema $\mathcal{TOY}$ soporta las restricciones de igualdad, desigualdad y aritméticas sobre los números reales. Las restricciones sobre dominios finitos es la extensión de $\mathcal{TOY}$ de la cual trata este trabajo.

$\mathcal{TOY}$ se basa en el estrechamiento perezoso [19] o lazy narrowing como mecanismo operacional, este mecanismo retrasa la evaluación de los argumentos de las funciones hasta que sea imprescindible esta evaluación para el computo de dicha función. En los programas que se utilizan restricciones el mecanismo de estrechamiento perezoso es combinado con el resolutor de restricciones y se puede obtener como respuesta a un objetivo una restricción en forma resuelta.

$\mathcal{TOY}$ tiene características que no se encuentran en la programación lógica ni en la programación funcional como son las funciones no deterministas que combinadas con la evaluación perezosa ayudan a mejorar la eficiencia de los problemas de búsqueda y otra característica propia de $\mathcal{TOY}$ son los patrones de orden superior.

3.2. Expresiones

Un programa $\mathcal{TOY}$ está construido con ayuda de expresiones, estas expresiones están formadas con variables y símbolos de función. Las variables comienzan por mayuscula y los símbolos de función comienzan por minúscula al igual que los tipos de datos. Un subconjunto importante de los símbolos de función son los símbolos de constructoras que son funciones que no necesitan reglas que la definan. Los predicados en $\mathcal{TOY}$ son tratados como funciones con resultado booleano.

Se denota con $c \in DC^n$ a un símbolo de constuctora c de n argumentos, $f \in FS^n$ es un símbolo de función de n argumentos. Los números y los caracteres son constructoras de aridad 0, (DC^0).

En $\mathcal{TOY}$ una expresión es una variable X o es un símbolo de constructora c o un símbolo de función f o la aplicación de una expresión e sobre otra expresión e_1 o una tupla $(e_1, ..., e_n)$ de expresiones.

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

Las aplicaciones de las funciones asocian por la izquierda y los paréntesis pueden ser omitidos debido a que $\mathcal{TOY}$ utiliza notación currificada. El siguiente predicado de *Prolog* necesita los paréntesis.

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

El mismo predicado en $\mathcal{TOY}$ se correspondería con:

1. Declaración de tipos. Si los tipos no son declarados entoncés serán inferidos. *Prolog* no tiene declaración de tipos.

```
append :: [A] -> [A] -> [A]
```

2. Reglas de definición.

```
append [] Ys = Ys
append [X|Xs] Ys = [X|append Xs Ys]
```

Además $\mathcal{TOY}$ posee operadores infijos y expresiones condicionales como `if then else`, los cuales se detalla en la sección 3.4.

Como subclases de expresiones están los *terminos*

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

que corresponden a los patrones de los lenguajes funcionales. Además existe otra subclase de expresión, los patrones de orden superior dónde los patrones pueden ser constructoras y funciones aplicadas parcialmente.

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

3.3. Tipos

$\mathcal{TOY}$ es un lenguaje de programación tipado basado en el polimorfismo de tipos de Damas-Milner ([4]). La sintáxis de los tipos en $\mathcal{TOY}$ se define de la siguiente manera.

Sea A una variable de tipos y $\delta \in TC_n$ un símbolo de constructora de tipos de aridad n.

$$\tau ::= A \mid \delta\ \tau_1 ... \tau_n \mid (\tau_1, ..., \tau_n) \mid (\tau_1 -> \tau)$$

$(\tau_1, ..., \tau_n)$ corresponde al producto cartesiano de los correspondientes tipos y $(\tau_1 -> \tau)$ corresponde al tipo de las funciones donde τ_1 es el tipo del argumento, τ el tipo del retorno

y $->$ es una constructora predefinida infija que asocia a la derecha. La notación $e :: \tau$ denota que la expresión e tiene el tipo τ

El algoritmo de inferencia de tipos descrito en el apendice E de [21] decide si una expresión está bien tipada, de ser asi se computa el tipo más general τ de e. Los tipos aportan expresividad al lenguaje y además ayudan a detectar los errores de programación.

Tipos de datos

Los tipos de datos se definen con la palabra reservada *data*

$$data \quad \delta\ A_1...A_n = c_0\tau_{0,1}...\tau_{0,m_0}|...|c_k\ \tau_{k,1}...\tau_{k,m_k} \qquad m_i \geq 0 \quad \forall i \in \{1..k\}$$

donde δ es el identificador del tipo y como es natural no puede haber dos identificadores con el mismo nombre. $A_1...A_n$ son las variables del tipo que se está definiendo y ha de ser lineal, es decir, no puede haber dos variables iguales. Además, $c_i(0 \leq i < k)$ son constructoras y $\tau_{i,1}...\tau_{i,n_i}$ una serie de tipos que correponden a las constructoras c_i. Las variables que están en el lado derecho de la declaración han de estar en el lado izquierdo de la declaración.

Si $n = 0$ entonces lo que se está definiendo es un tipo constante. Al ser constante es no polimorfico, un ejemplo sería

```
data pairInt = p int int
```

que introduce `pairInt` como un nuevo tipo sin parámetros y además también introduce un símbolo de constructora `p`.

Si $n = 0, m_i = 0 \quad \forall i \in \{1..k\}$ entonces se están definiendo constructoras constantes o tipos enumerados, por ejemplo

```
data taskName = t1 | t2 | t3 | t4 | t5 | t6
data resourceName = m1 | m2
```

que introduce `taskName` como un nuevo tipo para los datos y `t1 ... t6` son las constructoras o valores de dichos tipos.

Un tipo de dato es llamado *polimorfico o genérico* si $n > 0$ y tiene variables del tipo polimórfico, por ejemplo la variable A

```
data pair A = p A A
```

Los tipos de datos también se pueden definir recursivamente, la definición de los números naturales es recursiva.

```
data nat = zero | suc nat
```

Sinónimos de tipos o alias

Los alias permiten declarar un nuevo identificador para otro tipo y se utilizan para simplificar la forma de nombrar al tipo o porque se le quiera dar un nombre más significativo, por ejemplo `string` es más significativo que `[char]`.

La sintáxis de un alias es

$$type\ \nu\ A_1...A_n = \tau$$

donde ν es el identificador del alias, $A_1...A_n$ son los parámetros del alias, estas variables han de ser distintas para que sea lineal y τ es la construcción de la cual se está haciendo el alias, las variables de τ deben estar contenidas en $A_1...A_n$.

Un ejemplo es definir una cierta región del plano que se quiere embaldosar mediante un par de coordenadas cartesianas. El par `(x,y)` delimita la región comprendida entre: `(0,0)`, `(0,y)`, `(x,0)` y `(x,y)`.

```
type region = (int, int)
```

Al igual que en la definición de los tipos de datos, la parte izquierda ha de ser lineal, por lo tanto no puede haber dos variables repetidas y las variables de la parte derecha han de estar en la parte izquierda. Además no puede haber recursión.

Tipos predefinidos

$\mathcal{TOY}$ tiene incluidos los siguientes tipos predefinidos

- `bool`, tipo definido como `bool = true | false`
- `int, real`, tipos correspondientes a los números enteros y reales, están definidos a bajo nivel y no son visibles al usuario.
- `char`, tipo de caracteres, deben ir entre comillas simples como por ejemplo 'a', '1', '@'. Algunos caracteres corresponden a las secuencias de control y no son signos visibles, estos caracteres se escriben utilizando la secuencia de escape que corresponde con la barra invertida '\'.

Caracter $\mathcal{TOY}$	*significado*
'\\'	barra invertida
'\''	comilla
'\"'	comilla doble
'\n'	salto de línea
'\r'	retorno de carro
'\t'	tabulador horizontal
'\v'	tabulador vertical
'\f'	salto de pagina
'\a'	señal acústica
'\b'	espacio
'\d'	borrado
'\e'	escape

- `[A]`, tipo definido para las listas polimórficas, en su definición se utiliza variables del tipo polimórfico `A`, la constructora constante `[ ]` para las listas vacías y el operador infijo `:` para construir las listas no vacías `(X:Xs)` con la cabeza `X :: A` y la cola `Xs :: [A]`.

```
data [A] = [ ] | A : [A]
```

La notación al estilo de *Prolog* `[X|Xs]` es un alias de (X:Xs).

- `tuplas`, se pueden definir tuplas de una aridad determinada, como por ejemplo

```
data tuplas A B C D = t A B C D
```

El tipo cadena de caracteres no es un tipo predefinido de $\mathcal{TOY}$ pero está definido en el archivo *misc.toy* distribuido con el sistema. Un string se define como una lista de caracteres utilizando un Alias.

```
type string = [char]
```

3.4. Funciones

Para definir una función en $\mathcal{TOY}$ primero se expresa la declaración de tipos de la función y posteriormente las reglas de definición.

La declaración de tipos es opcional y si no es escrita entonces el sistema $\mathcal{TOY}$ toma el tipo más general posible mediante su propio *inferidor de tipos*. Si los tipos son declarados el tipo inferido ha de contener al tipo declarado porque de no ser así existiría conflicto de tipos. Una de las utilidades de declarar tipos muy generales es que podemos utilizar una misma función genérica y particularizarla sobre diferentes tipos concretos.

La sintaxis de la definición de una función es

$$\begin{array}{rcl} f & :: & \tau_1 -> \ldots \tau_n -> \tau \\ & \ldots & \\ ft_1 \ldots t_n & = & r <== C_1, \ldots, C_m \; where \; D_1, \ldots, D_p \\ & \ldots & \end{array}$$

es decir, $ft_1 \ldots t_n$ puede ser reducido a r si las restricciones $C_1, \ldots, C_m$ son satisfechas utilizando las definiciones locales $D_1, \ldots, D_p$.

$f \in FS^n$ es un símbolo de función de aridad n. Las reglas de definición han de ser lineales en su lado izquierdo, es decir, los patrones $t_1 ... t_n$ no pueden contener variables repetidas. Los patrones son expresiones que no contienen llamadas a funciones, esto quiere decir que son expresiones irreducibles o formas normales. Si $\mathcal{TOY}$ encuentra una variable repetida automáticamente la sustituye por una nueva variable e introduce una restricción de igualdad. Por ejemplo `g A A = 0` es traducida a `g A B = 0 <== A == B`.

El lado derecho de la regla de definición (r) está formado por expresiones construidas con variables, operadores, constructoras y funciones definidas. C_i son *expresiones condicionales o restricciones* y D_i son *definiciones locales*. Si la regla de definición no necesita condicionales se omite <== y si no necesita definiciones locales se omite *where*.

C_i son expresiones de la forma $e_1 \diamond e_2$ con $\diamond \in \{==, /=, <, >, <=, >=\}$. La expresión $e_1 == e_2$ representa la igualdad estricta y se resuelve evaluando e_1 y e_2 a un patrón común. Si una expresión es de la forma $e_1 == true$ se puede simplificar a e_1. Una explicación más detallada de la igualdad y la desigualdad $e_1 / = e_2$ se verá en 3.10.

D_i es una definición local de la forma $s_i = d_i$ donde d_i es una expresión y s_i es o es una variable o una constructora de aridad l. Se evalúan las expresiones d_i y se vinculan a las variables que hay en s_i

$$s_i = X | (cX_1, \ldots, X_l)$$

Además se ha de cumplir

- $\forall i\ 1 \leq i \leq p; s_i$ y $(ft_1...t_n)$ no tienen variables comunes. Es decir, las variables que se definen locales y las variables de la parte izquierda de la definición de la regla han de ser distintas.
- $\forall i\ 1 \leq i < j \leq m; s_i$ y s_j no tienen variables comunes. Las variables que se definen locales no coinciden con las variables de las restricciones.
- $\forall i\ 1 \leq i \leq m; 1 \leq j \leq i; s_i$ y d_j no tienen variables comunes. Es decir, las variables definidas en la definición local i puede ser usadas en la definición local j si $i < j$, esto quiere decir que las definiciones locales no pueden ser recursivas.

Un ejemplo de las definiciones locales puede ser calcular las raices de la ecuación $AX^2 + BX + C = 0$, la regla de definición utiliza la regla predefinida `sqrt` de la cual se hace referencia en 3.6.

```
type coeff = real
type root  = real
roots :: coeff -> coeff -> coeff -> (root,root)
roots A B C = ((-B-D)/E,(-B+D)/E) <== E <> 0
                    where D = sqrt(B*B-4*A*C), E = 2*A
```

La variable que se define localmente (D) es distinta a las variables de la parte izquierda de la definición (A, B y C), también es distinta a la variable que se implica en la restricción (E) y por supuesto no hay recursión en la difinición local. Por lo tanto se cumplen las restricciones impuestas a la definición de variables locales.

Aunque las funciones en $\mathcal{TOY}$ se definen como se acaba de mostrar, hay que tener en cuenta que $\mathcal{TOY}$ admite azucares sintácticos y predicados al estilo *Prolog*.

La expresión condicional `if b then` e_1 `else` e_2 es un azucar sintáctico de la función

```
if-then-else :: bool -> A -> A -> A
if-then-else true X Y  = X
if-then-else false X Y = Y
```

Un ejemplo de la aplicación de este azúcar sintáctico puede ser la función infija (!!) que está definida en el fichero *misc.toy* disponible con la distribución de $\mathcal{TOY}$

```
%% Xs!!N is the Nth-element of Xs
(!!) :: [A] -> int -> A
[X|Xs] !! N = if N==0 then X else Xs !! (N-1)
```

Como ya se dijo antes, los predicados en $\mathcal{TOY}$ son funciones que devuelven un valor booleano. Además $\mathcal{TOY}$ permite utilizar cláusulas con la parte derecha a `true` para definir reglas porque en $\mathcal{TOY}$ una cláusula no puede ser vacía.

Cada cláusula

$$p\ t_1 \ldots t_n\ :-\ C_1, \ldots, C_m\ where\ D_1, \ldots, D_p$$

es una regla de definición donde las condiciones corresponden al cuerpo de la cláusula

$$p\ t_1 \ldots t_n = true <== C_1, \ldots, C_m\ where\ D_1, \ldots, D_p$$

Un simple ejemplo puede ser

```
type plato = string
type comida = string
servir :: (plato,comida) -> bool
servir("primero","judias")   :- true
servir("primero","lentejas") :- true
```

3.5. Objetivos

Un objetivo en $\mathcal{TOY}$ es una serie de restricciones $C_1, ..., C_n$ de igualdad $e_1 == e_2$ o desigualdad $e_1 / = e_2$ que se quieren resolver. Si C_i es únicamente una expresión booleana e entonces el sistema entiende que es una restricción de igualdad $e == true$.

Si el objetivo es insatisfactible entonces el sistema responde *no*, y en el caso que sea satisfactible responde *yes* y muestra las restricciones computadas en forma resuelta como se detalla en 3.10, continuando con el cómputo si así lo desea el usuario hasta encontrar otra solución por *backtracking* o finalizar el cómputo.

Una de las caracteristicas importantes de la programación lógico funcional relacionada con los objetivos es *utilizar las funciones de forma reversible* análogamente a los predicados *Prolog*. La programación funcional no puede utilizar funciones de forma reversible porque no puede reducir expresiones que contengan variables.

Si tomamos la función `append` definida en 3.2, se supone que `append` está definida para recibir dos listas y devolver la lista resultante de concatenar las dos listas de entrada, sin embargo, como $\mathcal{TOY}$ puede lanzar objetivos para que se resuelvan de forma reversible estos objetivos pueden tener como variables los parámetros de entrada de la función.

```
Toy> F X [3,4] == [1,2,3,4]
      yes
      F == append
      X ==  [ 1, 2 ]
      Elapsed time: 0 ms.
```

La respuesta nos dice que la restricción que es parte del objetivo es cierta si la restricción que se da como resultado en forma resuelta es cierta.

Es posible que una restricción tenga más de una solución, el sistema irá mostrando todas las soluciones de una en una según lo vaya indicando el usuario. Este proceso se realiza a través del backtracking.

```
Toy> append X Y == [1,2]
      yes
      X == []
      Y ==  [ 1, 2 ]
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      yes
      X ==  [ 1 ]
      Y ==  [ 2 ]
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      yes
```

```
        X ==  [ 1, 2 ]
        Y == []
        Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
        no.
```

El orden en el cual las restricciones que forman parte de la respuesta son mostradas se ve en 3.10

3.6. Primitivas o Funciones Predefinidas

$\mathcal{TOY}$ tiene una serie de funciones primitivas o predefinidas que pueden ser utilizadas por cualquier programa sin tener la necesidad de definirlas de nuevo. Las definiciones de dichas funciones están incluidas en el fichero *basic.toy* que se muestra en el apéndice A. En este fichero se definen los tipos de datos, como por ejemplo

```
data bool = true | false
```

y el tipo de las funciones, como por ejemplo

```
primitive sqrt :: real -> real
```

3.7. Evaluación Perezosa y Estructuras de Datos Infinitas.

$\mathcal{TOY}$ es un leguaje de evaluación perezosa, es decir, no se evalúan los parámetros de las funciones hasta que esto no sea necesario para proseguir con el cómputo. Esta caracteristica hace que se puedan tratar estructuras de datos infinitas como parámetros de funciones sin crear en un cómputo no terminante, debido a que estas estructuras se generan según se vayan demandando. Un ejemplo clásico es la función `repeat` que recibe un argumento de entrada y genera una lista infinita de enteros iguales al argumento de entrada y la función `take` que recibe un entero N y una lista y devuelve una lista con los N primeros elementos de la lista que se le pasa por argumento, o la lista entera si el tamaño de la lista es menor o igual que N.

```
repeat :: A -> [A]
repeat X = [X|repeat X]

take :: int -> [A] -> [A]
take N [] = []
take N [X|Xs] = if N==0 then [] else [X|take (N-1) Xs]
```

Si se evalúa la expresión `repeat 8` obtenemos un cómputo infinito `[8,8,8,8,8,8,8....]`, pero al evaluar `take 5 (repeat 8)`, la pereza hace que el resultado sea `[8,8,8,8,8]`.

Aunque $\mathcal{TOY}$ se traduce a *Prolog*, debido a la pereza es más eficiente que *Prolog* en problemas que tratan estructuras de datos infinitas, porque si en *Prolog* se invoca un predicado que genera una estructura de datos infinita, entonces intentará crear la estructura completa que esto llevaría a un cómputo no terminante.

3.8. Funciones Indeterministas o Multivaluadas

Una de las caracteristica de la programación lógica es el indeterminismo. Una función es indeterminista si devuelve resultados diferentes para argumentos iguales. Por ejemplo, se define la función `coin` que tiene dos posibles valores como resultado, el 0 y el 1,

```
coin :: int
coin = 0
coin = 1
```

La evaluación de `coin` da como primer resultado el 0 y por *backtracking* dará como segundo resultado el 1

Si se define otra función `double` la cual duplica el valor que se le pasa por parámetro.

```
double :: int -> int
double X = X + X
```

La evaluación de `double coin == L` muestra como resultado `L == 0` y `L == 2`, esto es debido a que `double` recibe el argumento `coin` sin evaluar por ser un lenguaje perezoso, al reducir la expresión `coin + coin` primero reduce una de las dos expresiones `coin` y debido al *sharing*, la otra expresión `coin` toma el mismo valor, ya que las variables que aparencen más de una vez en la parte derecha de la regla de reescritura son compartidas. Esta semántica se la conoce como *call-time choice*.

El método *generate and test* resuelve problemas indeterministas generando candidatos a solución y comprobando si son solución. Este método es a menudo poco eficiente porque los candidatos son calculados totalmente antes de ser comprobados. $\mathcal{TOY}$ permite expresar un método *generate and test perezoso*, donde el generador es una función en vez de un predicado, los candidatos generados son pasados como argumentos al comprobador y como se tiene evaluación perezosa se generan los candidatos según sean requeridos por el comprobador y si el candidato no es válido y el sistema lo detecta antes de generarlo por completo, finaliza la generación.

Un **ejemplo** de esta caracteristica es la ordenación por permutación. Supongamos que se quiere dar un conjunto de elementos ordenados, una forma es generar permutaciones de esos elementos y comprobar que están ordenados. Veamos dos métodos:

- **Método de generación y comprobación impaciente**

  El predicado `permSort` sigue el esquema de *Prolog* pues genera una permutación *completa* `Ys` de `Xs` y después comprueba si esta ordenada

```
permSort :: [A] -> [A] -> bool
permSort Xs Ys = true <== permutation Xs Ys == true,
                                isSorted Ys == true

permutation :: [A] -> [A] -> bool
permutation [] []     :- true
permutation [X|Xs] Zs :- permutation Xs Ys,
                         insertion X Ys Zs

insertion :: A -> [A] -> [A] -> bool
```

```
insertion X [] [X]          :- true
insertion X [Y|Ys] [X,Y|Ys] :- true
insertion X [Y|Ys] [Y|Zs]   :- insertion X Ys Zs

isSorted :: [A] -> bool
isSorted []       = true
isSorted [X]      = true
isSorted [X,Y|Zs] = X <= Y /\ isSorted [Y|Zs]
```

Este esquema genera completamente la permutación y después comprueba si es ordenado siendo, por tanto, muy ineficiente.

- **Método de generación y comprobación perezoso**

```
permSort :: [A] -> [A]
permSort Xs = checkSorted (permutation Xs)

checkSorted :: [A] -> [A]
checkSorted Xs = Ys <== isSorted Ys == true

permutation :: [A] -> [A]
permutation []     = []
permutation [X|Xs] = insert X (permutation Xs)

insert :: A -> [A] -> [A]
insert X []     = [X]
insert X [Y|Ys] = [X,Y|Ys]
insert X [Y|Ys] = [Y|insert X Ys]
```

En este esquema la función `checkSorted` comprueba la ordenación de la permutación que genera la función indeterminista `permutation`. Con la evaluación perezosa la función `checkSorted` puede fallar sin completar la evaluación de su argumento provocando la vuelta atrás de la función `permutation` que intentará generar otro argumento para la función `checkSorted`, por lo tanto, según se va generando el candidato se comprueba que este está ordenado, si no está ordenado entonces falla y empieza a generar un nuevo candidato.

3.9. Funciones de Orden Superior

Una función es de orden superior si alguno de sus argumentos o el resultado es una función. Esta es una caracteristica propia de la programación funcional.

Por ejemplo la función `map` definida en 3.6 es de orden superior y si se aplica tomando como argumento la función `suma_diez`.

```
suma_diez:: int -> int
suma_diez N = N + 10
```

Un objetivo de orden superior es:

```
TOY> map suma_diez [1,30,1000, -40] == L
     yes
     L ==  [ 11, 40, 1010, -30 ]
     Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
     no.
     Elapsed time: 0 ms.
```

$\mathcal{TOY}$ también soporta predicados de orden superior con notación clausal, porque trata a los predicados como funciones que devuelven un valor booleano.

Ejemplo: si deseamos un caso particular de la función `map` definida antes, pero aplicada sólo a funciones con resultado booleano tendríamos la definición:

```
mapRel :: ( A -> B -> bool) -> [A] -> [B] -> bool
mapRel R [] [] :- true
mapRel R (X:Xs) (Y:Ys) :- R X Y, mapRel R Xs Ys
```

Si se va a procesar una lista infinita entonces es mejor `map` que `mapRel` porque como se ve en 3.7 la pereza hace que no se genere cómputos no terminantes.

Patrones de orden superior.

En $\mathcal{TOY}$ un *patrón* es una expresión irreducible o una forma normal. Existen *patrones de primer orden* que están formados por variables, números, tuplas o constructoras de aridad n y también existen *patrones de orden superior*, estos admiten como patrones expresiones en las cuales pueden aparecer constructoras y funciones aplicadas parcialmente.

Supongamos la función `map` definida en 3.6 y la función

```
and :: bool -> bool -> bool
and true X = X
and false X = false
```

En el objetivo `map (and W) [X,Y] == [true,false]` la función `(and W)` es un patrón de orden superior ya que es una función aplicada parcialmente y de tipo

```
(and W) :: bool -> bool
```

La resolución intuitiva de este objetivo sería:

1. aplicar `map` resultando: `[(and W X),(and W Y)] == [true,false]`
2. por tener ambas listas dos elementos: `and W X == true` y `and W Y == false`
3. aplicando `and` se obtiene como única solución: `W = true, X = true, Y = false`.

Variables lógicas de orden superior.

$\mathcal{TOY}$ permite hacer cómputos que contengan variables lógicas de orden superior, es decir, se puede lanzar como objetivo una restricción donde el identificador de la función a ejecutar sea una variable. Por ejemplo, dada la definición de la función *append* vista en 3.2. Es posible resolver el objetivo

```
Toy> F X [3,4] == [1,2,3,4]
     yes
     F == append
     X ==  [ 1, 2 ]
     Elapsed time: 0 ms.
```

Obteniendo como resultado que la variable lógica F toma el valor funcional append.

3.10. Restricciones

$\mathcal{TOY}$ puede ser descrito como un lenguaje lógico funcional basado en el estrechamiento, es decir, el mecanismo operacional es una combinación de unificación y reescritura.

Como ya se vio en 3.4, las funciones en $\mathcal{TOY}$ son de la forma

$$\begin{array}{rcl} f & :: & \tau_1 -> \ldots \tau_n -> \tau \\ & \ldots & \\ ft_1 \ldots t_n & = & r <== C_1, \ldots, C_m \; where \; D_1, \ldots, D_p \\ & \ldots & \end{array}$$

donde $ft_1 \ldots t_n$ puede ser reducido a r si las restricciones $C_1, \ldots, C_m$ son satisfechas utilizando las definiciones locales $D_1, \ldots, D_p$. C_i son *expresiones condicionales o restricciones* de la forma $e_1 \diamond e_2$ con $\diamond \in \{==, /=, <, >, <=, >=\}$. Las restricciones deben estar bien tipadas, es decir, e_1 y e_2 han de ser del mismo tipo para $==$ y $/=$ y e_1 y e_2 han de ser del mismo tipo numerico (int o float) para $<, >, <=, >=$. Recordemos que si una expresión es de la forma $e_1 == true$ se puede simplificar a e_1.

Para saber si las restricciones son satisfactibles se requiere de un resolutor de restricciones el cual es independiente del estrechamiento, aunque coopera con el estrechamiento cuando las restricciones involucran subexpresiones $f(e_1, ..., e_n)$.

Un resolutor de restricciones debe producir uno de los siguientes resultados:

- Un *fallo* si la restricción es insatisfactible.
- una restricción en *forma resuelta*. Es posible que una restricción sea transformada en una familia de expresiones en forma resuelta. Es decir, que la restricción original sea equivalente a la disyunción de las restricciones en forma resuelta. Las disyunciones corresponden a las distintas ramas del árbol de computación.

```
Toy> [X,0] /= [1,Y]
     yes
     { X /= 1 }
  Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
```

```
        yes
        { Y /= 0 }
  Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
        no.
```

- una *restricción suspendida*, cuando el resolutor de restricciones no puede asegurar su satisfactibilidad las restricciones se quedan suspendidas y pueden ser reactivadas en un paso posterior si alguna variable es instanciada. Esto ocurre en $\mathcal{TOY}$ con las restricciones aritméticas no lineales.

Durante la computación, las restricciones resueltas y suspendidas están en el *almacén de restricciones*. Cuando el cómputo termina, el almacén es parte de la respuesta. De hecho las respuestas pueden tener tres clases de restricciones que se mostraran en el siguiente orden:

- primero, las restricciones de igualdad, donde se muestra cómo se han ligado las variables.

```
Toy> X /= 3, append [1,Y] [X] == [1,3,4]
      yes
      X == 4
      Y == 3
  Elapsed time: 0 ms.
```

- después, las restricciones de desigualdad

```
Toy> [X,0] /= [1,Y]
      yes
      { X /= 1 }
  Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      yes
      { Y /= 0 }
  Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      no.
  Elapsed time: 0 ms.
```

- y por último, si el resolutor sobre los números reales esta activo, se mostrarán las restricciones correspondientes.

```
Toy(R)> 2 * X  - 3 * Y > 4, X > 0, Y >0
  yes
    { X-1.5*Y>2.0 }
    { Y>-0.0 }
  Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
  no.
  Elapsed time: 0 ms.
```

En este capítulo no se tiene en cuenta las restricciones sobre domnios finitos, dejándose para el siguiente capítulo dichas restricciones.

Las restricciones de igualdad y desigualdad sintáctica están embebidas en el sistema, por lo tanto siempre están disponibles y su definición de tipos es:

```
(==),(/=) :: A -> A -> bool
```

3.10.1. Restricciones de Igualdad (==)

La expresión $e_1 == e_2$ representa la restricción de igualdad, que se evalúa a *true* si las expresiones e_1 y e_2 pueden reducirse a una forma normal común. Se reduce a *false* si no pueden reducirse a una forma normal común.

En programación lógico funcional y en concreto en $\mathcal{TOY}$ se evalúan los términos y se unifican.

La evaluación es una caracteristica de la programación funcional, se evalúa $e_1 == e_2$ a *true* si se pueden reducir las expresiones e_1 y e_2 a las mismas formas normales. Por ejemplo

- En $\mathcal{TOY}$ $1 + 1 == 2$ se reduce a *true* y $1 + 2 == 2$ se reduce a $false$.
- En *Prolog* la igualdad es vista desde el punto de vista sintáctico y por lo tanto *1 + 1 == 2* se reduce a *false* porque *1 + 1* y *2* son términos distintos.

En programación funcional no se pueden reducir expresiones con variables, *Haskell*no se puede reducir algo de la forma $X == 0$. Sin embargo, *Prolog* si permite reducir expresiones con variables porque permite la unificación que liga un término a otro, si se tiene $X ==$ *0* entonces liga 0 a X.

En $\mathcal{TOY}$ se pueden evaluar términos como en la programación funcional y por lo tanto una restricción de la forma $1 + 1 == 2$ se reduce a *true* y también se pueden unificar términos ligando valores a las variables, así $X ==$ *0* se reduce a *true* y muestra como resultado la restricción $X ==$ *0*.

3.10.2. Restricciones de Desigualdad (/=)

Una desigualdad entre dos expresiones es cierta si ambas expresiones pueden reducirse a expresiones que contienen una constructora distinta en la misma posición.

En $\mathcal{TOY}$ la desigualdad también se caracteriza por la evaluación de términos como en funcional y por la unificación de la programación lógica.

Si la evaluación de las expresiones lleva a formas normales con conflicto de constructoras entonces devolverá *true* `[1+1,0] /= [1,0]` y si se reduce a la misma forma normal entonces será *false* `[1+0,0] /= [1,0]`. Además las desigualdades pueden contener variables

```
Toy> [X,0] /= [1,Y]
      yes
      { X /= 1 }
      Elapsed time: 0 ms.

    more solutions (y/n/d) [y]?
      yes
      { Y /= 0 }
      Elapsed time: 0 ms.
```

Que el sistema muestre como respuesta desigualdades es muy útil porque de no ser así en la anterior restricción el sistema tendría que dar infinitos valores como respuesta a la variable X (y otros tantos para Y), todos los valores que son distintos de 1, estos sería un número infinito de respuestas.

Al ser $\mathcal{TOY}$ un lenguaje indeterminista se puede dar que se cumpla la igualdad y desigualdad entre dos términos a la vez, este es el caso de `coin == 0, coin /= 0` que se reduce a *true*.

3.10.3. Restricciones Sobre los Números Reales

Si se intenta resolver en $\mathcal{TOY}$ la restricción `2 + X == 5` el sistema devuelve un error que nos dice que las varibles no están permitidas en operaciones aritméticas, igualmente ocurre con los operadores `>, <, >= y <=` . Esto se debe a que el objetivo no se puede reducir y ha de ser tratado por el resolutor de restricciones sobre reales de Sicstus. El usuario debe activar explícitamente el resolutor sobre reales con la sentencia /cflpr y el símbolo del sistema cambia de `Toy` a `Toy(R)` .

```
Toy(R)> 2 + X == 5
      yes
      X == 3
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
```

Si las restricciones no llevan variable entonces no es necesario activar el resolutor. La restricción `2 + 3 == 5` no necesita que el resolutor esté activo.

Las funciones aritméticas son primitivas del sistema y sus declaraciones están contenidas en el fichero *basic.toy* (A).

El resolutor de restricciones sobre reales necesita que las restricciones sean lineales, si no son lineales las deja suspendidas hasta que en un paso posterior sean instanciadas o hasta que el proceso finalice y se muestre la restricción como respuesta.

```
Toy(R)> X * X == 9
      yes
            { 9.0-X^2.0==0.0 }
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      no.
```

Capítulo 4

$\mathcal{TOY}(\mathcal{FD})$

Antes de mostrar el capítulo referente al lenguaje $\mathcal{TOY}(\mathcal{FD})$ se han introducido los dos capítulos anteriores debido a que el lenguaje que implementamos $\mathcal{TOY}(\mathcal{FD})$ es la extensión del lenguaje lógico funcional $\mathcal{TOY}$ (capítulo 3) con las restricciones sobre dominios finitos $\mathcal{FD}$ (capítulo 2).

En este capítulo se muestran los conceptos fundamentales de la sintáxis y de la disciplina de tipos, después de los conceptos teóricos se pasa a la implementación mostrando la definición, tipo y ejemplo de cada restricción de dominio finito que se ha implementado en $\mathcal{TOY}(\mathcal{FD})$. Porteriormente se resuelven problemas en $\mathcal{TOY}(\mathcal{FD})$, unos son análogos a los que se mostraron en el capítulo 2, pero es interesante ver que se pueden resolver problemas de $\mathcal{CFLP}(\mathcal{FD})$ que no se pueden resolver en $\mathcal{CLP}(\mathcal{FD})$, como por ejemplo generar una lista infinita que se trata con evaluación perezosa para no crear un cómputo no terminante, mostrando así una buena propiedad de $\mathcal{TOY}(\mathcal{FD})$. Por último, para poder implementar en $\mathcal{TOY}$ las restricciones sobre dominios finitos se han hecho modificaciones en el código, se detalla tanto las modificaciones que se han hecho en los ficheros existentes como los ficheros que se han creado nuevos.

4.1. Introducción

$\mathcal{TOY}(\mathcal{FD})$ es la extensión de las restricciones sobre dominios finitos en el lenguaje lógico funcional $\mathcal{TOY}$, por lo tanto está enmarcado en el ámbito de la programación CFLP(FD) de la cual se expondrán los conceptos fundamentales en la siguiente sección.

$\mathcal{TOY}(\mathcal{FD})$ incrementa la expresividad de la programación lógica con restricciones sobre dominios finitos ($\mathcal{CLP}(\mathcal{FD})$) al combinar la notación funcional, relacional y currificada de $\mathcal{TOY}$. Además $\mathcal{TOY}(\mathcal{FD})$ tiene las caracteristicas propias de $\mathcal{TOY}$ que se mostraron en el capítulo 3, esto hace que $\mathcal{TOY}(\mathcal{FD})$ tenga caracteristicas que no existen en el ámbito $\mathcal{CLP}(\mathcal{FD})$ como la notación currificada, tipos, funciones de orden superior (restricciones de orden superior), composición de restricciones, patrones de orden superior, aplicaciones parciales, evaluación perezosa y polimorfismo. Estas caracteristicas permiten al lenguaje tener buenas cualidades, como por ejemplo la evaluación perezosa permite al lenguaje manejar estructuras de datos infinitas; que el lenguaje sea tipado hace que se pueda aplicar un inferidor de tipos.

$\mathcal{TOY}(\mathcal{FD})$ integra las restricciones sobre dominios finitos (FD) como funciones, es decir, como ciudadanos de primera clase, por lo tanto estas restricciones se pueden utilizar en los mismos lugares en dónde se puede poner una variable, esto permite que una restricción

FD pueda aparecer como un argumento o como resultado de otra función o restricción. Por lo tanto se tienen patrones de orden superior (HO). Además $\mathcal{TOY}(\mathcal{FD})$ toma de la programación FD las variables de dominios, las restricciones y los propagadores.

4.2. Sintáxis de CFLP(FD)

En esta sección se van a exponer los conceptos fundamentales básicos sobre la sintáxis y la disciplina de tipos. Para esto se ha seguido la formalización dada por [12], [11] y [10].

4.2.1. Tipos y Signaturas

Para poder definir los tipos es necesario definir anteriormente $\mathcal{TVar}$ como un conjunto numerable de variables tipadas $\alpha,\ \beta,\ \ldots$ Las variables de los dominios finitos son variables con valores enteros. $TC = \bigcup_{n\in\mathbb{N}} TC^n$ es un conjunto numerable de todas las constructoras de tipos y $C \in TC^n$ es la constructora de tipo de aridad n.

Los tipos $\tau \in \mathit{Type}$ tienen la sintaxis

$$\tau ::= \alpha \quad | \, C\ \tau_1 \ldots \tau_n \quad | \, \tau \to \tau' \mid (\tau_1, \ldots, \tau_n)$$

Por notación $C\ \overline{\tau}_n$ abrevia $C\ \tau_1 \ldots \tau_n$, además "$\to$" asocia a la derecha, por lo tanto $\overline{\tau}_n \to \tau$ abrevia $\tau_1 \to \cdots \to \tau_n \to \tau$. Un tipo sin ninguna ocurrencia de "$\to$" es un *tipo de dato*. Los tipos $(\tau_1, \ldots, \tau_n)$ son n-tuplas. Se denota con $tvar(\tau)$ al conjunto de variables que aparecen en τ y diremos que τ es *monomorfico* si y sólamente si $tvar(\tau) = \emptyset$ y *polimorfico* en el caso contrario. En el lenguaje $\mathcal{TOY}$ existen tipos predefinidos que se muestran a continuación.

- `[A]` es el tipo predefinido para las listas polimórficas,
- `bool` predefine los booleanos con constantes `true` y `false`,
- `int` predefine los valores enteros,
- `char` predefine los caracteres con constantes `'a', 'b', ...` y
- `success` con la constante `⊤`.

Una *signatura polimorfica* sobre TC es $\Sigma = \langle TC,\ DC,\ FS\rangle$, donde $DC = \bigcup_{n\in\mathbb{N}} DC^n$ es un conjunto de constructoras de datos y $FS = \bigcup_{n\in\mathbb{N}} FS^n$ es el conjunto de los símbolos de funciones evaluables. Las funciones evaluable pueden ser funciones primitivas o funciones definidas por el usuario.

Cada $c \in DC^n$ tiene una declaración principal de tipos $c\ ::\ \overline{\tau}_n \to C\ \overline{\alpha}_k$, donde $n, k \geq 0, \alpha_1, \ldots, \alpha_k$ son diferentes, τ_i son tipos de datos, y $tvar(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\}$ para todo $1 \leq i \leq n$. También, cada $f \in FS^n$ tiene una declaración principal de tipos $f\ ::\ \overline{\tau}_n \to \tau$, donde τ_i, τ son tipos arbitrarios. Para cualquier signatura polimórfica Σ, se denota con $\Sigma_\perp$ a la signatura Σ extendida con DC^0. En $\Sigma_\perp$ la constructora de dato $\perp\ ::\ \alpha$ representa un valor indefinido que pertenece a cualquier tipo. DC^0 incluye las constantes `true, false` y `⊤` anteriormente introducidas, estos valores son útiles para representar los resultados que devuelven algunas funciones primitivas.

4.2.2. Patrones y Expresiones

Los *patrones y expresiones* son los dos dominios sintácticos más importantes. Los patrones son expresiones irreducibles, es decir, no se puede dar un paso más de evaluación sobre ellos. Las expresiones si se pueden reducir por medio de reglas.

Sea Σ una signatura dada (si no se da una signatura explícitamente siempre se tomará Σ), sea $\mathcal{V}ar$ un conjunto infinito numerable de variables $X, Y, \ldots$ y sea el conjunto de los números enteros $\mathbb{Z}$ de elementos básicos $u, v, \ldots$ que son disjuntos entre si y además son disjuntos con $\mathcal{TV}ar$ y Σ. Los elementos básicos van a representar valores enteros en conjuntos específicos de dominios finitos.

Las *expresiones parciales* $e \in Exp_\perp$ tienen la sintáxis

$$e ::= \perp \mid u \mid X \mid h \mid e\ e' \mid (e_1,\ \ldots,\ e_n)$$

donde $u \in \mathbb{Z},\ X \in \mathcal{V}ar,\ h \in DC \cup FS$. La expresión de la forma $e\,e'$ es la aplicación de la expresión e sobre el argumento e'. Las aplicaciones asocian a la izquierda, por lo tanto $e_0\ e_1 \ldots\ e_n$ abrevia $(\ldots(e_0\ e_1)\ldots)\ e_n$. La expresión $(e_1, \ldots, e_n)$ representa una tupla de n componentes. El conjunto de variables que aparencen en la expresión e se denota con $var(e)$. Además una expresión `e` es `lineal` si todas las variables contenidas en `e` son distintas, es decir, una expresión e es `lineal` si y sólo si cada $X \in var(e)$ tiene una única aparición en e

Una expresión e está en *forma normal de cabeza* si y sólo si e es una variable X o tiene la forma $c(\overline{e}_n)$ para alguna constructora $c \in DC^n$ $(n \geq 0)$ y alguna n-tupla $\overline{e}_n = (e_1,\ \ldots,\ e_n)$ donde e_i está en forma normal de cabeza.

Los *patrones parciales* $t \in Pat_\perp \subset Exp_\perp$ tienen la sintáxis

$$t ::= \perp \mid u \mid X \mid c\ t_1\ \ldots\ t_m \mid f\ t_1\ \ldots\ t_m$$

donde $u \in \mathbb{Z}$, $X \in \mathcal{V}ar$, $c \in DC^k$, $0 \leq m \leq k$, $f \in FS^n$, $0 \leq m < n$ y $t_i \in Pat_\perp$ para todo $1 \leq i \leq m$. Los patrones parciales representan *aproximaciones* a los valores de las expresiones. Los patrones parciales de la forma $f\ t_1 \ldots t_m$ con $f \in FS^n$ y $m < n$ sirven como una representación conveniente de las funciones como variables [12], de hecho, las aplicaciones parciales (aplicaciones con menos argumentos que su aridad correspondiente) de $c \in DC^k$ y $f \in FS^n$ son permitidas como patrones. A estas aplicaciones parciales se las llama *patrones de orden superior (HO)* debido a su comportamiento funcional. Por lo tanto, los símbolos de función, cuando se aplican parcialmente se comportan como constructoras de datos. Los patrones de orden superior pueden ser manipulados como cualquier otro patrón.

La expresiones y patrones en las cuales no aparece $\perp$ son *totales*. El conjunto de expresiones totales se denota por Exp y respectivamente el conjunto de patrones totales es Pat. Actualmente, el símbolo $\perp$ nunca aparece en un programa.

4.2.3. Observaciones Sobre Sustituciones

Existen dos tipos de sustituciones en el marco en el cual nos encontramos, la sustitución de datos y la sustitución de tipos.

Sustitución de datos. Se define una *sustitución* $\theta \in Sub_\perp$ como $\theta : \mathcal{V}ar \to Pat_\perp$ con extensión a $\hat{\theta} : Exp_\perp \to Exp_\perp$, la cual es también denotada como θ.

Sea Sub el conjunto de todas las *sustituciones totales* $\theta : \mathcal{V}ar \to Pat$ y $Sub_\perp$ el conjunto de todas las *sustituciones parciales* $\theta : \mathcal{V}ar \to Pat_\perp$. El *dominio* de una sustitición *dom(θ)*

es el conjunto de todas las variables X tal que $\theta(X) \neq X$. $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ son las sustituciones con dominio $\{X_1, \ldots, X_n\}$ que satisfacen $\theta(X_i) = t_i$ para todo $1 \leq i \leq n$.

Sustitución de tipos se puede definir como $\theta_t : \mathcal{TV}ar \rightarrow Type$ con una única extensión $\hat{\theta}_t : Type \rightarrow Type$, también denotada por θ_t. $TSub$ denota el conjunto de todas las sustituciones de tipos.

Se toma como notación $e\theta$ en vez de $\theta(e)$, y también se toma como notación $\theta\sigma$ para la composición de θ y σ de tal forma que $e(\theta\sigma) = (e\theta)\sigma$ para cualquier e.

Un ejemplo de sustitución de tipos puede ser el siguiente. Sea la función `map` definida de la siguiente forma:

```
map:: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [F X|map F Xs]
```

Si se resuelve el siguiente objetivo:

```
Toy(FD)> map (#> 20) [3,30] == L

      yes
      L ==  [ false, true ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

el tipo `A` se ha sustituido por `int` y `B` por `bool`.

4.2.4. Expresiones Bien Tipadas

Inspirado por el sistema de tipos de Milner [4] se va a introducir la noción de expresión bien tipada. Primero se define el *entorno de tipos* como el conjunto T de asunciones de tipo $X :: \tau$ para variables tal que T no incluye dos asunciones diferentes para la misma variable. El *dominio* de T que se escribe $dom(T)$ es el conjunto de todas las variables que aparecen T. Para cualquier variable $X \in dom(T)$, el único tipo τ tal que $(X :: \tau) \in T$ es denotado como $T(X)$. Sea $h \in DC \bigcup FC$, la notación $(h :: \tau) \in_{var} \Sigma$ es usada para indicar que Σ incluye la declaración de tipo $h :: \tau$ salvo renombramiento de variables de tipo. El *tipo inferido* $(\Sigma, T) \vdash_{WT} e :: \tau$ es derivado por medio de las siguientes reglas de *inferencia de tipos*:

UR $(\Sigma, T) \vdash_{WT} u :: \mathsf{int}$, para cada $u \in \mathbb{Z}$.
VR $(\Sigma, T) \vdash_{WT} X :: \tau$, si $T(X) = \tau$.
ID $(\Sigma, T) \vdash_{WT} h :: \tau\theta_t$, si $(h :: \tau) \in_{var} \Sigma_\perp,\ \theta_t \in TSub$.
AP $(\Sigma, T) \vdash_{WT} (e\ e_1) :: \tau$, si $(\Sigma, T) \vdash_{WT} e :: (\tau_1 \rightarrow \tau)$, $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1$, para algún $\tau_1 \in Type$.
TP $(\Sigma, T) \vdash_{WT} (e_1, \ldots, e_n) :: (\tau_1, \ldots, \tau_n)$, si $\forall i \in \{1, \ldots, n\} : (\Sigma, T) \vdash_{WT} e_i :: \tau_i$.

Una *expresión* $e \in Exp_\perp$ está *bien tipada* si y sólo si existe algún *entorno de tipos* T y algún tipo τ, tal que el *tipo inferido* $T \vdash_{WT} e :: \tau$ puede ser derivado. Las expresiones que admiten más de un tipo son llamadas *polimorficas*. Un expresión bien tipada siempre

admite un *tipo principal* (PT) que es el tipo más general. Un patrón cuyo PT determina los PTs de los subpatrones es llamado *transparente*.

Un *programa* $\mathcal{CFLP}(\mathcal{FD})$ P está *bien tipado* si es un conjunto de reglas definidas bien tipadas para los símbolos de función en esta signatura. Las reglas definidas para $f \in FS^n$ cuya declaración es $f :: \overline{\tau}_n \to \tau$ tienen la forma

$$\text{(R)} \quad \underbrace{f\ t_1\ \ldots\ t_n}_{\text{left hand side}} \quad = \quad \underbrace{r}_{\text{right hand side}} \quad \Leftarrow \quad \underbrace{C}_{\text{Condition}}$$

y debe satisfacer los siguientes requerimientos:

1. $t_1 \ldots t_n$ es una secuencia lineal de patrones transparentes y r es una expresión.

2. La *condición* C es una conjunción de restricciones $C_1, \ldots, C_k$, donde cada C_i puede ser cualquier sentencia de la forma:

   a) $e == e'$ donde $e, e' \in \mathit{Exp}$.

   b) $e /= e'$ donde $e, e' \in \mathit{Exp}$.

   c) $u \in D$ donde $u \in \mathbb{Z} \bigcup Var$, D es un patrón que representa una lista de variables o enteros.

   d) $a \otimes b \asymp c \in C$ donde $a, b, c \in \mathbb{Z} \bigcup Var$, $\otimes \in \{+, -, *, /\}$ y $\asymp \in \{=, \neq, <, \leq, >, \geq\}$

3. Existe algún entorno de tipo T con dominio $var(R)$ en el cual están bien tipadas las reglas definidas en el siguiente sentido:

   a) Para todo $1 \leq i \leq n$: $(\Sigma, T) \vdash_{WT} t_i :: \tau_i$.

   b) $(\Sigma, T) \vdash_{WT} r :: \tau$.

   c) Para cada $(e == e') \in C$, $\exists \mu \in \mathit{Type}$ s.t. $(\Sigma, T) \vdash_{WT} e :: \mu :: e'$.

   d) Para cada $(e/= e') \in C$, $\exists \mu \in \mathit{Type}$ s.t. $(\Sigma, T) \vdash_{WT} e :: \mu :: e'$.

   e) Para cada $(u \in D) \in C$: $(\Sigma, T) \vdash_{WT} u :: \texttt{int}, D :: [\texttt{int}]$.

   f) Para cada $(a \otimes b \asymp c) \in C$: $(\Sigma, T) \vdash_{WT} a, b, c :: \texttt{int}$ con $\otimes \in \{+, -, *, /\}$ y $\asymp \in \{=, \neq, <, \leq, >, \geq\}$.

donde $(\Sigma, T) \vdash_{WT} a :: \tau :: b$ denota $(\Sigma, T) \vdash_{WT} a :: \tau$, $(\Sigma, T) \vdash_{WT} b :: \tau$.

Intuitívamente, el significado de la regla de programa anterior (R) es que una llamada a la función f puede ser reducida a r si los parámetros concuerdan con los patrones t_i y si además las condiciones de igualdad, desigualdad o restricciones FD son satisfechas. Una condición $e == e'$ es satisfactible si se pueden evaluar e y e' a un mismo patrón común. Los predicados son funciones cuyo resultado es booleano, $p :: \overline{\tau}_n \to \texttt{bool}$. Como facilidad sintáctica se definen las *cláusulas* como reglas cuyo lado derecho es *true*. Esto nos permite escribir de la misma forma en la cual se escriben los predicados *Prolog*; cada cláusula $p\ t_1\ \ldots\ t_n\ :-\ C_1, \ldots, C_k$ abrevia una regla definida de la forma $p\ t_1\ \ldots\ t_n = \mathit{true} \Leftarrow C_1, \ldots, C_k$.

4.2.5. Objetivos

Un *objetivo bien tipado* tiene la misma forma que una expresión bien tipada. El sistema $\mathcal{CFLP}(\mathcal{FD})$ resuelve objetivos. La respuesta computada a estos objetivos es una tupla $\langle E, \sigma, C, \delta \rangle$ donde $E \in Exp$ es una expresión, $\sigma \subseteq Sub$ es el conjunto de sustituciones de variables, C es el conjunto de restricciones de desigualdad y δ es el conjunto de los dominios finitos podados.

Cuando el cómputo finaliza se muestra como respuesta a un objetivo las restricciones resueltas y suspendidas que permanecen en el almacén de restricciones. De hecho el almacén puede contener cuatro tipos de restricciones. Las tres primeras ya se vieron en 3.10 por lo tanto sólo se da un ejemplo de la respuesta correspondiente a los dominios finitos.

- las restricciones de igualdad.
- las restricciones de desigualdad
- si el resolutor sobre los números reales esta activo, se mostrarán las restricciones correspondientes.
- si el resolutor sobre los dominios finitos esta activo, se mostrarán las restricciones correspondientes.

```
Toy(FD)> domain [X,Y] 1 10, X #< F, F == Y

      yes
      Y == F
      Y in 2..10
      X in 1..9

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? y

      no.

      Elapsed time: 0 ms.
```

Como se ha visto, el sistema tiene tres modos de uso: sin restricciones, con restricciones sobre reales y con restricciones sobre dominios finitos. Cualquier objetivo que se pueda resolver sin utilizar restricciones se puede resolver con algún resolutor de restricciones activado, tanto el resolutor de reales como el de dominios finitos, pero tener el resolutor activado hace que se pierda eficiencia porque el resolutor necesita mantener información extra. Por lo tanto si el objetivo no necesita de algún resolutor específico, entonces es conveniente no tener ningún resolutor activo.

4.3. Implementación del Lenguaje $\mathcal{TOY}(\mathcal{FD})$

Un domino finito (FD) es la aplicación $\delta : \mathcal{V}ar \to \mathbb{Z}$, el conjunto de todos los FDs es denotado por $\mathcal{FD}$ y el conjunto de todas las aplicaciones parciales $\delta : \mathcal{V}ar \to \mathbb{Z}_\perp$ es denotado por $\mathcal{FD}_\perp$. $X_1 \in i_1, \ldots, X_n \in i_n$, para todo $1 \leq i \leq n$, donde $i_i \subseteq Integer$ y $dom(\delta) = \{X_1, \ldots, X_n\}$.

Al igual que en las sustituciones se escribe $e\delta$ en vez de $\delta(e)$, y $\delta\sigma$ para la composición de δ y σ, de tal forma que $e(\delta\sigma) = (e\delta)\sigma$ para cualquier e.

Se dice que δ es *inconsistente* y se escribe *inconsistente*(δ) si existe un X_j con $1 \leq j \leq n$ tal que i_j es el conjunto vacío, y se dice que es *consistente* en otro caso, (*consistente*(δ)).

4.3.1. Restricciones Sobre Dominios Finitos

Una restricción primitiva sobre dominio finito es una función bien tipada primitiva declarada con un tipo de alguna de las siguientes formas:

- `int` → `int` → `int` transforma pares de variables de FD a variables FD, como por ejemplo los operadores aritméticos `(#*), (#/), (#+), (#-) :: int` → `int` → `int`.
- `int` → `int` → `bool` transforma pares de variables de FD a valores booleanos, como por ejemplo los operadores relacionales `(#>), (#<), (#>=), (#<=), (#=), (#\=) :: int` → `int` → `bool`.
- $\overline{\tau}_n \to \tau$ tal que para todo τ_i de $\overline{\tau}_n$, τ y $\tau_i \in Type$. Los tipos más comunes son { `int`,`[int]`,`[labelType]`,`statistics`,`[options]`,`[serialOptions]` } $\subseteq Type$.

Algunos de estos tipos ya se habían definido en la sección 4.2.1, `int` es el tipo predefinido para los enteros y $[\tau]$ es el tipo 'lista de τ'. Los tipos de datos `labelingType`, `statistics`, `options` y `serialOptions` son tipos predefinidos para los dominios finitos y permiten elegir las diferentes opciones del estrechamiento, reactivar la búsqueda o visualizar estadísticas. Su definición completa se muestra en la tabla 4.1 y en el apéndice B que muestra el código del fichero `cflpfd.toy` en el cual están definidas las primitivas correspondientes a las restricciones sobre dominios finitos y los tipos de datos.

Cuadro 4.1: Tipos de datos predefinidos ($\mathcal{TOY}(\mathcal{FD})$)

```
data labelingType = ff | ffc | leftmost | mini | maxi | step | enum | bisect | up
        | down | each | toMinimize int | toMaximize int | assumptions int
data statistics = resumptions | entailments | prunings | backtracks | constraints
data typeprecedence = d (int,int,liftedInt)
data liftedInt = superior | lift int
data serialOptions = precedences [typeprecedence] | path_consistency bool
        | static_sets bool | edge_finder bool | decomposition bool
data reasoning = value | domains | range
data options = on reasoning | complete bool
data fdset = interval int int
data range = cte int int | uni range range | inter range range | compl range
```

En la siguiente sección se van a detallar las restricciones primitivas sobre dominios finitos que están definidas e implementadas en $\mathcal{TOY}(\mathcal{FD})$.

4.3.2. Primitivas Correspondientes a $\mathcal{TOY}(\mathcal{FD})$

$\mathcal{TOY}(\mathcal{FD})$ es un lenguaje que se traduce a *Prolog* y que utiliza el resolutor de dominios finitos de *Prolog*, por lo tanto en esta sección se va a tratar de relacionar mediante ejemplos, las primitivas de $\mathcal{TOY}(\mathcal{FD})$ con las primitivas de *Prolog* dadas en 2.4. Una de las principales diferencias es que $\mathcal{TOY}(\mathcal{FD})$ utiliza notación currificada mientras que *Prolog* es de primer orden. Los ejemplos que a continuación se muestran no son complejos, la intención de poner estos ejemplos es dar una idea simple del significado y sintaxis de cada función primitiva de $\mathcal{TOY}(\mathcal{FD})$. En la siguiente sección se dan ejemplos más complejos.

- ***Aritméticas*: `(#*)/2, (#/)/2, (#+)/2, (#-)/2.`**
  Relacionales: `(#>)/2, (#<)/2, (#>=)/2, (#<=)/2, (#=)/2, (#\=)/2.`

  Declaración de tipos de las restricciones aritméticas y relacionales.

`(#*), (#/), (#+), (#-) :: int → int → int`
`sum :: [int] → (int → int → bool) → int → bool`
`scalar_product :: [int] → [int] → (int → int → bool) → int → bool`
`(#>), (#<), (#>=), (#<=), (#=), (#\=) :: int → int → bool`

  Estos operadores tienen notación prefija (`#Op e e'`) e infija (`e #Op e'`) donde $Op \in$ `{ =, \=, <, <=, >, >=, +, -, *, /}` . Los operadores relacionales no varían con respecto a *Prolog* (2.4), pero en $\mathcal{TOY}(\mathcal{FD})$ los operadores `+, -, *, /` llevan `#` delante. Las declaraciones de tipos están definidas en el fichero *cflpfd.toy* de la distribución cuyo código se muestra en el apéndice B, en el cual también se muestran las prioridades

```
infix  30 #>,#<,#>=,#<=
infix  20 #=,#\=
infixr 90 #*
infixl 90 #/
infixl 50 #+,#-
```

  Ejemplo en $\mathcal{TOY}(\mathcal{FD})$ a nivel de comando:

```
Toy(FD)> X #> 0, X #< Y, Y #< Z, Z #< 10, X #+ Y #< Z

      yes
      Z in 4..9
      Y in 2..7
      X in 1..6

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? y

      no.

      Elapsed time: 0 ms.
```

sum Xs RelOp Value: `Xs` es una lista de variables, `RelOp` es un operador aritmético o relacional y `Value` es un entero. Esta expresión devuelve `true` si la suma de los valores de `Xs` se corresponde con `Value` dependiendo de `RelOp`.

```
Toy(FD)> sum [2,3,4] (#=) X

      yes
      X == 9

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? y

      no.

      Elapsed time: 0 ms.

Toy(FD)> sum [2,3,4] (#<) X

      yes
      X in 10..sup

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

scalar_product Coeffs Xs RelOp Value: donde `Coeffs` es una lista de enteros de longitud n, `Xs` es una lista de variables de longitud n y `Value` es un número entero. Esta expresión devuelve `true` si la suma del producto de los valores de la misma posición de `Coeffs` y de `Xs` se corresponde con `Value` dependiendo del operador aritmético o relacional `RelOp`.

```
Toy(FD)> scalar_product [1,2,3] [4,5,6] (#=) X

      yes
      X == 32

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> scalar_product [1,2,3] [4,5,6] (#<) X

      yes
      X in 33..sup

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

Observese que las funciones `sum/3`, `scalar_product`/4 y `count`/4 son restricciones de orden superior porque aceptan como argumento una restricción del tipo (`int` → `int` → `bool`) como por ejemplo las restricciones relacionales (#<) o (#>). Además, en estas primitivas el operador debe ir entre paréntesis para evitar error de tipos.

- ***dominios sobre variables:*** `domain/3`

  Declaración de tipos

  ```
  domain :: [int] → int → int → bool
  ```

  domain Variables Min Max: `Variables` es una lista de variables, `Min` es un entero o el átomo `inf` que determina la cota inferior de las variables de la lista `Variables` y `Max` es un entero o el átomo `sup` que denota la cota superior.

  ```
  Toy(FD)> domain [Y,Z] 2 8

        yes
        Z in 2..8
        Y in 2..8

        Elapsed time: 0 ms.

  more solutions (y/n/d) [y]?

        no.

        Elapsed time: 0 ms.
  ```

- ***Restricciones proposicionales:*** `#<=>/2`

  Declaración de tipos

  ```
  #<=>:: bool → bool → bool
  ```

  P `#<=>` Q**:** `es true si las restricciones P y Q son ambas true o son ambas false.` De la restricciones proposicionales que se mostraron en 2.4 sólo se ha traducido `#<=>` a $\mathcal{TOY}(\mathcal{FD})$.

  ```
  Toy(FD)> 4 #> 2 #<=> 4 #> 0

        yes

        Elapsed time: 0 ms.

  more solutions (y/n/d) [y]?

        no.

        Elapsed time: 0 ms.

  Toy(FD)> 4 #> 2 #<=> 4 #> 5

        no.

        Elapsed time: 0 ms.
  ```

- ***Restricciones combinatorias:*** `assignment/2` `all_different/1` `all_different'/2` `all_distinct/1` `all_distinct'/2` `circuit/1` `circuit'/2` `serialized/2` `serialized/3` `cumulative/4` `cumulative'/5` `exactly/3` `element/3` `count/4`.

  Declaraciones de tipos

```
assignment :: [int] → [int] → bool
all_different  :: [int] → bool
all_different' :: [int] → [options] → bool
all_distinct  :: [int] → bool
all_distinct' :: [int]→[options]→bool
circuit  :: [int] → bool
circuit' :: [int] → [int] → bool
serialized  :: [int] → [int] → bool
serialized' :: [int] → [int] →[serialOptions] → bool
cumulative  :: [int] → [int] → [int] → int → bool
cumulative' :: [int] → [int] → [int] → int → [serialOptions] → bool
exactly :: int → [int] → int → bool
element :: int → [int] → int → bool
count :: int → [int] → (int → int → bool) → int → bool
```

Estas restricciones son muy útiles para resolver problemas que están definidos sobre dominios discretos. A menudo, estas restricciones son llamadas *restricciones simbólicas*, [2].

count Val List RelOp Count: `Val` es un entero, `List` es una lista de variables, `Count` es un entero o una variable de dominio y `RelOp` es un operador relacional definido en las restricciones aritméticas. Este predicado devuelve `true` if `Count` es el número de veces que se cumple `val` en la lista `List` dependiendo de `RelOp`. Al igual que `sum` y `scalar_product` este predicado necesita paréntesis en `RelOp`.

```
Toy(FD)> count 10 [2,3,4] (#=) A

      yes
      A == 0

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> count 10 [2,3,4,2,6,1,2] (#=) A

      yes
      A == 0

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

element X CList Y es `true` si el elemento `X`-ésimo de la lista `CList` coincide con `Y`, `false` en el caso contrario y también es `false` si la lista tiene longitud menor que `X`.

```
Toy(FD)> element 2 [1,2,3,4] Y
```

```
      yes
      Y == 2

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> element 5 [1,2,3,4] Y

      no.

      Elapsed time: 0 ms.

Toy(FD)> element X [1,2,3,4] 2

      yes
      X == 2

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

all_different Variables, all_distinct Variables, estas dos funciones son semejantes. El resultado de esta función es `true` si los valores que toman las variables de la lista `Variables` son todos distintos.

```
Toy(FD)> all_different [2,3,4,5,6,7]

      yes

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> all_different [2,3,4,5,6,7,2]

      no.

      Elapsed time: 0 ms.
```

all_different' Variables Options, all_distinct' Variables Options, estas dos funciones también son semejantes. El resultado de esta función es `true` si los valores que toman las variables de la lista `Variables` son todos distintos. `Options` es una lista de las siguientes opciones, cuyas definiciones se mostraron en la tabla 4.1.

1. `on reasoning`, con esta opción se dice cómo se depiertan las restricciones. Si `reasoning` toma el valor `value` (que es el valor por defecto que toma `all_different/[1,2]`), se despierta la restricción cuando una variable llega a ser básica; si `reasoning` toma el valor `range` la restricción se despierta cuando cuando un límite de una variable es cambiado; si es `reasoning` toma

el valor `domains` (que es el valor por defecto de `all_distinct/[1,2]`), la restricción se despierta cuando el dominio de una variable es cambiado.

2. `complete bool`, si `bool` es `true` (valor por defecto de `all_distinct/[1,2]`), entonces se utiliza un algoritmo completo que mantiene la consistencia de los dominios [25]; si `bool` es `false` (valor por defecto de `all_different/[1,2]`) un algoritmo incompleto es usado.

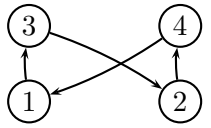
```
Toy(FD)> all_different' [2,3,8] [on(value),complete(true)]

      yes

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

assignment Xs Ys: es aplicada sobre dos listas de longitud N donde cada variable toma un valor entre 1 y N que es único.

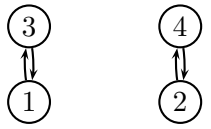
```
Toy(FD)> assignment [A,B] [C,D]

      yes
      D in 1..2
      C in 1..2
      B in 1..2
      A in 1..2

      Elapsed time: 0 ms.
```

circuit Succ: circuito hamiltoniano. `Succ` es una lista de N enteros en la cual el elemento i-ésimo es el sucesor (o predecesor en orden inverso), del vértice i del grafo.

```
Toy(FD)> circuit [3,4,2,1]

      yes

      Elapsed time: 0 ms.

Toy(FD)> circuit [3,4,1,2]

      no.

      Elapsed time: 0 ms.

Toy(FD)> circuit [3,4,A,B]

      yes
      A == 2
      B == 1

      Elapsed time: 0 ms.
```



yes



no

circuit' Succ Pred: circuito hamiltoniano. `Succ` (`Pred`) es una lista de N enteros o variables de dominio en la cual el elemento i-ésimo es el sucesor (o predecesor en orden inverso), del vértice i del grafo.

serialized Starts Durations: fabricar en serie. `Starts` es una lista de enteros que marca el comienzo de la tarea i-ésima y `Durations` es también una lista de enteros que representa el tiempo que se tarda en realizar la tarea i-ésima

```
Toy(FD)> domain [A,B] 20 50, serialized [1,20,53] [12,A,B]

     yes
     B in 20..50
     A in 20..50

     Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

  no.

  Elapsed time: 0 ms.
```

serialized' Starts Durations serialOptions: fabricar en serie. `Starts` es una lista de enteros que marca el comienzo de la tarea i-ésima y `Durations` es también una lista de enteros que representa el tiempo que se tarda en realizar la tarea i-ésima. `serialOptions` es una lista de las siguientes opciones, cuyas definiciones se mostraron en la tabla 4.1.

1. `precedences Ps`, da un conjunto de restricciones de precedencia para aplicar a las tareas, `Ps` debe ser de la forma `d(i,j,k)`, donde `i` y `j` deben ser números de tareas y `k` debe ser un número positivo precedido de la palabra `lift` o `superior` para indicar el valor mayor del dominio. Cada término añade la restricción

$$S_i + k \leq Sj \quad o \quad S_j \leq S_i \quad si\ k\ es\ un\ entero$$

$$S_j \leq S_i \quad si\ k\ es\ \texttt{superior}$$

2. `path_consistency Boolean` si Boolean es `true`, se usa un algoritmo de consistencia por camino para mejorar la poda.
3. `static_sets Boolean`, si Boolean es `true` se usa un algoritmo que razona sobre el conjunto de tareas que debe preceder a una tarea dada,
4. `edge_finder Boolean`, si Boolean es `true` se usa un algoritmo que intenta identificar las tareas que necesariamente preseden o son precedidas por algún conjunto de tareas.
5. `decomposition Boolean`, si Boolean es `true` se hace un intento de descomponer las restricciones cada vez que estas son reactivadas.

```
Toy(FD)> domain [S1,S2,S3] 0 20,serialized' [S1,S2,S3] [5,5,5]
[precedences [d(2,1,superior),d(3,2,superior)],path_consistency
true,static_sets true,edge_finder true, decomposition true]

     yes
     S3 in 10..20
     S2 in 5..15
     S1 in 0..10
```

```
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

cumulative Starts Durations Resources Limit: `Starts, Durations y Resources` son tres listas de enteros de la misma longitud N. `Starts y Durations` representan el comienzo y la duración de las N tareas, es decir, lo mismo que en `Serialized`. `Resources` son los recursos que consume cada tarea, el total de los recursos consumidos no puede exceder de `Limit`. `Serialized` es un caso particular de `cumulative`.

```
Toy(FD)> cumulative [2,4,6] [4,6,8] [3,2,1] 40

      yes

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> cumulative [2,4,6] [4,6,8] [3,2,1] 4

      no.

      Elapsed time: 0 ms.
```

cumulative' Starts Durations Resources Limit serialOptions: `Starts, Durations, Resources y Limit` son lo mismo que en `cumulative` y `serialOptions` son las mismas opciones que se han explicado en `serialized'` y que se definieron en la tabla 4.1.

```
Toy(FD)> domain [S1,S2,S3] 0 20,cumulative' [S1,S2,S3] [5,5,5]
[6,7,8] 20 [precedences [d(2,1,superior),d(3,2,superior)],
path_consistency true,static_sets true,edge_finder true,
decomposition true]

      yes
      S3 in 0..20
      S2 in 0..20
      S1 in 0..20

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

exactly X L N: es `true` si `X` aparece exactamente `N` veces en la lista `L` de enteros o variables de dominio. Esta restricción no está definida en *Prolog*.

```
Toy(FD)> exactly 2 [1,2,2,2,3] X

      yes
      X == 3

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

- ***Restricciones de enumeración o etiquetado:*** `labeling/2, indomain/1.`

  Declaraciones de tipos

```
labeling :: [labelType] → [int]→ bool
indomain :: int → bool
```

  Como se vio en 2.4.4 el resolutor de dominos finitos no es completo debido al alto coste de los resolutores completos, lo que se puede hacer es, primero usar el resolutor incompleto para podar los dominios todo lo que se pueda y así se reducen los intervalos, a continuación dando valores concretos pertenecientes a esos intervalos se vuelve a verificar que las relaciones son ciertas para dichos valores. Las siguientes restricciones realizan la búsqueda sistemática de una variables o de una lista de variables, en este último caso se tienen opciones para guiar a la búsqueda a la hora de determinar cúal es la siguiente variable a evaluar o cual es el siguiente valor de la variable que se está evaluándo para probar.

  Las retricciones de enumeración reactivan el proceso de búsqueda cuando la propagación ya no es posible.

  indomain X asigna valores factibles del dominio a la variable `X` vía backtracking.

```
Toy(FD)> domain [X,Y] 0 5, sum [X,Y] (#<) 3, indomain X

      yes
      X == 0
      Y in 0..2

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      X == 1
      Y in 0..1

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      X == 2
      Y in 0

      Elapsed time: 0 ms.
```

```
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

labeling Options Variables donde `Variables` es una lista de variables y `Options` es una lista de opciones de búsqueda. Es `true` si exite una asignación de valores del dominio a las variables que hagan satisfactible las restricciones. Las opciones (`Options`) controlan la naturaleza de la búsqueda y están divididas en cuatro grupos.

El primer grupo de `Options` controla el orden en el cual las variables son seleccionadas para asignarles valores del dominio 2.4.6

- `leftmost`, toma las variables en el orden en el cual se definieron, este es el valor por defecto.
- `ff` usa el principio first-fail, selecciona la variable de menor dominio.
- `ffc` extiende la opción `ff` seleccionando la variable envuelta en el mayor número de restricciones.
- `mini, maxi`, la variable con el valor menor/mayor. Estas opciones se corresponde con las opciones `min` y `max` de *Prolog*, se cambiaron los nombres debido a conflictos.

El segundo grupo de `Options` controla el orden en el cual los valores de las variables son seleccionados 2.4.7

- `enum`, elige los valores desde el menor hasta el mayor.
- `step`, selecciona el mayor o el menor valor.
- `bisect`, divide al dominio en dos y toma el valor medio
- `up`, controla que el dominio sea explorado en orden ascendente. Este valor se toma por defecto.
- `down`, controla que el dominio sea explorado en orden descendente.

El tercer grupo de `Options` controla cuántas soluciones se desean o cuales.

- `each`, se desean todas las soluciones, este valor es por defecto. Esta opción se corresponde con la opción `all` de *Prolog*, se cambio el nombre debido a conflictos con el fichero *misc.toy*.
- `toMinimize(X), toMaximize(X)`, utiliza un algoritmo "branch-and-bound" para encontrar una solución que minimice (maximice) el dominio de la variable `X` en `L`. Estas opciones se corresponde con las opciones `minimize` y `maximize` de *Prolog*, se cambiaron los nombres debido a conflictos.

El cuarto grupo de `Options` controla el número se asunciones

- `assumptions(K)`, K es el número de asunciones hechas durante la búsqueda.

En la primitiva `assigment` se dio como ejemplo a nivel de comando:
`Toy(FD)> assignment [A,B] [C,D]` y como respuesta se obtuvo que las variables `A, B, C` y `D` tenían valores comprendidos entre 1 y 2 (1..2), si se aplica `labeling` sobre estas variables entonces se obtienen respuestas para valores concretos. Al ser

un problema con un árbol de búsqueda pequeño no es muy interesante dar opciones para guiar la búsqueda y es por ello por lo que se dejan las opciones de búsqueda por defecto (`[]`).

```
Toy(FD)> assignment [A,B] [C,D], labeling [] [A,B,C,D]

      yes
      A == 1
      B == 2
      C == 1
      D == 2

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      A == 2
      B == 1
      C == 2
      D == 1

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

- ***Restricciones de estadísticas:*** `fd_statistics'/1, fd_statistics/2`

  Declaraciones de tipos

```
fd_statistics  :: statistics → int → bool
fd_statistics' :: bool
```

  fd_statistics' siempre devuelve `true` y muestra un conjunto de estadísticas.

  fd_statistics' Key Value devuelve `true` si

  - Value unifica con el número de restricciones creadas y `Key == constraints` o
  - Value unifica con el número de veces que
    1. una restricción es reasumida y `Key == resumptions`
    2. una desvinculación ha sido detectada por una restricción y `Key == entailments`
    3. un dominio ha sido podado y `Key == prunings`
    4. hay backtracking porque el dominio ha llegado a ser vacío `Key == backtracks`

  Un ejemplo de estadísticas se mostrará en el problema de las n reinas (4.3.3).

4.3.3. Ejemplos Introductorios a $\mathcal{TOY}(\mathcal{FD})$

Una caracteristica de $\mathcal{TOY}(\mathcal{FD})$ es que puede resolver todos los ejemplos de $\mathcal{CLP}(\mathcal{FD})$ y además también puede resolver problemas de la programación funcional.

Primeramente se mostrarán los problemas que se dieron en (2.4.1) y que correponden a la programación $\mathcal{CLP}(\mathcal{FD})$, veamos que también se pueden resolver con $\mathcal{TOY}(\mathcal{FD})$.

Puzzles aritméticos

El primer ejemplo que se vio es el clásico puzzle aritmético, en el cual cada letra de la siguiente suma expresa un dígito. Se trata de averiguar qué dígito corresponde a cada letra.

```
  D O N A L D
+ G E R A L D
--------------
= R O B E R T
```

Las Variables D, O, N, A, L, G, E, R, B, T tienen dominio finito $Dom(D, G, R) = \{1, .., 9\}$, $Dom(O, N, A, L, E, B, T) = \{0, .., 9\}$

Del enunciado se obtiene como retricción implícita que las 10 variables deben ser asignadas a valores diferentes. Además está la restricción explícita de la suma

```
      100000*D + 10000*O + 1000*N + 100*A + 10*L + D
+     100000*G + 10000*E + 1000*R + 100*A + 10*L + D,
=     100000*R + 10000*O + 1000*B + 100*E + 10*R + T;
```

A continuación se muestra su código $\mathcal{TOY}(\mathcal{FD})$

```
include "cflpfd.toy"

dgr :: [int] -> [labelingType] -> bool
dgr [D,O,N,A,L,G,E,R,B,T] Label = true <==
    domain [O,N,A,L,E,R,B,T] 0 9,
    domain [D,G] 1 9,
    all_different [D,O,N,A,L,G,E,R,B,T],
    100000#*D #+ 10000#*O #+ 1000#*N #+ 100#*A #+ 10#*L #+ D
#+  100000#*G #+ 10000#*E #+ 1000#*R #+ 100#*A #+ 10#*L #+ D
#=  100000#*R #+ 10000#*O #+ 1000#*B #+ 100#*E #+ 10#*R #+ T,
    labeling Label [D,O,N,A,L,G,E,R,B,T]
```

y su ejecución desde la línea de comandos con dos tipos de etiquetado diferentes, el primero `first fail, (ff)` y el segundo con el valor por defecto `leftmost`.

```
Toy(FD)> dgr L [ff]

      yes
      L ==  [ 5, 2, 6, 4, 8, 1, 9, 7, 3, 0 ]

      Elapsed time: 31 ms.
```

```
more solutions (y/n/d) [y]?

      no.

      Elapsed time: 16 ms.

Toy(FD)> dgr L []

      yes
      L ==  [ 5, 2, 6, 4, 8, 1, 9, 7, 3, 0 ]

      Elapsed time: 469 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 78 ms.
```

Colorear un mapa

En este problema se da un mapa y unos colores, en la sección 2.4.1 se puso como ejemplo el mapa de Australia (figura 2.1) y los colores {rojo, amarillo, azul}. Se trata de colorear el mapa con dichos colores de tal forma que no existan dos regiones adyacentes del mismo color.

Se disponen de siete variables (WA, NT, SA, O, NSW, V, T) cuyo dominio es {rojo, amarillo, azul}.

Las restricciones que capturan el hecho de que no puede haber dos regiones adjuntas con el mismo color son

$$WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q,$$
$$SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V$$

El código correspondiente en $\mathcal{TOY}(\mathcal{FD})$ es:

```
include "cflpfd.toy"

mapa_australia :: [int] -> [labelingType] -> bool
mapa_australia [T,WA,NT,SA,Q,V,NSW] Label = true <==
    domain [T,WA,NT,SA,Q,V,NSW] 1 3,
    WA #\= NT, WA #\= SA, NT #\= SA, NT #\= Q, Q #\= SA, Q #\= NSW,
    NSW #\= SA, NSW #\= V, SA #\= V,
    labeling Label [T,WA,NT,SA,Q,V,NSW]
```

Este problema tiene 18 soluciones, aquí sólo se muestran dos.

```
Toy(FD)> mapa_australia L []

      yes
      L ==  [ 1, 1, 2, 3, 1, 1, 2 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
```

```
yes
L ==  [ 1, 1, 3, 2, 1, 1, 3 ]

Elapsed time: 0 ms.
```

Problema de las n reinas

Este es el típico problema de las n reinas, se dispone de un tablero de dimensiones $n_x n$ y de n reinas, se deben colocar las n reinas sobre el tablero, de tal forma que ninguna reina esté amenazada por otra reina. Se define las variables R_i y C_i como la fila y la columna que ocupa la reina i.

La restricción de no amenaza entre dos las reinas implica que no puede haber dos reinas en la misma fila, ni dos reinas en la misma columna, ni dos reinas en la misma diagonal.

El código correspondiente al problema de las n reinas en $\mathcal{TOY}(\mathcal{FD})$ está disponible en la distribución y se muestra a continuación

```
include "cflpfd.toy"
include "misc.toy"

queens :: int -> [int] -> [labelingType] -> bool
queens N L Label = true <==
       length L == N,
       domain L 1 N,
       constrain_all L,
       labeling Label L

constrain_all :: [int] ->  bool constrain_all [] = true
constrain_all [X|Xs] = true <==
       constrain_between X Xs 1,
       constrain_all Xs

constrain_between :: int -> [int] -> int -> bool
constrain_between X [] N = true
constrain_between X [Y|Ys] N = true <==
        no_threat X Y N,
        N1 == N+1,
        constrain_between X Ys N1

no_threat:: int -> int -> int -> bool
no_threat X Y I = true <==
        X  #\= Y,
        X #+ I #\= Y,
        X #- I #\= Y
```

La solución al problema con 4 reinas es

```
Toy(FD)> queens 4 L [], fd_statistics'
Resumptions: 126
Entailments: 62
Prunings: 82
```

```
Backtracks: 2
Constraints created: 30

      yes
      L ==  [ 2, 4, 1, 3 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

Resumptions: 48
Entailments: 30
Prunings: 31
Backtracks: 0
Constraints created: 0

      yes
      L ==  [ 3, 1, 4, 2 ]

      Elapsed time: 0 ms.
```

Emparejamientos

Este es el último problema que se dió en 2.4.1 y trata de emparejar un conjunto de mujeres { María, Sara, Ana } y un conjunto de hombres { Pedro, Pablo, Juan} teniendo en cuenta ciertas restricciones que se muestran en el grafo de la figura 2.3, donde las aristas representan las preferencias.

Se definen las variables X_{Maria}, X_{Sara} , X_{Ana}, de dominio { Pedro, Pablo, Juan}. Como restricción implícita se tiene que estas variables han de ser distintas.

El correspondiente código $\mathcal{TOY}(\mathcal{FD})$ es el siguiente

```
include "cflpfd.toy"
parejas :: [int] -> int -> [labelingType] -> bool
parejas [MARIA,SARA,ANA] N Label = true <==
        domain [MARIA,SARA,ANA] 1 N,
        SARA #\= 2,
        MARIA #\= 2,
        all_different [MARIA,SARA,ANA],
        labeling Label [MARIA,SARA,ANA]
```

y su ejecución

```
Toy(FD)> parejas L 3 [ff]

      yes
      L ==  [ 1, 3, 2 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      L ==  [ 3, 1, 2 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
```

```
no.

Elapsed time: 0 ms.
```

Están disponibles en la distribución otros ejemplos: send more money, las secuencias mágicas, circuitos electrónicos, código de Hamming ...

4.3.4. Ejemplos Más Complejos

Planificación de tareas

Un problema típico de restricciones con dominios finitos es el problema de planificar unas ciertas tareas que requieren unos determinados recursos y tienen que cumplir unas restricciones de precedencia, es decir, unas tareas se han de ejecutar antes que otras. Sea tX^{Y}_{mZ} donde X es el identificador de la tarea t, Y representa el tiempo que tarda la tarea en realizarse y mZ es la máquina en la cual se realiza la tarea. Un grafo de precedencias para las seis tareas es el que se muestra en la figura 4.1.

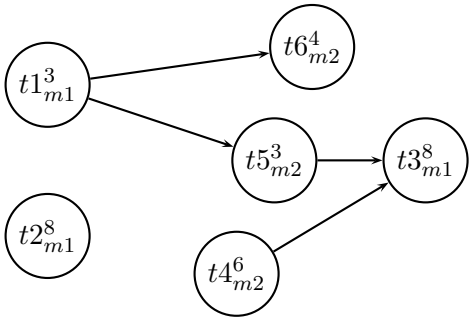


Figura 4.1: Grafo de precedencia de tareas

Esta planificación en concreto se resuelve mediante el siguiente programa que está diponible en la distribución *scheduling.toy*, en el cual se muestran caracteristicas de $\mathcal{TOY}(\mathcal{FD})$ que no existen en $\mathcal{CLP}(\mathcal{FD})$, como por ejemplo que los argumentos de las funciones no están encerrados entre paréntesis para permitir aplicaciones de orden superior, otra caracteristica es tener aplicaciones funcionales en argumentos, como por ejemplo `(End#-D)` en la segunda regla de definición de `horizon`. También se permite como azúcar sintáctico expresiones escritas con la sintaxis de *Prolog* que representan funciones de resultado booleano, este azúcar sintáctico está reflejado en la regla de definición de la función `sched`.

En el siguiente código se va a representar una lista de tareas (`task`) como una tupla de cinco elementos `task = (taskName, durationType, precedencesType, resourcesType, startType)` en la cual sus componentes son el nombre de la tarea, el tiempo que tarda en realizarse, una lista con las tareas que son precedentes, una lista con los recursos que requiere y el tiempo de inicio. La función principal de este programa es `schedule` que tiene como argumentos la lista de tareas que se desea planificar, el tiempo de inicio y el tiempo de finalización máximo de la planificación, es decir, todas las tareas se han debido ejecutar antes de este parámetro, estos dos últimos parámetros representan el intervalo de tiempo de la planificación. Si una tarea tarda en ejecutarse x y el intervalo de planificación es $[Start, End]$ entonces debe comenzar a ejecutarse en el intervalo $[Start, End - x]$, ya que si empieza en el intervalo $(End - x, End]$ no da tiempo a que se ejecute completamente, esta poda de los dominios que reflejan cuando han de comenzar las tareas esta

impementada en la función `horizon`. La función `scheduleTasks` impone las restricciones de precedencia y requerimientos de todas las tareas en la planificación a través de las funciones `precedeList` y `requireList`. También se tienen dos funciones `start` y `duration` que devuelven el tiempo de inicio y la duración respectivamente de la 5-tupla que determina una tarea para ser utilizadas posteriormente en la función `precedes`, esta función impone las restricciones de precedencia entre dos tareas. La función `requireList` impone las restricciones de los recursos haciendo que si dos tareas requieren el mismo recurso no se solapen, para ello es necesaria la función `noOverlaps` la cual dice que dos tareas no se solapan si una tarea cualquiera de las dos precede a la otra.

```
include "cflpfd.toy"
include "misc.toy"

data taskName = t1 | t2 | t3 | t4 | t5 | t6
data resourceName = m1 | m2
type durationType = int
type startType = int
type precedencesType = [taskName]
type resourcesType = [resourceName]
type task = (taskName, durationType, precedencesType, resourcesType,
             startType)

start :: task -> int
start (Name, Duration, Precedences, Resources, Start) = Start

duration :: task -> int
duration (Name, Duration, Precedences, Resources, Start) = Duration

schedule :: [task] -> int -> int -> bool
schedule TL Start End = true <== horizon TL Start End, scheduleTasks TL TL

horizon :: [task] -> int -> int -> bool
horizon [] S E = true
horizon [(N, D, P, R, S)|Ts] Start End = true <==
        domain [S] Start (End#-D),
        horizon Ts Start End

scheduleTasks :: [task] -> [task] -> bool
scheduleTasks [] TL = true
scheduleTasks [(N, D, P, R, S)|Ts] TL = true <==
        precedeList (N, D, P, R, S) P TL,
        requireList (N, D, P, R, S) R TL,
        scheduleTasks Ts TL

precedeList :: task -> [taskName] -> [task] -> bool
precedeList T [] TL = true
precedeList T1 [TN|TNs] TL = true <== belongs (TN, D, P, R, S) TL,
```

```
        precedes T1 (TN, D, P, R, S),
        precedeList T1 TNs TL

precedes :: task -> task -> bool
precedes T1 T2 = true <==  ST1 == start T1,
        DT1 == duration T1,
        ST2 == start T2,
        ST1 #+ DT1 #<= ST2

requireList :: task -> [resourceName] -> [task] -> bool
requireList T [] TL = true
requireList T [R|Rs] TL = true <==
        requires T R TL, requireList T Rs TL

requires :: task -> resourceName -> [task] -> bool
requires T R [] = true
requires (N1, D1, P1, R1, S1) R [(N2, D2, P2, R2, S2)|Ts] = true <==
        N1 /= N2,
        belongs R R2,
        noOverlaps (N1, D1, P1, R1, S1) (N2, D2, P2, R2, S2),
        requires (N1, D1, P1, R1, S1) R Ts
requires T1 R [T2|Ts] = true <== requires T1 R Ts

belongs :: A -> [A] -> bool
belongs R [] = false
belongs R [R|Rs] = true
belongs R [R1|Rs] = belongs R Rs

noOverlaps :: task -> task -> bool
noOverlaps T1 T2 = true <== precedes T1 T2
noOverlaps T1 T2 = true <== precedes T2 T1
```

Resolvemos mediante la línea de comandos el objetivo que representa la planificación del grafo dibujado en la figura 4.1

```
Toy(FD)> schedule [(t1,3,[t5,t6],[m1],S1), (t2,8,[],[m1],S2),
(t3,8,[],[m1],S3), (t4,6,[t3],[m2],S4), (t5,3,[t3],[m2],S5),
(t6,4,[],[m2],S6)] 1 20

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S6 in 10..16
      S5 in 7..9
      S4 in 1..3

      Elapsed time: 16 ms.
```

Si se desean valores concretos para S4, S5 y S6 entonces se añade `labeling` al objetivo, como el número de soluciones es muy alto sólo se muestran dos soluciones

```
Toy(FD)> schedule [(t1,3,[t5,t6],[m1],S1), (t2,8,[],[m1],S2),
(t3,8,[],[m1],S3), (t4,6,[t3],[m2],S4), (t5,3,[t3],[m2],S5),
(t6,4,[],[m2],S6)] 1 20, labeling [ff] [S1,S2,S3,S4,S5,S6]

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 7
      S6 == 10

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 7
      S6 == 11

      Elapsed time: 0 ms.
```

Observese que se puede dejar alguna variable sin etiquetar si así se desea, al igual que antes sólo se muestran dos soluciones

```
Toy(FD)> schedule [(t1,3,[t5,t6],[m1],S1), (t2,8,[],[m1],S2),
(t3,8,[],[m1],S3), (t4,6,[t3],[m2],S4), (t5,3,[t3],[m2],S5),
(t6,4,[],[m2],S6)] 1 20, labeling [ff] [S1,S2,S3,S4,S5]

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 7
      S6 in 10..16

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 8
      S6 in 11..16

Elapsed time: 0 ms.
```

Por último, si se añade al código la función `sched`, se puede lanzar un objetivo más corto. La inclusión del código se puede hacer directamente en el mismo fichero o en otro fichero que contenga la directiva `include "scheduling.toy"`. La función `sched` es

```
sched :: startType -> startType -> startType -> startType ->
```

```
          -> startType -> bool
sched S1 S2 S3 S4 S5 S6 :- Tasks == [(t1,3,[t5,t6],[m1],S1),
    (t2,8,[],[m1],S2),
    (t3,8,[],[m1],S3),
    (t4,6,[t3],[m2],S4),
    (t5,3,[t3],[m2],S5),
    (t6,4,[],[m2],S6)],
    schedule Tasks 1 20,
    labeling [ff] [S1,S2,S3,S4,S5,S6]
```

Y un objetivo en la línea de comandos es el siguiente del cual sólo se muestra dos soluciones

```
Toy(FD)> sched S1 S2 S3 S4 S5 S6
      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 7
      S6 == 10

      Elapsed time: 15 ms.

more solutions (y/n/d) [y]?

      yes
      S1 == 1
      S2 == 4
      S3 == 12
      S4 == 1
      S5 == 7
      S6 == 11

      Elapsed time: 0 ms.
```

4.3.5. Ejemplos que Utilizan la Evaluación Perezosa

Secuencias mágicas

El problema de las secuencias mágicas [17] dice que dada una serie de enteros no negativos $S = (s_0, s_1, \ldots, s_{n-1})$ es *mágica* si y sólo si hay s_i secuencias de i en S $\forall i \in \{1, \ldots, n-1\}$, por ejemplo, la serie (4,2,1,0,1,0,0,0) es una secuencia mágica de $S = (s_0, s_1, \ldots, s_7)$ pues el 0 se repite 4 veces, el 1 se repite 2 veces y análogamente con los demás elementos de la serie.

Una posible solución en $\mathcal{TOY}(\mathcal{FD})$ que no utiliza la evaluación perezosa es el código que se muestra a continuación y que esta disponible en la distribución *magicseq.toy*.

```
include "cflpfd.toy"
include "misc.toy"

magic :: int -> [int] -> [labelingType] -> bool
magic N L Label = true <==
    length L == N,
    domain L 0 (N-1),
```

```
    constrain L  L  0  Cs,
    sum L (#=) N,
    scalar_product Cs  L  (#=) N,
    labeling Label L

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain [] A  B  [] = true
constrain [X|Xs]  L  I  [I|S2] = true <==
    count I  L  (#=)  X,
    constrain Xs  L  (I+1)  S2
```

Como se puede observar se han utilizado las funciones `sum/3, scalar_product/4` y `count/4`, [7] que son primitivas de $\mathcal{TOY}(\mathcal{FD})$ de orden superior, ya que aceptan una restricción relacional de tipo `(int → int → bool)` como argumento, por ejemplo las restricciones `(#=)` o `(#<)`. También se puede observar una aplicación funcional como argumento en la segunda regla de definición de `constrain` porque tiene como argumento `(I+1)`. Su ejecución en la línea de comandos es

```
Toy(FD)> magic 8 L []

      yes
      L ==  [ 4, 2, 1, 0, 1, 0, 0, 0 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? n
```

Este código que se acaba de ver se puede enmarcar en la programación $\mathcal{CLP}(\mathcal{FD})$, a continuación se va a mostrar el código que utiliza la evaluación perezosa para generar una lista infinita de variables cuyo dominio está comprendido entre 0 y N-1 donde N es la longitud de la secuencia que se quiere calcular.

```
include "misc.toy"    %% To use take/2, map/2 and ./2
include "cflpfd.toy"

generateFD :: int -> [int]
generateFD N = [X | generateFD N] <== domain [X] 0 (N-1)

lazymagic :: int -> [int]
lazymagic N = L <== take N (generateFD N) == L,   %% Lazziness evaluation
    constrain L  L  0  Cs,
    sum L (#=) N,                 %% HO FD constraint
    scalar_product Cs  L  (#=) N, %% HO FD constraint
    labeling [ff] L

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain [] A  B  [] = true
constrain [X|Xs]  L  I  [I|S2] = true <==
    count I  L  (#=)  X,        %% HO FD constraint
    I1 == I+1,
    constrain Xs  L  I1  S2
```

Esta es una parte del código que se encuentra en la distribución *lazymagicseq.toy*. Un objetivo `lazymagic N == L`, devuelve en `L` la serie mágica de longitud `N`, que cumple la restricción `take N (generateFD N)` que es evaluada perezosamente ya que `generateFD` produce una lista infinita 3.7. Si ejecutamos

```
Toy(FD)> lazymagic 8 == L

      yes
      L ==  [ 4, 2, 1, 0, 1, 0, 0, 0 ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? n
```

Obtenemos una respuesta análoga a la anterior, pero se puede tener soluciones mucho más interesantes, por ejemplo, dados dos números N y M devolver una lista de cuyos componentes son secuencias mágicas de longitudes N..N+M-1. Para obtener estas nuevas soluciones es necesario añadir el siguiente código

```
magicfrom :: int -> [[int]]
magicfrom N = [lazymagic N | magicfrom (N+1)]
```

La resolución para N=8 y M=3 en la línea de comandos es

```
Toy(FD)> take 3 (magicfrom 8) == L

      yes
      L ==  [  [ 4, 2, 1, 0, 1, 0, 0, 0 ],  [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ],
      [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ] ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? n
```

Una solución análoga a la anterior que utiliza el orden superior se puede obtener añadiendo el código siguiente.

```
from :: int -> [int] from N = [N|from (N+1)]
```

Observese que en esta nueva solución no interviene la función `magicfrom`.

```
Toy(FD)> take 3 (map lazymagic (from 8)) == L

      yes
      L ==  [  [ 4, 2, 1, 0, 1, 0, 0, 0 ],  [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ],
      [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ] ]

      Elapsed time: 16 ms.

more solutions (y/n/d) [y]? n
```

Además de la notación currificada y el orden superior de funciones se puede utilizar la composición de funciones

```
lazyseries :: int -> [[int]]
lazyseries = map lazymagic.from
```

```
Toy(FD)> take 3 (lazyseries 8) == L

      yes
      L ==  [  [ 4, 2, 1, 0, 1, 0, 0, 0 ],  [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ],
      [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ] ]

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]? n
```

En este ejemplo se muestran caracteristicas de $\mathcal{TOY}(\mathcal{FD})$ que no existen en $\mathcal{CLP}(\mathcal{FD})$ como la capacidad de manejar listas infinitas mediante la evaluación perezosa, la notación currificada de funciones, el orden superior y la composición de funciones.

4.3.6. Traducción a *Prolog*

$\mathcal{TOY}$ es un sistema que está implementado en *Prolog*, y además los programas $\mathcal{TOY}$ se traducen a *Prolog* y este código es el que se ejecuta. Cuando se compila un fichero desde toy, por ejemplo si se compila el fichero `queens.toy` mediante el comando `/compile`, `Toy>/compile(queens.toy)` o `Toy>/compile(queens)`, se crea un fichero `queens.pl`, posteriormente se carga en el sistema el fichero `queens.pl` mediante el comando `/load`, `Toy>/load(queens)`, lo que permite ejecutar objetivos correspondientes al código definido por `queens.toy`.

Una alternativa más utilizada es el comando `Toy>/run(queens)` que hace el proceso de `/compile` y `/load` en un único comando. Cuando se compila y carga un fichero $\mathcal{TOY}$ se hace la traducción a un fichero *Prolog*, para realizar esta traducción se se crean ciertos ficheros temporales: `queens.tmp.toy, queens.tmp.tmp` y `queens.tmp.out`.

$\mathcal{TOY}(\mathcal{FD})$ es una extensión de $\mathcal{TOY}$, por lo tanto, para implementar las restricciones sobre dominios finitos en $\mathcal{TOY}$ lo que se ha hecho es añadir código para que el sistema sea capaz de activar el resolutor de dominios finitos de *Prolog* mediate el comando `/cflpfd` y una vez activo se cambie el prompt del sistema a `Toy(FD)>`. $\mathcal{TOY}(\mathcal{FD})$ tiene una serie de primitivas correspondientes a los dominios finitos.

4.3.7. Módulos del Sistema

Antes de detallar las modificaciones que se hicieron en el código es conveniente saber que el sistema está basado en módulos. En *Prolog* un programa se puede dividir en diferentes módulos cada uno con su espacio de nombres, donde los predicados son locales al módulo, pero los términos son globales. El sistema de módulos es plano, es decir, todos los módulos se ven unos a otros. Un predicado es visible en el módulo donde es definido, pero si un predicado es importado (`imports`) por otro módulo entonces es visible también en ese otro módulo. Los predicados declarados como públicos pueden ser exportados. Cualquier predicado puede ser llamado desde otro módulo anteponiendo al predicado con el nombre del módulo, el operador `:`.

Se define un módulo mediante el comando

```
:- module(ModuleName, ExportList).
```

Cuando el fichero es cargado, todos los predicados del fichero quedan contenidos en el módulo `ModuleName` y los predicados de la lista de `ExportList` puede ser exportados. Cuando una declaración de módulo es procesada, todos los predicados existentes en el módulo son eliminados antes de que los nuevos predicados sean cargados.

Cuando un módulo es cargado (`load_files`) desde otro módulo todos los predicados públicos son importados por el módulo que hace la llamada o se puede dar una lista de los predicados a importar.

Para cargar el sistema $\mathcal{TOY}$ lo que se hace es consultar el fichero *toy.pl* desde *Prolog*

```
| ?- ['toy.pl'].
```

En el fichero *toy.pl* se define el módulo toy y se cargan los módulos `initToy, tools` y `dyn`, a continuación se muestran las sentencias correspondientes contenidas en el fichero *toy.pl*

```
:- module(toy,[initiate/0]).
:- load_files(initToy,[if(changed),imports([initiate/0])]).
:- load_files(tools,[if(changed),imports([append/3])]).
:- load_files(dyn,[if(changed)]).
```

El predicado `initiate/0` es exportado desde el módulo `toy`, que a su vez es importado desde `initToy`. Del módulo `dyn` son importados todos los predicados.

Cuando se carga *toy.pl* desde *Prolog* se pueden ver una serie de mensajes informativos en los cuales vemos que el módulo `user` ha importado el módulo `toy` este a se vez importa el módulo `initToy` y este a su vez importa el módulo `process` y así sucesivamente. A continuación se muestra una selección de estos mensajes.

```
%  module toy imported into user
%   module initToy imported into toy
%    module process imported into initToy
%     module compil imported into process
%      module errortoy imported into compil
%       module tools imported into errortoy
%       module system imported into tools
%       module dyn imported into errortoy
%      module token imported into compil
%      module tools imported into token
%      module dyn imported into token
%       module gramma_toy imported into token
%       module errortoy imported into gramma_toy
%       module compil imported into gramma_toy
%       module tools imported into gramma_toy
%        module outgenerated imported into gramma_toy
%     module gramma_toy imported into compil
%     module tools imported into compil
%     module dyn imported into compil
%      module plgenerated imported into compil
%      module tools imported into plgenerated
%       module toycomm imported into plgenerated
```

```
%       module dyn imported into toycomm
        . . .
%  module dyn imported into toy
%  module terms imported into toy
```

En esta sección se van a listar todos los módulos del sistema detallando los cambios que se han hecho para introducir $\mathcal{TOY}(\mathcal{FD})$ versión 2.1.0.

Como notación tomaremos: nombre del módulo $\hookrightarrow$ nombre del fichero.

toy $\hookrightarrow$ toy.pl; éste es el módulo que carga todo el sistema $\mathcal{TOY}$, esto es posible debido a que en este fichero se encuentra la inicialización (`:- initialization continue.`) que se ejecuta automáticamente cuando el fichero se carga. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: user
- Este módulo importa los siguientes módulos: initoy, tools y dyn

initToy $\hookrightarrow$ initToy.pl; éste es el módulo que interactúa con el usuario, lee lo que contenga la línea de comandos, lo analíza sintácticamente y si comienza por / entonces lo leido es un comando, si comienza por > entonces lo leido es una expresión y sino es un objetivo. Una vez que está analizado y si es correcto se ejecuta.

Con la implementación de los dominios finitos hay dos nuevos comandos para hacer la carga y descarga (`/cflpfd` y `/nocflpfd`), por lo tanto la regla que procesa los comandos `processC` se amplía con dos reglas de definición nuevas.

```
processC("cflpfd") :-
    !,
    loadCflpfd.

processC("nocflpfd") :-
    !,
    unloadCflpfd.
```

- `loadCflpfd`: Comprueba que no esté activo ni el resolutor de reales ni el resolutor de dominios finitos. Si alguno está activo escribe un mensaje y finaliza. Si ninguno está activo entonces se carga un nuevo juego de primitivas que está contenido en el fichero `cflpfd.toy` y que se muestra en el apéndice B. El mecanismo para cargar este fichero es el siguiente, se construye la cadena de caracteres `"run('C:\\...\\nadacflpfd')"` con la ruta correspondiente y se lanza mediante el comando `processC`, esto hace que se compile y se cargue el fichero `nadacflpfd.toy` el cual contiene como única sentencia: `include "cflpfd.toy"`.

  Una vez cargadas las primitivas correspondientes a los dominios finitos se aserta un flag `cflpfd_active` mediante el predicado `assertcflpfd_active` que se importa del módulo `dyn`. Este flag indica que los dominios finitos están activos y finalmente se muestra un mensaje.

```
loadCflpfd:-
      clpr_active, !,
      nl, write('Cannot load both Real Domain Constraints and
                  Finite Domain Constraints libraries.'), nl.
loadCflpfd:-
      cflpfd_active, !,
      nl, write('Finite Domain Constraints library already
                  loaded.'), nl.
loadCflpfd:-
      complete_root_filename(nadacflpfd,F), name(F,FS),
      concatLsts(["run(",FS,")"], Command),
      processC(Command),
      assertcflpfd_active,
      nl, write('Finite Domain Constraints library loaded.'), nl.
```

- `unloadCflpfd:` para desactivar las restricciones sobre dominios finitos primero se comprueba si están activadas, después se desactiva el flag `cflpfd_active`, borrando el hecho `cflpfd_active` mediante `my_abolish(cflpfd_active/0)`

```
unloadCflpfd :-
        cflpfd_active, !,
        my_abolish(cflpfd_active/0),
        nl, write('Finite Domain Constraints library unloaded.'), nl.

unloadCflpfd :-
      nl, write('Cannot unload: Finite Domain Constraints library
                  is not loaded.'), nl.
```

Cada vez que procesa la línea de comandos con `process` y ha sido correcto se ejecuta de nuevo `execute` que lo primero que hace es escribir el prompt dependiendo del flag que esté activo, por ello se añade la siguiente regla de definición a `write_prompt`

```
write_prompt :-
  cflpfd_active, !,
write('Toy(FD)> ').
```

Además es necesario comprobar que los dominios finitos no estén activos cuando se ejecuta el comando `\cflpr`

```
loadClpr:-
      cflpfd_active, !,
      nl, write('Cannot load both Real and Finite Domain
                  Constraints libraries.'), nl.
```

- Este módulo es importado por: toy, evalexp, goals y transdebug.

- Este módulo importa los siguientes módulos: process, goals, transob, tools, dyn, basic, evalexp, dds, transfun, codfun, inferred, toycomm, writing, newout, osystem, transdebug, si clpr_active entonces importa el módulo primitivCodClpr sino primitivCod, si io_active entonces importa el módulo primitivCodIo, si iographic_active entonces importa el módulo primitivCodGra.

tools ↪ tools.pl; contiene predicados generales, por ejemplo `concatLsts`. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: toy, initToy, process, compil, plgenerated (basic), cflpr, toycom, codfun, gramma_toy (cabecera_gramma_toy), connected, dds, entailment, errortoy, evalexp, goals, gramma_toy (gramma_toy), basicFact, inferrer, navigating, osystem, primitivCod, primitivCodClpr, primitivCodGra, primitivCodIo, pVal, sailing, gramma_toy (transdcg), token, transdebug, transfun, transob y uncompile.
- Este módulo no importa módulos.

dyn ↪ dyn.pl; este módulo contiene predicados dinámicos. Los predicados dinámicos permiten añadir y eliminar cláusulas de los predicados presentes en el programa y hacerlo en tiempo de ejecución. Para insertar cláusulas de un predicado dinámico se utiliza el predicado `assert`. Una utilidad de los predicados dinámicos es la implementación del estado en los programas Prolog.

Para la implementación de los dominios finitos se define

```
:- dynamic cflpfd_active/0.
assertcflpfd_active:- assert(cflpfd_active).
```

Además se introducen en la lista de predicados que se pueden exportar.

```
:- module(dyn,[ ... cflpfd_active/0,assertcflpfd_active/0, ... ]).
```

- Este módulo es importado por: toy, initToy, process, compil, toycomm, connected, errortoy, evalexp, goals, navigating, osystem, inferrer, token, pVal, transdebug, primitivCod, primitivCodClpr, primitivCodGra y primitivCodIo.
- Este módulo no importa módulos.

process ↪ process.pl; este es el módulo principal de la generación de código. Supongamos que se quiere procesar el fichero `File.toy`, primero comprueba que existe y luego se crea un nuevo fichero `File.tmp.toy` uniendo los ficheros `File.toy` y `basic.toy` (apéndice A). En `File.tmp.toy` están las primitivas del sistema y las funciones definidas por el usuario. Este fichero se compila llamando al predicado `compil` del módulo `compil`, esta llamada realiza el análisis léxico y sintáctico dando como resultado el fichero `File.tmp.out` y usando como fichero intermedio `File.tmp.tmp`. Si ha habido algún error entonces no se genera el código y se muestra un error. Si todo ha ido bien se genera el código `File.pl` mediante `processOut/3`. Por último se borran los ficheros temporales.

Para la implementación de los dominios finitos sobre $\mathcal{TOY}$ es necesario reorganizar el código, para ello en el predicado `processOut` se añade el objetivo:

```
if(cflpfd_active,arrange_fdcode(NameF),true).
```

El predicado `arrange_fdcode/1` borra del fichero `File.pl` todas las funciones que están entre `init_fd` y `end_fd` (apéndice B) y añade las definiciones de funciones de dominios finitos que se encuentran en el fichero `cflpfdfile.pl` (apéndice C) al final del fichero File.pl, para añadir estas definiciones utiliza el predicado `append_cflpfdfile/1` que recibe como parámetro el fichero `File.pl`.

```
arrange_fdcode(NameF) :-
%Open file for reading
    name(Name,NameF),
    openFile(Name,".pl",read,HandleIn),
    openFile(voidfd,".pl",write,HandleOut),
%Remove void fd function definitions
    remove_void_defs(HandleIn,HandleOut),
    closeFile(HandleIn),
%Append the clpfdfile
    append_cflpfdfile(HandleOut),
    closeFile(HandleOut),
%Delete the source file
    append(NameF,".pl",NamePL),
    name(NamePLAtom,NamePL),
    delete_file(NamePLAtom),
%Rename the voidfd file as the source file
    rename_file('voidfd.pl',NamePLAtom).

append_cflpfdfile(HandleOut) :-
    complete_root_filename(cflpfdfile,File),
    openFile(File,".pl",read,HandleIn),
    copy_items(HandleIn,HandleOut),
    closeFile(HandleIn).
```

En el fichero `cflpfdfile.pl` se encuentra la directiva que carga el resolutor de dominios finitos de Sicstus.

```
:- use_module(library(clpfd)).
```

- Este módulo es importado por: initToy, inferrer y transdebug.
- Este módulo importa los siguientes módulos: compil, transfun, inferrer, tools, dyn, writing, newout, primFunct, si clpr_active importa el módulo primitivCod-Clprsino primitivCod, si io_active importa el módulo primitivCodIo, si iographic_active importa primitivCodGra y osystem.

compil ↪ compil.pl; este módulo genera el código intermedio (`File.tmp.out`). El fichero `File.tmp.toy` que se generó en el módulo `process` y en el cual están las primitivas del sistema y las funciones definidas por el usuario se compila llamando al predicado `compil`, esta llamada realiza el análisis léxico, sintáctico y parte del semántico dando como resultado el fichero `File.tmp.out` y usando como fichero intermedio `File.tmp.tmp`. Si ha habido algún error entonces no se genera el código.

Dentro del análisis semántico hay que añadir

```
semantic(HS,includecflpfd(Chain),[],
            _SinIn,includecflpfd,Tables,Error) :-
   !,
  % pasamos el nombre de fichero a cadena Prolog
   chainToyToProlog(Chain,ChainProlog),
  % pasamos el nombre de fichero a  atomo
   name(FileInc,ChainProlog),
   assert(initToy:cflpfdfile(FileInc)).
```

- Este módulo es importado por: process, transob y gramma_toy.
- Este módulo importa los siguientes módulos: errortoy, token, gramma_toy, tools, dyn, primFunct, primitivCod, si clpr_active importa el módulo primitivCodClpr, si io_active importa el módulo primitivCodIo, si iographic_active importa primitivCodGra.

plgenerated ↪ basic.pl; Contiene las funciones predefinidas y los tipos del sistema. Para verificar la consistencia de tipos y sintáctica es necesario saber la aridad y los tipos de las constructoras y funciones. Para cada constructora se define una cláusula de la forma:

```
const(Nombre,Aridad,Tipo,Tipo_destino).
```

donde `Nombre` es el identificador de la constructora, `Aridad` es la aridad de la aplicación total de la constructora, `Tipo` es el tipo que tiene asociado y el argumento `Tipo_destino` es el tipo de la constructora aplicada totalmente. En el fichero *basic.pl* están definidas las clausulas para las constructoras predefinidas.

De forma análoga se definen las cláusulas para las funciones, también están definidas en el fichero *basic.pl*.

```
funct(Nombre,Aridad,Ar_Tipo,Tipo,Tipo destino).
```

Las funciones en $\mathcal{TOY}$ pueden utilizar la notación infija, para ello se usa la función `infix` con los parámetros: símbolos son operadores, si asocian a la derecha `infixr`, a la izquierda `infixl` o si no asocian y , la precedencia que tienen.

```
infix(Operador, Asociatividad, Precedencia).
```

Lo que se hace es cargar inicialmente el fichero `basic.pl` que contiene el módulo `plgenerated` inicial, pero cuando se genere un `fichero.pl` y se cargue (en el módulo `process`), como se genera como módulo `plgenerated`, se borra el módulo `plgenerated` antiguo y se carga el nuevo.

- Este módulo es importado por: initToy, toycomm y codfun.
- Este módulo importa los siguientes módulos: tools, toycomm, primFunct y primitivCod

plgenerated ↪ cflpr.pl; análogo al anterior basic.pl pero para las restricciones sobre reales. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo no es importado.
- Este módulo importa los siguientes módulos: tools, toycomm, primFunct y primitivCod.

toycomm ↪ toycomm.pl; , este módulo contiene todos los predicados comunes a todos los programas $\mathcal{TOY}$. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, plgenerated (basic), cflpr basicFact, evalexp, navigating y primitivCod, primitivCodClpr, primitivCodGra y pVal.
- Este módulo importa los siguientes módulos: dyn, plgenerated (basic), codfun, tools, primFunct, si clpr_active importa el módulo primitivCodClpr y sino importa primitivCod, si io_active importa el módulo primitivCodIo, si iographic_active importa primitivCodGra, goals y pval.

codfun ↪ codfun.pl; este módulo genera al código de una función usando el árbol de definición. Este módulo no se ha modificado en la implementación de restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, toycomm, evalexp, transfun, primitivCod, primitivCodClpr, primitivCodGra y primitivCodIo.
- Este módulo importa los siguientes módulos: tools, basic, newout, primFuct,

basicFact ↪ basicFact.pl; Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: navigating.
- Este módulo importa los siguientes módulos: tools y toycomm

gramma_toy ↪ cabecera_gramma_toy.pl; este módulo sólo carga ficheros. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo no es importado.
- Este módulo importa los siguientes módulos: errortoy, compil, tools y newout.

connected ↪ connected.pl; , busca los componentes conectados del grafo de dependencias, concretamente las dependencias funcionales de las funciones declaradas por el usuario y las dependencias standard. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: inferrer.
- Este módulo importa los siguientes módulos: tools, dyn y newout.

dds ↪ dds.pl; genera el árbol definicional (dds) asociado a la función. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, evalexp y transfun.
- Este módulo importa los siguientes módulos: tools y newout.

entailment ↪ entailment.pl; vincula dos hechos básicos. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: navigating.
- Este módulo importa los siguientes módulos: navigating y tools.

errortoy ↪ errortoy.pl; , este módulo se usa para el tratamiento de errores en la compilación. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: compil, gramma_toy (cabecera_gramma_toy), goals, gramma_toy (gramma_toy), gramma_toy (transdcg), transfun, transob, transdebug, primitivCod, primitivCodClpr, primitivCodGra y primitivCodIo y pVal.
- Este módulo importa los siguientes módulos: tools y dyn.

evalexp ↪ evalexp.pl; evalúa las expresiones. $\mathcal{TOY}$ evalúa expresiones además de comandos y objetivos. Muestra la evaluación en pantalla. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy.
- Este módulo importa los siguientes módulos: dds, writing, inferrer, dyn, toycomm, tools, codfun, transfun y initToy.

goals ↪ goals.pl; este módulo muestra las respuestas a los objetivos. Para la implementación de los dominios finitos se ha introducido:

```
:- use_module(library(clpfd)).

/*********************************************************/
/*      SALIDA INTERVALOS DE DOMINIOS FINITOS            */
/*********************************************************/
% Llamada showIntervals(Handle, L/_).
showIntervalsIfNeeded(Handle, L/_Ls) :-
      cflpfd_active,
      !,
```

```
      showIntervals(Handle, L, _Ls).
showIntervalsIfNeeded(Handle, L1/L2).

showIntervals(Handle, [], []).

showIntervals(Handle, [(Val,Nom)|Xs], Ls) :-
      var(Val), !,
      nl, tab(6), write(Nom), write(' in '),
      fd_dom(Val,Range), write(Range),
      showIntervals(Handle, Xs, Ls).
showIntervals(Handle, [X|Xs], [X|Ls]) :-
      showIntervals(Handle, Xs, Ls).
```

- Este módulo es importado por: initToy, toycomm, navigating, sailing y transdebug.
- Este módulo importa los siguientes módulos: tools, dyn, writing, transdebug, initToy y errorToy.

gramma_toy ↪ gramma_toy.pl; este módulo contiene la sintáxis del lenguaje. Este fichero se ha modificado para implementar los dominios finitos en la siguiente cláusula:

```
topdecl(includecflpfd(_368),_670,_671,_672,_673,_674,_675) :-
    reserved(includecflpfd,_670,_840,_672,_673,_843,_675),
    !,
    chain(_368,_840,_671,_672,_843,_674,_675).
```

- Este módulo es importado por: compil, transob y token.
- Este módulo importa los siguientes módulos: errortoy, compil, tools y newout.

inferrer ↪ inferrer.pl; para inferir los tipos se deja en la BD cláusulas de la forma functiontype(F,T) para cada función F. Si F no tiene declaración de tipos entonces T es el tipo inferido. Si F tiene declaración de tipos entonces T es el tipo declarado si es unificable con el inferido. Si hay errores en la inferencia entonces se muestra un mensaje por pantalla y para el proceso de compilación. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, process y evalexp.
- Este módulo importa los siguientes módulos: connected, dyn, primitives, primFunct, si clpr_active importa el módulo primitivCodClpr, si io_active importa el módulo primitivCodIo, si iographic_active importa primitivCodGra, basic, newout, tools y writing.

navigating ↪ navigating.pl; navega por el árbol de depuración. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: entailment y transdebug.

- Este módulo importa los siguientes módulos: tools, goals, pVal, entailment, dyn, basicFact y toycomm.

outgenerated ↪ newout.pl; en este módulo se declaran dinámicamente los siguientes predicados `cdata/4, infix/4, ftype/4, fun/4, depends/2` y `primitive/3`. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, process, codfun, gramma_toy que se locliza en cabecera_gramma_toy.pl, conected, dds, gramma_toy (gramma_toy), inferrer, gramma_toy (transdcg), transdebug y transfun.
- Este módulo no importa módulos.

osystem ↪ osystem.pl; en este módulo están los predicados que contienen llamadas al sistema. Este módulo depende de la plataforma en la cual se trabaje: Linux, Windows ... Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, process y primitivCodGra.
- Este módulo importa los siguientes módulos: tools y dyn.

primFunct ↪ primFunct.pl; módulo de primitivas del sistema. Define los operadores infijos y las funciones anteponiendo prim y primitive respectivamente. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: process, compil, plgenerated (basic), cflpr, toycomm, codfun, inferrer, primitivCod y primitivCodGra.
- Este módulo no importa módulos.

primitivCod ↪ primitivCod.pl; módulo de primitivas del sistema. Este módulo no se ha modificado para la implementación de los dominios finitos

- Este módulo es importado por: initToy, compil, plgenerated (basic), cflpr, toycomm, inferrer, y primitivCodGra.
- Este módulo importa los siguientes módulos: toycomm, tools, codfun, dyn, errortoy, primFunct, pVal y compil.

primitivCod ↪ primitivCodClpr.pl; este módulo es análogo al primitives.pl y se usa cuando el resolutor sobre reales está activado.

- Este módulo es importado por: initToy, procces, compil, toycomm e inferrer.
- Este módulo importa los siguientes módulos: toycomm, tools, codfun, dyn y errortoy.

primitivCodGra ↪ primitivCodGra.pl; módulo de primitivas del sistema. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, process, compil, toycomm, inferrer, y primitivCodGra.

- Este módulo importa los siguientes módulos: primFunct, primitivCod, dyn, osystem, si io_active importa el módulo primitivCodIo, toycomm, tools, codfun y errortoy.

primitivCodIo ↪ primitivCodIo.pl; módulo de primitivas del sistema. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: initToy, process y compil.
- Este módulo importa los siguientes módulos: tools, codfun, dyn y errortoy

pVal ↪ pVal.pl; este módulo es el encargado del predicado pVal para la depuración. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: toycomm, navigating, primitivCod y transdebug.
- Este módulo importa los siguientes módulos: dyn, writing, tools, transdebug, toycomm y errortoy.

sailing ↪ sailing.pl; navega por el árbol de depuración. Este módulo no se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo no es importado.
- Este módulo importa los siguientes módulos: tools y goals.

gramma_toy ↪ transdcg.pl; hace la traslación de dcg´s a cláusulas prolog. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo no es importado.
- Este módulo importa los siguientes módulos: errortoy, compil, tools y newout.

token ↪ token.pl; módulo con la definición de predicados para el análisis lexicográfico. No se ha modificado para implementar las restricciones sobre dominios finitos.

- Este módulo es importado por: compil y transob.
- Este módulo importa los siguientes módulos: tools, dyn y gramma_toy.

transdebug ↪ transdebug.pl; transforma un programa ya compilado `File.tmp.out` generado tras el análisis sintáctico, en otro ya preparado para la depuración.

- Este módulo es importado por: initToy, goals y pVal.
- Este módulo importa los siguientes módulos: tools, initToy, pVal, newout, topDown, compil, errortoy, dyn, process, goals y navigating.

transfun ↪ transfun.pl; devuelve el código asociado a una función a partir del nombre de la función y genera el árbol definicional.

- Este módulo es importado por: initToy, process y evalexp.
- Este módulo importa los siguientes módulos: dds, codfun, tools, newout y errortoy.

transob ↪ transob.pl; toma el primer carácter primer para ver si es una expresión a evaluar, un comando o un objetivo.

- Este módulo es importado por: initToy.
- Este módulo importa los siguientes módulos: token, errortoy, compil, gramma_toy y tools

uncompile ↪ uncompile.pl; produce un fichero `file.toy` desde un fichero `file.tmp.out`.

- Este módulo no es importado.
- Este módulo importa los siguientes módulos: tools y writing.

writing ↪ writing.pl; es el encargado de la escritura de atomos.

- Este módulo es importado por: initToy, process, evalexp, goals, inferrer, pVal y uncompile.
- Este módulo importa los siguientes módulos: dyn, si io_active entonces carga basicIo o si (iographic_active o ioclpr_active) carga basicGra, basic y newout.

Además de los ficheros anteriormente nombrados existen en las distribución de $\mathcal{TOY}(\mathcal{FD})$ otros ficheros como por ejemplo:
`primFunctIo.pl` y `primFunctGra.pl` los cuales contienen primitivas del sistema.
`aditionalClauses.toy` contiene definiciones.
`basicIo.pl` y `basicIo.toy` contienen primitivas de la entrada/salida de ficheros.
`basicGra.pl` y `basicGra.toy` contienen primitivas de la entrada/salida gráfica.
`toy.ini` son las opciones de configuración.
`nadacflpfd.toy` y `nada.toy` son ficheros especificos para la implementación.
`tcltk.toy` y `tcltk.pl` son bibliotecas para la integración de tcltk en $\mathcal{TOY}(\mathcal{FD})$.
`cflpfd.pl` y `cflpfdfile.pl` estos dos ficheros se muestran en B y C y corresponden a la extensión de las restricciones sobre dominios finitos.

4.3.8. Observaciones sobre los apéndices

De los apéndices A y B no se comenta nada porque el apéndice A contiene las declaraciones de las funciones y tipos predefinidos de $\mathcal{TOY}$ y B contiene las declaraciones de las funciones y tipos predefinidos de la extensión sobre los dominios finitos de $\mathcal{TOY}$, es decir, $\mathcal{TOY}(\mathcal{FD})$.

El apéndice C muestra el fichero *cflpfdfile.toy* el cual contiene la invocación al resolutor de restricciones sobre de dominios finitos de *Prolog*.

```
:- use_module(library(clpfd)).
```

Además contiene el código que traduce cada restricción $\mathcal{TOY}(\mathcal{FD})$ a su correspondiente restricción *Prolog*. El resolutor de *Prolog*, anteriormente activado, es el encargado de resolver las restricciones. La función `exactly` está definida en este fichero porque no existe en Sicstus *Prolog*.

```
% exactly
'$exactly'(NEle, VL, N, Out, Cin, Cout):-
        Out=true,
```

```
        hnf(NEle, HNEle, Cin, Cout1),
        hnf(VL, HVL, Cout1, Cout2),
        toyListToPrologList(HVL, PrologHVL),
        hnf(N, HN, Cout2, Cout),
        exactly(HNEle,PrologHVL,HN).

exactly(_,[],0). exactly(X,[Y|L],N):-
          X #=Y #<=> B,
          N #= M+B,
          exactly(X,L,M).
```

Para traducir cada restricción $\mathcal{TOY}(\mathcal{FD})$ a su correspondiente restricción *Prolog* se deben calcular las formas normales de cabeza de cada argumento y con estas formas normales se hace la llamada a la correspondiente restricción de *Prolog*. Las variables `Cin` y `Cout` representan el almacen de restricciones de entrada y de salida.

```
% #>
$#>(L, R, Out, Cin, Cout):-
      hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
      ((Out=true,  HL #>  HR);
       (Out=false, HL #=< HR)).
```

En esta definición se pueden observar dos llamadas a restricciones *Prolog* (la restricción $\mathcal{TOY}(\mathcal{FD})$ comienza por \$). Esto se debe a que se pueden resolver objetivos que unifican con `true` o con `false`.

```
Toy(FD)> X #> 3 == true

      yes
      X in 4..sup

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.

Toy(FD)> X #> 3 == false

      yes
      X in inf..3

      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

      no.

      Elapsed time: 0 ms.
```

Si la restricción contiene una lista esta debe ser "plegada"(pasar de 1:2:3:[] a [1,2,3]) mediante el predicado:

```
toyListToPrologList([],[]).
toyListToPrologList(A:R,[A|RR]):-toyListToPrologList(R,RR).
```

Por ejemplo la función `sum` tiene como primer argumento una lista, que se pasa a forma normal de cabeza y se le aplica `toyListToPrologList` (sum :: [int] → (int → int → bool) → int → bool)

```
'$sum'(VL, Op, O, Out, Cin, Cout):-
        Out=true,
        hnf(VL, HVL, Cin, Cout1),
        toyListToPrologList(HVL, PrologHVL),
        hnf(Op, HOp, Cout1, Cout2),
        hnf(O, HO, Cout2, Cout),
        sum(PrologHVL,HOp,HO).
```

Las variables de los dominios finitos toman valores enteros, por este motivo en ciertas restricciones se evita pasar un valor real como argumento.

```
'$domain'(VL, Low, Upp, Out, Cin, Cout):-
        hnf(VL, HVL, Cin, Cout1),
        hnf(Low, HLow, Cout1, Cout2),
        hnf(Upp, HUpp, Cout2, Cout),
        HLow1 is integer(HLow),
        HUpp1 is integer(HUpp),
        toyListToPrologList(HVL, PrologHVL),
        ((domain(PrologHVL, HLow1, HUpp1), Out=true)
         ;
         (HLow2 is integer(HLow1 - 1), HUpp2 is integer(HUpp1 + 1),
          domain_remove(PrologHVL, HLow2, HUpp2),
          Out=false)).

domain_remove([], _HLow2, _HUpp2).
domain_remove([X|Xs], HLow2, HUpp2) :-
        X in (inf..HLow2) \/ (HUpp2..sup),
        domain_remove(Xs, HLow2, HUpp2).
```

Capítulo 5

Conclusiones y Trabajo Futuro

En este trabajo hemos visto que la programación $\mathcal{CFLP}(\mathcal{FD})$ contiene propiedades que no existen en la programación $\mathcal{CLP}(\mathcal{FD})$ como la notación funcional y currificada, los tipos, las funciones y restricciones de orden superior, composición de restricciones, patrones de orden superior, evaluación perezosa, polimorfismo ... Estas caracteristicas hacen que $\mathcal{CFLP}(\mathcal{FD})$ tenga mejores herramientas que $\mathcal{CLP}(\mathcal{FD})$, por ejemplo se puede hacer una comprobación de tipos. Además $\mathcal{CFLP}(\mathcal{FD})$ es más expresivo debido a la combinación de la notación funcional, relacional y currificada. En especial, la evalución perezosa permite utilizar estructuras de datos infinitas imposibles de manejar en $\mathcal{CLP}(\mathcal{FD})$.

En este trabajo se presenta una implementación concreta de un lenguaje $\mathcal{CFLP}(\mathcal{FD})$, describiendo la sintáxis, la disciplina de tipos y cómo se integran las restricciones de dominios finitos como funciones sobre el lenguaje existente $\mathcal{TOY}$. En este nuevo lenguaje que denominamos $\mathcal{TOY}(\mathcal{FD})$ se permite usar las funciones en los mismos lugares dónde puede aparecer un dato (resultados, argumentos, elementos de estructuras de datos...) proporcionando un mecanismo lo bastante potente como para poder definir restricciones de orden superior.

Mostramos cómo es posible construir una implementación de un lenguaje $\mathcal{CFLP}(\mathcal{FD})$ conectando el lenguaje lógico funcional preexistente $\mathcal{TOY}$ con el resolutor de restricciones de dominios finitos que nos permite la realización conjunta de cómputos perezosos simultaneándolo con la resolución de restricciones de dominios finitos. Más concretamente, hemos utilizado la implementación anterior del lenguaje lógico funcional $\mathcal{TOY}$ y la hemos conectado al eficiente resolutor de dominios finitos de Sicstus *Prolog*.

La integración de las restricciones de dominios finitos dentro de los lenguajes lógico funcionales nos permiten recibir beneficios de ambos mundos, como se ha visto mediante los ejemplos expuestos en los cuales tenemos patrones de orden superior, aplicaciones parciales, indeterminismo, evaluación perezosa, variables lógicas, tipos, dominios finitos y restricciones. Desde el punto de vista declarativo hemos conseguido una gran capacidad de expresión en nuestro lenguaje que no hubiésemos conseguido utilizando otros paradigmas.

Habiendo hecho un curso de doctorado sobre la programación con restricciones, me interesó el tema y tras la realización del presente trabajo en el que he profundizado en los aspectos, tanto prácticos como teóricos, de la integración de resolutores en un marco lógico funcional tengo la intención de seguir trabajando en esta misma línea en dos sentidos, una es la integración de los resolutores de dominios finitos y de dominios reales dentro del propio $\mathcal{TOY}$ y la otra línea es estudio de algunos resolutores generales como por ejemplo OLP e intentar integrarlos en $\mathcal{TOY}$ con vistas a mejorar la eficiencia de cómputos del lenguaje.

Apéndice A

basic.toy

En este apéndice se muestra el fichero *basic.toy* el cual contiene las declaraciones de las funciones y tipos predefinidos de $\mathcal{TOY}$.

```
data bool = true | false

% unary integer and real functions
primitive uminus, % unary minus operator
          abs     % absolute value
          :: real -> real

% unary real functions
primitive sqrt,
          ln,exp, % natural logarithm and exponential
          sin,cos,tan,cot,
          asin,acos,atan,acot,
          sinh,cosh,tanh,coth,
          asinh,acosh,atanh,acoth
          :: real -> real

% binary arithmetic operators and functions for reals and integers
primitive (+),(-),(*),min,max :: real -> real -> real

% binary real functions
primitive (/),
          (**),log   % Exponentiation and logarithm
          :: real -> real -> real

% integer powers
primitive (^) :: real -> int -> real


% binary integer functions
primitive div, mod, gcd :: int -> int -> int

% rounding and truncating functions
primitive round,trunc,floor,ceiling :: real -> int

% integer to real conversion
primitive toReal   :: int -> real

% relational operators
primitive (<),(<=),(>),(>=) :: real -> real -> bool

% equality and disequality functions
primitive (==),(/=) :: A -> A -> bool

primitive ord:: char -> int
```

```
primitive chr:: int -> char

/************************    E/S ****************************/

% operaciones basicas

primitive putChar:: char -> (io unit)
primitive done:: io unit
primitive getChar:: io char
primitive return:: A -> io A

% secuencializadoras

primitive (>>):: io A -> io B -> io B
primitive (>>=):: io A -> (A -> io B) -> io B

% operaciones adicionales

primitive putStr:: [char] -> io unit
primitive putStrLn:: [char] -> io unit
primitive getLine:: io [char]
primitive cont1:: char -> io [char]
primitive cont2:: char -> [char] -> io [char]

% operaciones de fichero

primitive writeFile:: [char] -> [char] -> io unit
primitive readFile:: [char] -> io [char]
primitive readFileContents:: handle -> io [char]

/*************************************************************************/

primitive dVal:: A -> pVal primitive dValToString:: A -> [char]
primitive showVal::A->B primitive delayVal::A->B->pVal primitive
selectWhereVariableXi:: A -> int -> int -> B -> C

/*************************************************************************/

% infix operator precedences

infix  98 ^,**
infixl 90 /
infixr 90 *
infixl 50 +,-
infix  30 <,<=,>,>=
infix  10 ==,/=
infixl 11 >>,>>=

% 'if_then_else' and 'if_then' functions are equivalent to their 'sugared'
% version 'if .. then .. else' and 'if .. then'. They are
% to write partial applications.

primitive if_then_else :: bool -> A -> A -> A
%if_then_else true X Y = X
%if_then_else false X Y = Y

primitive if_then :: bool -> A -> A
%if_then true X = X

% 'flip' function is necessary for syntactic management of operator sections.
primitive flip :: (A -> B -> C) -> B -> A -> C
% flip F X Y = F Y X
```

Apéndice B

cflpfd.toy

En este apéndice se muestra el fichero *cflpfd.toy* el cual contiene las declaraciones de las funciones y tipos predefinidos de la extensión sobre dominios finitos de $\mathcal{TOY}$, es decir, $\mathcal{TOY}(\mathcal{FD})$. Además están declaradas funciones y tipos que se utilizan a bajo nivel. A continuación de cada declaración hay una regla de definición la cual es necesaria por la forma en la cual está implementado $\mathcal{TOY}$.

```
%%%%%%%%%%%%%%%%%%% BEGIN %%%%%%%%%%%%%%%%%%%%
init_fd :: int
init_fd = 0

%%%%%%%%%%%%%%%%%%% MEMBERSHIP CONSTRAINTS %%%%%%%%%%%%%%%%%%%%

domain :: [int] -> int -> int -> bool
domain [0] 0 0 = true

subset :: int -> int -> bool
subset 0 0 = true

setcomplement :: int -> int -> bool
setcomplement 0 0 = true

inset :: int -> int -> bool
inset 0 0 = true

intersect :: int -> int -> int
intersect 0 0 = 0

%%%%%%%%%%%%%%%%%%% PROPOSITIONAL CONSTRAINTS %%%%%%%%%%%%%%%%%%%%

infix  15 #<=>

(#<=>) :: bool -> bool -> bool
true #<=> true = true

%%%%%%%%%%%%%%%%%%% RELATIONAL CONSTRAINTS  %%%%%%%%%%%%%%%%%%%%%

infix  30 #>,#<,#>=,#<=

(#>) :: int -> int -> bool
0 #> 0 = true

(#<) :: int -> int -> bool
0 #< 0 = true

(#>=) :: int -> int -> bool
0 #>= 0 = true
```

```
(#<=) :: int -> int -> bool
0 #<= 0 = true

infix  20 #=,#\=

(#=) :: int -> int -> bool
0 #= 0 = true

(#\=) :: int -> int -> bool
0 #\= 0 = true

%%%%%%%%%%%%%%%%%%% ARITHMETIC OPERATORS %%%%%%%%%%%%%%%%%%%%%

infixr 90 #*

(#*) :: int -> int -> int
0 #* 0 = trunc 0

infixl 90 #/

(#/) :: int -> int -> int
0 #/ 0 = trunc 0

infixl 50 #+,#-

(#+) :: int -> int -> int
0 #+ 0 = trunc 0

(#-) :: int -> int -> int
0 #- 0 = trunc 0

infixl 90 #&

(#&) :: int -> int -> int
0 #& 0 = trunc 0

%%%%%%%%%%%%%%%%%%% REIFIED CONSTRAINTS %%%%%%%%%%%%%%%%%%%%%%%

exactly :: int -> [int] -> int -> bool
exactly 0 [0] 0 = true

%%%%%%%%%%%%%%%%%%% ARITHMETIC CONSTRAINTS %%%%%%%%%%%%%%%%%%%%%

sum :: [int] -> (int -> int -> bool) -> int -> bool
sum [0] (>) 0 = true

scalar_product :: [int] -> [int] -> (int -> int -> bool) -> int ->
bool
scalar_product [0] [0] (>) 0 = true

%%%%%%%%%%%%%%%%%%% COMBINATORIAL CONSTRAINTS %%%%%%%%%%%%%%%%%%%
count:: int -> [int] -> (int -> int -> bool) ->  int -> bool
count 0 [0] (>) 0 = true

element :: int -> [int] -> int -> bool
element  0 [0] 0 = true

circuit :: [int] -> bool
circuit [0] = true

circuit' :: [int] -> [int] -> bool
circuit' [0] [0] = true

serialized :: [int] -> [int] -> bool
serialized [0] [0] = true

data typeprecedence = d (int,int,liftedInt)
```

```
data liftedInt = superior | lift int

data serialOptions = precedences [typeprecedence]
           | path_consistency bool
           | static_sets bool
           | edge_finder bool
           | decomposition bool

serialized' :: [int] -> [int] -> [serialOptions] -> bool
serialized' [0] [0]  [precedences([d(2,1,lift(2))])] = true

all_different :: [int] ->  bool
all_different    [0]    =   true

data reasoning = value
               | domains
               | range

data options =   on reasoning
               | complete bool

all_different' :: [int] -> [options] -> bool
all_different' [0] [on value] = true

all_distinct :: [int] -> bool
all_distinct [0] = true

all_distinct' :: [int] -> [options] -> bool
all_distinct'[0] [on value] = true

assignment :: [int] -> [int] -> bool
assignment [0] [0] = true

cumulative :: [int] ->  [int] ->  [int] -> int  -> bool
cumulative [0] [0] [0] 0 = true

cumulative' :: [int] ->  [int] ->  [int] -> int  ->
[serialOptions] -> bool
cumulative' [0] [0] [0] 0 [precedences([d(2,1,lift(2))])] = true

%%%%%%%%%%%%%%%%%% LABELING %% %%%%%%%%%%%%%%%%%%%%%%%%

data labelingType = ff
                  | ffc
                  | leftmost
                  | mini
                  | maxi
                  | step
                  | enum
                  | bisect
                  | up
                  | down
                  | each
                  | toMinimize int
                  | toMaximize int
                  | assumptions int

labeling :: [labelingType] -> [int] -> bool
labeling [ff] [0] = true

indomain :: int -> bool
indomain 0 = true

minimize :: bool -> int -> bool
minimize true 0 = true
```

```
maximize :: bool -> int -> bool
maximize true 0 = true

%%%%%%%%%%%%%%%%%% STATISTICS  %%%%%%%%%%%%%%%%%%%%%%%%

data statistics =   resumptions
                  | entailments
                  | prunings
                  | backtracks
                  | constraints

fd_statistics :: statistics -> int -> bool
fd_statistics resumptions 0 = true

fd_statistics' ::  bool fd_statistics' = true

%%%%%%%%%%%%%%%%%% INDEXICAL CONSTRAINTS %%%%%%%%%%%%%%%%%%%%%%

isin:: int -> (int,int) -> bool
isin 0 (0,0) = true

minimum :: int -> int
minimum 0 = 0

%%%%%%%%%%%%%%%%%% RESTRICCIONES BAJO NIVEL %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% REFLECTION PREDICATES    %%%%%%%%%%%%%%%%%%%%

sup::int sup = 0
inf::int inf = 0

fd_var :: int -> bool
fd_var 0  = true

fd_min :: int -> int
fd_min 0  = 0

fd_max :: int -> int
fd_max 0 = 0

fd_size :: int -> int
fd_size 0 = 0

fd_degree :: int -> int
fd_degree 0 = 0

fd_closure :: [int] -> [int]
fd_closure [0] = [0]

fd_neighbors :: int -> [int]
fd_neighbors 0 = [0]

data fdset = interval int int

fd_set :: int -> [fdset] -> bool
fd_set 0 [interval 0 0] = true

is_fdset :: [fdset] -> bool
is_fdset [interval 0 0] = true

empty_fdset :: [fdset] -> bool
empty_fdset [interval 0 0]  = true

fdset_parts :: [fdset] -> int -> int -> [fdset] -> bool
fdset_parts [interval 0 0] 0 0 [interval 0 0] = true

empty_interval :: int -> int -> bool
empty_interval 0 0 = true
```

```
fdset_interval :: [fdset] -> int -> int -> bool
fdset_interval [interval 0 0] 0 0 = true

fdset_singleton :: [fdset] -> int -> bool
fdset_singleton [interval 0 0] 0 = true

fdset_min ::  [fdset] -> int
fdset_min [interval 0 0] = 0

fdset_max ::  [fdset] -> int
fdset_max [interval 0 0] = 0

fdset_size ::  [fdset] -> int
fdset_size [interval 0 0] = 0

list_to_fdset ::  [int] -> [fdset]
list_to_fdset [0] = [interval 0 0]

fdset_to_list ::  [fdset] -> [int]
fdset_to_list [interval 0 0] = [0]

fdset_add_element ::  [fdset] -> int -> [fdset]
fdset_add_element [interval 0 0] 0 = [interval 0 0]

fdset_del_element ::  [fdset] -> int -> [fdset]
fdset_del_element [interval 0 0] 0 = [interval 0 0]

fdset_disjoint :: [fdset] -> [fdset] -> bool
fdset_disjoint [interval 0 0] [interval 0 0] = true

fdset_intersect :: [fdset] -> [fdset] -> bool
fdset_intersect [interval 0 0] [interval 0 0] = true

fdset_intersection :: [fdset] -> [fdset] -> [fdset]
fdset_intersection [interval 0 0] [interval 0 0] = [interval 0 0]

fdset_intersection' :: [fdset] -> [fdset]
fdset_intersection' [interval 0 0] = [interval 0 0]

fdset_member ::  int -> [fdset] -> bool
fdset_member 0 [interval 0 0] = true

fdset_belongs :: int -> int -> bool
fdset_belongs 0 0 = true

fdset_equal :: [fdset] -> [fdset]-> bool
fdset_equal [interval 0 0] [interval 0 0] = true

fdset_subset :: [fdset] -> [fdset]-> bool
fdset_subset [interval 0 0] [interval 0 0] = true

fdset_subtract :: [fdset] -> [fdset] -> [fdset]
fdset_subtract [interval 0 0] [interval 0 0] = [interval 0 0]

fdset_union :: [fdset] -> [fdset] -> [fdset]
fdset_union [interval 0 0] [interval 0 0] = [interval 0 0]

fdset_union' :: [fdset] -> [fdset]
fdset_union' [interval 0 0] = [interval 0 0]

fdset_complement :: [fdset] -> [fdset]
fdset_complement [interval 0 0] = [interval 0 0]

data range =  cte    int int         %% a..a
            | uni    range range     %% range \/ range
            | inter  range range     %% range /\ range
```

```
            | compl  range           %% \range

fd_dom :: int -> range -> bool
fd_dom 0 (cte 0 0) = true

range_to_fdset :: range -> [fdset]
range_to_fdset (cte 0 0) = [interval 0 0]

fdset_to_range :: [fdset] -> range
fdset_to_range [interval 0 0] = (cte 0 0)

data wakeoptions = domm int
                 | minn int
                 | maxx int
                 | minmax int
                 | vall int

fd_global :: bool -> State -> [wakeoptions] -> bool  %% State may be
                                  %% any data constructor applied to a tuple
                                  %% e.g., st(Xs,4).
fd_global true A [(domm 0)] = true

%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%
end_fd :: int end_fd = 0
```

Apéndice C

cflpfdfile.pl

En este apéndice se muestra el fichero *cflpfdfile.toy* el cual contiene la invocación al resolutor de restricciones sobre dominios finitos de *Prolog*, el código que traduce cada retricción $\mathcal{TOY}(\mathcal{FD})$ a su correspondiente restricción *Prolog*, el código de la función `exactly` ya que no está definida en *Prolog* y el código de las funciones definidas a bajo nivel que son necesarias.

```
%%%%%%%%%%%  BEGIN OF CFLPFDFILE   %%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%  CFLPFD CONSTRAINTS    %%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(clpfd)).

/**************    CODE FOR FUNCTIONS    **************/

%%%%%%%%%%%%%%%%%% CONSTRAINTS     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%% MEMBERSHIP CONSTRAINTS  %%%%%%%%%%%%%%%%%%%%%%

% domain
'$domain'(VL, Low, Upp, Out, Cin, Cout):-
        hnf(VL, HVL, Cin, Cout1),
        hnf(Low, HLow, Cout1, Cout2),
        hnf(Upp, HUpp, Cout2, Cout),
        HLow1 is integer(HLow),
        HUpp1 is integer(HUpp),
        toyListToPrologList(HVL, PrologHVL),
        ((domain(PrologHVL, HLow1, HUpp1), Out=true)
         ;
         (HLow2 is integer(HLow1 - 1), HUpp2 is integer(HUpp1 + 1),
          domain_remove(PrologHVL, HLow2, HUpp2),
         Out=false)).

domain_remove([], _HLow2, _HUpp2).
domain_remove([X|Xs], HLow2, HUpp2) :-
        X in (inf..HLow2) \/ (HUpp2..sup),
        domain_remove(Xs, HLow2, HUpp2).

subset(X,Y) +:
  X in dom(Y)/\dom(X).

setcomplement(X,Y) +:
  X in \dom(Y).

% subset
'$subset'(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
```

```
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  subset(HL,HR));
         (Out=false, setcomplement(HL,HR))).

% setcomplement
'$setcomplement'(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  setcomplement(HL,HR));
         (Out=false, subset(HL,HR))).

% inset
'$inset'(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
    hnf(R, HR, Cout1, Cout),
        ((Out=true,  fd_set(HR,SR), HL in_set SR);
         (Out=false, fd_set(HR,SR), fdset_complement(SR,CSR), HL in_set CSR)).

% intersect
'$intersect'(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
    hnf(R, HR, Cout1, Cout),
        fd_set(HR,SR),
        fd_set(HL,SL),
        fdset_intersection(SR,SL,SI),
        Out in_set SI.

%%%%%%%%%%%%%%%%%% PROPOSITIONAL CONSTRAINTS   %%%%%%%%%%%%%%%%%%

from_bool_to_int(false,0).
from_bool_to_int(true,1).

% #<=>
$#<=>(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
        from_bool_to_int(HL,HLINT),
        from_bool_to_int(HR,HRINT),
        (HLINT #<=> HRINT,!,Out=true);
        (Out=false).

%%%%%%%%%%%%%%%%%% RELATIONAL CONSTRAINTS      %%%%%%%%%%%%%%%%%%%%

% #>
% #>
$#>(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #>  HR);
         (Out=false, HL #=< HR)).

% #<
$#<(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #<  HR);
         (Out=false, HL #>= HR)).

% #>=
$#>=(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
    hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #>= HR);
         (Out=false, HL #< HR)).

% #<=
$#<=(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
```

```
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #=< HR);
         (Out=false, HL #> HR)).

% #=
$#=(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #=  HR);
         (Out=false, HL #\= HR)).

% #\=
$#\=(L, R, Out, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
    hnf(R, HR, Cout1, Cout),
        ((Out=true,  HL #\= HR);
         (Out=false, HL #=  HR)).

%%%%%%%%%%%%%%%%%% ARITHMETIC OPERATORS    %%%%%%%%%%%%%%%%%%%%

% #*
$#*(L, R, O, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout2),
      hnf(O, HO, Cout2, Cout),
        HL * HR #= HO.

% #/
$#/(L, R, O, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout2),
      hnf(O, HO, Cout2, Cout),
        HL / HR #= HO.

% #+
$#+(L, R, O, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout2),
      hnf(O, HO, Cout2, Cout),
        HL + HR #= HO.

% #-
$#-(L, R, O, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout2),
      hnf(O, HO, Cout2, Cout),
        HL - HR #= HO.

% #&
$#&(L, R, O, Cin, Cout):-
        hnf(L, HL, Cin, Cout1),
      hnf(R, HR, Cout1, Cout2),
      hnf(O, HO, Cout2, Cout),
        HL mod HR #= HO.


%%%%%%%%%%%%%%%%%% REIFIED    CONSTRAINTS      %%%%%%%%%%%%%%%%%%%%

% exactly
'$exactly'(NEle, VL, N, Out, Cin, Cout):-
        Out=true,
        hnf(NEle, HNEle, Cin, Cout1),
        hnf(VL, HVL, Cout1, Cout2),
        toyListToPrologList(HVL, PrologHVL),
        hnf(N, HN, Cout2, Cout),
        exactly(HNEle,PrologHVL,HN).

exactly(_,[],0).
```

```
exactly(X,[Y|L],N):-
           X #=Y #<=> B,
           N #= M+B,
           exactly(X,L,M).

%%%%%%%%%%%%%%%%%% ARITHMETIC CONSTRAINTS      %%%%%%%%%%%%%%%%%%

% sum
'$sum'(VL, Op, O, Out, Cin, Cout):-
        Out=true,
        hnf(VL, HVL, Cin, Cout1),
        toyListToPrologList(HVL, PrologHVL),
        hnf(Op, HOp, Cout1, Cout2),
        hnf(O, HO, Cout2, Cout),
        sum(PrologHVL,HOp,HO).

% scalar_product
'$scalar_product'(VL1, VL2, Op, O, Out, Cin, Cout):-
        Out=true,
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1, PrologHVL1),
        hnf(VL2, HVL2, Cout1, Cout2),
        toyListToPrologList(HVL2, PrologHVL2),
        hnf(Op, HOp, Cout2, Cout3),
        hnf(O, HO, Cout3, Cout),
        scalar_product(PrologHVL1,PrologHVL2,HOp,HO).


%%%%%%%%%%%%%%%%%% COMBINATORIAL CONSTRAINTS   %%%%%%%%%%%%%%%%%%

% count
'$count'(V, VL, Op, O, Out, Cin, Cout):-
        Out=true,
        hnf(V, HV, Cin, Cout1),
        hnf(VL, HVL, Cout1, Cout2),
        toyListToPrologList(HVL,PrologHVL),
        hnf(Op, HOp, Cout2, Cout3),
        hnf(O, HO, Cout3, Cout),
        count(HV,PrologHVL,HOp,HO).


% element
'$element'(I, VL, V, Out, Cin, Cout):-
        Out=true,
        hnf(I, HI, Cin, Cout1),
        hnf(VL, HVL, Cout1, Cout2),
        hnf(V, HV, Cout2, Cout),
        toyListToPrologList(HVL,PrologHVL),
        element(HI, PrologHVL, HV).

% circuit
'$circuit'(VL, Out, Cin, Cout):-
        Out=true,
        hnf(VL, HVL, Cin, Cout),
        toyListToPrologList(HVL,PrologHVL),
        circuit(PrologHVL).

% serialized
 '$serialized'(VL1, VL2, Out, Cin, Cout):-
        Out=true,
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1,PrologHVL1),
        hnf(VL2, HVL2, Cout1, Cout),
        toyListToPrologList(HVL2,PrologHVL2),
        serialized(PrologHVL1,PrologHVL2).
```

```
% 'serialized\''
'$serialized\''(VL1,VL2, OpL, Out, Cin, Cout):-
        Out=true,
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1,PrologHVL1),
        hnf(VL2, HVL2, Cout1, Cout2),
        toyListToPrologList(HVL2,PrologHVL2),
        hnf(OpL, HOpL, Cout2, Cout),
        toyListToPrologList(OpL,PrologOpL),
        process_options(PrologOpL,ProcessedPrologOpL),
        serialized(PrologHVL1,PrologHVL2,ProcessedPrologOpL).

process_options([],[]).
process_options([precedences(_T)|Ls],[precedences(T1)|Lss]):-
                                hnf(_T, _TT,_D2, _E),
                                toyListToPrologList(_TT,_TTT),
                          process_precedences(_TTT,T1),
                                process_options(Ls,Lss).
process_options([X|Ls],[X|Lss]):-process_options(Ls,Lss).

process_precedences([],[]).
process_precedences([d('$$tup'((I1,I2,I3)))|Ls],[d(I1,I2,O3)|Lss]):-
                                 process_value(I3,O3),
                                 process_precedences(Ls,Lss).
process_precedences([X|Ls],[X|Lss]):-process_precedences(Ls,Lss).

process_value(lift(I3),I3).
process_value(superior,sup).

% 'circuit\''
'$circuit\''(VL1, VL2, Out, Cin, Cout):-
        Out=true,
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1,PrologHVL1),
        hnf(VL2, HVL2, Cout1, Cout),
        toyListToPrologList(HVL2,PrologHVL1),
        circuit(PrologHVL1,PrologHVL2).

% all_different
'$all_different'(IL, Out, Cin, Cout):-
        Out=true,
        hnf(IL, HIL, Cin, Cout),
        toyListToPrologList(HIL, PrologHIL),
        all_different(PrologHIL).

% 'all_different\''
'$all_different\''(IL, OpL, Out, Cin, Cout):-
        Out=true,
        hnf(IL, HIL, Cin, Cout1),
        toyListToPrologList(HIL, PrologHIL),
        hnf(OpL, HOpL, Cout1, Cout),
        toyListToPrologList(HOpL, PrologHOpL),
        all_different(PrologHIL,PrologHOpL).

%all_distinct
'$all_distinct'(IL, Out, Cin, Cout):-
        Out=true,
        hnf(IL, HIL, Cin, Cout),
        toyListToPrologList(HIL, PrologHIL),
        all_distinct(PrologHIL).

% 'all_distinct\''
'$all_distinct\''(IL, OpL, Out, Cin, Cout):-
        Out=true,
        hnf(IL, HIL, Cin, Cout1),
```

```
        toyListToPrologList(HIL, PrologHIL),
        hnf(OpL, HOpL, Cout1, Cout),
        toyListToPrologList(HOpL, PrologHOpL),
        all_distinct(PrologHIL,PrologHOpL).

% assignment
'$assignment'(VL1, VL2, Out, Cin, Cout):-
        Out=true,
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1, PrologHVL1),
        hnf(VL2, HVL2, Cout1, Cout),
        toyListToPrologList(HVL2, PrologHVL2),
        assignment(PrologHVL1,PrologHVL2).

% cumulative
'$cumulative'(SL, DL, RL, LIM, Out, Cin, Cout):-
        Out=true,
        hnf(SL, HSL, Cin, Cout1),
        toyListToPrologList(HSL, PrologHSL),
        hnf(DL, HDL, Cout1, Cout2),
        toyListToPrologList(HDL, PrologHDL),
        hnf(RL, HRL, Cout2, Cout3),
        toyListToPrologList(HRL, PrologHRL),
        hnf(LIM, HLIM, Cout3, Cout),
        cumulative(PrologHSL, PrologHDL, PrologHRL, HLIM).

% 'cumulative\''
'$cumulative\''(SL, DL, RL, LIM, OL, Out, Cin, Cout):-
        Out=true,
        hnf(SL, HSL, Cin, Cout1),
        toyListToPrologList(HSL, PrologHSL),
        hnf(DL, HDL, Cout1, Cout2),
        toyListToPrologList(HDL, PrologHDL),
        hnf(RL, HRL, Cout2, Cout3),
        toyListToPrologList(HRL, PrologHRL),
        hnf(LIM, HLIM, Cout3, Cout4),
        hnf(OL, HOL, Cout4, Cout),
        toyListToPrologList(HOL, PrologHOL),
        process_options(PrologHOL,ProcessedPrologHOL),
        cumulative(PrologHSL, PrologHDL, PrologHRL, HLIM,ProcessedPrologHOL).

%%%%%%%%%%%%%%%%%% LABELING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

'$labeling'(LTL, VL, true, Cin, Cout):-
        hnf(LTL, HLTL, Cin, Cout1),
        hnf(VL, HVL, Cout1, Cout),
        toyListToPrologList(HLTL, PrologHLTL),
        toyListToPrologList(HVL, PrologHVL),
        processLabelingType(PrologHLTL, LabPrologHLTL),
        labeling(LabPrologHLTL, PrologHVL).

processLabelingType([], []) :- !.
processLabelingType([mini|As], [min|Bs]) :-
        !, processLabelingType(As, Bs).
processLabelingType([maxi|As], [max|Bs]) :-
        !, processLabelingType(As, Bs).
processLabelingType([toMinimize(X)|As], [minimize(X)|Bs]) :-
        !, processLabelingType(As, Bs).
processLabelingType([toMaximize(X)|As], [maximize(X)|Bs]) :-
        !, processLabelingType(As, Bs).
processLabelingType([each(X)|As], [all(X)|Bs]) :-
        !, processLabelingType(As, Bs).
processLabelingType([Option|As], [Option|Bs]) :-
```

```
        !, processLabelingType(As, Bs).

% indomain
'$indomain'(I, true, Cin, Cout):-
         hnf(I, HI, Cin, Cout),
         indomain(HI).

% minimize
'$minimize'(B, I, true, Cin, Cout):-
         hnf(B, HB, Cin, Cout1),
         hnf(I, HI, Cout1, Cout),
         minimize(HB, HI).

% maximize
'$maximize'(B, I, true, Cin, Cout):-
         hnf(B, HB, Cin, Cout1),
         hnf(I, HI, Cout1, Cout),
         maximize(HB, HI).

%%%%%%%%%%%%%%%%%% STATISTICS PREDICATES   %%%%%%%%%%%%%%%%%%

% fd_statistics
'$fd_statistics'(S, I, true, Cin, Cout):-
         hnf(S, HS, Cin, Cout1),
         hnf(I, HI, Cout1, Cout),
         fd_statistics(HS,HI).

% 'fd_statistics\''
'$fd_statistics\''(true, Cin, Cin):-fd_statistics.

%%%%%%%%%%%%%%%%%% INDEXICAL CONSTRAINTS   %%%%%%%%%%%%%%%%%%%%%%

% isin
'$isin'(_A, _B, true, _C, _D):-
        hnf(_B, '$$tup'(_E), _C, _F),
        hnf(_E, ','(_G, _H), _F, _F1),
    hnf(_G,_GG,_F1,_F2),
    hnf(_H,_HH,_F2,_F3),
    hnf(_A,_AA,_F3,_D),
    lanzar(_AA,_GG,_HH).

lanzar(_AA,_GG,_HH)+:
        _GG=..[min,_YY],
    _AA in min(_YY).._HH.

% minimum
'$minimum'(_A,min(_AA), _C, _D):-
    hnf(_A,_AA,_C,_C1),
        printf("\n Las variables son _AA = %w\n",[_AA]).

%%'$sup'(sup, Cin, Cin).
'$sup'(9999999, Cin, Cin).

% inf
'$inf'(-9999999, Cin, Cin).


%%%%%%%%%%%%%%%%%% REFLECTION FUNCTIONS    %%%%%%%%%%%%%%%%%

% fd_min

% fd_var
'$fd_var'(V, true, Cin, Cout):-
    hnf(V,HV,Cin,Cout),
    fd_var(HV).

'$fd_min'(I,Min,Cin,Cout):-
    hnf(I,HI,Cin,Cout1),
```

```
    hnf(Min,HMin,Cout1,Cout),
    fd_min(HI, HMin).

% fd_max

'$fd_max'(I,Max,Cin,Cout):-
    hnf(I,HI,Cin,Cout1),
    hnf(Max,HMax,Cout1,Cout),
    fd_max(HI, HMax).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% fd_size
'$fd_size'(V, S, Cin,Cout):-
    hnf(V,HV,Cin,Cout1),
    hnf(S,HS,Cout1,Cout),
    fd_size(HV, HS).

% fd_degree
'$fd_degree'(V, S, Cin,Cout):-
    hnf(V,HV,Cin,Cout1),
    hnf(S,HS,Cout1,Cout),
    fd_degree(HV, HS).

% fd_closure
'$fd_closure'(VL1, VL2, Cin, Cout):-
        hnf(VL1, HVL1, Cin, Cout1),
        toyListToPrologList(HVL1,PrologHVL1),
        fd_closure(PrologHVL1,PrologHVL2),
        toyListToPrologList(HVL2,PrologHVL2),
        hnf(VL2, HVL2, Cout1, Cout).

% fd_neighbors
'$fd_neighbors'(V, S, Cin, Cout):-
    hnf(V,HV,Cin,Cout1),
    hnf(S,HS,Cout1,Cout),
    fd_neighbors(HV,HS).


hnf_recursive([], [], _, _).
hnf_recursive(_B, [[_F|_G] | _Resto], Cin, Cout):-
        hnf(_T, interval(_F, _G), Cin, Cout1),
        hnf_recursive(_R, _Resto, Cout1, Cout2),!,
        hnf(_B, :(_T, _R), Cout2, Cout).

% fd_set
'$fd_set'(V, L, true, Cin, Cout):-
        hnf(V, HV, Cin, Cout1),
        fd_set(HV,HB),
        hnf_recursive(L, HB, Cout1, Cout),!.

% is_fdset
'$is_fdset'(_A, true, _B, _C):-
        hnf_recursive(_A, _AA, _B, _H),
        is_fdset(_AA),!.

% empty_fdset
'$empty_fdset'(_A, true, _B, _C):-
        hnf_recursive(_A, _AA, _B, _C),
        empty_fdset(_AA),!.

% fdset_parts
'$fdset_parts'(_A, _B, _C, _D, true, _E, _F2):-
        hnf_recursive(_A, _AA, _E, _I),
        hnf(_B,_BB,_I,_Cout1),
        hnf(_C,_CC,_Cout1,_II),
```

```
        hnf_recursive(_D, _DD, _II, _F2),
        fdset_parts(_AA,_BB,_CC,_DD),!.

% empty_interval
'$empty_interval'(_A, _B, true, _C, _D):-
        hnf(_A,_A1,_C,Cout1),
        hnf(_B,_B1,Cout1,_D),
        empty_interval(_A1,_B1),!.

% fdset_interval
'$fdset_interval'(_A, _B, _C, true, _D, _E):-
        hnf_recursive(_A, _AA, _D, _H),
        hnf(_B,_B1,_H,Cout1),
        hnf(_C,_C1,Cout1,_E),
        fdset_interval(_AA,_B1,_C1),!.

% fdset_singleton
'$fdset_singleton'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _AA, _C, _H),
        hnf(_B,_B1,_H,_D),
        fdset_singleton(_AA,_B1),!.

% fdset_min
'$fdset_min'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _Resto, _C, _H),
        hnf(_B,_B1,_H,_D),
        fdset_min(_Resto,_B1),!.

% fdset_min
'$fdset_min'(_A, _O, _B, _C):-
        hnf_recursive(_A, _AA, _B, _H),
        fdset_min(_AA,_Output),
        hnf(_O,_Output, _H, _C),!.

% fdset_max
'$fdset_max'(_A, _O, _B, _C):-
        hnf_recursive(_A, _AA, _B, _H),
        fdset_max(_AA,_Output),
        hnf(_O,_Output, _H, _C),!.

% fdset_size
'$fdset_size'(_A, _O, _B, _C):-
        hnf_recursive(_A, _AA, _B, _H),
        fdset_size(_AA,_Output),
        hnf(_O,_Output, _H, _C),!.

% list_to_fdset
'$list_to_fdset'(_A, _O, _B, _C):-
        hnf(_A, _F, _B, _D),
        toyListToPrologList(_F,_FF),
        list_to_fdset(_FF,_BB),
        hnf_recursive(_O, _BB, _D, _C),!.

% fdset_to_list
'$fdset_to_list'(_A, _B, _C, _E):-
        hnf_recursive(_A, _AA, _C, _D),
        fdset_to_list(_AA,_BB),
        toyListToPrologList(_B1,_BB),
        hnf(_B, _B1, _D, _E),!.

% fdset_add_element
'$fdset_add_element'(_A, _B, _C, _D, _E):-
        hnf_recursive(_A, _AA, _D, _F),
        hnf(_B,_B1,_F,_G),
        fdset_add_element(_AA,_B1,_CC),
        hnf_recursive(_C, _CC, _G, _E),!.
```

```
% fdset_del_element
'$fdset_del_element'(_A, _B, _C, _D, _E):-
        hnf_recursive(_A, _AA, _D, _F),
        hnf(_B,_B1,_F,_G),
        fdset_del_element(_AA,_B1,_CC),
        hnf_recursive(_C, _CC, _G, _E),!.

% fdset_disjoint
'$fdset_disjoint'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _AA, _C, _E),
        hnf_recursive(_B, _BB, _F, _D),
        fdset_disjoint(_AA, _BB),!.

% fdset_intersect
'$fdset_intersect'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _AA, _C, _E),
        hnf_recursive(_B, _BB, _F, _D),
        fdset_intersect(_AA, _BB),!.

% fdset_intersection
'$fdset_intersection'(_A, _B, _C, _D, _E):-
        hnf_recursive(_A, _AA, _D, _F),
        hnf_recursive(_B, _BB, _F, _G),
        fdset_intersection(_AA, _BB, _CC),
        hnf_recursive(_C, _CC, _G, _E),!.

% 'fdset_intersection\''
'$fdset_intersection\''(_A, _B, _C, _D):-
        hnf_recursive(_A, _AA, _C, _E),
        fdset_intersection(_AA, _BB),
        hnf_recursive(_B, _BB, _E, _D),!.

% fdset_member
'$fdset_member'(_A, _B, true, _C, _D):-
        hnf(_A,_A1,_C,_H),
        hnf_recursive(_B, _BB, _H, _D),
        fdset_member(_A1,_BB).


% fdset_belongs
'$fdset_belongs'(_A, _B, true, _C, _C):-
        hnf(_A,_A1,_C,_H),
        hnf(_B, _B1, _H, _D),
        fd_set(_B1, FDSet),
        fdset_member(_A1,FDSet).

% fdset_equal
'$fdset_equal'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _AA, _C, _F),
        hnf_recursive(_B, _BB, _F, _D),
        fdset_eq(_AA, _BB),!.

% fdset_subset
'$fdset_subset'(_A, _B, true, _C, _D):-
        hnf_recursive(_A, _AA, _C, _F),
        hnf_recursive(_B, _BB, _F, _D),
        fdset_subset(_AA, _BB),!.

% fdset_subtract
'$fdset_subtract'(_A, _B, _C, _D, _E):-
        hnf_recursive(_A, _AA, _D, _F),
        hnf_recursive(_B, _BB, _F, _G),
        fdset_subtract(_AA, _BB, _CC),
        hnf_recursive(_C, _CC, _G, _E),!.

% fdset_union
'$fdset_union'(_A, _B, _C, _D, _E):-
```

```
        hnf_recursive(_A, _AA, _D, _F),
        hnf_recursive(_B, _BB, _F, _G),
        fdset_union(_AA, _BB, _CC),
        hnf_recursive(_C, _CC, _G, _E),!.

% 'fdset_union\''
'$fdset_union\''(_A, _B, _C, _D):-
        hnf_recursive(_A, _AA, _C, _F),
        fdset_union(_AA, _BB),
        hnf_recursive(_B, _BB, _F, _D),!.

% fdset_complement
'$fdset_complement'(_A, _B, _C, _D):-
        hnf_recursive(_A, _AA, _C, _F),
        fdset_complement(_AA, _BB),
        hnf_recursive(_B, _BB, _F, _D),!.

hnf_recursive3(_B, _E..F, _C, _D):-
        hnf(_B, cte(_E, _F), _C, _D).

hnf_recursive3(_B, _EE \/_FF, _C, _D):-
        hnf_recursive3(_E,_EE,_C,_H),
        hnf_recursive3(_F,_FF,_H,_I),
        hnf(_B, uni(_E, _F), _I, _D).

hnf_recursive3(_B, _EE \/_FF, _C, _D):-
        hnf_recursive3(_E,_EE,_C,_H),
        hnf_recursive3(_F,_FF,_H,_I),
        hnf(_B, inter(_E, _F), _I, _D).

hnf_recursive3(_B, _BB, _C, _D):-
        _BB=..[\,_EE],
        hnf(_B, compl(_E), _C, _H),
        hnf_recursive3(_E,_EE,_H,_D).

% fd_dom
'$fd_dom'(_A, _B, true, _C, _D):-
        hnf(_A, _A1, _C, _G),
        fd_dom(_A1,_BB),
        hnf_recursive3(_B,_BB,_G,_D).

% range_to_fdset
'$range_to_fdset'(_A, _B, _C, _D):-
        hnf_recursive3(_A,_AA,_C,_F),
        range_to_fdset(_AA,_BB),
        hnf_recursive(_B, _BB, _F, _D).

% fdset_to_range
'$fdset_to_range'(_A, _B, _C, _D):-
        hnf_recursive(_A,_AA,_C,_F),
        fdset_to_range(_AA,_BB),
        hnf_recursive3(_B, _BB, _F, _D).

traduction(domm(X),dom(X)).
traduction(minn(X),min(X)).
traduction(maxx(X),max(X)).
traduction(vall(X),val(X)).
traduction(minmax(X),minmax(X)).

traduce_list([],[]).
traduce_list([X|Xs],[Y|Ys]):-traduction(X,Y),traduce_list(Xs,Ys).

traduce_args([],[]).
traduce_args([X:Xs|Rs],[Y|Ys]):-toyListToPrologList(X:Xs,Y),!,
                    traduce_args(Rs,Ys).
traduce_args([X|Rs],[X|Ys]):-traduce_args(Rs,Ys).
```

```
% fd_global
'$fd_global'(_A, _B, _C, _O, _D, _E):-
        hnf(_A, _A1, _D, _F),
        hnf(_B, _BB, _F, _I),       %%_B = st('$$tup'((Args)))
        _BB=..[st,Tuple],
        Tuple =..['$$tup',Ar],    %% Tuple = '$$tup'(Args)
        Ar =..[C|Args],          %% C = '.',
        traduce_args(Args, PrologArgs),
        _B1  =..[st|PrologArgs],
        hnf(_C, _CC, _I, _E),
        toyListToPrologList(_CC, CCC),
        traduce_list(CCC,C1),
        printf("\n Las variables son _A1 = %w \n  _B1 = %w \n  _C1 = %w
            \n",[_A1,_B1,C1]),
        fd_global(A1,_B1,C1).


%%%%%%%%%%% Predicados auxiliares   %%%%%%%%%%%%%%%%%%

toyListToPrologList([],[]).
toyListToPrologList(A:R,[A|RR]):-toyListToPrologList(R,RR).

%------      INPUT/OUTPUT              ------------------------------
%------------------- Predicates for I/O ----------------------------
%------------------- Predicates of formatting ----------------------
%------------------- printf: formatted ouput -----------------------

% control chars '\.'

printf([92,110|F],Ts) :- !, nl,      printf(F,Ts).  % 92,110 = \n
printf([92,116|F],Ts) :- !, put(9),  printf(F,Ts).  % 92,116 = \t
printf([92, 92|F],Ts) :- !, put(92), printf(F,Ts).  % 92, 92 = \\
printf([92, 37|F],Ts) :- !, put(37), printf(F,Ts).  % 92, 37 = \%
printf([92, 34|F],Ts) :- !, put(34), printf(F,Ts).  % 92, 34 = \"

% format chars '%.'

printf([37,119|F],[T|Ts]) :- !, write_term(T,[portrayed(true)]), printf(F,Ts).
printf([37,113|F],[T|Ts]) :- !, writeq(T,[portrayed(true)]), printf(F,Ts).
printf([37,100|F],[T|Ts]) :- !, display(T), printf(F,Ts).
printf([37,115|F],[T|Ts]) :- !, puts(T),    printf(F,Ts).
printf([37,118|F],[V,T|Ts]) :- !, nameTermVars(T,V,Tn), printf(F,[Tn|Ts]).

% plain chars '.'

printf([Ch|F],Ts) :- !, put(Ch), printf(F,Ts).

printf([],_).

printf(F) :- printf(F,[]).

puts([]).
puts([Ch|S]) :- put(Ch), puts(S).

print_list([]):-!.
print_list([(X=A)|L]):-ground(A),
                       printf("    %w = %w \n",[X,A]),
                       print_list(L),!,
                       assert(sol(true)).
print_list([(_X=_A)|L]):-print_list(L),!.

%%%%%%%%%%%%%%  END OF CFLPFDFILE  %%%%%%%%%%%%%%%%%%%%%%%%%
```

Bibliografía

[1] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] N. Beldiceanu. *Global constraints as graph properties on a structured network of elementary constraints of the same type*. In Dechter, R., editor, 6th International Conference on Principles and Practice of Constraint Programming (CP'97), number 1894 in LNCS, pp. 52–66, Singapore. Springer-Verlag. 2000.

[3] M. Carlsson and G. Ottosson and B. Carlson. An open-ended finite domain constraint solver, 9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97), U. Montanari and F. Rossi, Number 1292 in LNCS, Springer-Verlag, Southampton, UK, pp. 191-206, 1997.

[4] L. Damas and R. Milner. *Principal Type Schemes for Functional Programs*. Proc. ACM Symp. on Principles of Programming Languages (POPL'82) ACM Press, pp. 207–212, 1982.

[5] J. Darlington and Y.K. Guo and H. Pull. *A New Perspective on the Integration of Functional and Logic Languages*. International Conf. on Fifth Generation Computer Systems (FGCS'92), IOS Press, pp. 682–693, 1992.

[6] A.J. Fernández, M.T. Hortalá-González and F. Sáenz-Pérez. *Solving combinatorial problems with a constraint functional logic language*. In P. Wadler and V. Dahl editors. 5th International Symposium of Practical Aspects of Declarative Languages (PADL'2003), number 2562 in LNCS, Springer-Verlag, pp. 320–338, New Orleans, Louisiana, USA, 2003.

[7] A.J. Fernández, T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD) v1.4: System and User manual. Available at `http://www.lcc.uma.es/~afdez/cflpfd/`, 2003.

[8] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*, Cognitive Technologies Series, Springer, 2003.

[9] J.C. González-Moreno. *Programación lógica de orden superior con combinadores*. PhD thesisDIA-UCM, Madrid, 1994.

[10] J.C. González-Moreno and M.T. Hortalá-González and M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming*. In 14th International Conference on Logic Programming (ICLP'97), The MIT Press, pp. 153-167,1997.

[11] J.C. González-Moreno and M.T. Hortalá-González and F.J. López-Fraguas and M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. The Journal of Logic Programming 40, 1,pages 47-87,July 1999.

[12] J.C. González-Moreno and M.T. Hortalá-González and M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. In 4th International Symposium on Functional and Logic Programming (FLOPS'99), A Middeldorp and Taisuke Sato, Eds. Number 1722 in LNCS, Springer-Verlag, Tsukuba, Japan, pp. 1–20, 1999. Also published in a special issue of the Journal of Functional and Logic Programming, 2001.

[13] C.A. Gunter and D. Scott, *Semantic Domains*. Handbook of Theoretical Computer Science. Vol. B: Formal Models and Sematics, J. van Leeuwen editor, Elsevier and The MIT Press, pp. 633-674, 1990.

[14] M. Hanus. *The Integration of Functions into Logic Programming: A Survey*. The Journal of Logic Programming, Special issue: Ten Years of Logic Programming, vol. 19-20, pp. 583–628, 1994.

[15] R. Haralick and G. Elliott. *Increasing tree search efficiency for constraint satisfaction problems*. Artificial Inteligence, 14:263-313, 1980.

[16] M. Henz and T. Müller. *An overview of finite domain constraint programming*. 5th Conference of the Association of Asia-Pacific Operational Research Societies, Singapore, 2000.

[17] P. van Hentenryck. *Constraint Satisfaction in logic programming*. The MIT Press, Cambridge, MA. (1989).

[18] J. Jaffar and M. Maher *Constraint logic programming: a survey*. The Journal of Logic Programming, vol 19-20, pp. 503–581, 1994.

[19] R. Loogen, F.Lopez-Fraguas, and M.Rodríguez-Artalejo. *A demand driven computation strategy for lazy narrowing*. Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming, pp. 184–200. Springer LNCS 714, 1993.

[20] F. López-Fraguas. *A General Scheme for Constraint Functional Logic Programming*. 3rd International Conference on Algebraic and Logic Programming (ALP'92), H. Kirchner and G. Levi, Eds. number 632 in LNCS, Springer-Verlag, Volterra, Italy, pp. 213–227, 1992.

[21] F. López-Fraguas. and J. Sánchez-Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*. Proc. RTA´99, Springer LNCS 1631, pp. 244–247, 1999.

[22] F. López-Fraguas and M. Rodríguez-Artalejo and R. del Vado-Vírseda. *Constraint Functional Logic Programming Revisited*. 5th International Workshop on Rewriting Logic and its Applications (WRLA'2004), Elsevier ENTCS series, Barcelona, Spain, 2004.

[23] L. López-Fraguas and M. Rodríguez-Artalejo and R. del Vado-Vírseda. *A Lazy Narrowing Calculus for Declarative Constraint Programming*. 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04), ACM Press, Verona, Italy. 2004.

[24] K. Marriot and P. J. Stuckey. *Programming With Constraints. An Introduction*. The Mit Press; Cambridge, Massachusetts; London, England, 1998.

[25] J.C. Regin. *A filtering algorithm for constraints of difference in CSPs*. Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94), pp. 362–367, 1994.

[26] M. Rodríguez-Artalejo, *Functional and Constraint Logic Programming*, H. Comon and C. Marché and R. Trainen, Eds. Number 2002 in LNCS, Springer-Verlag, pp. 202-270, 2001.

[27] J. Sánchez-Hernández and F. López-Fraguas. *Un lenguaje lógico funcional con restricciones*. DSIP Complutense University of Madrid, Research Report, 1998.

[28] *SICStus Prolog User's Manual*. Swedish Institute of Computer Science. PO Box 1263. SE-164 29 Kista, Sweden Release 3.8.4. May 2000.

[29] E. Tsang. *Foundations of constraint satisfaction*. Academic Press, London and San Diego, 1993.

[30] N.F. Zhou. *Channel Routing with Constraint Logic Programming and Delay*. 9th International Conference on Industrial Applications of Artificial Intelligence, Gordon and Breach Science Publishers, pp. 217–231, 1996.