Integración de ILOG CP en TOY

UNIVERSIDAD COMPLUTENSE
MADRID

Proyecto Fin de Máster en Programación y Tecnología Software
Máster en Investigación en Informática.
Facultad de Informática, Universidad Complutense de Madrid
Curso 2008-2009

Autor:
Ignacio Castiñeiras Pérez

Director:
Dr. Francisco Javier López Fraguas

Colaboradores externos de dirección:
Dr. Fernando Sáenz Pérez
Dra. Teresa Hortalá González

Madrid, Septiembre 2009

Este trabajo ha sido financiado parcialmente por los proyectos
TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 y
UCM-BSCH-GR58/08-910502.

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Integración de ILOG CP en $\mathcal{TOY}$", realizado durante el curso académico 2008-2009 bajo la dirección del doctor Francisco Javier López Fraguas [y con la colaboración externa de dirección de los doctores Fernando Sáenz Pérez y Teresa Hortalá González] en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo: Ignacio Castiñeiras Pérez

Resumen

La programación con restricciones (del inglés Constraint Programming: CP) es una rama que lleva años estudiando problemas con una fuerte componente combinatoria. El esquema CP está basado en el planteamiento de restricciones, que pueden ser vistas como relaciones entre el dominio de ciertas variables. Cualquier esquema CP debe ser instanciado a un dominio $\mathcal{D}$ concreto, CP($\mathcal{D}$). Este dominio proporciona tipos de datos, un lenguaje de restricciones basadas en operaciones primitivas y un resolutor de restricciones. Los lenguajes declarativos son más apropiados para formular problemas CP que los lenguajes imperativos por su mayor nivel de abstracción. Los lenguajes funcionales y lógicos son los dos paradigmas declarativos más importantes, y su combinación ha sido un tema de investigación en las dos últimas décadas. En este trabajo se estudia el sistema lógico-funcional con restricciones (del inglés Constraint Functional Logic Programming: CFLP($\mathcal{D}$)) $\mathcal{TOY}$, que incluye los dominios de restricciones $\mathcal{FD}$ (del inglés Finite Domains: $\mathcal{FD}$), $\mathcal{R}$ (del inglés Reals: $\mathcal{R}$), $\mathcal{H}$ (del inglés Herbrand: $\mathcal{H}$) y un dominio $\mathcal{M}$ (del inglés Mediator: $\mathcal{M}$).

Actualmente el sistema $\mathcal{TOY}(\mathcal{FD})$ (al que denominamos en este trabajo $\mathcal{TOY}(\mathcal{FD}s)$) utiliza el sistema de restricciones $\mathcal{FD}$ externo ofrecido por SICStus Prolog en su biblioteca clpfd. En este trabajo se intercambia clpfd por ILOG CP 1.4 como nuevo sistema de restricciones $\mathcal{FD}$ externo del sistema $\mathcal{TOY}$, dando lugar al nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$. El primer capítulo es introductorio y estudia el sistema de restricciones $\mathcal{TOY}(\mathcal{FD})$ desde un punto de vista genérico. Se detalla la tecnología clpfd utilizada y se enumeran los inconvenientes de su utilización. A continuación se presenta la tecnología ILOG CP 1.4, incidiendo en las ventajas que ofrece y estudiando en detalle la arquitectura básica de sus aplicaciones. En el segundo capítulo se detalla la nueva arquitectura de componentes del sistema $\mathcal{TOY}$ que permite integrar la tecnología ILOG CP como sistema de restricciones $\mathcal{FD}$ externo. Se estudia el marco de comunicación entre SICStus y C++. Se detalla cómo adaptar este marco al sistema $\mathcal{TOY}$ y a ILOG CP. Se desarrolla una primera versión básica del sistema $\mathcal{TOY}(\mathcal{FD}i)$ que permite modelar y resolver con ILOG CP un pequeño repertorio de restricciones $\mathcal{FD}$ soportadas por el lenguaje $\mathcal{TOY}$. En el tercer capítulo se amplía esta versión del sistema $\mathcal{TOY}(\mathcal{FD}i)$, dotándolo de una mayor funcionalidad mediante backtracking, procedimientos de búsqueda, sincronización de las restricciones de igualdad y desigualdad gestionadas en el dominio de restricciones $\mathcal{H}$ y definiendo un nuevo propagador que permita sincronizar información desde ILOG a $\mathcal{TOY}$. El capítulo 4 muestra el rendimiento del sistema $\mathcal{TOY}(\mathcal{FD}i)$ sobre algunos ejemplos, comparando este rendimiento con el del sistema $\mathcal{TOY}(\mathcal{FD}s)$ y el de la tecnología ILOG CP actuando en solitario. El capítulo 5 presenta las conclusiones obtenidas al desarrollar el trabajo y el trabajo futuro que éste desencadena.

Los resultados de este trabajo han sido aceptados para su presentación en el 18^{th} International Workshop on Functional (Constraint) Logic Programming (WFLP'09), celebrado en Brasilia el 28 de Junio. El mismo artículo ha sido invitado para una segunda revisión que le permita aparecer en el volumen de Lecture Notes in Computer Science asociado al congreso. Asimismo, los resultados de este trabajo han sido aceptados para su presentación en las 9^{as} Jornadas sobre Programación y Lenguajes (PROLE'09), celebrado en San Sebastian del 9 al 11 de Septiembre.

Palabras clave

Programación con restricciones, Programación lógico-funcional, Sistemas de restricciones, $\mathcal{TOY}$, ILOG CP, SICStus Prolog, Implementación e integración de lenguajes.

Summary

Combinatorial problems have been studied in the Constraint Programming area since many years. The CP scheme is based on posing constraints, which are basically relations among domain variables. Any CP scheme must be instantiated by a parametrically given constraint domain $\mathcal{D}$. This domain provides specific data values, a constraint language based on primitive operations and a constraint solver. Due to its higher level of abstraction, declarative languages seem to be more suitable to formulate CP problems than imperatives languages. Functional languages and logic languages are the most paradigmatic declarative languages, and its combination has been studied in the last two decades. In this work, we study the Constraint Functional Logic Programming system CFLP($\mathcal{D}$) $\mathcal{TOY}$, which includes the constraint domains $\mathcal{FD}$, $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{M}$.

Currently, the system $\mathcal{TOY}(\mathcal{FD})$ (which we denote here as $\mathcal{TOY}(\mathcal{FD}s)$) uses the external $\mathcal{FD}$ constraint system offered by SICStus Prolog via its library clpfd. In this work, we replace clpfd by ILOG CP 1.4 as a new external $\mathcal{FD}$ constraint system for $\mathcal{TOY}$, given rise to the new system $\mathcal{TOY}(\mathcal{FD}i)$. Int chapter 1 we introduce our work and study the system $\mathcal{TOY}(\mathcal{FD})$ from a generic point of view. Also, we detail the clpfd technology used, enumerating the disadvantages that $\mathcal{TOY}$ suffer by using it. Next, we present the ILOG CP 1.4 technology, enumerating the advantages it offers, and studying in detail the basic architecture of its applications. In chapter 2 we show the new component architecture of the system $\mathcal{TOY}$, which uses ILOG CP as a new external $\mathcal{FD}$ constraint system. We study the communication framework that SICStus offers to communicate SICStus and C++. We detail how to adapt this framework to our particular context between $\mathcal{TOY}$ and ILOG CP. We develop a very basic first release of the system $\mathcal{TOY}(\mathcal{FD}i)$, which gives support to model and solve a little set of $\mathcal{FD}$ constraints of the language $\mathcal{TOY}$. In chapter 3 we enhance the first release of $\mathcal{TOY}(\mathcal{FD}i)$. We develop a second release of $\mathcal{TOY}(\mathcal{FD}i)$ which includes, backtracking, labeling, a synchronization of the equality and disequality constraints managed in the constraint system $\mathcal{H}$ and a new propagator which allows to synchronize information from ILOG to $\mathcal{TOY}$. In chapter 4 we show the $\mathcal{TOY}(\mathcal{FD}i)$ performance for some examples. We compare the $\mathcal{TOY}(\mathcal{FD}i)$ performance to $\mathcal{TOY}(\mathcal{FD}s)$ and ILOG CP. In chapter 5 we present some conclusions and future work.

The results of this work have been accepted for its presentation in the 18^{th} International Workshop on Functional (Constraint) Logic Programming (WFLP'09), held in Brasilia on June 28^{th}. This article has been invited for a second review to appear in the Lecture Notes in Computer Science publication associated to the conference. Also, the results of this work have been accepted for its presentation in the 9^{as} Jornadas sobre Programación y Lenguajes (PROLE'09), held in San Sebastian on September 9^{th}-11^{th}.

Keywords

Constraint Programming, Functional Logic Programming, Constraint Systems, $\mathcal{TOY}$, ILOG CP, SICStus Prolog, Implementation and integration of languages.

Agradecimientos

Quiero empezar dándole las gracias a Paco. Por enseñar una asignatura como Calculabilidad y Complejidad, por impartirla escribiendo en la pizarra y por hacernos pensar en cosas muy raras que me daban vueltas a la cabeza fuera del horario de clases. Quiero darle las gracias por ofrecerme un trabajo, por confiar en mí y darme la oportunidad de dedicarme a la investigación realizando el doctorado dentro del GPD. Por último, quiero darle las gracias por preocuparse (con distancia pero con constancia) de que las cosas me fueran bien.

Quiero darles las gracias a Fernando y a Teresa. Aquí me puede más lo personal que lo profesional, pero voy a empezar por lo último, que a fin de cuentas es la mitad que da sentido a la otra. Quiero agradecerles que me hayan enseñado la programación con restricciones. Con ello me han mostrado que muchas curiosidades que me asaltaban desde niño tienen una ciencia detrás, y que esa ciencia puede estudiarse y mejorarse. Quiero agradecerles que me hayan enseñado a desarrollar un proyecto, a entender las diferentes etapas que lo componen, y a solventarlas con paciencia, sensatez y trabajo. Quiero agradecerles su tranquilidad, su trato exquisito cada día, sus consejos, los valores que transmiten y su saber estar. Quiero remarcar que en todo momento, sobre todo cuando las cosas pintaban mal, he sentido en ellos un valor seguro, que me transmitía la confianza necesaria para superar las dificultades y alcanzar los objetivos propuestos. Me siento muy afortunado de haber podido trabajar con ellos. Sin su ayuda, probablemente este trabajo no sería ahora una realidad.

Gracias por último a Enrique, Juan, Adrián, Roberto, Javier, Carlos, Manu y Miky. Porque ponen su mejor cara en el trabajo (en lugar de la peor, como ocurre en muchos otros sitios). Por ser tan buena gente, generar buen rollo en el despacho y hacer que ir a trabajar sea mucho más divertido.

Índice general

Índice de figuras

Capítulo 1

Introducción y antecedentes

La evolución tecnológica, marcada en gran medida por las necesidades económicas, convierte la optimización en la asignación y utilización de recursos en un valor fundamental en el desarrollo de cualquier empresa u organización. Detrás de estas tareas se esconden problemas con una estructura combinatoria que muchas veces los convierte en NP-completos, por lo que no existen eficientes algoritmos generales para resolverlos. Como consecuencia, para abordar estos problemas se requiere tiempo y experiencia, tanto en la formalización del dominio de aplicación del problema como en el diseño del algoritmo que lo va a resolver. La programación con restricciones (del inglés Constraint Programming: CP) es una rama que lleva años estudiando este tipo de problemas.

El esquema CP está basado en el planteamiento de restricciones, que pueden ser vistas como relaciones entre el dominio de ciertas variables. Se debe instanciar este esquema a un cierto dominio $\mathcal{D}$ de restricciones, CP($\mathcal{D}$). Este dominio proporciona tipos de datos, un lenguaje de restricciones basadas en operaciones primitivas y un resolutor de restricciones. Diferentes dominios dan lugar a diferentes instancias del esquema CP:

- CP($\mathcal{H}$) proporciona restricciones de igualdad y desigualdad sintáctica sobre el universo de Herbrand ($\mathcal{H}$).
- CP($\mathcal{R}$) proporciona restricciones aritméticas sobre números reales ($\mathcal{R}$).
- CP($\mathcal{FD}$) proporciona restricciones de dominio finito (del inglés Finite Domain: $\mathcal{FD}$) sobre números enteros.

La programación con restricciones sobre dominios finitos, CP($\mathcal{FD}$) [26, 14], emergió como uno de los paradigmas de programación más pujantes en las últimas décadas. En la actualidad, existen numerosos instancias de sistemas CP($\mathcal{FD}$), entre las que destacamos ILOG CP [19], Gecode [11] y Minion [28].

Los lenguajes declarativos son más apropiados que los lenguajes imperativos para formular problemas sobre restricciones, debido a su capacidad para definir restricciones como relaciones entre objetos. La integración de la programación con restricciones CP en sistemas de programación lógica (del inglés Logic Programming: LP) da lugar a lenguajes de programación lógica con restricciones (del inglés Constraint Logic Programming: CLP). El esquema CLP($\mathcal{D}$) [20, 21, 22] proporciona un práctico y potente marco de trabajo para la programación con restricciones, donde ésta convive con el estilo declarativo y la semántica de la programación lógica. Estos sistemas CLP constituyen herramientas de programación de alto nivel, proporcionando una balanza razonable entre la formulación de un problema

y la eficiencia en su resolución. La alta flexibilidad con la que están implementados estos sistemas permite utilizarlos como herramientas de desarrollo para la implementación de aplicaciones y como plataformas donde se pueden investigar conceptos clave de la implementación de lenguajes de programación. Por todo ello, el diseño, implementación y optimización de sistemas CLP se ha convertido en uno de los temas de mayor interés dentro de las áreas de CP y LP. De hecho, la instancia CLP($\mathcal{FD}$) es quizá una de las instancias más exitosas de CP($\mathcal{FD}$). Entre las muchos sistemas que siguen el esquema CLP($\mathcal{FD}$) destacamos aquí SICStus Prolog [33], B-Prolog [2], Eclipse [7], GNU Prolog [12], SWI Prolog [34] y Ciao Prolog [5].

La combinación de la programación lógica con restricciones CLP y la programación funcional (del inglés Functional Programming) da lugar a lenguajes de programación lógico-funcional con restricciones (del inglés Constraint Functional Logic Programming: CFLP). Los lenguajes lógico-funcionales integran lenguajes lógicos y funcionales para heredar lo mejor de cada uno. De la programación lógica, los lenguajes lógico-funcionales heredan la unificación, la potencia de las variables lógicas, los mecanismos de búsqueda indeterminista y la posibilidad de trabajar con estructuras de datos parciales. De la programación funcional heredan la expresividad de las funciones, el empleo de tipos, el orden superior y un mecanismo de evaluación más eficiente (determinismo y evaluación perezosa). En [23] se propone un esquema para CFLP($\mathcal{D}$) que sirve de marco lógico y semántico para la programación lógico-funcional con restricciones. Además se propone una semántica operacional basada en un cálculo de estrechamiento perezoso. De estos resultados surge el sistema $\mathcal{TOY}$ [1, 24], sobre el que se desarrolla este trabajo.

$\mathcal{TOY}$ es un sistema implementado en SICStus Prolog. Incluye los dominios de restricciones $\mathcal{FD}$ $\mathcal{R}$, $\mathcal{H}$ y un dominio $\mathcal{M}$ para la comunicación entre ellos [8]. A la hora de resolver objetivos sobre un determinado programa $\mathcal{TOY}$ diferenciamos entre los sistemas de restricciones $\mathcal{FD}$, $\mathcal{R}$, $\mathcal{H}$ y un dominio mediador $\mathcal{M}$, que permite la cooperación de los anteriores. La Figura 1.1 muestra los distintos sistemas de restricciones de $\mathcal{TOY}$. Cada restricción es enviada a su correspondiente sistema de restricciones, donde es gestionada. Mientras que los sistemas de restricciones $\mathcal{FD}$ y $\mathcal{R}$ utilizan sistemas de resolución de restricciones proporcionados por un sistema de restricciones externo [10], los sistemas de restricciones $\mathcal{H}$ y $\mathcal{M}$ tienen una gestión explícita dentro del sistema [8]. La versión actual del sistema $\mathcal{TOY}$($\mathcal{FD}$) incluye como sistema de restricciones $\mathcal{FD}$ externo el proporcionado por SICStus Prolog. El propósito de este trabajo es integrar el sistema ILOG CP 1.4 como nuevo sistema de restricciones $\mathcal{FD}$ externo del sistema $\mathcal{TOY}$.

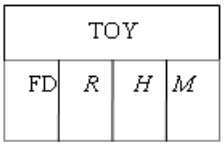

Figura 1.1: Sistemas de restricciones $\mathcal{TOY}$.

Por otro lado, las distintas instancias de CP($\mathcal{D}$) permiten formalizar gran cantidad de problemas cuya naturaleza está asociada a un único dominio de aplicación. Sin embargo, para formalizar problemas heterogéneos, (problemas cuyas restricciones involucran más de un dominio) la única solución aparente consiste en adaptar artificialmente esas restricciones a un dominio concreto. En contra de este criterio, en los últimos años se ha puesto mucho énfasis en la investigación de la combinación y cooperación entre los reso-

lutores de diferentes dominios. Mencionamos aquí [3, 4, 29, 30, 13, 25, 15, 31, 16] como una selección que ilustra las diferentes aproximaciones al problema. En [8] se describe una implementación del sistema $\mathcal{TOY}$ donde los distintos resolutores $\mathcal{FD}$, $\mathcal{R}$, $\mathcal{H}$ y $\mathcal{M}$ cooperan entre si durante la resolución de objetivos. En este caso los resolutores $\mathcal{FD}$ y $\mathcal{R}$ utilizan los resolutores de SICStus Prolog. Podemos ambientar la integración de ILOG CP en $\mathcal{TOY}(\mathcal{FD})$ como una primera pieza a la que se unirá la integración de ILOG CPLEX en $\mathcal{TOY}(\mathcal{R})$. Esto permitirá reproducir el esquema seguido en [8] utilizando los resolutores de restricciones proporcionados por ILOG.

Objetivos del trabajo

- Estudiar el sistema de restricciones $\mathcal{TOY}(\mathcal{FD})$ desde el punto de vista de la tecnología clpfd utilizada actualmente, enumerando los inconvenientes de su utilización.
- Presentar la tecnología ILOG CP 1.4, incidiendo en las ventajas que ofrece y estudiando en detalle la arquitectura básica de sus aplicaciones.
- Interconectar el sistema $\mathcal{TOY}$ con la tecnología ILOG CP para utilizarla como sistema de restricciones $\mathcal{FD}$ externo. Describir la nueva arquitectura de componentes resultante.
  - Estudiar el marco de comunicación entre SICStus y C++.
  - Detallar cómo adaptar este marco al sistema $\mathcal{TOY}$ y a ILOG CP.
- Desarrollar una primera versión básica del sistema $\mathcal{TOY}(\mathcal{FD}i)$ que modele y resuelva con ILOG CP un pequeño repertorio de restricciones $\mathcal{FD}$ soportadas por el lenguaje $\mathcal{TOY}$.
- Ampliar esta versión del sistema $\mathcal{TOY}(\mathcal{FD}i)$, dotándolo de una mayor funcionalidad.
- Mostrar el rendimiento del sistema $\mathcal{TOY}(\mathcal{FD}i)$ sobre algunos ejemplos. Comparar este rendimiento con el del sistema $\mathcal{TOY}(\mathcal{FD}s)$ y el de la tecnología ILOG CP actuando en solitario.
- Presentar las conclusiones obtenidas en el desarrollo del trabajo y el trabajo futuro que éste desencadena.

Los resultados de este trabajo han sido aceptados para su presentación en el 18^{th} International Workshop on Functional (Constraint) Logic Programming (WFLP'09), celebrado en Brasilia el 28 de Junio. El mismo artículo ha sido invitado para una segunda revisión que le permita aparecer en el volumen de Lecture Notes in Computer Science asociado al congreso. Asimismo, los resultados de este trabajo han sido aceptados para su presentación en las 9^{as} Jornadas sobre Programación y Lenguajes (PROLE'09), celebrado en San Sebastian del 9 al 11 de Septiembre.

Organización del trabajo

La distribución del trabajo es la siguiente. En el resto de este capítulo se estudia el estado actual del sistema $\mathcal{TOY}(\mathcal{FD})$ y la tecnología ILOG CP 1.4 que se va a integrar. El segundo capítulo describe la interconexión entre el sistema $\mathcal{TOY}$ e ILOG CP 1.4 e

implementa una primera versión básica del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$. En el tercer capítulo se presenta una ampliación del sistema $\mathcal{TOY}(\mathcal{FD}i)$, dotándolo de una mayor funcionalidad.

1.1. El sistema $\mathcal{TOY}(\mathcal{FD})$

Estudiamos en detalle el sistema de restricciones de dominios finitos del sistema $\mathcal{TOY}$. Una primera subsección que estudia el sistema $\mathcal{TOY}(FD)$ desde un punto de vista genérico. Una segunda subsección que estudia el sistema $\mathcal{TOY}(\mathcal{FD}s)$, incluido en la actual distribución del sistema $\mathcal{TOY}$.

1.1.1. $\mathcal{TOY}(FD)$ genérico

Un objetivo $\mathcal{TOY}$ puede encontrar, entre las restricciones a evaluar, restricciones $\mathcal{FD}$. Cada nueva restricción $\mathcal{FD}$ encontrada se añade a la conjunción de restricciones $\mathcal{FD}$ evaluada hasta el momento. Esta conjunción de restricciones $\mathcal{FD}$ es modelada y resuelta por un sistema de restricciones $\mathcal{FD}$ externo. Éste esta compuesto básicamente por un almacén de restricciones y un resolutor de restricciones. La conjunción de restricciones $\mathcal{FD}$ que es almacenada y resuelta está formada por la propia conjunción de restricciones $\mathcal{FD}$ y el conjunto de variables lógicas $\mathcal{FD}$ que aparecen en estas restricciones $\mathcal{FD}$. Por ejemplo, el objetivo `X #> Y, Z == T, X #> 4` contiene la conjunción de restricciones $\mathcal{FD}$ formada por las restricciones `X #> Y` y `X #> 4`, y las variables lógicas $\mathcal{FD}$ `X` e `Y`.

La resolución de una conjunción de restricciones $\mathcal{FD}$ utilizando un sistema de restricciones externo se compone de tres fases:

i) Se trasmite desde $\mathcal{TOY}$ hacia el sistema de restricciones externo la conjunción de restricciones $\mathcal{FD}$ planteada.

ii) El sistema de restricciones externo resuelve la conjunción de restricciones $\mathcal{FD}$.

iii) Se accede desde $\mathcal{TOY}$ hacia el sistema de restricciones externo para obtener la solución computada.

La transmisión de la conjunción de restricciones $\mathcal{FD}$ desde $\mathcal{TOY}$ hacia un sistema de restricciones externo exige un modelado de dicha conjunción de restricciones $\mathcal{FD}$ dentro de dicho sistema de restricciones. El modelado consiste en construir una nueva conjunción de restricciones $\mathcal{FD}$', equivalente a la conjunción de restricciones $\mathcal{FD}$, pero formalizada en el lenguaje utilizado por el sistema de restricciones. En la práctica esto supone traducir cada restricción $\mathcal{FD}$ a una nueva restricción $\mathcal{FD}$' y cada variable lógica $\mathcal{FD}$ a una nueva variable lógica $\mathcal{FD}$'. La Figura 1.2 muestra gráficamente el proceso.

Una vez modelada la conjunción de restricciones $\mathcal{FD}$', el sistema de restricciones aplica sus técnicas de resolución, cuyos resultados pueden verse en la Figura 1.3. En concreto el resultado de la resolución puede:

- Determinar la insatisfactibilidad de la conjunción de restricciones $\mathcal{FD}$', o bien
- Obtener una forma resuelta de la conjunción de restricciones $\mathcal{FD}$', esto es, el conjunto de valores que las variables lógicas $\mathcal{FD}$' deben adoptar para satisfacer la conjunción de restricciones $\mathcal{FD}$'.

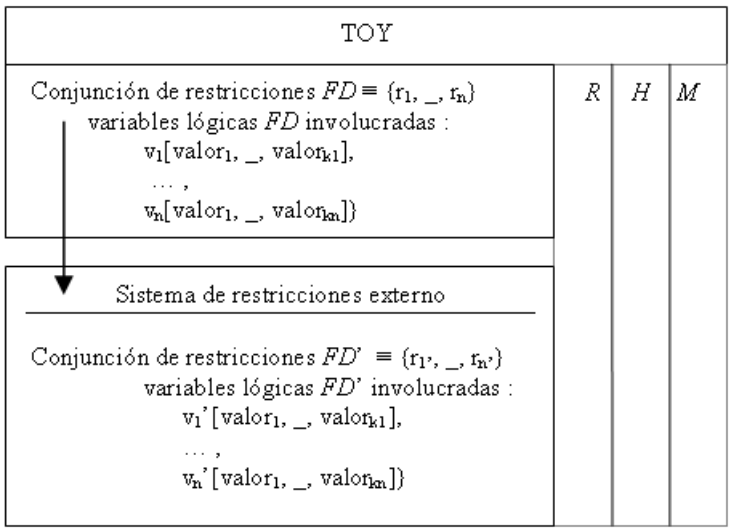

Figura 1.2: Traducción de la conjunción de restricciones $\mathcal{FD}$ al sistema de restricciones.

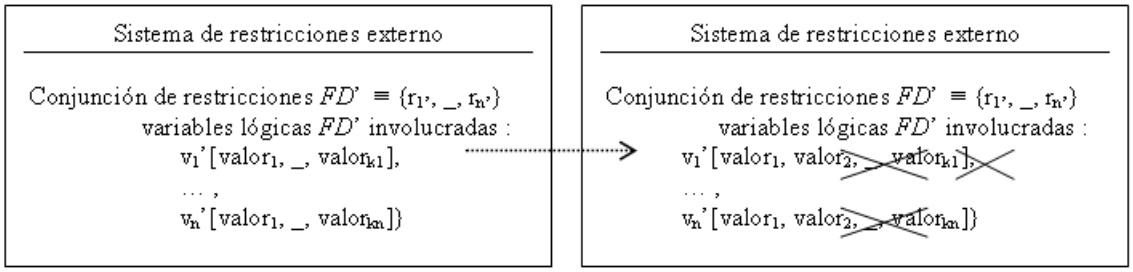

Figura 1.3: Resolución de la conjunción de restricciones $\mathcal{FD}$'.

Para que $\mathcal{TOY}$ utilice la solución obtenida por el sistema de restricciones externo se necesita acceder a la conjunción de restricciones $\mathcal{FD}$' resuelta. A partir de ésta se debe reconstruir la conjunción de restricciones $\mathcal{FD}$ planteada en el objetivo $\mathcal{TOY}$. La Figura 1.4 muestra este proceso.

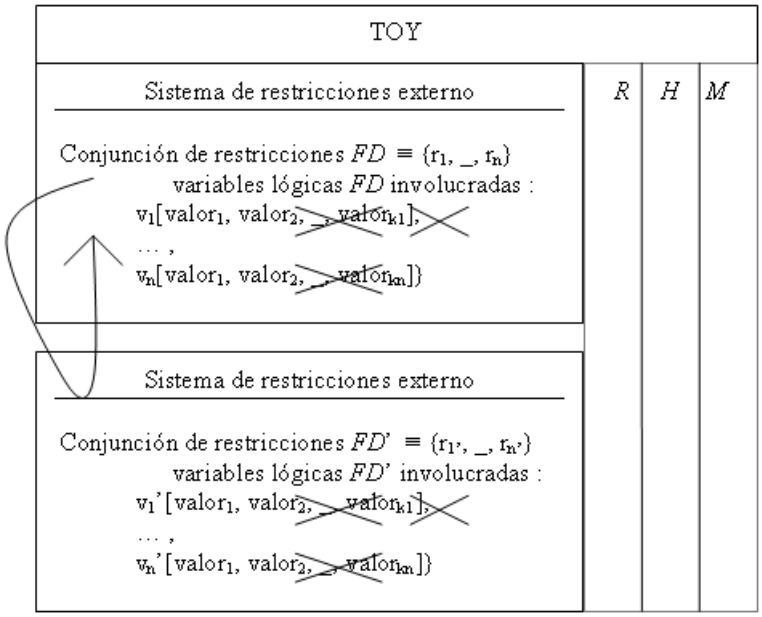

Figura 1.4: Acceso a la solución encontrada.

La Figura 1.5 muestra la arquitectura de componentes de un sistema $\mathcal{TOY}(FD)$ genérico. Esta arquitectura es independiente del sistema de restricciones externo utilizado. Supongamos que implementamos el sistema $\mathcal{TOY}(\mathcal{FD}k)$, que comunica al sistema $\mathcal{TOY}$ con un cierto sistema de restricciones externo `k`. Basándose en esta conexión, nuestro sistema $\mathcal{TOY}(\mathcal{FD}k)$ implementa un total de `n` restricciones $\mathcal{FD}$ `r1,...,rn`. Estas restricciones $\mathcal{FD}$ pueden ser utilizadas en cualquier programa $\mathcal{TOY}(\mathcal{FD}k)$, ya que son modeladas y resueltas en el sistema de restricciones `k`.

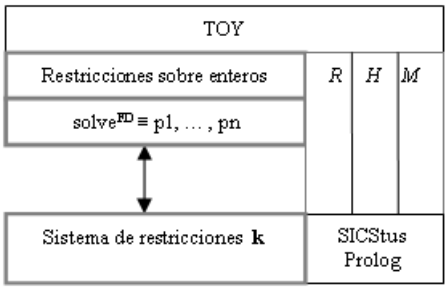

Figura 1.5: Arquitectura de componentes genérica $\mathcal{TOY}(FD)$.

Definimos el componente $solve^{FD}$, que contiene `n` predicados Prolog `p1,...,pn`. Cada predicado Prolog `pi` se encarga de gestionar una restricción $\mathcal{FD}$ de tipo `ri`, realizando las tres fases descritas anteriormente:

- Suponemos que `ri` es una restricción $\mathcal{FD}$ n-aria, y que `v1,...,vn` son sus n variables

lógicas $\mathcal{FD}$ involucradas.

Se traduce la restricción $\mathcal{FD}$ `ri(v1,...,vn)` a la restricción $\mathcal{FD}$' `ri'(v1',...,vn')` equivalente. Esta nueva restricción $\mathcal{FD}$' es añadida al sistema de restricciones `k`. Ahora `k` modela la conjunción de restricciones $\mathcal{FD}$' equivalente a la conjunción de restricciones $\mathcal{FD}$ evaluada por el objetivo $\mathcal{TOY}$.

- El sistema de restricciones `k` resuelve la nueva conjunción de restricciones $\mathcal{FD}$' modelada, obteniendo una nueva forma resuelta.
- Se accede a `k` para comprobar la forma resuelta de la conjunción de restricciones $\mathcal{FD}$', que es transmitida a $\mathcal{TOY}$. En caso de encontrar insatisfactibilidad de la conjunción de restricciones $\mathcal{FD}$', $\mathcal{TOY}$ devuelve fallo. En caso de que la conjunción de restricciones $\mathcal{FD}$' sea satisfactible, $\mathcal{TOY}$ continúa con la evaluación del objetivo.

Durante el proceso de resolución de un objetivo $\mathcal{TOY}$, el cálculo de estrechamiento perezoso detecta cada restricción primitiva $\mathcal{FD}$ `ri` que aparece entre las restricciones a evaluar. Esta nueva restricción `ri` se añade a la conjunción de restricciones $\mathcal{FD}$ evaluada hasta el momento por el objetivo $\mathcal{TOY}$. Para mantener la equivalencia entre las conjunciones de restricciones $\mathcal{FD}$ y $\mathcal{FD}$' se debe gestionar la nueva restricción $\mathcal{FD}$ `ri`. El cálculo de estrechamiento transfiere entonces el control de programa al predicado `pi`, que se encarga de gestionar `ri`.

1.1.2. $\mathcal{TOY}(FD)$ con `clpfd`: $\mathcal{TOY}(\mathcal{FD}s)$

La biblioteca clpfd, contenida en la distribución de SICStus Prolog, suministra un almacén de restricciones y un resolutor $\mathcal{FD}$ para el modelado y resolución de conjunciones de restricciones $\mathcal{FD}$. La distribución actual del sistema $\mathcal{TOY}$ utiliza esta tecnología como sistema de restricciones externo $\mathcal{FD}$. Denominamos a esta versión del sistema $\mathcal{TOY}$ con la nomenclatura $\mathcal{TOY}(\mathcal{FD}s)$. La Figura 1.6 muestra la arquitectura de componentes de $\mathcal{TOY}(\mathcal{FD}s)$, que conecta al sistema $\mathcal{TOY}$ con el almacén y resolutor de clpfd.

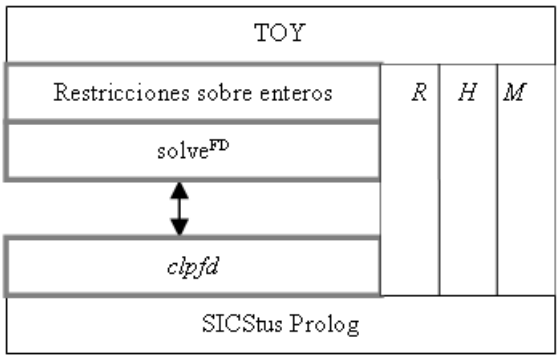

Figura 1.6: Arquitectura genérica $\mathcal{TOY}(FD)$.

Haciendo uso de clpfd, el sistema $\mathcal{TOY}(\mathcal{FD}s)$ desarrolla un amplio repertorio de restricciones $\mathcal{FD}$ comparable al de otros sistemas de programación lógica con restricciones. En [1] se estudia en detalle este repertorio de restricciones $\mathcal{FD}$. El componente $solve^{FD}$ de $\mathcal{TOY}(\mathcal{FD}s)$ contiene un predicado Prolog `pi` para gestionar cada uno de estos tipos de restricciones $\mathcal{FD}$. La comunicación entre el sistema $\mathcal{TOY}$ y el sistema cflpfd se produce

de manera natural, al estar ambos implementados en SICStus Prolog. En $\mathcal{TOY}(\mathcal{FD}s)$ las conjunciones de restricciones $\mathcal{FD}$ y $\mathcal{FD}$' son idénticas, y por lo tanto equivalentes.

Estudiamos a continuación la solución por intervalos encontrada por el sistema $\mathcal{TOY}(\mathcal{FD}s)$ para el siguiente objetivo $\mathcal{TOY}$:

```
TOY(FDs)> domain [X] 5 12, domain [Y] 2 17, X #+ Y == 17, X #- Y == 5
    yes
    { 5 # + Y #= X,
      X # + Y #= 17,
      X in 10..12,
      Y in 5..7 }
    Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
    no
    Elapsed time: 0 ms.
```

Esta solución por intervalos incluye:

- Toda restricción $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$ que no resulte ser trivial. Una restricción $\mathcal{FD}$ es trivial cuando todas las variables lógicas $\mathcal{FD}$ que esta restricción $\mathcal{FD}$ involucra han sido acotadas a valores enteros.

  En nuestro caso, como ni `X` ni `Y` han resultado acotadas a ningún valor, las dos restricciones $\mathcal{FD}$ `X #+ Y == 17` y `X #- Y == 5` deben mostrarse al usuario.

- Toda variable lógica $\mathcal{FD}$ involucrada en la conjunción de restricciones $\mathcal{FD}$. Cada variable debe mostrar los valores de su dominio que no han sido podados por el resolutor de clpfd.

  En nuestro caso, se muestran los dominios restantes de las variables lógicas $\mathcal{FD}$ `X` e `Y`. Podemos ver como el resolutor ha podado el dominio de `X` de 5..12 a 10..12. Además ha podado el dominio de `Y` de 2..17 a 5..7.

El resolutor de clpfd obtiene la solución por intervalos de la conjunción de restricciones $\mathcal{FD}$ aplicando propagación de restricciones. Además, incluye la posibilidad de realizar un etiquetado sobre algunas de las variables lógicas $\mathcal{FD}$ involucradas en esta conjunción de restricciones $\mathcal{FD}$. Este etiquetado incluye propagación de restricciones durante la búsqueda, para alcanzar una mayor eficiencia. Explicaremos en detalle todas estas técnicas en la subsección 1.2,1, donde resolvemos esta misma conjunción de restricciones $\mathcal{FD}$ utilizando el resolutor de la tecnología ILOG CP 1.4.

Posibles mejoras del sistema de restricciones clpfd

La implementación del sistema $\mathcal{TOY}(\mathcal{FD}s)$ es competitiva con el rendimiento de otros sistemas lógicos de resolución de restricciones, como puede verse en 'cite_revista'. Sin embargo, clpfd presenta algunos inconvenientes que nos hacen pensar en la necesidad de utilizar otra tecnología como sistema de restricciones externo:

- Trabajos recientes [6] han demostrado limitaciones en su capacidad de cómputo para hacer frente a problemas complejos.

- No es posible interactuar con el resolutor de restricciones *durante* los procesos predefinidos de búsqueda con objeto de podar el árbol de búsqueda más eficazmente con información específica de la conjunción de restricciones a resolver.

- La respuesta generada por el propio resolutor no incluye ninguna noción que facilite la depuración. De este modo, si la conjunción de restricciones tomada como entrada es insatisfactible, el resolutor no muestra el subconjunto de restricciones que no ha podido ser satisfecho.

1.2. ILOG CP 1.4

ILOG CP 1.4 [17] ofrece una de las tecnologías industriales más competitivas del mercado. Su resolutor trabaja como caja transparente, permitiendo la interacción durante el proceso de resolución. Incluye técnicas de depuración que ayudan al usuario a descubrir los fragmentos insatisfactibles del modelo de restricciones. Además, permite usar diferentes resolutores para el mismo dominio de aplicación. Por todo esto parece una buena alternativa a clpfd. En este trabajo integramos ILOG CP 1.4 como nuevo sistema de restricciones $\mathcal{FD}$ externo de $\mathcal{TOY}$.

ILOG CP 1.4 está compuesto por las bibliotecas ILOG Concert 2.6, ILOG Solver 6.6, ILOG Scheduler 6.6 e ILOG Distpacher 4.6. Mediante el uso de estas bibliotecas se pueden modelar y resolver conjunciones de restricciones $\mathcal{FD}$. La naturaleza de estas bibliotecas es declarativa, por lo que el usuario tan sólo debe especificar la conjunción de restricciones $\mathcal{FD}$ sobre la que desea trabajar, sin especificar en modo alguno la forma en que esta conjunción de restricciones $\mathcal{FD}$ debe ser resuelta. Las bibliotecas ofrecen un API C++, por lo que cualquier aplicación ILOG CP debe estar implementada en este lenguaje imperativo.

La estructura de las aplicaciones C++ ILOG CP se basa en un aislamiento entre los objetos que modelan la conjunción de restricciones $\mathcal{FD}$ planteada por el usuario y los objetos encargados de la posterior resolución de esta conjunción de restricciones $\mathcal{FD}$.

Siguiendo esta filosofía, en primer lugar se modela la conjunción de restricciones $\mathcal{FD}$ mediante un conjunto de objetos proporcionados por la biblioteca ILOG Concert 2.6. Esta biblioteca proporciona un lenguaje de modelado genérico. Gracias a la utilización de este lenguaje genérico, una misma conjunción de restricciones $\mathcal{FD}$ modelada puede ser traducida a distintos resolutores de restricciones.

Un resolutor de restricciones traduce una conjunción de restricciones $\mathcal{FD}$ modelada con ILOG Concert 2.6 a una nueva conjunción de restricciones $\mathcal{FD}$' equivalente, pero con una estructura interna dirigida al resolutor. Para ello se traducen uno a uno los objetos que modelan la conjunción de restricciones $\mathcal{FD}$. ILOG CP 1.4 proporciona los siguientes tres bibliotecas para la resolución de conjunciones de restricciones $\mathcal{FD}$:

- ILOG Solver 6.6, para la resolución de cualquier tipo de conjunción de restricciones $\mathcal{FD}$.
- ILOG Scheduler 6.6, para la resolución específica de conjunciones de restricciones $\mathcal{FD}$ referentes a la planificación de calendarios.
- ILOG Distpacher 4.6, para la resolución específica de conjunciones de restricciones $\mathcal{FD}$ referentes a la planificación de rutas.

La herramienta paradigmática representante de esta filosofía es ILOG OPL Studio [18].

En este trabajo nos centramos en el desarrollo de aplicaciones ILOG CP que utilizan exclusivamente las bibliotecas ILOG Concert 2.6 e ILOG Solver 6.6. La estructura de la sección es la siguiente:

- En la primera subsección se describen los principales objetos de ILOG Concert 2.6 e ILOG Solver 6.6 utilizados en el desarrollo de una aplicación ILOG CP. Además, se describe el proceso de resolución seguido por el resolutor de restricciones de ILOG Solver 6.6.
- En la segunda subsección se describen los comandos de compilación y vinculación necesarios para construir un ejecutable o una biblioteca dinámica a partir de los archivos fuente de una aplicación ILOG CP.

  La filosofía que seguiremos en este trabajo será utilizar en todo momento el comando más sencillo que alcance la funcionalidad deseada. Explicaremos en detalle cada comando utilizado.

1.2.1. Una primera aplicación ILOG CP: `intro.cpp`

La aplicación ILOG CP `intro` es suministrada en la distribución de ILOG CP como ejemplo introductorio. Está compuesta por un único archivo `intro.cpp`. Éste modela y resuelve la siguiente conjunción de restricciones $\mathcal{FD}$ utilizando la tecnología ILOG CP: `domain [X] 5 12, domain [Y] 2 17, X #+ Y == 17, X #- Y == 5, labeling [] [X,Y].` Hemos escrito la conjunción de restricciones $\mathcal{FD}$ en el lenguaje $\mathcal{TOY}$. Lo hacemos así por compatibilidad con el resto del trabajo.

Esta aplicación ILOG CP `intro` es representativa ya que contiene los principales objetos de ILOG Concert 2.6 e ILOG Solver 6.6 que utilizamos a lo largo de este trabajo. La Figura 1.7 muestra estos objetos tras las etapas de modelado y resolución de la conjunción de restricciones $\mathcal{FD}$.

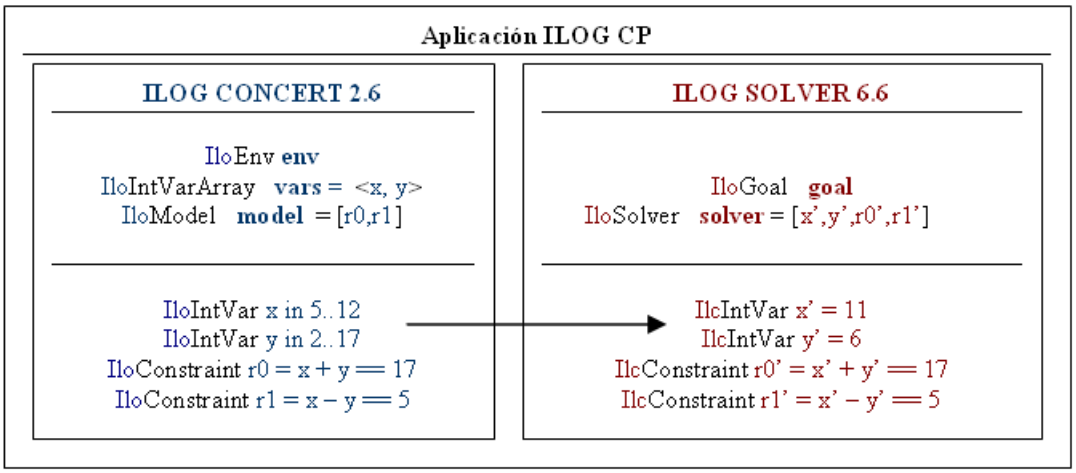

Figura 1.7: Objetos que componen una aplicación ILOG CP.

Estudiamos el contenido de `intro.cpp`:

```
        / intro.cpp /
-------------------------------------------------------------------
(0) #include <ilconcert/ilomodel.h>
    #include <ilsolver/ilosolverint.h>
(1) ILOSTLBEGIN
```

```
void main(){
(2)  IloEnv env;
(3)  IloModel model(env);
(4)  IloIntVarArray vars(env);
(5)  IloIntVar x(env, 5, 12);
(6)  vars.add(x);
     IloIntVar y(env, 2, 17);
     vars.add(y);
(7)  IloConstraint r0 = vars[0] + vars[1] == 17;
(8)  model.add(r0);
     IloConstraint r1 = vars[0] - vars[1] == 5;
     model.add(r1);

(9)  IloSolver solver(env);
(10)  solver.extract(model);
(11) if (solver.propagate()){
(12)    IloGoal goal = IloGenerate(env,vars,IloChooseFirstUnboundInt);
(13)    if (solver.solve(goal)){
(14)       solver.out() << "x = " << solver.getValue(x) << endl;
           solver.out() << "y = " << solver.getValue(y) << endl;
        }
     }
(15) env.end();
}
-----------------------------------------------------------------
```

En (0) importamos las cabeceras de ILOG Concert 2.6 e ILOG Solver 6.6 que vamos a necesitar para diseñar `intro`. En (1) se utiliza la macro ILOSTLBEGIN para utilizar la Standard Template Library de C++.

La aplicación contiene una única función `main` que se encarga de modelar y resolver la conjunción de restricciones $\mathcal{FD}$.

Esta función comienza creando un contexto o entorno de trabajo `IloEnv` (2). Este objeto gestiona la memoria de todos los objetos que vayan a ser creados dentro de la aplicación, tanto los pertenecientes a la fase de modelado como de resolución. Cuando la aplicación finaliza, el método `IloEnv::end()` (15) libera la memoria ocupada por todos los objetos de la aplicación.

A continuación la función `main` inicia la fase de modelado de la conjunción de restricciones $\mathcal{FD}$. Esto implica la creación en (3) del objeto `model`. Este *modelo* `model` representa un contenedor que almacena el conjunto de objetos que modelan la conjunción de restricciones.

Utilizamos un objeto `IloConstraint` para modelar cada una de las restricciones $\mathcal{FD}$ que componen la conjunción de restricciones. Utilizamos un objeto `IloIntVar` para modelar cada una de las variables lógicas $\mathcal{FD}$ involucradas en alguna restricción $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$.

En `model` sólo es necesario almacenar las `IloConstraint`s que modelan la conjunción. Las `IloIntVar`s no es necesario almacenarlas, ya que el propio `model` accede a ellas a través de las `IloConstraint`s en que estas `IloIntVar`s están involucradas. Al contener tan sólo las `IloConstraint`s, decimos que `model` es el objeto donde se hace explícita la conjunción de restricciones $\mathcal{FD}$ modelada.

Para poder acceder desde un único objeto a todas las variables lógicas $\mathcal{FD}$ de la conjunción de restricciones modelada creamos en (4) el vector de `IloIntVars` `vars`.

A continuación empezamos a modelar una a una las restricciones $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$. Por ello modelamos la restricción $\mathcal{FD}$ `X #+ Y == 17`. Para modelar una nueva restricción $\mathcal{FD}$ es necesario que todas sus variables lógicas $\mathcal{FD}$ involucradas hayan sido previamente modeladas. Por ello creamos en (5) una nueva `IloIntVar` que modela a `X`. El dominio de esta variable `5..12` viene determinado por su cota inferior y superior, suministrados como argumentos. En (6) almacenamos esta `IloIntVar` en el vector `vars`, para poder acceder a ella posteriormente. Toda nueva `IloIntVar` es añadida en la última posición de `vars`. En este caso, al estar `vars` vacío, la posición que ocupa la `IloIntVar x` es `vars[0]`. Repetimos el proceso con la `IloIntVar y`, que almacenamos en `vars[1]`.

Ahora que `X` e `Y` ya están modeladas (como `x` e `y`, respectivamente), podemos modelar la restricción $\mathcal{FD}$ `X #+ Y == 17`. Para ello creamos en (7) un nuevo objeto `IloConstraint r0 = vars[0] + vars[1] == 17`. En su definición vemos como accede a `vars` para utilizar sus `IloIntVars` involucradas. En (8) añadimos `r0` al contenedor `model`. Repetimos el proceso con la restricción $\mathcal{FD}$ `X #- Y == 5`.

Tras modelar esta última restricción hemos concluído el modelado de la conjunción de restricciones $\mathcal{FD}$. El estado de los objetos de la aplicación puede verse en la Figura 1.8.

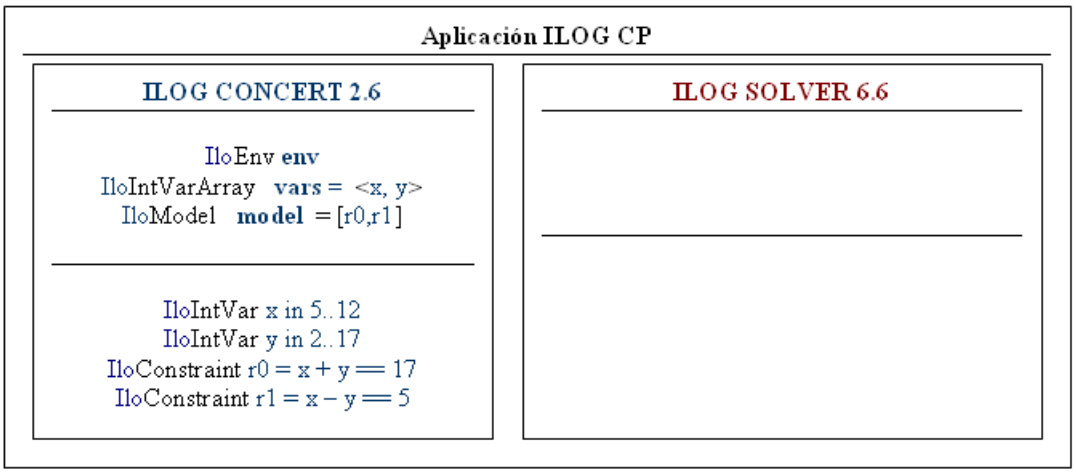

Figura 1.8: Estado de la aplicación tras el modelado.

Comenzamos ahora la fase de resolución. Para ello creamos en (9) el objeto resolutor de restricciones `solver`. Este objeto es una instancia de la clase definida en la biblioteca ILOG Solver 6.6.

En (10) traducimos la conjunción de restricciones $\mathcal{FD}$ modelada en `model` a una nueva conjunción de restricciones $\mathcal{FD}$' sobre la que trabajará nuestro resolutor `solver`. El método `IloSolver::extract(IloModel m)` itera en orden las `IloContraints` contenidas en `model`. Para cada `IloConstraint ri(v1,...,vn)` accede a las `v1,...,vn` involucradas, contenidas en `vars`. Para cada `IloIntVar vi` que no haya sido traducida previamente `solver` crea un nuevo objeto `IlcIntVar vi'` equivalente. Este nuevo objeto `vi'` está orientado a las técnicas de resolución que `solver` utiliza para resolver la conjunción de restricciones $\mathcal{FD}$' sobre la que trabaja. Tras traducir las `IloIntVars` involucradas, `solver` traduce la `IloConstraint ri`, creando un nuevo objeto `IlcConstraint ri'` equivalente.

El resultado de esta traducción puede verse en la Figura 1.9. El resolutor `solver` contiene la conjunción de restricciones $\mathcal{FD}$' equivalente a la conjunción de restricciones $\mathcal{FD}$ modelada.

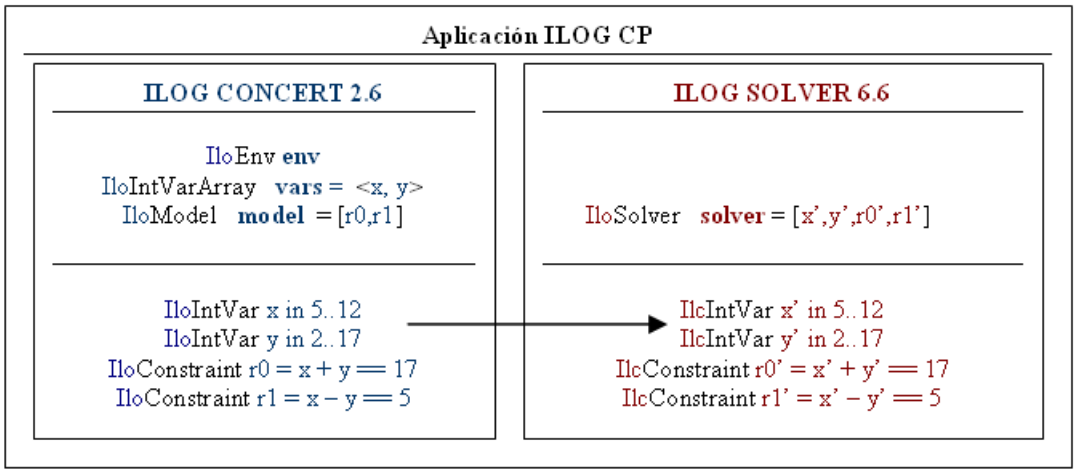

Figura 1.9: Estado de la aplicación tras la traducción al resolutor.

A continuación `solver` efectúa sus estrategias para la búsqueda de soluciones. Estas técnicas son:

- Propagación de restricciones.

  La propagación de restricciones aplica límite consistencia sobre los dominios de las `IlcIntVar`s, eliminando aquellos valores que no respetan alguna de las `IlcConstraint`s de la conjunción de restricciones $\mathcal{FD}$' sobre la que trabaja `solver`.

  Tras aplicar la propagación de restricciones obtenemos una solución por intervalos para la conjunción de restricciones $\mathcal{FD}$'. Esta solución por intervalos está formada por:

  - Las `IlcConstraint`s que componen la conjunción de restricciones $\mathcal{FD}$'.
  - Las `IlcIntVar`s, con sus dominios restantes tras aplicar la propagación de restricciones.

- Procedimiento de búsqueda.

  El procedimiento de búsqueda realiza un etiquetado sobre un subconjunto de las `IlcIntVar`s contenidas en `solver`. Este etiquetado impone que en cualquier solución obtenida las `IlcIntVar`s etiquetadas contengan un único valor en su dominio. Esto puede provocar que una conjunción de restricciones tenga una única solución por intervalos, pero varias soluciones por etiquetado.

  Para llevar a cabo el procedimiento de búsqueda se genera un árbol de búsqueda. Éste contiene tantas hojas como combinaciones posibles de valores pueden efectuarse sobre las `IlcIntVar`s a etiquetar. El procedimiento de búsqueda explora el árbol en el orden marcado por una estrategia pasada como parámetro. Si al explorar una rama del árbol se detecta insatisfactibilidad con respecto a la conjunción de restricciones, entonces se considera la rama como fallida y se pasa a explorar la siguiente rama

del árbol. Cuando se alcanza una hoja del árbol se ha encontrado una solución por etiquetado. El procedimiento de búsqueda puede continuar, para buscar nuevas soluciones en las ramas del árbol aún no exploradas.

Estudiamos más en detalle estas técnicas.

Propagación de restricciones

La límite consistencia se aplica a una `IlcConstraint ri'` con n `IlcIntVar`s `v1',...,vn'` involucradas. Cada `IlcIntVar vj'` tiene una cota inferior y una cota superior en su dominio. Para cada `vj'` la límite consistencia comprueba si su cota inferior (cota superior) puede satisfacer la `IlcConstraint ri'`. Es decir, si existe una combinación de valores `v1' -> val1, ..., vj' -> cota inferior (cota superior), ..., vn' -> valn` que satisface `ri'`.

- Si encuentra esa combinación, entonces la cota inferior (cota superior) de `vj` está justificada.
- Si no encuentra esa combinación de valores, entonces la cota inferior (cota superior) de `vj` no puede satisfacer `ri'`. Se poda la cota inferior (cota superior) y se aplica límite consistencia sobre la nueva cota inferior (cota superior) de `vj`.

Denominamos 'propagación completa' a aplicar la técnica de límite consistencia a todas las `IlcConstraint`s contenidas en `solver`. El método `solver.propagate()` realiza tantas 'propagaciones completas' como sean necesarias hasta encontrar uno de estos dos casos:

- Se han podado todos los valores del dominio de alguna de las `IlcIntVar`s contenidas en `solver`. En este caso se concluye que la conjunción de restricciones $\mathcal{FD}$' no es satisfactible, y por tanto no tiene solución. El método `solver.propagate()` devuelve el valor 0.
- Todas las `IlcIntVar`s tienen al menos un valor en su dominio. Además, la última 'propagación completa' no ha podado el dominio de ninguna de las `IlcIntVar`s contenidas en `solver`. En este caso se ha obtenido una solución por intervalos a la conjunción de restricciones $\mathcal{FD}$. El método `solver.propagate()` devuelve el valor 1.

En (11) `int solver.propagate()` efectúa la propagación de `r0' = x' + y' == 17; r1' = x' - y' == 5`. Esto provoca la poda de los dominios de `x' = 5..12; y' = 2..17`.
Veamos el resultado de la primera 'propagación completa':

i) `r0'` impone `x' + y' == 17`.

   a) Límite consistencia sobre `x'`.
      - Cota inferior 5. La combinación `x' = 5, y' = 12` satisface `r0'`.
      - Cota superior 12. La combinación `x' = 12, y' = 5` satisface `r0'`.

      El dominio de `x'` continúa como 5..12.

   b) Límite consistencia sobre `y'`.

- Cota inferior 2. No existe una combinación `x' in 5..12, 2` que satisfazca `r0'`. Se elimina el valor 2 del dominio de `y'`. La nueva cota inferior es 3. La límite consistencia continúa podando también los valores 3 y 4. La nueva cota inferior de `y'` será 5, para la que la combinación `x' = 12, y' = 5` satisface `r0'`.
- Cota superior 17. No existe una combinación `x' in 5..12, 17` que satisfazca `r0'`. Se elimina el valor 17 del dominio de `y'`. La nueva cota superior es 16. La límite consistencia continúa podando también los valores 16, 15, 14 y 13. La nueva cota inferior de `y'` será 12, para la que la combinación `x' = 5, y' = 12` satisface `r0'`.

El nuevo dominio de `y'` es 5..12.

ii) `r1''` impone `x' - y' == 5`.

a) Límite consistencia sobre `x'`.

- Cota inferior 5. No existe una combinación `x' = 5, y' in 5..12` que satisfazca `r1'`. Se elimina el valor 5 del dominio de `x'`. La nueva cota inferior es 6. La límite consistencia continúa podando también los valores 6, 7, 8 y 9. La nueva cota inferior de `x'` será 10, para la que la combinación `x' = 10, y' = 5` satisface `r1'`.
- Cota superior 12. La combinación `x' = 12, y' = 7` satisface `r1'`.

El nuevo dominio de `x'` es 10..12.

b) Límite consistencia sobre `y'`.

- Cota inferior 5. La combinación `x' = 10, y' = 5` satisface `r1'`.
- Cota superior 12. No existe una combinación `x' in 10..12, 12` que satisfazca `r1'`. Se elimina el valor 12 del dominio de `y'`. La nueva cota superior es 11. La límite consistencia continúa podando también los valores 11, 10, 9 y 8. La nueva cota inferior de `y'` será 7, para la que la combinación `x' = 12, y' = 7` satisface `r1'`.

El nuevo dominio de `y'` es 5..7.

Todas las `IlcIntVar` contenidas en `solver` tienen al menos un valor en su dominio, pero los dominios de `x'` e `y'` han cambiado durante la última 'propagación completa'. Por lo tanto tiene sentido realizar una segunda 'propagación completa':

i) `r0'` impone `x' + y' == 17`.

a) Límite consistencia sobre `x'`.

- Cota inferior 10. La combinación `x' = 10, y' = 7` satisface `r0'`.
- Cota superior 12. La combinación `x' = 12, y' = 5` satisface `r0'`.

El dominio de `x'` continúa como 10..12.

b) Límite consistencia sobre `y'`.

- Cota inferior 5. La combinación `x' = 12, y' = 5` satisface `r0'`.
- Cota superior 7. La combinación `x' = 10, y' = 7` satisface `r0'`.

El dominio de `y'` continúa como 5..7.

ii) `r1'` impone `x' - y' == 5`.

a) Límite consistencia sobre `x'`.
   - Cota inferior 10. La combinación `x' = 10, y' = 5` satisface `r1'`.
   - Cota superior 12. La combinación `x' = 12, y' = 7` satisface `r1'`.

   El dominio de `x'` continúa como 10..12.

b) Límite consistencia sobre `y'`.
   - Cota inferior 5. La combinación `x' = 10, y' = 5` satisface `r1'`.
   - Cota superior 7. La combinación `x' = 12, y' = 7` satisface `r1'`.

   El dominio de `y'` continúa como 5..7.

Todas las `IlcIntVar` contenidas en `solver` tienen al menos un valor en su dominio. Además, los dominios de `x'` e `y'` no han cambiado con esta segunda 'propagación completa'. Por lo tanto hemos encontrado la siguiente solución por intervalos a la conjunción de restricciones $\mathcal{FD}$': `x' in 10..12, y' in 5..7`, que puede verse en la Figura 1.10. El método `solver.propagate()` devuelve el valor 1.

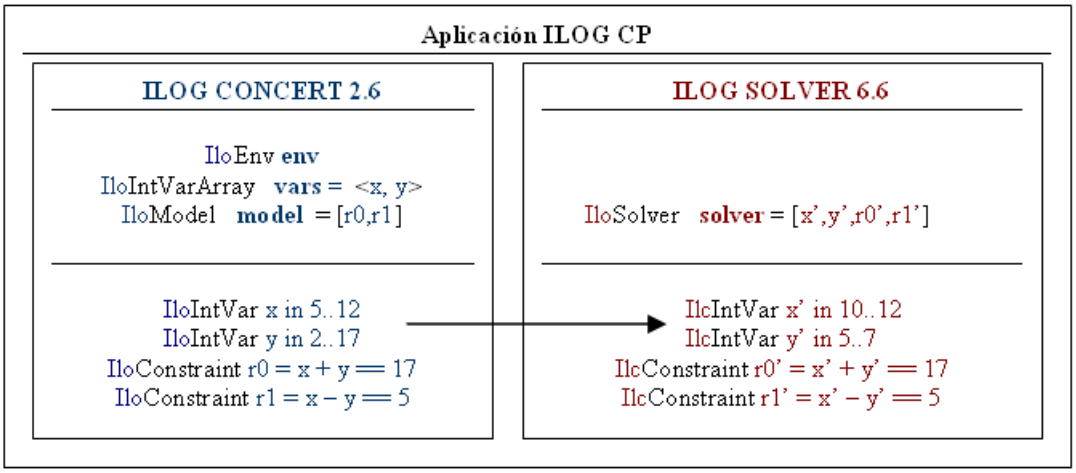

Figura 1.10: Estado de la aplicación tras la propagación de restricciones.

Procedimientos de búsqueda

La propagación de restricciones encuentra una solución por intervalos a la conjunción de restricciones $\mathcal{FD}$'. Sin embargo, existen casos en los que nos puede interesar obtener una solución un poco más específica, donde conozcamos el valor exacto que toman algunas de las `IlcIntVar`s contenidas en `solver`. Se denomina a esta situación etiquetado sobre algunas variables lógicas $\mathcal{FD}$ involucradas en la conjunción de resticciones $\mathcal{FD}$.

Por ejemplo, la conjunción de restricciones $\mathcal{FD}$ `domain [A,B,C] 0 2, A #> B`, tiene una solución por intervalos: `A in 1..2, B in 0..1, C in 0..2`. Sin embargo, nos puede interesar imponer que en las soluciones encontradas las variables `A` e `B` tomen un único valor en su dominio. Entonces la conjunción de restricciones $\mathcal{FD}$ tiene las soluciones `A = 1, B = 0, C in 0..2` y `A = 2, B = 1, C in 0..2`. Como vemos, una conjunción de restricciones $\mathcal{FD}$ con una única solución por intervalos puede dar pie a varias soluciones por etiquetado.

En nuestro caso vamos a realizar un etiquetado sobre las IlcIntVars x' e y'. Además, nos centramos tan sólo en buscar una primera solución al etiquetado. En caso de que existan más soluciones, no nos ocupamos de encontrarlas.

La biblioteca ILOG Solver 6.6 contiene la clase IloGoal, que implementa el etiquetado de IlcIntVars mediante procedimientos de búsqueda. El usuario puede crear sus propios procedimientos de búsqueda o utilizar los predefinidos en la biblioteca. Los procedimientos de búsqueda predefinidos están implementados en la clase IloGenerate, que hereda de IloGoal. El constructor de la clase IloGenerate es el siguiente:
IloGenerate(IloEnv e, IloIntVarArray v, IloChoose s):

- El primer argumento es el contexto env de la aplicación ILOG CP.
- El segundo argumento es un objeto IloIntVarArray indicando las IloIntVars asociadas a las IlcIntVars que se van a etiquetar. En nuestro caso, queremos etiquetar las IlcIntVars x' e y'. Sus IloIntVars asociadas son x e y. Como vars=<x,y> podemos utilizar directamente el IloIntVarArray vars como argumento de goal. Si vars contuviera IloIntVars adicionales entonces deberíamos crear un nuevo IloIntVarArray que contuviera únicamente a x e y.
- El tercer argumento define la estrategia que va a seguir goal al etiquetar las IlcIntVars x' e y'. Las dos estrategias que vamos a utilizar en este trabajo están predefinidas en IloGenerate:
  - Una estrategia de búsqueda estática que etiqueta las IlcIntVars en el orden en que sus IloIntVars asociadas están contenidas en el IloIntVarArray pasado como argumento. Está implementada en la clase IloChooseFirstUnboundInt. Por lo tanto, mediante la instrucción
    IloGoal goal = IloGenerate(env,vars,IloChooseFirstUnboundInt)
    creamos un procedimiento de búsqueda que etiqueta x' e y' siguiendo esta estrategia.
  - La estrategia dinámica first_fail. Cada vez que goal va a etiquetar una nueva IlcIntVar, la estrategia ordena las IlcIntVars aún no etiquetadas según el número de valores que tienen en su dominio. Después, goal etiqueta la IlcIntVar que contiene menor número de valores en su dominio. La estrategia first_fail está implementada en la clase IloChooseMinSizeInt. Por lo tanto, mediante la instrucción
    IloGoal g = IloGenerate(env,vars,IloChooseMinSizeInt) creamos un procedimiento de búsqueda que etiqueta x' e y' siguiendo esta estrategia.

  En ambas estrategias los valores del dominio de cada IlcIntVar se etiquetan en orden ascendente, comenzando por su cota inferior hasta llegar a su cota superior.

En (12) creamos el procedimiento de búsqueda goal para etiquetar x' e y' mediante la estrategía de búsqueda estática. Siguiendo este goal, solver etiqueta en primer lugar los valores de x' en orden ascendente. Despues etiqueta los valores de y, también en orden ascendente. En la Figura 1.11 mostramos el árbol de exploración que resulta de esta estrategia.

1. El árbol tiene tantos niveles como IlcIntVars se van a etiquetar. Cada IlcIntVar ocupa un nivel del árbol. En nuestro caso, x' ocupa el primer nivel e y' el segundo nivel.

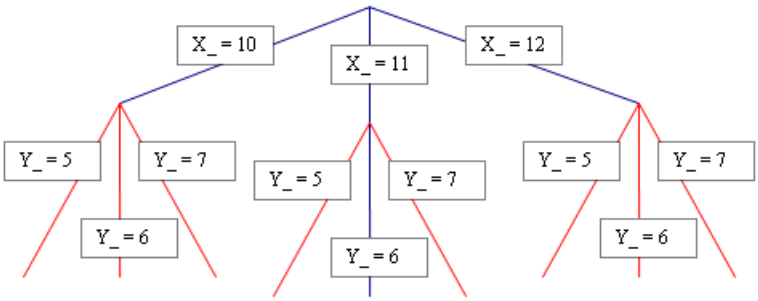

Figura 1.11: Árbol de búsqueda.

2. Si la `IlcIntVar` que ocupa el nivel `i` del árbol tiene un total de `k` valores en su dominio, entonces el nivel `i` del árbol tiene `k` ramas de cómputo. En nuestro caso, `x'` tiene dominio 10..12, así que el primer nivel del árbol tiene tres ramas de cómputo: la rama `x' = 10`, la rama `x' = 11` y la rama `x' = 12`. La `IlcIntVar y'` tiene dominio 5..7, así que el segundo nivel del árbol tiene también tres ramas de cómputo: la rama `y' = 5`, la rama `y' = 6` y la rama `y' = 7`.

3. El árbol de exploración contiene tantas hojas como posibles combinaciones de valores se pueden efectuar sobre las `IlcIntVar`s a etiquetar. Cada una de estas hojas representa un candidato a solución del etiquetado. En nuestro caso el árbol contiene nueve candidatos a solución:

```
1. x'=10, y'=5,   2. x'=10, y'=6,   3. x'=10, y'=7,
4. x'=11, y'=5,   5. x'=11, y'=6,   6. x'=11, y'=7,
7. x'=12, y'=5,   8. x'=12, y'=6,   9. x'=12, y'=7
```

4. El orden en que se exploran las hojas (candidatos) del árbol viene determinado por la estrategia de `goal`. En nuestro caso, debido a la estrategia de búsqueda estática, las hojas son exploradas en el orden en que las hemos numerado.

5. Además, `solver` implementa la técnica de propagación de restricciones durante la búsqueda. Con esta técnica `solver` detecta lo antes posible aquellas ramas del árbol que no contienen soluciones entre sus hojas. Cuando se detecta una rama `r` que no contiene soluciones entre sus hojas, se descarta la exploración de `r` y se continúa la búsqueda en la siguiente rama del árbol. La propagación de restricciones durante búsqueda consiste en aplicar propagación de restricciones cada vez que `solver` accede a una nueva rama del árbol.

Para comenzar la búsqueda de soluciones, `goal` debe aplicarse sobre `solver`. En este punto `solver` permite dos formas de proceder:

a) Si sólo se va a buscar una primera solución al etiquetado, se puede utilizar el método `int solver.solve(goal)`. Éste explora el árbol hasta encontrar una primera hoja solución. El valor devuelto indica si se ha encontrado esta solución o no. Tras finalizar, el método elimina el árbol de búsqueda, así como toda estructura creada específicamente para buscar soluciones al etiquetado.

b) Si se quieren obtener varias soluciones al etiquetado, entonces se deben utilizar los siguientes métodos:

i) `void solver.startNewSearch(goal)`, que asocia el procedimiento de búsqueda `goal` al resolutor `solver`, pero que no inicia la exploración del árbol en búsqueda de soluciones.

ii) `int solver.next()`, que busca una nueva solución al etiquetado. El valor devuelto indica si se ha encontrado esta solución o no. En caso de encontrarla, el procedimiento de búsqueda permanece en la hoja solución, para reanudar desde este punto la búsqueda de una nueva solución, en caso de que le sea solicitado.

iii) `void solver.endSearch()`, que es llamado cuando no se quieren buscar nuevas soluciones al etiquetado, o cuando la exploración del árbol ha concluido sin encontrar una nueva solución. Elimina el árbol de búsqueda, así como toda estructura creada específicamente para buscar soluciones al etiquetado.

En (13) aplicamos `goal` a `solver` y optamos por buscar una única solución al etiquetado. Estudiamos en detalle la forma en que se explora el árbol de búsqueda para encontrar esta solución.

En primer lugar se explora la rama `x' = 10`, que etiqueta `x'`. La propagación de restricciones durante la búsqueda aplica límite consistencia sobre el dominio de `y'`:

- `r0'` impone `x' + y' == 17`. En este caso `10 + y' == 17`.
  - Cota inferior 5. La combinación `x' = 10, y' = 5` no satisface `r0'`. Se elimina el valor 5 del dominio de `y'`. La nueva cota inferior es 6. La límite consistencia poda también el valor 6. La nueva cota inferior de `y'` es 7, para la que la combinación `x' = 10, y' = 7` satisface `r0'`.
  - Cota superior 7. Acabamos de comprobar que `x' = 10, y' = 7` satisface `r0'`.

  El dominio de `y'` se acota al valor 7.
- `r1'` impone `x' - y' == 5`. En este caso `10 - y' == 5`.
  - Cota inferior y superior 7. La combinación `x' = 10, y' = 7` no satisface `r1'`. Se elimina el valor 7 del dominio de `y'`. Como en este momento `y'` no tiene ningún valor en su dominio hemos detectado que la conjunción de restricciones $\mathcal{FD}$' es insatisfactible al etiquetar `x' = 10`.

Esto hace que `solver` pode todas las subramas `y' = 5`, `y' = 6`, `y' = 7` de la rama del árbol `x' = 10`. Al no contener ninguna subrama, la exploración de la rama `x' = 10` concluye.

Mediante la propagación de restricciones durante la búsqueda hemos evitado la exploración de los candidatos `1. x'=10, y'=5;` `2. x'=10, y'=6;` `3. x'=10, y'=7`; que no forman parte de las soluciones al etiquetado.

La búsqueda de soluciones continúa, explorando la rama `x' = 11`. Previamente `solver` debe restablecer el contenido que tenía antes de explorar la rama `x' = 10`. Esto implica restablecer el dominio de `y'` a 5..7.

Al explorar la rama `x' = 11` se aplica de nuevo propagación de restricciones sobre el dominio de `y'`:

- `r0'` impone `x' + y' == 17`. En este caso `11 + y' == 17`.

- Cota inferior 5. La combinación `x' = 11, y' = 5` no satisface `r0'`. Se elimina el valor 5 del dominio de `y'`. La nueva cota inferior es 6, que sí satisface `r0'`.
- Cota superior 7. La combinación `x' = 11, y' = 7` no satisface `r0'`. Se elimina el valor 7 del dominio de `y'`. La nueva cota superior es 6, que acabamos de comprobar que sí satisface `r0'`.

El dominio de `y'` se acota al valor 6.

- `r1'` impone `x' - y' == 5`. En este caso `11 - y' == 5`.
  - Cota inferior y superio 6. La combinación `x' = 11, y' = 6` satisface `r1'`.

El dominio de `y'` continúa acotado al valor 6.

Se realiza una segunda 'propagación completa', que no modifica el dominio de `y'`, por lo que finaliza la propagación de restricciones. Esto hace que `solver` pode las subramas `y' = 5`, `y' = 7`. Mediante la propagación de restricciones durante la búsqueda hemos evitado la exploración de los candidatos `4. x'=11, y'=5`; `6. x'=11, y'=7`, que no forman parte de las soluciones al etiquetado.

La rama `x' = 11` contiene una única subrama, `y' = 6`, que pasa a explorarse etiquetando a `y'`. Hemos alcanzado la hoja del candidato `5. x'=11, y'=6`. Se aplica propagación de restricciones:

- `r0'` impone `x' + y' == 17`. En este caso `11 + 6' == 17`, que se satisface trivialmente.
- `r1'` impone `x' - y' == 5`. En este caso `11 - 6' == 5`, que se satisface trivialmente.

Al satisfacerse todas las `IlcConstraint`s de la conjunción de restricciones $\mathcal{FD}'$, concluimos que el candidato `5. x'=11, y'=6` constituye una primera solución al etiquetado. El método `int solver.solve(goal)` finaliza devolviendo el valor 1. Esto elimina el árbol de búsqueda y toda estructura de datos utilizada durante el etiquetado. El estado de la aplicación ILOG CP puede verse en la Figura 1.12

En (14) `solver` muestra por pantalla la solución por etiquetado encontrada.

1.2.2. Una primera aplicación ILOG CP: `intro.exe` o `intro.dll`

Hemos explicado en detalle el archivo fuente `intro.cpp`, que modela y resuelve una conjunción de restricciones $\mathcal{FD}$ concreta. Describimos ahora los comandos necesarios para crear una aplicación ILOG CP `intro.exe` o `intro.dll` a partir del archivo fuente `intro.cpp`. Este ejemplo puede ser extendido a cualquier otra aplicación ILOG CP.

La creación de una aplicación ILOG CP implica dos fases:

1. Compilar a código máquina `archivo.obj` cada uno de los archivos fuente `archivo.cpp` que implementan la aplicación .
2. Vincular los archivos `*.obj` generados y resolver las referencias con las bibliotecas utilizadas.

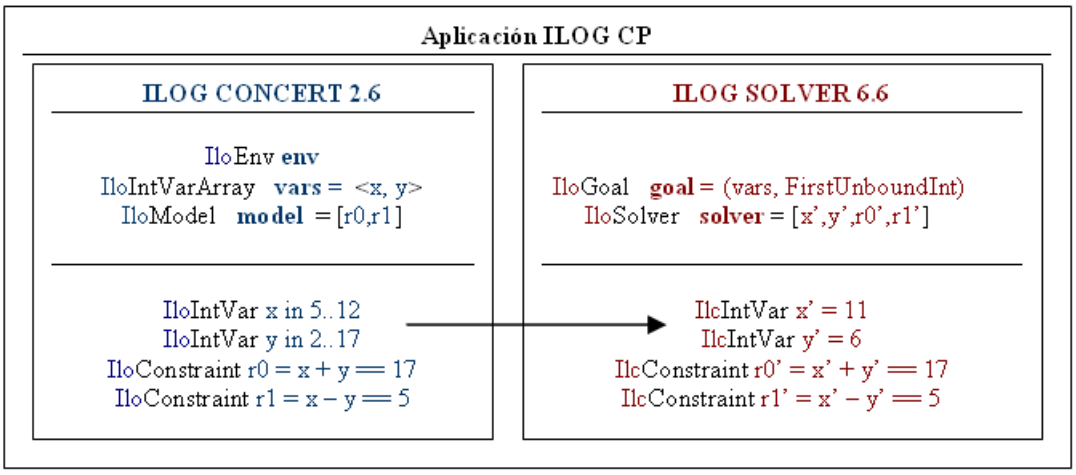

Figura 1.12: Estado de la aplicación tras la resolución.

Para realizar las dos fases se necesita un compilador y un vinculador de aplicaciones C++. Para un sistema operativo Windows XP Profesional con una arquitectura de 32 bits, ILOG CP 1.4 soporta las herramientas de compilación y vinculación distribuidas en los entornos de desarrollo Microsoft Visual Studio 2005 y 2008. Por compatibilidad con SICStus Prolog 3.12.8 nos decantamos por la más antigua de estas versiones. Instalamos y utilizamos el entorno de desarrollo Microsoft Visual Studio 2005 [27] (Versión 8.0.50727.42). Éste contiene Microsoft Visual C++ 6.0, que incluye el compilador `cl` (Versión 14.00.50727.42) y el vinculador `link` (Versión 8.00.50727.42) entre sus herramientas para el desarrollo de aplicaciones.

Para utilizar desde consola de comandos las herramientas `cl` y `link` se debe ejecutar el archivo `vsvars32.bat`. Este archivo por lotes está incluido en la distribución de Microsoft Visual Studio 2005, en la dirección
`<Microsoft Visual Studio 8 DIR>\Common7\Tools`. Configura el entorno para utilizar las herramientas de Microsoft Visual Studio.

Creamos una nueva carpeta `intro` en nuestro directorio. Su contenido inicial debe ser:

- El archivo `intro.cpp`.
- El nuevo archivo `vs2005.bat`. Mediante la ejecución de este archivo accedemos a una consola en la que podamos utilizar las herramientas de Microsoft Visual Studio. El contenido de `vs2005.bat` es el siguiente:

```
cmd /K "C:\Archivos de programa\Microsoft Visual Studio 8\
                                        Common7\Tools\vsvars32.bat"
pause
cd "<Intro DIR>"
pause
```

Compilación de los archivos fuente

El siguiente comando compila el archivo fuente `intro.cpp`:

```
cl intro.cpp -Fointro.obj -I"C:\ILOG\Concert26\include"
         -I"C:\ILOG\Solver66\include" -I. -MD -EHsc -DIL_STD -c
```

Explicamos en detalle los distintos elementos que constituyen el comando:

1. `cl` representa la llamada al compilador de Microsoft Visual Studio.

   El resto del comando son los argumentos utilizados en la llamada al compilador. Éste discrimina los argumentos entre archivos y opciones. Todo argumento no identificado como opción se toma como archivo.

2. `intro.cpp` es el archivo de entrada que se va a compilar.
3. `-Fointro.obj` indica al compilador que genere el archivo de código máquina `intro.obj`.
4. `-I"<DIR>"` busca archivos de inclusión en el directorio indicado. Debemos incluir:
   - Los directorios de inclusión de las bibliotecas ILOG Concert 2.6 e ILOG Solver 6.6. Serán necesarios, ya que `intro.cpp` contiene llamadas a estas bibliotecas. `<Concert DIR>\include` y `<Solver DIR>\include`.
   - El directorio actual, que contiene al propio archivo `intro.cpp`.
5. `-MD` selecciona DLL multiproceso como biblioteca en tiempo de ejecución.
6. `-EHsc` captura sólo las excepciones de C++.
7. `-DMacro` define una macro. Las aplicaciones ILOG CP precisan la macro de preprocesado `IL_STD`, por lo que debemos incluir la opción `DIL_STD`.
8. `-c` compila sin vincular.

Como resultado de la compilación el archivo `intro.obj` es creado en el directorio `<Intro DIR>`. El nuevo contenido de `<Intro DIR>` es:

- `vs2005.bat`
- `intro.cpp`
- `intro.obj`

Vinculación de una aplicación ILOG CP

Distinguimos los comandos para la creación de un ejecutable o una biblioteca dinámica:

Creación de un ejecutable `intro.exe`

El siguiente comando crea el ejecutable `intro.exe` vinculando `intro.obj` con las bibiotecas utilizadas:

```
link intro.obj -OUT:intro.exe
"C:\ILOG\Concert26\lib\x86_.net2005_8.0\stat_mda\concert.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solver.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solveriim.lib"
wsock32.lib
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

Explicamos en detalle los distintos elementos que constituyen el comando:

1. `link` representa la llamada al vinculador de Microsoft Visual Studio.

   El resto del comando son los argumentos utilizados en la llamada al vinculador. Éste discrimina los argumentos entre archivos y opciones. Todo argumento no identificado como opción se toma como archivo.

2. `intro.obj` es el único archivo a vincular. Éste contiene referencias a bibliotecas aún sin resolver. Estas bibliotecas deben incluirse como argumentos en la llamada al vinculador.

3. `-OUT:intro.exe` especifica el archivo ejecutable que se generará. Éste contendrá nuestra aplicación ILOG CP `intro`.

4. Las bibliotecas `concert.lib`, `solver.lib` y `solveriim.lib` se incluyen para resolver las referencias de `intro.obj`. ILOG CP 1.4 ofrece varias versiones de estas bibliotecas. Si se ha utilizado el argumento `-MD` en la compilación de los archivos fuente, entonces se debe utilizar la versión `stat_mda` de las bibliotecas ILOG. Esta versión de cada biblioteca se encuentra en las direcciones:
   `<Concert DIR>\lib\x86_net2005_8.0\stat_mda\`
   `<Solver DIR>\lib\x86_.net2005_8.0\stat_mda\`

5. `link` precisa como argumentos un conjunto de bibliotecas dependientes de la arquitectura del sistema:

   ```
   kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
   shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
   ```

   Para vincular una aplicación ILOG CP hay que incluir también la biblioteca `wsock32.lib`.

Como resultado de la vinculación los archivos `intro.exe` e `intro.exe.manifest` son creados en el directorio `<Intro DIR>`. El nuevo contenido de `<Intro DIR>` es:

- `vs2005.bat`
- `intro.cpp`
- `intro.obj`
- `intro.exe`
- `intro.exe.manifest`

Creación de una biblioteca dinámica *`intro.dll`*

El siguiente comando crea la bilioteca dinámica `intro.dll` vinculando `intro.obj` con las bibiotecas utilizadas:

```
link intro.obj -OUT:intro.dll
-dll
"C:\ILOG\Concert26\lib\x86_.net2005_8.0\stat_mda\concert.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solver.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solveriim.lib"
```

```
wsock32.lib
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

Los únicos argumentos que cambian con respecto a la creación de un ejectable son:

- `-OUT:intro.dll` para especificar el nombre de la biblioteca dinámica que se generará.
- `-dll` genera una dll.

Como resultado de la vinculación los archivos `intro.dll` e `intro.dll.manifest` son creados en el directorio `<Intro DIR>`. El nuevo contenido de `<Intro DIR>` es:

- `vs2005.bat`
- `intro.cpp`
- `intro.obj`
- `intro.dll`
- `intro.dll.manifest`

Independencia de la aplicación ILOG CP

Si queremos exportar nuestra aplicación `intro.exe` o `intro.dll` a otra carpeta de nuestro directorio, entonces hay que hacer que este ejectuable o biblioteca dinámica sea independiente del archivo manifiesto que lleva adjunto.

Los siguientes comandos realizan esta acción para el ejecutable y biblioteca dinámica, respectivamente. Utilizan la herramienta `mt` (Versión 5.2.3790.2014), suministrada en Microsoft Visual Studio 2005 para la gestión de archivos manifiesto.

```
1. mt -manifest intro.exe.manifest -outputresource:intro.exe;#2
2. mt -manifest intro.dll.manifest -outputresource:intro.dll;#2
```

Ahora ya disponemos de nuestra aplicación ILOG CP completamente autocontenida en `intro.exe` o `intro.dll`. Los archivos `intro.cpp`, `intro.obj` e `intro.*.manifest` pueden ser eliminados.

Capítulo 2

$\mathcal{TOY}(\mathcal{FD}i)$: Interconexión e implementación básica

En este capítulo describimos la interconexión del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$, que permite al sistema $\mathcal{TOY}$ modelar y resolver conjunciones de restricciones $\mathcal{FD}$ utilizando el sistema de restricciones externo ILOG CP 1.4. En la segunda sección describimos la implementación de una primera versión mínima de dicho sistema, que soporta ocho tipos de restricciones $\mathcal{FD}$.

2.1. $\mathcal{TOY}(\mathcal{FD}i)$: Interconexión entre $\mathcal{TOY}$ e ILOG CP 1.4

En la primera sección del capítulo anterior se describió la arquitectura de componentes genérica del sistema $\mathcal{TOY}(\mathcal{FD})$, que permite comunicar al sistema $\mathcal{TOY}$ con cualquier sistema de restricciones $\mathcal{FD}$ externo. En la segunda sección se describió la aplicación ILOG CP `intro`, que modela y resuelve una conjunción de restricciones $\mathcal{FD}$ concreta. Se describió además la forma de crear un ejecutable o una bibioteca dinámica para una aplicación ILOG CP concreta.

Utilizando todos estos conceptos previos describimos a continuación la interconexión del sistema $\mathcal{TOY}(\mathcal{FD}i)$, que permite utilizar ILOG CP como sistema de restricciones $\mathcal{FD}$ externo del sistema $\mathcal{TOY}$.

Para poder implementar el sistema $\mathcal{TOY}(\mathcal{FD}i)$ necesitamos crear una aplicación genérica ILOG CP. Esta aplicación genérica ILOG CP no dispone de una función `main` para modelar y resolver una conjunción de restricciones concreta. En lugar de eso, dispone de un repertorio de funciones que permiten modelar y resolver cualquier conjunción de restricciones $\mathcal{FD}$ suministrada externamente. Es decir, hasta ahora hemos utilizado ILOG CP como un sistema sobre el que trabajar directamente. En cambio, en la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ se va a utilizar ILOG CP como el nuevo sistema de restricciones $\mathcal{FD}$ externo del sistema $\mathcal{TOY}$. Las restricciones $\mathcal{FD}$ que aparecen en el objetivo $\mathcal{TOY}$ serán transmitidas a este componente ILOG CP. La aplicación genérica ILOG CP modela y resuelve la conjunción de restricciones $\mathcal{FD}$ que conforman las restricciones $\mathcal{FD}$ enviadas desde $\mathcal{TOY}$.

El sistema $\mathcal{TOY}$ está implementado en SICStus Prolog 3.12.8. La aplicación genérica ILOG CP es una aplicación C++. Estudiamos la forma de comunicar ambos sistemas a través de los siguientes tres hitos:

1. Comunicación entre una aplicación SICStus y una aplicación C++.

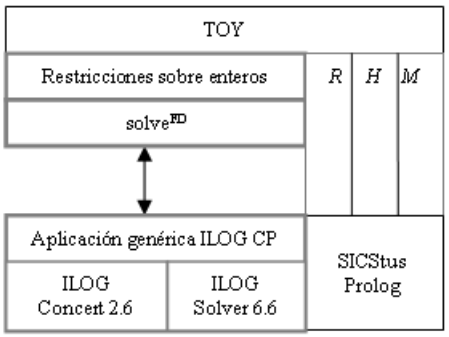

Figura 2.1: Arquitectura de componentes del sistema $\mathcal{TOY}(\mathcal{FD}i)$.

2. Adaptación de la aplicación C++ para que sea la aplicación genérica ILOG CP.
3. Adaptación de la aplicación SICStus para que sea el sistema $\mathcal{TOY}$.

2.1.1. Comunicación entre una aplicación SICStus y una aplicación C++

SICStus Prolog soporta predicados Prolog que están implementados en funciones C++. De este modo se pueden escribir aplicaciones SICStus donde parte del código está implementado en C++. Si durante la evaluación de un objetivo nos encontramos uno de estos predicados Prolog `pi`, entonces el control de programa se transfiere a la función C++ `fi` que lo implementa. Cuando `fi` finaliza su ejecución finaliza también la evaluación de `pi`. El control de programa retorna a la aplicación SICStus, que continúa la evaluación del objetivo.

Para que una aplicación SICStus `aplicacion.pl` utilice los predicados Prolog `p1,...,pn` implementados en las funciones C++ `f1,...,fn` es necesario:

i) Crear un recurso ajeno `recurso.dll` que identifica a todos los predicados Prolog `p1,...,pn` (implementados en `f1,...,fn`) que se van a utilizar en `aplicacion.pl`.

ii) Cargar el recurso ajeno `recurso.dll` en `aplicacion.pl`, para poder utilizar los predicados Prolog `p1,...,pn`. La directiva `:- load_foreign_resource(recurso)` se encarga de ello.

Creación de un recurso ajeno e integración en SICStus Prolog

Para crear un nuevo recurso ajeno `recurso.dll` se necesitan definir previamente dos archivos: `recurso.pl` y `recurso.cpp`.

- El archivo `recurso.cpp` es una aplicación C++. Debe contener las funciones C++ `f1,...,fn`. Puede además contener tantas funciones C++ auxiliares, variables estáticas, macros e `#include` como sean necesarios para implementar a las funciones `f1,...,fn`.
- El archivo `recurso.pl` sirve de nexo entre la aplicación SICStus `aplicacion.pl` y la aplicacion C++ contenida en `recurso.cpp`. Contiene n llamadas al predicado Prolog `foreign` y una llamada al predicado Prolog `foreign_resource`.
    - Cada llamada al predicado Prolog `foreign(fi,pi)` vincula el predicado Prolog `pi` con la función C++ `fi`. Al vincular `pi` con `fi` se debe especificar la conversión entre los argumentos de `pi` y los argumentos de `fi`.

- La llamada a `foreign_resource([f1,...,fn])` agrupa en una lista a todas las funciones C++ `f1,...,fn` que implementan a predicados Prolog.

Estudiamos en primer lugar las conversiones de argumentos entre un predicado Prolog `pi` y una función C++ `fi` permitidas en `foreign`. A continuación describimos en detalle la creación de un primer recurso ajeno `recurso.dll`.

Conversión de argumentos entre Prolog y C++

Tanto el predicado Prolog `pi` como la función C++ `fi` deben tener el mismo número de argumentos. Los argumentos de entrada de `pi` son transmitidos a `fi`. Los argumentos de salida de `pi` son computados en `fi`. En este trabajo utilizamos las cuatro siguientes conversiones permitidas en SICStus:

1. Prolog: +integer, C++: long
   El predicado Prolog transmite a la función C++ un número entero. La función C++ lo recibe como un argumento de tipo long.

2. Prolog: +term, C++: `SP_term_ref`
   El predicado Prolog transmite a la función C++ un término Prolog. La función C++ maneja estos términos Prolog como una variable de tipo `SP_term_ref`.

3. Prolog: -integer, C++: long*
   La función C++ computa en su cuerpo una variable de tipo long. Tras finalizar la ejecución de la función C++ el control de programa vuelve al predicado Prolog, que recibe como argumento el valor entero equivalente.

4. Prolog: -term, C++: `SP_term_ref`
   La función C++ computa en su cuerpo una variable de tipo `SP_term_ref`. Tras finalizar la ejecución de la función C++ el control de programa vuelve al predicado Prolog, que recibe como argumento el término Prolog equivalente.

SICStus permite manejar términos Prolog dentro de las funciones C++ mediante variables de tipo `SP_term_ref`. Estudiamos las operaciones soportadas por SICStus para `SP_term_ref`s:

- Funciones para manipular `SP_term_ref`s:
  - Creación de un nuevo `SP_term_ref`:
    `SP_term_ref SP_new_term_ref(void)`
  - Asignación del contenido de un `SP_term_ref from` a otro `SP_term_ref to`:
    `void SP_put_term(SP_term_ref to, SP_term_ref from)`

- Funciones para almacenar tipos de datos básicos C++ en una variable `SP_term_ref`:
  - Asignación de un valor entero a una variable `SP_term_ref`:
    `int SP_put_integer(SP_term_ref t, long l)`
    Devuelve el valor 1 (0) indicando si ha tenido éxito (o no) en la operación.
  - Asignación de una lista Prolog, cuya cabeza y cola están contenidas en los `SP_term_ref`s `head` y `tail`, a una variable `SP_term_ref t`:
    `int SP_cons_list(SP_term_ref t, SP_term_ref head, SP_term_ref tail)`

- Funciones para chequear y obtener el contenido de un `SP_term_ref`:

  - Chequea si una variable `SP_term_ref` contiene un valor entero:
    `int SP_is_integer(SP_term_ref t)`
  - Almacena el valor entero contenido en una variable `SP_term_ref` dentro de una variable long:
    `int SP_get_integer(SP_term_ref t, long *l)`
  - Chequea si una variable `SP_term_ref` es una lista:
    `int SP_is_list(SP_term_ref t)`
  - Extráe la cabeza y la cola de una lista contenida en una variable `SP_term_ref`. Almacena la cabeza y la cola en sendas variables `SP_term_ref`:
    `int SP_get_list(SP_term_ref t, SP_term_ref head, SP_term_ref tail)`

- Una función `int SP_compare(SP_term_ref x, SP_term_ref y)` para comparar si un mismo término Prolog está contenido en dos variables `SP_term_ref` diferentes.
- Una función `int SP_unify(SP_term_ref x, SP_term_ref y)` para unificar dos términos Prolog contenidos en variables `SP_term_ref`.

Un primer recurso ajeno *`recurso.dll`*

Creamos un primer recurso ajeno en la carpeta `<Recurso DIR>` de nuestro directorio. Añadimos a esta carpeta los siguientes archivos:

- `vs2005.bat`
- `aplicacion.pl`
- `recurso.cpp`
- `recurso.pl`

El directorio `<SICStus Prolog 3.12.8 DIR>\bin` contiene entre otros el archivo `sicstus.exe`, ejecutable del sistema SICStus Prolog 3.12.8. Si añadimos este directiorio a las variables de entorno de nuestro sistema operativo Windows XP, entonces podemos acceder al sistema SICStus Prolog 3.12.8 desde cualquier directorio, en concreto desde `<Recurso DIR>`.

Para añadir una nueva variable de entorno en Windows XP se debe acceder a `Mi PC --> Propiedades --> Opciones avanzadas --> Variables de entorno`. Creamos la nueva variable de entorno `PATH` con valor `<SICStus Prolog 3.12.8 DIR>\bin`.

Estudiamos el contenido de los archivos `recurso.pl`, `recurso.cpp` y `aplicacion.pl`.

```
-----------------------------------------------
    \recurso.pl\
foreign(f_suma,p_suma(+integer,+integer)).
foreign(f_resta,p_resta(+integer,+integer)).
foreign_resource(recurso,[f_suma,f_resta]).
-----------------------------------------------
    \recurso.cpp\
#include <sicstus/sicstus.h>
#include <stdio.h>
#include "recurso_glue.h"
```

```
void f_suma(long a, long b) {
  long add = a + b;
  printf("add =%lu\n", add);
}

void f_resta(long a, long b) {
  long sub = a - b;
  printf("sub =%lu\n", sub);
}
-----------------------------------------------
    \aplicacion.pl\
:- load_foreign_resource(recurso).

evalua(A,B) :-
  p_suma(A,B),
  p_resta(A,B).
-----------------------------------------------
```

El predicado Prolog `evalua` del archivo `aplicacion.pl` utiliza los predicados Prolog `p_suma` y `p_resta`, implementados en las funciones C++ `f_suma` y `f_resta`, respectivamente. Para poder utilizarlos debemos:

- Crear el recurso ajeno `recurso.dll` a partir de los archivos `recurso.cpp` y `recurso.pl`.
- Cargar este recurso en `aplicacion.pl` mediante la directiva `:- load_foreign_resource(recurso).`

SICStus ofrece la herramienta `splfr` [32] para crear un recurso ajeno `recurso.dll` a partir de los archivos `recurso.pl` y `recurso.cpp`. La herramienta `splfr` es una macro diseñada por SICStus. Realiza una serie de llamadas a las herramientas de Microsoft Visual Studio 2005 para generar este recurso ajeno.

El siguiente comando crea el recurso ajeno `recurso.dll` a partir de los archivos `recurso.pl` y `recurso.cpp`:

```
splfr recurso.pl recurso.cpp --verbose
```

Explicamos en detalle los distintos elementos que constituyen el comando:

1. `splfr` representa la llamada a la macro de SICStus para la generación de recursos ajenos.

   El resto del comando son los argumentos utilizados en la llamada a la macro. Ésta discrimina los argumentos entre archivos y opciones. Todo argumento no identificado como opción se toma como archivo.

2. `recurso.pl` y `recurso.cpp` son los archivos necesarios para generar `recurso.dll`.
3. `--verbose` es una opción para mostrar las llamadas internas realizadas por la macro `splfr`.

Como resultado de la macro los archivos `recurso.dll` y `recurso.dll.manifest` son creados en el directorio `<Recurso DIR>`. El nuevo contenido de `<Recurso DIR>` es:

- `vs2005.bat`
- `aplicacion.pl`
- `recurso.cpp`
- `recurso.pl`
- `recurso.dll`
- `recurso.dll.manifest`

Sin embargo, hay ocasiones en las que el usuario no puede crear un recurso ajeno utilizando directamente `splfr`. Esto es debido a que las opciones soportadas por `splfr` no son suficientes para indicar los argumentos necesarios para la creación del recurso ajeno. En estos casos, la recomendación de SICStus es reconstruir las llamadas internas que realiza `splfr` y customizarlas con los parámetros deseados.

Estudiamos en detalle los comandos utilizados internamente por `splfr recurso.pl recurso.cpp --verbose` para la creación de `recurso.dll`:

1. Comando para la creación de los archivos de interfaz `recurso_glue.c` y `recurso_glue.h`.

   ```
   sicstus -f --goal"(prolog:splfr_prepare_foreign_resource('recurso',
       'recurso.pl','recurso_glue.c','recurso_glue.h'),halt);halt(1)."
   ```

   Este comando llama a SICStus para que, utilizando el contenido del archivo `recurso.pl`, cree los dos nuevos archivos `recurso_glue.c` y `recurso_glue.h`. Estos archivos contienen el código pegamento para vincular los predicados Prolog `p_suma` y `p_resta` con las funciones C++ `f_suma` y `f_resta`. Estos archivos preveen la utilización de la macro `SPDLL` de SICStus.

   Para utilizar este código pegamento nuestro archivo `recurso.cpp` debe importar el archivo `recurso_glue.h`.

   Como resultado de la llamada a SICStus los archivos `recurso_glue.c` y `recurso_glue.h` son creados en el directorio `<Recurso DIR>`.

2. Comando para la compilación de `recurso.cpp`.

   ```
   cl recurso.cpp -Forecurso.obj
   -I"C:\Archivos de programa\SICStus Prolog 3.12.8\include" -I.
   -MD -EHsc -DIL_STD -DSPDLL -c
   ```

   Las únicas novedades con respecto al comando `cl` utilizado en la subsección 1.2.2 son:

   - `recurso.cpp` contiene referencias a las bibliotecas de SICStus. Se añade el directorio de inclusión de SICStus Prolog 3.12.8.

     `<SICStus Prolog 3.12.8 DIR>\include`
   - `recurso.cpp` contiene referencias al archivo de código pegamento `recurso_glue.c`. Como este archivo requiere la macro `SPDLL` se añade el argumento `DSPDLL`.

Como resultado de la compilación el archivo `recurso.obj` es creado en el directorio `<Recurso DIR>`.

3. Comando para la compilación de `recurso_glue.c`.

```
cl recurso_glue.c -Forecurso_glue.obj
-I"C:\Archivos de programa\SICStus Prolog 3.12.8\include" -I.
-MD -EHsc -DIL_STD -DSPDLL -c
```

Como resultado de la compilación el archivo `recurso_glue.obj` es creado en el directorio `<Recurso DIR>`.

4. Comando para la vinculación de `recurso.obj` y `recurso_glue.obj`.

```
link recurso.obj recurso_glue.obj -OUT:recurso.dll -dll
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

La única novedad con respecto al comando `link` utilizado en la subsección 1.2.2 es que ahora son dos archivos los que se vinculan.

Como resultado de la vinculación los archivos `recurso.dll` y `recurso.dll.manifest` son creados en el directorio `<Recurso DIR>`.

5. Comando para la integración del archivo manifiesto en el recurso ajeno.

```
mt -manifest recurso.dll.manifest -outputresource:recurso.dll;#2
```

Ahora ya disponemos de nuestro recurso ajeno completamente autocontenido en `recurso.dll`. Los archivos `recurso.cpp`, `recurso.pl`, `recurso_glue.c`, `recurso_glue.h`, `recurso.obj`, `recurso_glue.obj` y `recurso.dll.manifest` pueden ser eliminados.

Este recurso ajeno `recurso.dll` puede ser ahora cargado y utilizado en cualquier aplicación SICStus de nuestro directorio, en concreto en la aplicación SICStus `aplicacion.pl` contenida en `<Recurso DIR>`.

Ejecutamos ahora dos objetivos sobre `aplicacion.pl`:

- `evalua(3,2)`, que suma y resta los números 3 y 2.

```
| ?- evalua(3,2).
add =5
sub =1
yes
```

- `evalua([],4)`, que debe dar error de tipos.

```
| ?- evalua([],4).
! Type error in argument 1 of user:p_suma/2
! number expected, but [] found
! goal:  p_suma([],4)
```

2.1.2. Comunicación entre una aplicación SICStus y la aplicación genérica ILOG CP

Como explicamos al inicio de esta sección nuestra aplicación genérica ILOG CP contiene un repertorio de funciones C++ `f1,...,fn`, que permiten modelar y resolver conjunciones de restricciones $\mathcal{FD}$. En esta subsección explicamos los pasos necesarios para adaptar la aplicación C++ involucrada en el recurso ajeno para que sea en realidad la aplicación genérica ILOG CP.

Crearemos un nuevo recurso ajeno `ilog_cp.dll` en la carpeta `<Ilog_cp DIR>` de nuestro directorio. En primer lugar añadimos a esta carpeta los siguientes archivos:

- `vs2005.bat`
- `aplicacion.pl`
- `ilog_cp.cpp`
- `ilog_cp.pl`

A partir de la aplicación ILOG CP `intro` definida en la subsección 1.2.1 creamos una primera versión muy precaria de nuestra aplicación genérica ILOG CP `ilog_cp.cpp`. Ésta primera versión de `ilog_cp.cpp` contiene las funciones C++ `modelado` y `resolucion`. Estas funciones realizan la fase de modelado y resolución de la conjunción de restricciones $\mathcal{FD}$ de la aplicación ILOG CP `intro`. Creamos un archivo `ilog_cp.pl` para vincular predicados Prolog a estas funciones C++. Creamos una aplicación SICStus `aplicacion.pl` que hace uso de estos predicados implementados en las funciones de la aplicación genérica ILOG CP.

Estudiamos el contenido de los archivos `ilog_cp.pl`, `ilog_cp.cpp` y `aplicacion.pl`.

```
-----------------------------------------------
        / ilog_cp.pl /
foreign(modelado,modelado).
foreign(resolucion,resolucion).
foreign_resource(ilog_cp,[modelado,resolucion]).
-----------------------------------------------
        / ilog_cp.cpp /
#include <sicstus/sicstus.h>
#include <stdio.h>
#include "ilog_cp_glue.h"
#include <ilconcert/ilomodel.h>
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

static IloEnv env;
static IloModel model(env);
static IloIntVarArray vars(env);
static IloSolver solver(env);

void modelado(){
  IloIntVar x(env, 5, 12);
  vars.add(x);
```

```
  IloIntVar y(env, 2, 17);
  vars.add(y);
  IloConstraint r0 = vars[0] + vars[1] == 17;
  model.add(r0);
  IloConstraint r1 = vars[0] - vars[1] == 5;
  model.add(r1);
}

void resolucion(){
  solver.extract(model);
  if (solver.propagate()){
    IloGoal goal = IloGenerate(env,vars,IloChooseFirstUnboundInt);
    if (solver.solve(goal)){
      solver.out() << "x = " << solver.getValue(vars[0]) << endl;
      solver.out() << "y = " << solver.getValue(vars[1]) << endl;
    }
  }
}
-----------------------------------------------
        / aplicacion.pl /
:- load_foreign_resource(ilog_cp).

evalua :-
  modelado,
  resolucion.
-----------------------------------------------
```

En este caso hemos preferido que `modelado` y `resolucion` sea tanto el nombre del predicado Prolog como de la función C++ que lo implementa. El resto del recurso ajeno sigue la misma estructura que el del recurso ajeno `recurso` estudiado en la subsección anterior.

Creamos el recurso ajeno `ilog_cp.dll` a partir del contenido de `ilog_cp.cpp` e `ilog_cp.pl`. En este caso no podemos utilizar directamente la herramienta `splfr`:

a) En la subsección anterior estudiamos las llamadas internas que realiza `splfr`. Vimos que la creación del recurso ajeno `ilog_cp.dll` implicará, entre otros, la compilación del archivo `ilog_cp.cpp`.

b) En la subsección 1.2.2 estudiamos los argumentos necesarios para compilar con `cl` una aplicación ILOG CP concreta, por ejemplo la aplicación ILOG CP `intro.cpp`. Entre esos argumentos se debía incluir `-DIL_STD`, para utilizar la macro `IL_STD` durante el preprocesado. Pues bien, `splfr` no permite incluir este tipo de argumentos entre sus opciones.

Reproducimos los comandos internos utilizados por `splfr` para la creación de `recurso.dll`. Customizamos estos comandos con las necesidades específicas de una aplicación ILOG CP, descritas en la subsección 1.2.2.

1. Comando para la creación de los archivos de interfaz `ilog_cp_glue.c` e `ilog_cp_glue.h`.

```
sicstus -f --goal "(prolog:splfr_prepare_foreign_resource('ilog_cp',
'ilog_cp.pl','ilog_cp_glue.c','ilog_cp_glue.h'),halt);halt(1)."
```

Como resultado de la llamada a SICStus los archivos `ilog_cp_glue.c` e `ilog_cp_glue.h` son creados en el directorio `<Ilog_cp DIR>`.

2. Comando para la compilación de `ilog_cp.cpp`.

```
cl ilog_cp.cpp -Foilog_cp.obj
-I"C:\Archivos de programa\SICStus Prolog 3.12.8\include"
-I"C:\ILOG\Concert26\include" -I"C:\ILOG\Solver66\include" -I.
-MD -EHsc -DIL_STD -DSPDLL -c
```

Las únicas novedades con respecto al comando `cl` utilizado en la subsección 2.1.1 es que `ilog_cp.cpp` contiene referencias a las bibliotecas de ILOG. Se añade el directorio de inclusión de ILOG Concert 2.6 e ILOG Solver 6.6

Como resultado de la compilación el archivo `ilog_cp.obj` es creado en el directorio `<Ilog_cp DIR>`.

3. Comando para la compilación de `ilog_cp_glue.c`.

```
cl ilog_cp_glue.c -Foilog_cp_glue.obj
-I"C:\Archivos de programa\SICStus Prolog 3.12.8\include"
-I"C:\ILOG\Concert26\include" -I"C:\ILOG\Solver66\include"
-I. -MD -EHsc -DIL_STD -DSPDLL -c
```

Como resultado de la compilación el archivo `ilog_cp_glue.obj` es creado en el directorio `<Ilog_cp DIR>`.

4. Comando para la vinculación de `ilog_cp.obj` e `ilog_cp_glue.obj`.

```
link ilog_cp.obj ilog_cp_glue.obj -OUT:ilog_cp.dll -dll
"C:\ILOG\Concert26\lib\x86_.net2005_8.0\stat_mda\concert.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solver.lib"
"C:\ILOG\Solver66\lib\x86_.net2005_8.0\stat_mda\solveriim.lib"
wsock32.lib
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

La única novedad con respecto al comando `link` utilizado en la subsección 2.1.1 es que ahora hay que resolver referencias con las bibliotecas de ILOG.

Como resultado de la vinculación los archivos `ilog_cp.dll` e `ilog_cp.dll.manifest` son creados en el directorio `<Ilog\_cp DIR>`.

5. Comando para la integración del archivo manifiesto en el recurso ajeno.

```
mt -manifest ilog_cp.dll.manifest -outputresource:ilog_cp.dll;#2
```

Ahora ya disponemos de nuestro recurso ajeno completamente autocontenido en `ilog_cp.dll`. Los archivos `ilog_cp.cpp`, `ilog_cp.pl`, `ilog_cp_glue.c`, `ilog_cp_glue.h`, `ilog_cp.obj`, `ilog_cp_glue.obj` y `ilog_cp.dll.manifest` pueden ser eliminados.

Este recurso ajeno `ilog_cp.dll` puede ser ahora cargado y utilizado en cualquier aplicación SICStus de nuestro directorio, en concreto en la aplicación SICStus `aplicacion.pl` contenida en `<Ilog\_cp DIR>`.

Ejecutamos ahora el objetivo `evalua` sobre `aplicacion.pl`, que modela y resuelve la conjunción de restricciones $\mathcal{FD}$ de la aplicación ILOG CP `intro`:

```
| ?- evalua.
x = 11
y = 6
yes
```

2.1.3. Comunicación entre $\mathcal{TOY}$ y la aplicación genérica ILOG CP: $\mathcal{TOY}(\mathcal{FD}i)$

En esta subsección conectamos al sistema $\mathcal{TOY}$ con la aplicación genérica ILOG CP contenida en el recurso ajeno `ilog_cp.dll`. Esta conexión posibilita al sistema $\mathcal{TOY}$ definir nuevos predicados Prolog que estén implementados en las funciones C++ que fueron definidas en `ilog_cp.cpp`. Gracias a estos predicados Prolog, el sistema $\mathcal{TOY}$ puede modelar y resolver conjunciones de restricciones $\mathcal{FD}$ utilizando las bibliotecas de ILOG CP. Es decir, estamos posibilitando una nueva versión del sistema $\mathcal{TOY}$ que utiliza como sistema de restricciones externo la tecnología ILOG CP. Denominamos a esta nueva versión del sistema $\mathcal{TOY}$ como $\mathcal{TOY}(\mathcal{FD}i)$.

Utilizamos la distribución actual del sistema $\mathcal{TOY}$ (Versión 2.3.1), que suponemos contenida en el directorio `<TOY DIR>`. La distribución contiene, entre otros, un archivo `toy.pl`. Para ejecutar el sistema $\mathcal{TOY}$ basta con compilar desde SICStus el archivo `toy.pl`. La propia distribución de $\mathcal{TOY}$ incluye el ejecutable `toy.exe` que realiza esta acción de manera oculta al usuario.

Añadimos los archivos `ilog_cp.cpp` e `ilog_cp.pl` al directorio `<TOY DIR>` y reproducimos los comandos ejecutados en la subsección anterior para crear el recurso ajeno `ilog_cp.dll`. Para poder utilizar los predicados `modelado` y `resolución`, bastaría con que $\mathcal{TOY}$ cargase el recurso ajeno `ilog_cp.dll` dentro del sistema mediante la directiva `load_foreign_resource(ilog_cp)`. Esta tarea, que era inmediata en las aplicaciones SICStus `aplicacion.pl` utilizadas en las secciones 4.1 y 4.2, se vuelve un poco más complicada en el caso del sistema $\mathcal{TOY}$. Esto se debe a lo siguiente:

- Varios archivos distintos de la distribución de $\mathcal{TOY}$ hacen uso de las funciones C++ del recurso ajeno `ilog_cp.dll`.
  - El archivo `cflpfdfile.pl` contiene los predicados Prolog que gestionan las restricciones $\mathcal{FD}$. En la futura implementación del sistema $\mathcal{TOY}(\mathcal{FD}i)$ parece claro que estos predicados Prolog necesitarán comunicarse con la aplicación genérica ILOG CP.
  - El archivo `goals.pl` contiene el predicado Prolog `writeSolution`, que muestra la solución de un objetivo $\mathcal{TOY}$ al usuario. Si dicho objetivo contiene restricciones $\mathcal{FD}$, entonces `writeSolution` necesitará comunicarse con la aplicación genérica ILOG CP.
- Estos archivos pueden realizar la carga de predicados Prolog procedentes de otros archivos de la distribución $\mathcal{TOY}$. Cuando se produce la carga de nuevos archivos se puede producir la descarga del recurso ajeno `ilog_cp.dll`.

Por lo tanto:

1. Si cargamos el recurso ajeno `ilog_cp.dll` al compilar el archivo `toy.pl`, entonces no podemos utilizar el recurso ajeno en los predicados de otros archivos Prolog de la distribución $\mathcal{TOY}$. Concretamente no podremos utilizarlo en los archivos `cflpfdfile.pl` y `goals.pl`, por lo no podemos utilizar a ILOG CP como sistema de restricciones externo.

2. Si cargamos el recurso ajeno en cada archivo de la distribución que lo necesite entonces tampoco funciona, ya que el recurso se carga una única vez por archivo. Si dos predicados Prolog de un mismo archivo necesitan usar el recurso ajeno, y entre la ejecución de uno y otro se produce la descarga del recurso ajeno, entonces el segundo predicado Prolog no podrá utilizar el recurso ajeno.

La solución que utilizamos consiste en:

1. Modificar el contenido del archivo `ilog_cp.pl`:

   1.1. Incluímos la directiva `load_foreign_resource(ilog_cp)` en este archivo.

   1.2. Añadimos la directiva `:- module(ilog_cp,[modelado/0,resolucion/0])`, que define un nuevo módulo en el sistema $\mathcal{TOY}$ con los predicados Prolog `modelado` y `resolucion`.

   El nuevo contenido de `ilog_cp.pl` es:

```
:- module(ilog_cp,[modelado/0,resolucion/0]).
foreign(modelado,modelado).
foreign(resolucion,resolucion).
foreign_resource(ilog_cp,[modelado,resolucion]).
:- load_foreign_resource(ilog_cp).
```

2. Modificar el contenido de todos los archivos de la distribución de $\mathcal{TOY}$ que necesiten hacer uso del recurso `ilog_cp.dll`.

   2.1. Añadimos la directiva
   `:- load_files(ilog_cp,[if(changed),imports([modelado/0,resolucion/0])]).`

3. Reproducir de nuevo los comandos ejecutados en la subsección anterior para crear el recurso ajeno `ilog_cp.dll`.

Ahora, todos los archivos de la distribución de $\mathcal{TOY}$ que precisen hacer uso del recurso ajeno importan los predicados Prolog contenidos definidos en el módulo `ilog_cp.pl`. Este módulo contiene los predicados Prolog `modelado` y `resolucion`, implementados como funciones C++ en el recurso ajeno.

1. Cada vez que un archivo de la distribución $\mathcal{TOY}$ necesite utilizar el recurso ajeno importará el modulo `ilog_cp`, contenido en el archivo `ilog_cp.pl`. Esto implica compilar `ilog_cp.pl`.

2. Como `ilog_cp.pl` contiene la directiva
   `:- load_foreign_resource(ilog_cp)` se cargará el recurso ajeno `ilog_cp.dll`.

Por lo tanto, conseguimos que el sistema $\mathcal{TOY}$ cargue el recurso ajeno cada vez que necesita utilizarlo. Es decir, podemos acceder desde cualquier punto del sistema $\mathcal{TOY}$ a la aplicación genérica ILOG CP. Esto supone el punto de arranque a la implementación del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$.

Mostramos como ejemplo una primera comunicación absurda entre $\mathcal{TOY}$ e ILOG CP. Importamos el módulo `ilog_cp` en los archivo `cflpfdfile.pl` y `goals.pl`. Modificamos el predicado Prolog `$#>` de `cflpfdfile.pl`, incluyendo una llamada al predicado Prolog `modelado`. Modificamos el predicado Prolog `writeSolution` de `goals.pl`, incluyendo una llamada al predicado Prolog `resolucion`.

Ejecutamos ahora el objetivo $\mathcal{TOY}$ `X #> 3`. Este objetivo llama al predicado Prolog `$#>(X,3,Out,Cin,Cout)` para gestionar la restricción $\mathcal{FD}$. Como el objetivo tiene éxito se llama al predicado Prolog `writeSolution`, que muestra la solución al usuario. Mostramos la ejecución de dicho objetivo en nuestra primera versión rudimentaria del sistema $\mathcal{TOY}(\mathcal{FD}i)$:

```
Toy(FD)> X #> 3
x = 11
y = 6
      { X in 4..sup }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Vemos que los predicados `S#>` y `writeSolution` han realizado correctamente la evaluación de los predicados Prolog `modelado` y `resolucion`. De hecho, el predicado `resolucion` muestra por pantalla la solución que ha encontrado a la conjunción de restricciones $\mathcal{FD}$ de la aplicación `intro`.

La comunicación entre $\mathcal{TOY}$ e ILOG CP se realiza correctamente. Por lo tanto, comenzamos la implementación del sistema $\mathcal{TOY}(\mathcal{FD}i)$.

Terminamos la subsección describiendo dos tareas extra:

- La forma en que adaptamos la distribución de $\mathcal{TOY}$ a la versión de SICStus Prolog 3.12.8.
- Los comandos necesarios para crear el nuevo archivo ejecutable `toy.exe`.

Adaptación de $\mathcal{TOY}$ a SICStus Prolog 3.12.8

La versión 2.3.1 del sistema $\mathcal{TOY}$ está implementada en SICStus Prolog 3.11.1. Como el sistema $\mathcal{TOY}(\mathcal{FD}i)$ se va a implementar sobre SICStus Prolog 3.12.8, sustituimos aquellos ficheros de $\mathcal{TOY}$ (Versión 2.3.1) importados de la distribución de SICStus Prolog 3.11.1 por sus equivalentes en la distribución de SICStus Prolog 3.12.8:

i) Renombramos la carpeta `sp311` a `sp312`. Reemplazamos el archivo `sprt.sav` de `<TOY DIR>\SP312\bin` por el archivo `sprt.sav` contenido en la distribución de SICStus Prolog 3.12.8.

ii) Eliminamos la carpeta `sp-3.11.1`, que ya no es necesaria para el funcionamiento del sistema.

iii Sustituimos los archivos `spcon.dll` y `sprt311.dll` por `spcon.dll` y `sprt312.dll` contenidos en SICStus Prolog 3.12.8.

Creación del ejecutable `toy.exe`

El archivo ejecutable `toy.exe` depende de la biblioteca dinámica `sp311.dll`. Al reemplazar esta biblioteca dinámica por su equivalente de SICStus Prolog 3.12.8 `sp312.dll` el ejecutable `toy.exe` deja de funcionar. SICStus ofrece la herramienta `spld` para crear ejecutables. Estudiamos esta herramienta para crear un nuevo `toy.exe` que utilice la biblioteca `sp312.dll`. Las dos características que tiene `toy.exe` son:

1. Realiza como única acción `compile(toy).`
2. Utiliza como directorio respaldo el directorio `<TOY DIR>`, donde está siendo creado.

Para crear un ejecutable indicándole las acciones a realizar son necesarios cuatro pasos:

1. Crear un archivo `ejecutable.pl` cuyo contenido sea:

   ```
   user:runtime_entry(start) :-
       "codigo a ejecutar"
   ```

   En nuestro caso creamos el archivo `ejecToy.pl` con el contenido:

   ```
   user:runtime_entry(start) :-
       compile(toy).
   ```

2. Ejecutar SICStus Prolog 3.12.8 para compilar el archivo `ejecToy.pl` y guardarlo como el archivo de restauración textttoy.sav. Realizamos estas acciones mediante el comando:
   `sicstus --goal "compile(ejecToy), save_program('toy.sav'), halt."`
3. Utilizar el argumento `--main=restore`. Esta opción indica a `spld` que existe un archivo de restauración `*.sav` que contiene el código a ejecutar por el ejecutable.
4. Utilizar el argumento `--resources=toy.sav=/toy.sav`. Esta opción indica a `spld` el archivo de restauración que contiene el código a ejecutar por el archivo ejecutable.
5. Utilizar el argumento `--respath=.`, que indica a `spld` que utilice el directorio actual como directorio de respaldo.

El comando
`spld --main=restore --respath=. --resources=toy.sav=/toy.sav --output=toy.exe`
crea el nuevo archivo ejecutable `toy.exe` de la distribución $\mathcal{TOY}$. Este archivo ejecuta el sistema $\mathcal{TOY}$ utilizando la biblioteca dinámica `sp312.dll` del sistema SICStus Prolog 3.12.8.

2.2. $\mathcal{TOY}(\mathcal{FD}i)$: Un sistema básico

En esta sección describimos la implementación de una primera versión mínima del sistema $\mathcal{TOY}(\mathcal{FD}i)$. Esta primera versión da soporte a ocho tipos de restricciones $\mathcal{FD}$, como son `L #> R`, `L #>= R`, `L #< R`, `L #<= R`, `L #+ R`, `L #- R`, `L #* R` y `L #/ R`. Para ello:

- Ampliamos la funcionalidad de la aplicación genérica ILOG CP `ilog_cp.cpp` implementando nuevas funciones C++ `f1,...,fn` que permitan modelar y resolver estos tipos de restricciones.
- Modificamos `ilog_cp.pl` definiendo nuevos predicados Prolog `p1,...,pn` que estén implementados en estas funciones C++ `f1,...,fn`.
- Creamos un nuevo recurso ajeno `ilog_cp.dll` que permite al sistema $\mathcal{TOY}(\mathcal{FD}i)$ utilizar estas funciones C++ `f1,...,fn` a través de los predicados `p1,...,pn`.

Los predicados Prolog `$#>`, `$#>=`, `$#<`, `$#<=`, `$#+`, `$#-`, `$#*` y `$#/` pertenecen al componente $solve^{FD}$. Estos predicados están definidos en la actual distribución de $\mathcal{TOY}(\mathcal{FD}s)$, y se encargan de la gestión de los ocho tipos de restricciones $\mathcal{FD}$ que queremos soportar en esta primera implementación de $\mathcal{TOY}(\mathcal{FD}i)$. Por lo tanto, modificamos el contenido de estos predicados para que utilicen ILOG CP como sistema de restricciones $\mathcal{FD}$ externo.

Para poder resolver una conjunción de restricciones $\mathcal{FD}$ en el sistema $\mathcal{TOY}(\mathcal{FD}i)$ son necesarias tres fases:

i) Se transmite desde $\mathcal{TOY}$ a la aplicación genérica ILOG CP la conjunción de restricciones $\mathcal{FD}$ planteada.

ii) La aplicación genérica ILOG CP resuelve dicha conjunción de restricciones $\mathcal{FD}$.

iii) Se accede desde $\mathcal{TOY}$ a la aplicación genérica ILOG CP para obtener la solución computada.

La sección se estructura de la siguiente manera. En las tres primeras subsecciones se describe una primera implementación de estos predicados para poder realizar las fases i), ii), e iii), respectivamente. En la cuarta subsección se describe una implementación más sólida de estos predicados, que permite modelar y resolver conjunciones de restricciones que incluyan restricciones $\mathcal{FD}$ compuestas.

2.2.1. Transmisión de la conjunción de restricciones $\mathcal{FD}$

La transmisión a la aplicación genérica ILOG CP de la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$ exige una traducción de cada restricción $\mathcal{FD}$ a una nueva restricción $\mathcal{FD}$' equivalente en el formato requerido por la biblioteca ILOG Concert 2.6. La nueva conjunción de restricciones $\mathcal{FD}$' representa dentro de la aplicación genérica ILOG CP a la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$.

En la sección 1.1.1 describimos la forma genérica para traducir la conjunción de restricciones $\mathcal{FD}$ al sistema de restricciones externo. En la subsección 1.2.1 describimos los objetos básicos para representar una conjunción de restricciones $\mathcal{FD}$' en una aplicación ILOG CP. Para el objetivo `X #> Y, X #> 4` la asociación entre la conjunción de restricciones $\mathcal{FD}$ y la conjunción de restricciones $\mathcal{FD}$' en la aplicación ILOG CP puede verse en la Figura 2.2.

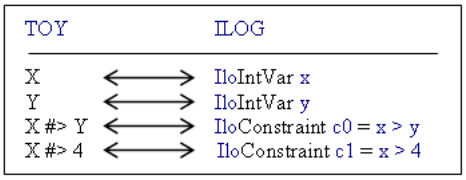

Figura 2.2: Asociación entre objetos $\mathcal{TOY}$ e ILOG.

Si el cálculo de estrechamiento perezoso detecta una restricción primitiva $\mathcal{FD}$ `C`, entonces esta restricción `C` modifica la conjunción de restricciones $\mathcal{FD}$ `C1,...,Cn` tratada en el objetivo $\mathcal{TOY}$ hasta el momento. La nueva conjunción de restricciones $\mathcal{FD}$ es ahora `C1,...,Cn,C`, que incluye además a toda nueva variable lógica $\mathcal{FD}$ `V` que esté involucrada en `C`. Esta nueva conjunción de restricciones $\mathcal{FD}$ no es consistente con la conjunción de restricciones $\mathcal{FD}$' `c1,...,cn` contenida en la aplicación genérica ILOG CP. Para recuperar la consistencia se debe:

- Traducir la nueva restricción $\mathcal{FD}$ `C` a su restricción $\mathcal{FD}$' equivalente `c`. Esto implica traducir previamente toda nueva variable lógica $\mathcal{FD}$ `V` involucrada en `C` a su variable lógica $\mathcal{FD}$' equivalente `v`.
- Transmitir la nueva restricción $\mathcal{FD}$' `c` a la aplicación genérica ILOG CP.

Tras realizar esas dos tareas, la aplicación genérica ILOG CP contendrá la conjunción de restricciones $\mathcal{FD}$' `c1,...,cn,c`, consistente con la conjunción de restricciones $\mathcal{FD}$ `C1,...,Cn,C` tratada por el objetivo $\mathcal{TOY}$.

Para traducir y transmitir `C` se utiliza un predicado Prolog del componente $solve^{FD}$ (recordar Figura 2.1). Este componente $solve^{FD}$ va a contener tantos predicados Prolog `p1,...,pk` como tipos de restricciones $\mathcal{FD}$ soporte el sistema $\mathcal{TOY}(\mathcal{FD}i)$. Si suponemos que `C` es, por ejemplo, una restricción de tipo `L #> R`, entonces `C` será transferida al predicado Prolog `$#>` del componente $solve^{FD}$. Este predicado `$#>` se encarga, entre otras tareas, de traducir y transmitir `C` a la aplicación genérica ILOG CP.

Por ejemplo, dado el objetivo `X #> Y, X #> 4, Z #> Y`, donde suponemos que las dos primeras restricciones `X #> Y, X #> 4` ya han sido evaluadas, el contenido de la aplicación ILOG CP puede verse en la Figura 2.3.

El cálculo de estrechamiento perezoso evalúa ahora la restricción `Z #> Y`. Para su resolución, el cálculo transmite esta restricción $\mathcal{FD}$ al predicado Prolog `$#>(+L,+R,-Out,+Cin,-Cout)`, que traduce y transmite esta nueva restricción `Z #> Y` a la aplicación ILOG CP. Veasé la Figura 2.4.

El predicado Prolog `$#>(Z,Y,O,Cin,Cout)` recupera la consistencia entre la conjunción de restricciones $\mathcal{FD}$' representada en `model` (`[x>y, x>4]`) y la nueva conjunción de restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ evaluado hasta el momento (`[X#>Y, X#>4, Z#>Y]`) al traducir y transmitir `Z #> Y` mediante:

1) Gestión de los argumentos.

   - Verifica que la nueva variable lógica $\mathcal{FD}$ `Z` no contiene ningún objeto `IloIntVar` asociado en la conjunción de restricciones $\mathcal{FD}$'. Esto es, que `Z` no está representada en la conjunción de restricciones $\mathcal{FD}$'.

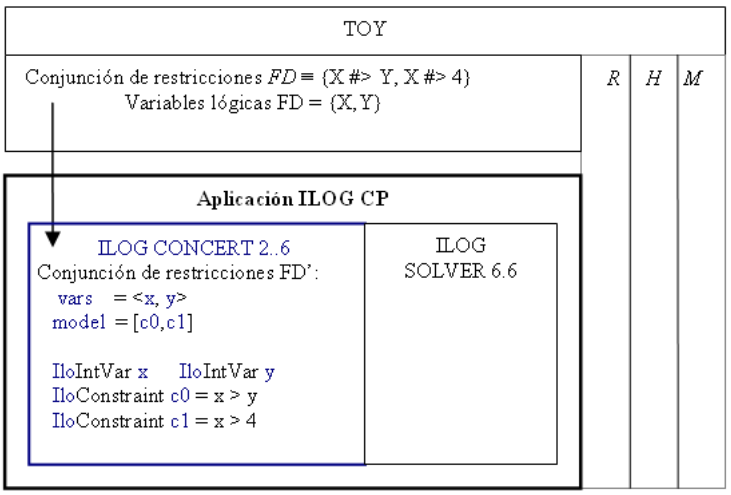

Figura 2.3: Estado de $\mathcal{TOY}(\mathcal{FD}i)$ tras evaluar las dos primeras restricciones.

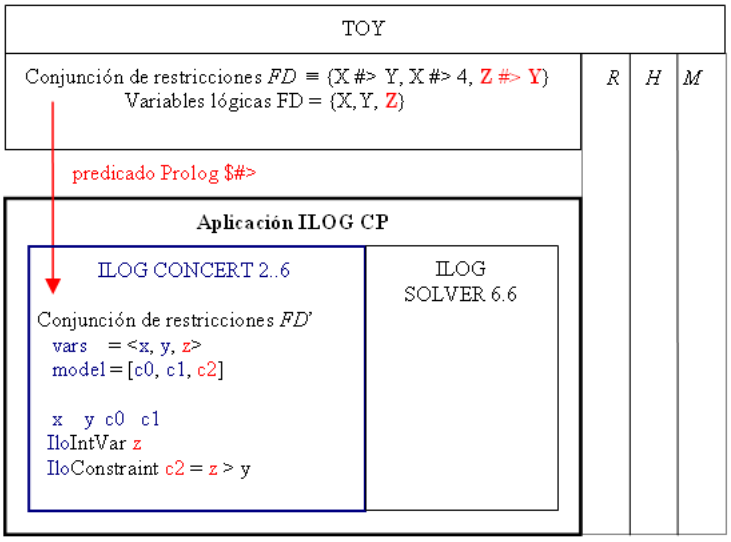

Figura 2.4: Resolución del predicado Prolog.

- Crear el nuevo objeto `IloIntVar z` que representa dentro de la conjunción de restricciones $\mathcal{FD}$' (explicitada en `model`) a la variable lógica $\mathcal{FD}$ `Z` de la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$.
- Añade esta nueva `IloIntVar z` al vector de variables lógicas $\mathcal{FD}$' `vars`. De este modo tiene acceso directo a `z` en todo punto de cómputo a partir de la transmisión de la restricción `Z #> Y`.

- Verifica que la variable lógica $\mathcal{FD}$ `Y` está asociada al objeto `IloIntVar y` contenido en `vars[1]`.

2) Gestión de la propia restricción $\mathcal{FD}$ `Z #> Y`.

- Crea el nuevo objeto `IloConstraint c2` que representa dentro de la conjunción de restricciones $\mathcal{FD}$' (explicitada en `model`) a la restricción $\mathcal{FD}$ `Z #> Y` de la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$. Esta `IloConstraint c2` tiene a `vars[2]` (`z`) y `vars[1]` (`y`) como `IloIntVar`s involucradas.
- Añade `c2` a `model`. En este momento tanto `IloConstraint c2` como sus `IloIntVar`s involucradas forman parte de la conjunción de restricciones $\mathcal{FD}$'.

Para posibilitar estas tareas:

- Ampliamos la funcionalidad de la aplicación genérica ILOG CP, añadiendo a `ilog_cp.cpp` dos nuevas funciones C++:
  - Una función C++ `create_new_Var`, que crea una nueva `IloIntVar` y la inserta en `vars`.
  - Una función C++ `post_greater`, que crea una nueva `IloConstraint` involucrando ciertas `IloIntVar`s contenidas en `vars` e inserta esta `IloConstraint` en `model`.
- Ampliamos el contenido de `ilog_cp.pl`, definiendo dos nuevos predicados Prolog `create_new_Var` y `post_greater` que están implementados en las funciones C++ `create_new_Var` y `post_greater`, respectivamente.

  El nuevo contenido de `ilog_cp.pl` es:

```
:- module(ilog_cp,[create_new_Var/0,post_greater/4]).
foreign(create_new_Var,create_new_Var).
foreign(post_greater,post_greater(+integer,+integer,+integer)).
foreign_resource(interface,[create_new_Var,post_greater]).
:- load_foreign_resource(ilog_cp).
```

- Ejecutamos los comandos descritos en la subsección 1.2.2 para crear una nueva versión del recurso ajeno `ilog_cp.dll` que incluya las nuevas funciones C++. Esto permite utilizar los predicados Prolog `create_new_Var` y `post_greater` en el sistema $\mathcal{TOY}$.

El predicado `$#>` de $solve^{FD}$ contiene tantas llamadas a estos predicados `create_new_Var` y `post_greater` como sean necesarias para poder traducir y transmitir la restricción `Z #> Y` a la aplicación genérica ILOG CP. La Figura 2.5 muestra gráficamente la comunicación.

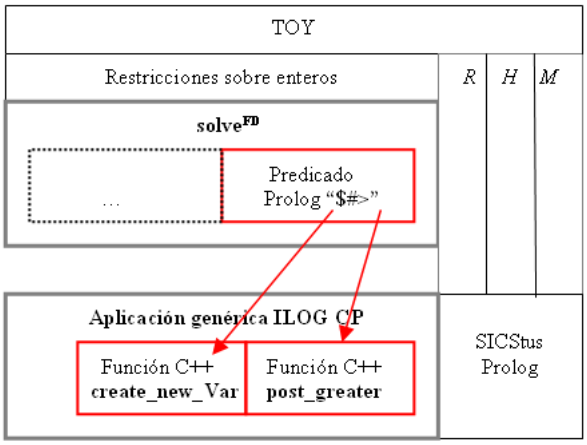

Figura 2.5: Marco de comunicación.

Hemos visto sobre el objetivo `X #> Y, X #> 4, Z #> Y` el comportamiento que pretendemos alcanzar. A continuación detallamos los dos intentos que hemos realizado para implementar este marco de comunicación que permite la transmisión de la conjunción de restricciones $\mathcal{FD}$ a la aplicación genérica ILOG CP. El primer intento utiliza únicamente la función C++ `post_greater`, pero resulta fallido. El segundo intento utiliza las funciones C++ `create_new_Var` y `post_greater` definidas anteriormente.

Primer intento de transmisión de la conjunción de restricciones $\mathcal{FD}$ a la aplicación ILOG CP

En el objetivo `X #> Y, X #> 4, Z #> Y`, la resolución de `Z #> Y` implica detectar que `Z` no tiene ninguna `IloIntVar z` asociada, y que `Y` tiene a `IloIntVar y` contenida en `vars[1]` como `IloIntVar` asociada.

Un primer intento realiza la traducción de las `IloIntVar` asociadas a las variables lógicas $\mathcal{FD}$ íntegramente dentro de la aplicación ILOG CP. Como explicamos en la subsección 2.1.1, SICStus permite la conversión de cualquier término Prolog a una variable de tipo `SP_term_ref` equivalente. Basándose en esta posibilidad, este intento maneja un `vector<SP_term_ref>` paralelo al vector `vars`. Este vector contiene la conversión de las variables lógicas $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$. Las variables lógicas $\mathcal{FD}$ se añaden al vector en el orden textual en que estas variables aparecen involucradas en restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$. Cada variable lógica $\mathcal{FD}$ contenida en una cierta posición `i` del vector contiene su `IloIntVar` asociada en la misma posición `i` de `vars`.

De este modo, podemos crear una primera versión de la función C++ `post_greater(SP_term_ref L, SP_term_ref R)`, que incluya como argumentos los dos términos Prolog involucrados en la restricción $\mathcal{FD}$ que se está transmitiendo. En nuestro caso, el predicado Prolog `$#>` realizaría una única llamada a `post_greater(Z,Y)` para traducir y transmitir la restricción $\mathcal{FD}$ `Z #> Y` a la aplicación genérica ILOG CP. Esta llamada `post_greater(Z,Y)`:

1) Evalúa los argumentos `Z` e `Y` de la restricción $\mathcal{FD}$:

   i) Utiliza el método `int SP_is_variable(SP_term_ref Arg)`, que indica si `Arg` es una variable lógica $\mathcal{FD}$.

i) En caso de serlo, comprueba si esta variable lógica $\mathcal{FD}$ está contenida en el vector. Para ello utiliza el método `int SP_compare(SP_term_ref Arg, SP_term_ref t)` sobre las distintas variables lógicas $\mathcal{FD}$ contenidas en el vector. Este método devuelve 0 cuando ambos `SP_term_ref` corresponden al mismo término Prolog (en este caso una misma variable lógica $\mathcal{FD}$).

iii) Si la variable lógica $\mathcal{FD}$ está contenida en la posición `k` del vector entonces concluimos que su `IloIntVar` asociada está contenida en `vars[k]`.

iv) Si la variable lógica $\mathcal{FD}$ no está contenida en el vector entonces podemos asegurar que estamos ante una nueva variable lógica $\mathcal{FD}$ que no contiene ninguna `IloIntVar` asociada dentro de la aplicación genérica ILOG CP. Siendo `n` el tamaño de `vars` y vector, donde los diferentes elementos se numeran de 0 a `n-1`:

- Se añade la variable lógica $\mathcal{FD}$ `Arg` al vector en la posición `n`.
- Se crea un nuevo objeto `IloIntVar arg`, que se añade a `vars` en la posición `vars[n]`.

2) Una vez que todas las variables lógicas $\mathcal{FD}$ `V1,...,Vn` involucradas en la restricción $\mathcal{FD}$ `C` tienen sus `IloIntVars` `v1,...,vn` asociadas, se crea la nueva `IloConstraint c` que involucra a estas `IloIntVars` `v1,...,vn`. Esta `IloConstraint c` se añade a `model`.

Observamos el comportamiento que deseamos alcanzar en la Figura 2.6 y 2.7.

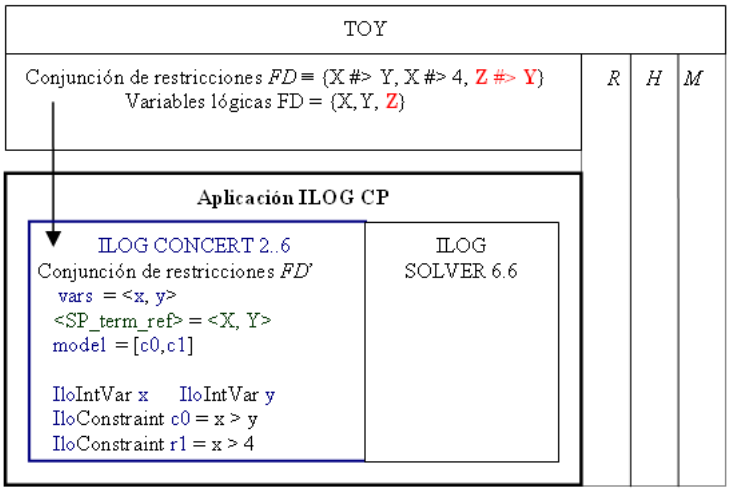

Figura 2.6: Intento de comunicación con un vector de `SP_term_ref`.

Sin embargo este primer intento fracasa. La razón se encuentra en las reglas que gobiernan el ámbito de los `SP_term_ref`. Así, cuando `post_greater` finaliza su ejecución, todos los objetos `SP_term_ref` contenidos en ella (bien sea como argumentos de entrada o dinámicamente creados durante su ejecución) se convierten automáticamente en no válidos. Por lo tanto el contenido del vector deja de ser válido. Esto impide el correcto reconocimiento de las variables lógicas $\mathcal{FD}$ involucradas en futuras restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$. Veámoslo de nuevo sobre el objetivo `X #> Y, X #> 4, Z #> Y`. Al comen-

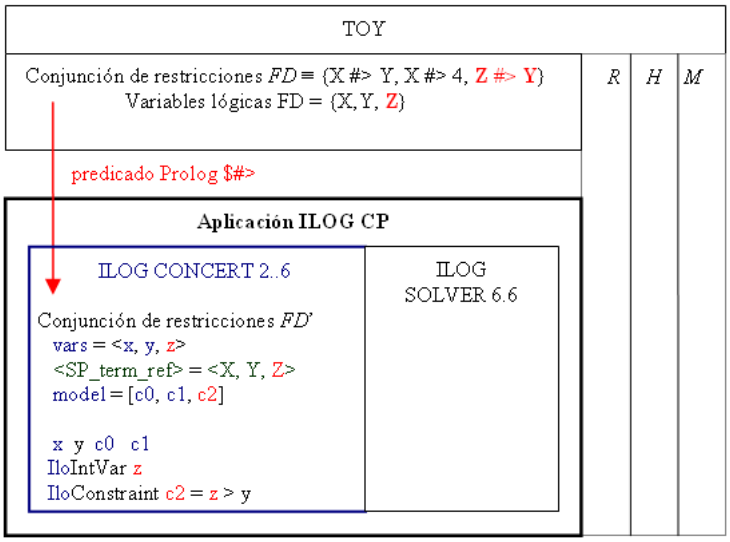

Figura 2.7: Intento de comunicación con un vector de `SP_term_ref`.

zar la evaluación del objetivo $\mathcal{TOY}$, tanto el vector `vars` como el `vector<SP_term_ref>` están vacíos.

La primera restricción del objetivo, `X #> Y`, es una restricción $\mathcal{FD}$ de tipo `#>`. Es gestionada por el predicado Prolog `$#>`, que traduce y transmite la restricción mediante el predicado `post_greater(X,Y)`, implementado en la función C++ `post_greater`.

Esta función C++ `post_greater` detecta a `X` como nueva variable lógica $\mathcal{FD}$, por lo que:

- Crea una nueva `IloIntVar x`, que añade a `vars[0]`.
- Añade `X` a `vector<SP_term_ref>[0]`.

Posteriormente detecta a `Y` como nueva variable lógica $\mathcal{FD}$, ya que el método `SP_compare(Y,vector[0])` devuelve un valor distinto de 0. Crea una nueva `IloIntVar y`, que añade a `vars[1]` y añade `Y` a `vector<SP_term_ref>[1]`.

Una vez que `post_greater` ha evaluado todos sus argumentos, crea la nueva `IloConstraint c0 = vars[0] > vars[1]`, que añade a `model`. Con esto concluye la función C++ `post_greater`, y el contenido del vector se convierte en no válido. El control de programa vuelve al predicado `$#>`, que ha traducido y transmitido correctamente la restricción `X #> Y`.

La segunda restricción del objetivo, `X #> 4`, es una restricción $\mathcal{FD}$ de tipo `#>`. De nuevo es gestionada por el predicado Prolog `$#>`, que contiene una llamada a `post_greater(X,4)`. Como el contenido del vector ya no es válido, al evaluar el argumento `X`, el método `SP_compare(X,vector[0])` indica que son términos distintos, al igual que `SP_compare(X,vector[1])`. Por eso `post_greater` detecta a `X` como una nueva variable lógica $\mathcal{FD}$ no contenida en `vector<SP_term_ref>`. Crea una nueva `IloIntVar x` y la añade a `vars[2]`. Además añade `X` a `vector<SP_term_ref>[2]`.

La ejecución de `post_greater` continúa y detecta a `4` como un valor entero. Una vez que todos los argumentos de la restricción $\mathcal{FD}$ han sido evaluados, `post_greater` crea la nueva `IloConstraint c1 = vars[2] > 4`, en lugar de `IloConstraint c1 = vars[0] > 4`. De este modo, la `IloConstraint c1` no es equivalente a la restricción $\mathcal{FD}$ `X #> 4`, y las conjunciones de restricciones $\mathcal{FD}$ y $\mathcal{FD}$' no son consistentes.

Es necesario un segundo intento que permita la correcta transmisión de la conjunción de restricciones $\mathcal{FD}$ a la aplicación ILOG CP.

Segundo intento de transmisión de la conjunción de restricciones $\mathcal{FD}$ a la aplicación ILOG CP

Hasta ahora no tenemos un almacén explícito que contenga a la conjunción de restricciones $\mathcal{FD}$. Cada restricción $\mathcal{FD}$ que es detectada por el cálculo de estrechamiento, así como las variables lógicas $\mathcal{FD}$ involucradas en alguna restricción $\mathcal{FD}$ sólo son accesibles durante la traducción de la propia restricción $\mathcal{FD}$. Pero no existe un objeto referenciable que reúna a todos estas restricciones y variables, es decir, no existe un objeto conjunción de restricciones $\mathcal{FD}$. Por lo tanto, toda restricción $\mathcal{FD}$ traducida en un momento dado no podrá ser referenciada en la traducción de futuras restricciones $\mathcal{FD}$ detectadas por el cálculo de estrechamiento.

Por otro lado, toda `IloConstraint` e `IloIntVar` conforman la conjunción de restricciones $\mathcal{FD}$' que se está representando en la aplicación ILOG CP. Pero además, el objeto `model` contiene la lista con todas las `IloConstraint`s, desde las que consigue también acceso a las `IloIntVar`s involucradas. En definitiva, el objeto `model` es un almacén que contiene la conjunción de restricciones $\mathcal{FD}$'. Incluso disponemos del objeto `vars` para tener acceso directo a cada `IloIntVar` involucrada en alguna `IloConstraint` de `model`.

Mediante `model` y `vars`, en la traducción de una nueva restricción $\mathcal{FD}$ se puede acceder a los elementos que conforman la conjunción de restricciones $\mathcal{FD}$'. Sin embargo, no se puede acceder a los elementos que conforman la conjunción de restricciones $\mathcal{FD}$. Es decir, en el objetivo anterior `X #> Y, X #> 4, Z #> Y`, el predicado Prolog que gestionaba `Z #> Y` podía acceder a las variables lógicas $\mathcal{FD}$' contenidas en `vars`, y a las restricciones $\mathcal{FD}$' contenidas en `model`, pero no podía acceder a las variables lógicas $\mathcal{FD}$ `X` e `Y`, o a las restricciones $\mathcal{FD}$ `X #> Y` y `X #> 4`.

El fracaso sufrido en el primer intento de transmisión de la conjunción de restricciones $\mathcal{FD}$ a la aplicación ILOG CP, que impedía referenciar a una misma variable lógica $\mathcal{FD}$ involucrada en dos restricciones $\mathcal{FD}$ diferentes, pone de manifiesto que es preciso otra ruta para posibilitar la comunicación. Al gestionar `Z #> Y`, la única manera de conocer si `Z` está contenida en `vars`, así como la posición que ocupa en dicha lista es comparándola con `X` e `Y`, es decir, con las variables lógicas $\mathcal{FD}$ involucradas en la conjunción de restricciones $\mathcal{FD}$ traducida hasta el momento. Esto evidencia que es necesario crear el almacén que contenga la conjunción de restricciones $\mathcal{FD}$. Todo predicado Prolog que traduzca una nueva restricción $\mathcal{FD}$ debe tener acceso a este almacén.

En general debemos valorar que, debido a la complejidad de comunicar SICStus con C++, hay algunas funcionalidades que la aplicación ILOG CP no es capaz de resolver, por lo que estas funcionalidades deben ser resueltas en el componente $solve^{FD}$. En la práctica esto significa que cada predicado Prolog que gestiona una restricción $\mathcal{FD}$ debe resolver en los objetivos de su cuerpo funcionalidades que, siendo necesarias, no puedan ser resueltas con llamadas a funciones C++. La solución que adoptamos (y que permite integrar con éxito ILOG CP en $\mathcal{TOY}$) exige que cada predicado Prolog de $solve^{FD}$ tenga acceso a un conjunto de estructuras de datos. Estas estructuras de datos deben contener toda la información necesaria que permita a cada predicado Prolog resolver por si mismo las funcionalidades que la aplicación ILOG CP no es capaz de resolver.

Por ejemplo, necesitamos hacer explícita una lista con el conjunto de variables lógicas $\mathcal{FD}$ involucradas en las restricciones $\mathcal{FD}$ evaluadas hasta el momento en el objetivo $\mathcal{TOY}$.

Las variables lógicas $\mathcal{FD}$ se añaden a esta lista en el orden textual en que van apareciendo involucradas en restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$. Cada variable lógica $\mathcal{FD}$ contenida en una cierta posición `i` de esta lista contiene su `IloIntVar` asociada en la misma posición `i` de `vars`. Vemos que en este segundo intento esta lista juega el mismo papel que jugaba en el primer intento el `vector<SP_term_ref>`.

Cada predicado Prolog que gestione una restricción $\mathcal{FD}$ debe tener acceso a esta lista de variables lógicas $\mathcal{FD}$. Dicho predicado Prolog modifica el contenido de esta lista, añadiendo las nuevas variables lógicas $\mathcal{FD}$ que esta restricción $\mathcal{FD}$ involucra. Asimismo, y una vez que todas las variables lógicas $\mathcal{FD}$ involucradas en la restricción $\mathcal{FD}$ están contenidas en la lista, se pueden identificar los índices que estas variables lógicas $\mathcal{FD}$ ocupan, obteniendo automáticamente los índices de sus `IloIntVar` asociadas. Utilizamos esos índices para seleccionar las `IloIntVar`s involucradas en la nueva `IloConstraint`.

Por tanto, en la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ decidimos crear un almacén desde el que acceder en cada momento a la conjunción de restricciones $\mathcal{FD}$ que han sido traducidas. Reunimos esta conjunción de restricciones $\mathcal{FD}$ en un único elemento almacén `store(Restricciones,Variables)`, donde `Restricciones` y `Variables` representan la lista de restricciones $\mathcal{FD}$ y la lista de variables lógicas $\mathcal{FD}$ involucradas en alguna de las restricciones $\mathcal{FD}$. Cada predicado Prolog de $solve^{FD}$ que procesa una nueva restricción $\mathcal{FD}$ necesita acceder a este almacén `store(Restricciones,Variables)` para gestionar su restricción $\mathcal{FD}$. Como SICStus no permite el uso de variables lógicas globales, `store(Restricciones,Variables)` tiene que ser común a todos los predicados Prolog de $solve^{FD}$. La implementación de $\mathcal{TOY}(\mathcal{FD}s)$ incluía dos variables lógicas, `Cin` y `Cout`, utilizadas respectivamente como argumentos de entrada y de salida en todos los predicados Prolog de $solve^{FD}$. Esta variable `Cin` [9] surgió con la idea de representar las restricciones de desigualdad de Herbrand, y parece un buen lugar para almacenar el elemento `store(Restricciones,Variables)`.

La funcionalidad que permite al predicado Prolog `$#>` traducir su restricción `L #> R` es la siguiente:

1. Accede al contenido actual de la conjunción de restricciones $\mathcal{FD}$ y conjunción de restricciones $\mathcal{FD}$'. Para la conjunción de restricciones $\mathcal{FD}$' utiliza las funciones C++ que permiten el acceso a `model` y `vars`. Para la conjunción de restricciones $\mathcal{FD}$ utiliza su argumento de entrada `Cin` para extraer el elemento `store(Restricciones,Variables)`.

   Definimos un nuevo predicado Prolog `find_store_in_Cin` que busca y extrae el elemento `store` de `Cin`. Sus cláusulas son:

   ```
   %find_store_in_Cin/5(+Cin,-Vars,-Size,-Constraints,-Cout).

   (i)   find_store_in_Cin([],[],0,[],[]).

   (ii)  find_store_in_Cin([store(Vars,Size,Constraints)|Ys],Vars,Size,
                                                  Contraints,Ys) :- !.

   (iii) find_store_in_Cin([Y|Ys],Vars,Size,Contraints,[Y|R]) :-
           find_store_in_Cin(Ys,Vars,Size,Contraints,R).
   ```

   La cláusula `(i)` representa el caso base. Si se ha explorado toda la lista almacén `Cin` y no se ha encontrado el elemento `store`, entonces concluímos que `store` es vacío. Devolvemos `Variables = []`, `Tamaño = 0`, `Restricciones = []`.

La cláusula `(ii)` encuentra el elemento `store` en la cabeza de la lista. Retira el elemento del almacén, unifica `Variables`, `Tamaño` y `Restricciones` y no evalúa el almacén restante.

La cláusula `(iii)` no encuentra el elemento `store` en la cabeza de la lista. Continúa la búsqueda en el almacén aún no evaluado.

Por motivos de simpleza hemos omitido la presencia del valor entero `Size` en `store` durante la descripción de esta subsección. Este elemento representa el tamaño de la lista Variables. Se almacena por motivos de eficiencia, evitando así repetidas llamadas al predicado Prolog `length(Variables,Size)` en diferentes objetivos del cuerpo de `$#>`.

2. Traduce y transmite la restricción `L #> R`. Esto implica:

   2.1 Una traducción y transmisión de los argumentos `L` y `R` involucrados en la restricción.

   2.2 Una traducción y transmisión de la propia restricción `L #> R`.

   Modifica el contenido de la conjunción de restricciones $\mathcal{FD}$' mediante `model` y `vars`. Modifica el contenido de la conjunción de restricciones $\mathcal{FD}$ mediante `Restricciones` y `Variables`. Aunque las variables Prolog no son mutables, por simplicidad en la descripción en este trabajo hablamos permanentemente de una única variable `Restricciones` o `Variables`.

3. Almacena el nuevo contenido de las estructuras de datos `Restricciones` y `Variables`, para que otros predicados Prolog que gestionen futuras restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ tengan acceso a dichas estructuras.

Explicamos en detalle como realizar los puntos 2.1 y 2.2

Traducción y transmisión de los argumentos de la restricción $\mathcal{FD}$

Por cada argumento `Arg` de la restricción $\mathcal{FD}$ se genera como información un par de valores enteros `(IsVar,Value)`. `IsVar` es una variable lógica que se unifica con el valor 1 si el argumento es una variable lógica $\mathcal{FD}$, o con el valor 0 si el argumento es un valor entero. `Value` es una variable lógica que, en el caso de que `IsVar=1`, representa la posición que ocupa dicha variable lógica $\mathcal{FD}$ en la lista `Variables`. Si `IsVar=0` entonces `Value` representa el valor entero del argumento.

Definimos los predicados Prolog `is_var_in_vars_list` y `manage_contraint_argument`, que permiten generar el par `(IsVar,Value)` para cada argumento `Arg` de la restricción $\mathcal{FD}$.

El predicado `is_var_in_vars_list` busca una variable lógica $\mathcal{FD}$ en la lista `Variables`. Si la encuentra devuelve el índice asociado. Si no la encuentra devuelve el índice en que debería ser añadida. Como la numeración va de `0` a `n-1`, la nueva posición coincide con el tamaño actual de la lista. Si el tamaño es 3, la numeración actual va de `0` a `2`, y una nueva variable lógica $\mathcal{FD}$ debe ser añadida en la posición `3`.

```
%is_var_in_vars_list/4(+Var,+Vars,+EvaluatedPosition,-Index).

(i)   is_var_in_vars_list(_,[],E,E).
```

```
(ii)  is_var_in_vars_list(V,[X|_],E,E):-
        V == X,
        !.

(iii) is_var_in_vars_list(V,[X|Xs],E,I) :-
        NE is E+1,
        is_var_in_vars_list(V,Xs,NE,I).
```

El predicado `manage_constraint_argument` generar el par `(IsVar,Value)` para un argumento `Arg` de la restricción $\mathcal{FD}$.

```
%manage_constraint_argument/7(+Arg,+Vars,+Size,-NewVars,-NewSize,
                                              -IsVar,-Value).

(i) manage_constraint_argument(Arg,Vars,Size,Vars,Size,0,V) :-
        integer(Arg),
        !.

(ii) manage_constraint_argument(Arg,Vars,Size,NewVars,NewSize,1,I) :-
        var(Arg),
        is_var_in_vars_list(Arg,Vars,0,Index),
        ( Index == Size ->
            (append(Vars, [Arg], NewVars),
            NewSize is Size+1,
            create_new_Var
            )
        ;
            (NewVars = Vars,
            NewSize = Size
            )
        ),
        !.
```

Si el argumento `Arg` es el valor entero `k`, se genera el par `(0,k)`.
Si el argumento `Arg` es una variable lógica $\mathcal{FD}$ contenida en la posición `i` de `Variables`, se genera el par `(1,i)`.
Si un argumento `Arg` es una variable lógica $\mathcal{FD}$ no contenida en la lista `Variables`, entonces podemos concluir que se trata de una nueva variable lógica $\mathcal{FD}$ no tratada hasta ahora en el objetivo $\mathcal{TOY}$. Se añade esta variable `Arg` al final de `Variables`. Esto provoca una inconsistencia con `vars`, ya que ahora `Variables` tiene un elemento más que `vars`. Esto indica que la variable `Arg`, contenida en la última posición de `Variables`, no tiene una `IloIntVar arg` asociada en `vars`. Por ello se llama al predicado `create_new_Var`, implementado en la función C++ `create_new_Var`. Esta función cre una nueva `IloIntVar arg` y la añade al final del vector `vars`. Esto produce una asociación entre `Arg`, contenida en la última posición de `Variables` y `arg`, contenida en la última posición de `vars`.

Traducción y transmisión de la propia restricción $\mathcal{FD}$

La restricción `L #> R` debe ser almacenada en la lista `Restricciones` de la conjunción de restricciones $\mathcal{FD}$. Esto provoca una inconsistencia con la conjunción de restricciones $\mathcal{FD}$' contenida en `model`. Para recuperar la consistencia traducimos y transmitimos la restricción `L #> R` a la aplicación genérica ILOG CP. Para ello creamos una segunda versión de la función C++
`post_greater(long IsVarL, long ValueL, long IsVarR, long ValueR)`. Esta función crea una nueva `IloConstraint`. Utiliza los pares de información `(IsVar,Value)` generados para identificar los valores enteros o las `IloIntVar` de `vars` involucradas en la nueva `IloConstraint`. Añade esta nueva `IloConstraint` a `model`.

Implementación del predicado Prolog `$#>`

Presentamos una primera versión del predicado Prolog `$#>`, aún muy incipiente, que nos permitirá traducir y transmitir correctamente la conjunción de restricciones $\mathcal{FD}$ del objetivo que estamos viendo como ejemplo.

```
% $#>/5(+L,+R,-Out,+Cin,-Cout).

(0)  $#>(L, R, Out, Cin, [store(Vars2,Size2,[HL>HR|Constraints])|Cinter1]):-
(1)    find_store_in_Cin(Cin,Vars,Size,Constraints,Cinter1),
(2)    manage_constraint_argument(L,Vars,Size,Vars1,Size1,IsL,ValL),
(3)    manage_constraint_argument(R,Vars1,Size1,Vars2,Size2,IsR,ValR),
(4)    post_greater(IsL,ValL,IsR,ValR).
```

La línea `(0)` contiene la definición del predicado Prolog `$#>`. Esta definición incluye a `L` y `R` como argumentos de entrada. Ambos representan los argumentos izquierdo y derecho del operador aritmético *mayor*. El argumento `Out` se utiliza para la reificación de la restricción $\mathcal{FD}$. Si `Out=true` se impone al resolutor la restricción `L #> R`, mientras que si `Out=false` se impone al resolutor la restricción `L #<= R`. La reificación queda fuera del alcance de este trabajo, si bien representa una de las tareas a realizar como trabajo futuro. Los otros argumentos son `Cin` y `Cout`, de entrada y salida, respectivamente. Éstos contienen el estado del almacén mixto de restricciones al inicio y finalización de la ejecución del predicado `$#>`. El predicado utiliza el almacén `Cin` que se le suministra como entrada, realiza sobre este almacén las modificaciones necesarias para la gestión de la restricción $\mathcal{FD}$ y devuelve el nuevo estado del almacén en `Cout`. Cada una de las modificaciones necesarias representa un estado intermedio del contenido del almacén en la gestión de la restricción $\mathcal{FD}$. Caracterizamos a esos estados intermedios con variables `Cinter` n.

En `(1)` se accede a `Cin` para buscar y extraer el contenido actual de la conjunción de restricciones $\mathcal{FD}$. En `(2)` y `(3)` traduce y transmite los argumentos de la restricción $\mathcal{FD}$ y en `(4)` traduce y transmite la propia restricción $\mathcal{FD}$. Además almacena en `Cout` el nuevo estado de la conjunción de restricciones $\mathcal{FD}$ tras la traducción y transmisión de la restricción $\mathcal{FD}$ que ha gestionado.

Describimos el comportamiento sobre el objetivo mostrado en esta subsubsección: `X #> Y, X #> 4, Z #> Y`.

Al comenzar la evaluación del objetivo, la conjunción de restricciones $\mathcal{FD}$ es vacía, por lo que el elemento `store(Restricciones,Variables)` no está contenido en `Cin`. Asimismo la conjunción de restricciones $\mathcal{FD}$' es también vacía, por lo que `model` y `vars` no contienen ningún elemento.

La primera restricción del objetivo se gestiona mediante el predicado Prolog
`$#>(X,Y,O,[],[store([X#>Y],[X,Y])])`.

1. Este predicado busca en `Cin` el término `store(Restricciones,Variables)`. Al no encontrarlo, unifica `R` = [] y `V` = [] para construir posteriormente el término `store(R,V)` y almacenarlo en `Cout`.

2. Gestiona la restricción `X #> Y`.

   2.1 Evalúa el argumento `X`.

   - Identifica a `X` como una variable lógica $\mathcal{FD}$.
   - Busca `X` en la lista `V` = [].
   - Como `X` no pertenece a `V` hay que añadirla al final de la lista. Mantenemos por simplicidad la misma nomenclatura para la variable lógica `V`. Ahora `V` = [`X`]. Esto provoca una inconsistencia con `vars = []`. Es preciso llamar a la función C++ `create_new_Var`, que crea una nueva `IloIntVar x` que añade a `vars` = [`x`]. Se obtiene que `X` y `x` están asociados por ocupar el mismo índice de `V` y `vars`. Se genera la siguiente información relacionada con el argumento `X`: `(1,0)` (recordemos que la primera componente indica que `X` es una variable lógica $\mathcal{FD}$ y la segunda que se encuentra en la posición 0 de la lista).

   2.2 Evalúa el argumento `Y`.

   - Identifica a `Y` como una variable lógica $\mathcal{FD}$.
   - Busca `X` en la lista `V` = [`X`].
   - Como `Y` no pertenece a las variables lógicas $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$ hay que añadirla al final de la lista. Ahora `V` = [`X`,`Y`]. Esto provoca una inconsistencia con `vars = [x]`. Es preciso llamar a la función C++ `create_new_Var`, que crea una nueva `IloIntVar y` que añade a `vars` = [`x`,`y`]. Se genera la siguiente información relacionada con el argumento `Y`: (1,1).

   2.2 Gestiona la propia restricción $\mathcal{FD}$ `X #> Y`. Almacena `X #> Y` en `R` = []. Ahora `R` = [`X #> Y`]. Esto provoca una inconsistencia con `model = []`. Es preciso llamar a la función C++ `post_greater`, que crea una nueva `IloConstraint c0` utilizando los pares (1,0) y (1,1) de los argumentos `X` e `Y`. Por lo tanto, se crea la `IloConstraint c0 = vars[0] > vars[1]`.

3. Tras la resolución de la restricción $\mathcal{FD}$, tanto la conjunción de restricciones $\mathcal{FD}$ y la conjunción de restricciones $\mathcal{FD}$' son consistentes. Es necesario almacenar el nuevo contenido de la conjunción de restricciones $\mathcal{FD}$ para que otros predicados Prolog tengan acceso a `R` y `V` en la resolución de futuras restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$. Almacenamos `store(R,V)` en la variable de salida `Cout` del predicado Prolog `$#>`. El predicado Prolog que gestione la siguiente restricción $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ encontrará el elemento `store([ X #> Y ],[X, Y])` en su variable de entrada `Cin`.

Vemos que este segundo intento resulta satisfactorio para transmitir la conjunción de restricciones $\mathcal{FD}$ a la aplicación ILOG CP.

En la Figura 2.8 mostramos gráficamente el contenido del elemento `store` que recibe el predicado Prolog `$#>(Z,Y,O,Cin,Cout)`. Este predicado Prolog gestiona la restricción $\mathcal{FD}$ `Z #> Y`. La Figura 2.9 muestra la forma en que se realiza la gestión del argumento Z. La Figura 2.10 muestra la gestión de la propia restricción $\mathcal{FD}$ `Z #> Y`.

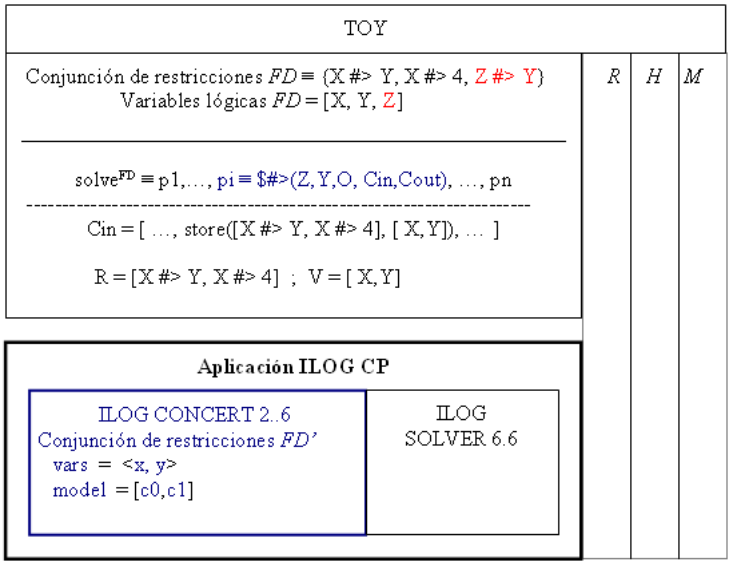

Figura 2.8: Contenido del almacén al inicio de la resolución de `Z` mayor que `Y`.

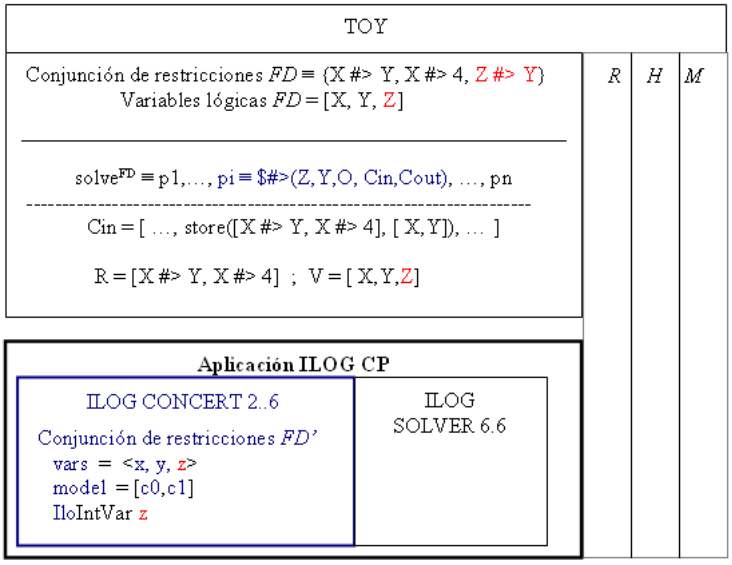

Figura 2.9: Gestión del argumento `Z`.

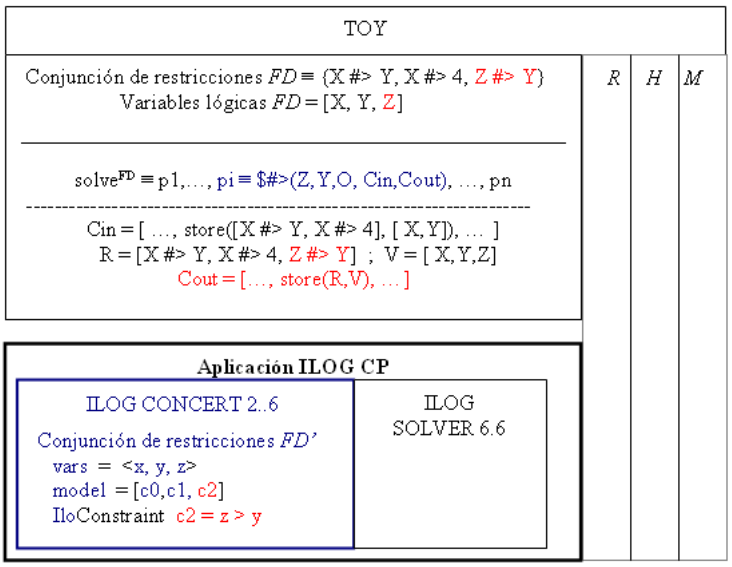

Figura 2.10: Gestión de la restricción $\mathcal{FD}$ `Z` mayor que `Y`.

2.2.2. Resolución de la conjunción de restricciones $\mathcal{FD}$'

La aplicación genérica ILOG CP resuelve la conjunción de restricciones $\mathcal{FD}$' que le es transmitida desde el objetivo $\mathcal{TOY}$. Esta conjunción de restricciones $\mathcal{FD}$' está explícitamente representada en `model`. Para conseguir resolver la conjunción de restricciones, la aplicación genérica ILOG CP utiliza un resolutor de restricciones concreto. En este trabajo utilizamos exclusivamente el resolutor de restricciones de dominios finitos de la biblioteca ILOG Solver 6.6, implementado en la clase `IloSolver`. Crearemos por lo tanto un objeto `solver` de tipo `IloSolver`. ILOG permite utilizar diferentes resolutores de restricciones (incluso resolutores que operen sobre diferentes dominios de restricciones) para resolver una misma conjunción de restricciones $\mathcal{FD}$' contenida en `model` aplicando técnicas de resolución diferentes.

Para aplicar esas técnicas de resolución concretas, cada resolutor de restricciones debe primero traducir la conjunción de restricciones $\mathcal{FD}$' a una nueva conjunción de restricciones $\mathcal{FD}$". Esta nueva conjunción de restricciones $\mathcal{FD}$" es equivalente la conjunción de restricciones $\mathcal{FD}$'. Sin embargo, la estructura interna de sus objetos está orientada a las técnicas de resolución concretas que el resolutor de restricciones vaya a aplicar.

Esta subsección describe las etapas en que `solver` traduce y resuelve el contenido de `model`.

1. Traducción.

   - Para traducir el contenido de `model`, `solver` itera por todas las `IloConstraints` que contiene `model`. Crea un nuevo objeto `IlcConstraint c'` o `IlcIntVar v'` por cada objeto `IloConstraint c` o `IloIntVar v`, respectivamente, involucrado en la conjunción de restricciones $\mathcal{FD}$'.
     - Cada `IloConstraint c` involucra un conjunto de `IloIntVars v1,...,vn`. Crea un nuevo objeto `IlcIntVar v'` asociado a cada `IloIntVar v` que no hubiera sido previamente traducida.

    - Crea un nuevo objeto `IlcConstraint c'` asociado a cada objeto `IloConstraint c` que no hubiera sido previamente traducido.
  - Todo objeto `IloConstraint c` o `IloIntVar v` que no esté directa o indirectamente contenido en `model` no se traduce. Este último punto cobrará especial relevancia cuando describamos la implementación del backtracking en la sección 3.3.
  - El resolutor `solver` permite acceder a cada objeto `IlcConstraint c'` e `IlcIntVar v'` a través de su objeto `IloConstraint c` e `IloIntVar v` asociado, respectivamente. Para ello proporciona los métodos `IlcIntVar solver.getIntVar(IloIntVar v)` e `IlcConstraint solver.getConstraint(IloConstraint c)`. Este modo de acceso es suficiente para nuestros intereses, por lo que no utilizaremos contenedores adicionales, como en el caso de `vars` para las `IloIntVar`s de la conjunción de restricciones $\mathcal{FD}$'.

2. Resolución.

  - Se permite la propagación de cualquier subconjunto de las `IlcConstraint`s que conforman la conjunción de restricciones $\mathcal{FD}$". Esta propagación podará valores de los objetos `IlcIntVar`. Sin embargo, los objetos `IloIntVar` mantendrán su dominio intacto. Para conocer el resultado de la propagación, hay que acceder a los objetos `IlcIntVar` a través de sus objetos `IloIntVar` asociados.
  - Se permite utilizar procedimientos de búsqueda que etiqueten valores sobre cualquier subconjunto de las `IlcIntVar`s involucradas en la conjunción de `IlcConstraint`s.
  - Salvo cuando utilicemos explícitamente restricciones primitivas $\mathcal{FD}$ de etiquetado, que explicaremos en la sección 3.4, nuestra intención es obtener una solución por intervalos a la conjunción de restricciones $\mathcal{FD}$ propuesta, por lo que sólo utilizaremos la técnica de propagación de restricciones.
  - Tras cada tentativa de resolución, bien sea sólo con propagación de restricciones o con propagación más procedimiento de búsqueda, `solver` dispone de un método para determinar si la conjunción de restricciones $\mathcal{FD}$" es o no satisfactible. Denominamos forma resuelta al conjunto de:
    - Las `IlcIntVar`s, con sus dominios posiblemente podados tras las técnicas de resolución aplicadas.
    - Las `IlcConstraint`s.

Ahora son ya tres las conjunciones de restricciones que deben ser consistentes, como puede verse en la Figura 2.11

a) La conjunción de restricciones $\mathcal{FD}$ explicitada en `store(Restricciones,Variables)` de $solve^{FD}$.

b) La conjunción de restricciones $\mathcal{FD}$' contenida en `model`.

c) La conjunción de restricciones $\mathcal{FD}$" creada explícitamente por `solver` a partir del contenido de `model`.

Es importante recordar que $\mathcal{TOY}$ gestiona las restricciones $\mathcal{FD}$ de una en una, a medida que van siendo detectadas por el cálculo de estrechamiento. De este modo el predicado

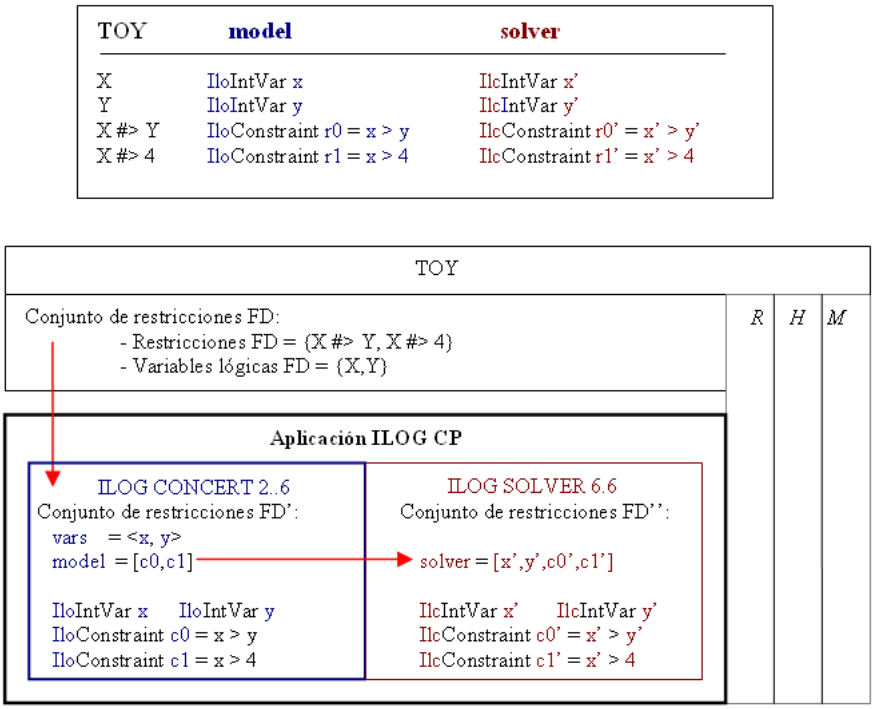

Figura 2.11: Restricciones en $solve^{FD}$, `model` y `solver`.

Prolog que gestiona cada restricción $\mathcal{FD}$ traduce en primer lugar la restricción $\mathcal{FD}$ a la conjunción de restricciones $\mathcal{FD}$' explicitada en `model`. En nuestro caso, el predicado Prolog `$#>` llamaba al predicado `post_constraint` para realizar esta tarea. Con esto se alcanzaba de nuevo la consistencia entre la conjunción de restricciones $\mathcal{FD}$ y la conjunción de restricciones $\mathcal{FD}$'. Definimos una tercera versión de la función C++ `post_constraint(long IsL, long ValL, long IsR, long ValR, long* feasible)`. Esta nueva función se encarga de:

- Traducir la restricción $\mathcal{FD}$ `C` a su `IloConstraint c` equivalente. Añadir `c` a `model`.
- Restablecer la consistencia entre la nueva conjunción de restricciones $\mathcal{FD}$' y la conjunción de restricciones $\mathcal{FD}$" traduciendo todo objeto `o` contenido en `model` a su objeto `o'` equivalente orientado al resolutor `solver`.
- Encontrar una forma resuelta a la nueva conjunción de restricciones $\mathcal{FD}$" contenida en `solver` e indicar mediante el valor booleano `feasible` si esta forma resuelta ha podido ser encontrada o no.

Como describimos en la subsección 1.2.1 ILOG proporciona los métodos `solver.extract(model)` y `solver.propagate()`. El primero itera por todas las `IloConstraints` de `model` identificando los objetos que no han sido traducidos por `solver`. Debido a que desde la última traducción de `solver` sólo hemos añadido una nueva `IloConstraint` el proceso resulta un tanto ineficiente. Algo similar ocurre con la propagación, ya que `solver.propagate()` propaga todas las `IlcConstraints` contenidas en `solver`. Desde la última vez que se obtuvo una forma resuelta tan sólo se ha añadido una nueva `IlcConstraint`. Es seguro que

hasta que la nueva `IlcConstraint` que acabamos de añadir no propague no lo hará ninguna de las `IlcConstraints` previamente añadidas. Por tanto, no parece razonable utilizar el método `solver.propagate()`, que intenta propagar todas las `IlcConstraints`.

Esta propuesta parece consistente con el tipo de uso que concibe ILOG. El método de trabajo para el que está concebido ILOG aisla las fases de modelado y resolución de un problema. Por ello está pensado para un modelado completo de la conjunción de restricciones $\mathcal{FD}$'. Sólo tras finalizar esta tarea, el resolutor traduce la conjunción de restricciones $\mathcal{FD}$' a una conjunción de restricciones $\mathcal{FD}$" equivalente. Posteriormente el resolutor propaga sus `IlcConstraints` asociadas a las `IloConstraints` de `model`.

Sin embargo, nuestra aproximación desde el sistema $\mathcal{TOY}$ combina constantemente las fases de traducción y resolución, aplicadas para cada nueva restricción $\mathcal{FD}$ que es detectada por el cálculo de estrechamiento. Para una aplicación más eficiente de la traducción y resolución de cada nueva restricción, utilizamos el método alternativo `int solver.propagate(IloConstraint)`. Este método se centra en propagar únicamente la `IloConstraint` propuesta, con lo que evitamos la posible ineficiencia en la resolución. Como puede observarse, el propio método requiere una `IloConstraint` como argumento, y no una `IlcConstraint`, como cabría esperar. Esto produce que antes de proceder a su propagación, `solver` se encargue de traducir exclusivamente el contenido de esta `IloConstraint`, evitando así iteraciones innecesarias por las restantes `IloConstraint` contenidas en `model`. Además el método devuelve un valor entero indicando si ha podido completar la propagación encontrando una nueva forma resuelta, o si por el contrario la nueva conjunción de restricciones $\mathcal{FD}$" es insatisfactible. Utilizamos ese valor entero como el entero `feasible` computado por la función `post_greater`.

En las Figuras 2.12 y 2.13 puede verse un ejemplo acerca de la eficacia en la estrategia de propagación. Supongamos el objetivo `X #> Y, X #> Y #+ 1`, donde las variables lógicas $\mathcal{FD}$ tienen un dominio entre los valores $0 \ldots 2$. La segunda restricción `X #> Y #+ 1` no es primitiva, por lo que el cálculo de estrechamiento la descompone en `Y #+ 1 == _Z, X #> _Z`. En la subsubsección 2.2.4 se estudiará en más detalle la forma en que se realiza esta descomposición. La Figura 2.12 muestra el estado de la aplicación ILOG CP tras la traducción y resolución de las restricciones `X #> Y, Y #+ 1 == _Z`. La Figura 2.13 detalla las modifaciones que ocurren en la aplicación ILOG CP con la traducción y resolución de la restricción `X #> _Z`.

La estrategia de propagación que ILOG utiliza por defecto efectuaría `solver.extract(model)` y `solver.propagate()`. La extracción de `model` accedería a `c0` y `c1`, que detectaría como traducidos. Posteriormente accedería a `c2`, creando el nuevo objeto `IlcConstraint c2'` asociado a `c2`. La propagación intentaría propagar `c0'` y `c1'`, sin efecto alguno. A continuación propagaría `c2'`, que vincularía `x'= 2` y `z = 1`. Esto despertaría la propagación de `c1'`, vinculando `c1 = 0`.

Con esto concluimos que es las únicas operaciones que producen cambios son la extracción de `c2` y la propagación de `c2'` (pudiendo despertar la propagación de otras `IlcConstraints`). Estas dos operaciones son efectuadas mediante la instrucción `solver.propagate(c2)`, que es la única instrucción que efectuamos en nuestra estrategia de propagación.

Aunque será explicado con mayor detalle en la sección 3.3, adelantamos aquí que esta estrategia sólo se seguirá ante cambios monótonos en `model`. Cuando el backtracking produzca cambios destructivos en el modelo será preciso realizar de nuevo una traducción del contenido completo de `model`, para que `solver` pueda descartar los objetos sobre los que ya no es necesario que siga trabajando.

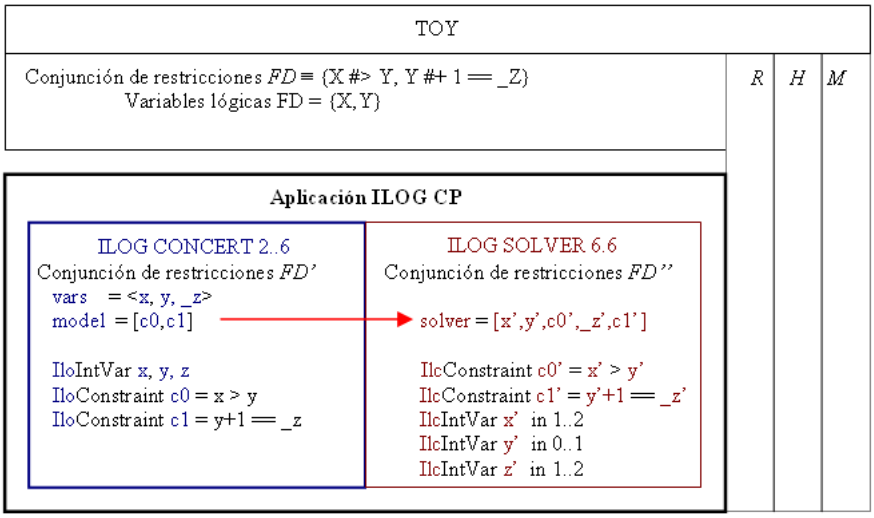

Figura 2.12: Cambios producidos por una nueva restricción $\mathcal{FD}$.

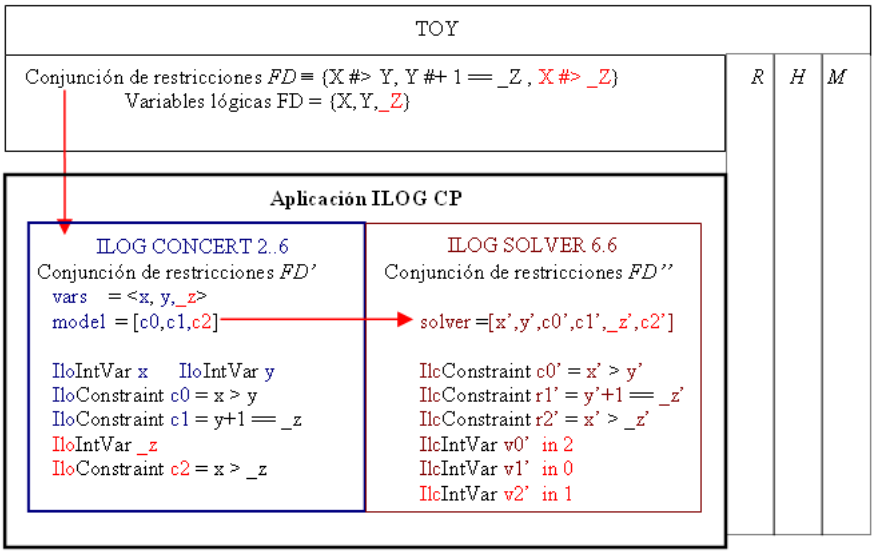

Figura 2.13: Cambios producidos por una nueva restricción $\mathcal{FD}$.

Retomando de nuevo el hecho de que ahora son tres las conjunciones de restricciones que deben ser consistentes, parece sensato plantearse la conveniencia de utilizar la fase de traducción de cada restricción $\mathcal{FD}$ a una restricción $\mathcal{FD}$', en contra de un nuevo intento de traducción que traduzca las restricciones $\mathcal{FD}$ directamente sobre `solver`, evitando el paso por `model`. Con ello sólo utilizaríamos la conjunción de restricciones $\mathcal{FD}$ y la conjunción de restricciones $\mathcal{FD}$". Además, una de las ventajas de utilizar `model` radica en que puede ser extraído a varios resolutores, pero sin embargo en este trabajo utilizamos un único resolutor, por lo que no hacemos uso de esa funcionalidad del sistema. Sin embargo decidimos utilizar la fase de traducción a `model` por las siguientes razones:

1. Simpleza: modelar directamente sobre `solver` es más complejo.
2. Estandarización: es la opción elegida por defecto en ILOG y todos sus ejemplos siguen este formato.
3. Eficiencia: a la hora de modificar el contenido de la conjunción de restricciones $\mathcal{FD}$ tras la aplicación de backtracking.

El primer y tercer punto serán tratados con mayor detalle en la sección 3.3.

Por último mostramos la nueva versión del predicado `$#>`, que permite realizar las fases de traducción y propagación de cada restricción `L #> R`. Vemos que en `(4)` se incluye ahora el nuevo argumento `Feasible`, que indica si se ha podido encontrar o no una forma resuelta a la conjunción de restricciones $\mathcal{FD}$".

```
% $#>/5(+L,+R,-Out,+Cin,-Cout).

(0)  $#>(L, R, Out, Cin, [store(Vars2,Size2,[HL>HR|Constraints])|Cin1]):-
(1)    find_store_in_Cin(Cin,Vars,Size,Constraints,Cin1),
(2)    manage_constraint_argument(L,Vars,Size,Vars1,Size1,IsL,ValL),
(3)    manage_constraint_argument(R,Vars1,Size1,Vars2,Size2,IsR,ValR),
(4)    post_greater(IsL,ValL,IsR,ValR,Feasible).
```

2.2.3. Acceso a la solución obtenida

Para completar el proceso de resolución de una conjunción de restricciones $\mathcal{FD}$ utilizando el sistema de restricciones externo ILOG CP necesitamos tener acceso a la forma resuelta que ILOG CP obtiene para su conjunción de restricciones $\mathcal{FD}$" planteada. La información que precisamos obtener es diferente según el punto del proceso de resolución del objetivo $\mathcal{TOY}$ en el que nos encontremos.

- La adición de una nueva restricción $\mathcal{FD}$ a la conjunción de restricciones $\mathcal{FD}$ planteada hasta el momento puede provocar que esta nueva conjunción de restricciones $\mathcal{FD}$ sea insatisfactible. Precisamente la estrategia que utilizamos de traducir y propagar cada nueva restricción $\mathcal{FD}$ en la aplicación ILOG CP nos permite detectar esa insatisfactibilidad por cada nueva restricción $\mathcal{FD}$ procesada. Por ejemplo, en la gestión de una nueva restricción `L #> R` basta con exigir que la llamada a `post_greater` sea `post_greater(IsL,ValL,IsR,ValR,1)`. Así, si la conjunción de restricciones $\mathcal{FD}$" no es satisfactible se producirá un fallo en el predicado `$#>`. Si la conjunción es satisfactible la llamada a `post_greater` no produce fallo y la evaluación del objetivo $\mathcal{TOY}$ continúa. Esto agiliza la búsqueda de soluciones, permitiendo la detección temprana de ramas de cómputo fallidas.

Por ejemplo, en el objetivo `X #> 2, X #< 1, ...` no es preciso evaluar el objetivo restante para saber que la respuesta será fallo, ya que la conjunción de restricciones `X #> 2, X #< 1` es insatisfactible al no encontrar ningún posible valor del dominio de `X` que haga cierta la conjunción.

- Si la evaluación de un objetivo $\mathcal{TOY}$ ha concluido con éxito, es preciso mostrar al usuario la respuesta computada por el sistema. Para la conjunción de restricciones $\mathcal{FD}$ planteada en el objetivo $\mathcal{TOY}$ esta respuesta consiste en:

  1. Mostrar para cada variable lógica $\mathcal{FD}$ los valores de su dominio que no han sido podados.
  2. Mostrar cada restricción $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$ que incluya al menos una variable sin instanciar (restricción no trivial).

  El predicado Prolog `showAnswer` se encarga de mostrar al usuario por pantalla la respuesta computada al objetivo $\mathcal{TOY}$ propuesto. Para mostrar esta información, este predicado Prolog necesita acceder a las `IlcIntVar`s e `IlcConstraint`s contenidas en `solver`. El procedimiento que permite acceder a cualquier elemento de la conjunción de restricciones $\mathcal{FD}$", sobre el que trabaja `solver`, es el siguiente:

  a) Este predicado Prolog no pertenece al sistema de restricciones $\mathcal{FD}$, pero igualmente recibe como argumento de entrada la variable `Cin`, por lo que tiene acceso al elemento `store(Restricciones,Variables)`. Mediante las listas `Restricciones` y `Variables` dispone del acceso a todos los elementos que conforman la conjunción de restricciones $\mathcal{FD}$ sobre el que tenemos que mostrar la solución obtenida.

  b) Las listas `Restricciones` y `Variables` tienen sus listas asociadas en la aplicación ILOG CP, como vimos en la subsección 2.2.1. De este modo, cualquier variable lógica $\mathcal{FD}$ contenida en la posición `i` de `Variables` contiene su `IloIntVar` asociada en `vars[i]`, que permite un acceso directo al objeto.

     Del mismo modo, cualquier restricción $\mathcal{FD}$ contenida en la posición `i` de `Restricciones` contiene su `IloConstraint` asociada en la `i`-esima posición de `model`. Se puede acceder a esta `IloConstraint` aplicando un iterador para acceder en orden a los elementos que están contenidos en `model`. El acceso, como se ve, es algo más ineficiente.

  c) Utilizando los métodos `IlcIntVar solver.getIntVar(IloIntVar)` e `IlcConstraint solver.getConstraint(IloConstraint)` cada elemento de la conjunción de restricciones $\mathcal{FD}$' puede acceder a su elemento asociado del conjunto de restricciones $\mathcal{FD}$", como describimos en la subsección 2.2.2.

  El predicado Prolog `showAnswer` muestra el dominio de cada una de las variables lógicas $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$. Para obtener estos dominios:

  i) El predicado Prolog `showAnswer` realiza una llamada a la función C++ `get_vars_domain`, pasándole como argumento de entrada la lista `Variables` contenida en `Cin` y recibiendo como argumento computado por la función una nueva lista de listas `DominioVariables`. Para cada `IloIntVar` de `vars`, la función C++ accede a su `IlcIntVar` asociada. Una vez que tiene acceso a ésta, la función computa una lista de pares, que representa los distintos intervalos que puede tomar dicha `IlcIntVar` en su dominio.

- Para una `IlcIntVar` cuyo dominio está acotado a un valor concreto `k`, computa la lista cuyo único par es (`k`,`k`).
- Para una `IlcIntVar` cuyo dominio se compone de un conjunto de intervalos, donde cada intervalo contiene todos los valores enteros entre cotas mínima y máxima, se computa una lista con un par por cada intervalo, donde cada par contiene la cota mínima y la máxima. Por ejemplo, para una variable cuyo dominio contiene todos los valores enteros comprendidos entre 0 y 5 a excepción del 3, la lista computada es [(0,2), (4,5)].

Por tanto, creamos una lista de pares para cada `IlcIntVar` asociada a la `IloIntVar` contenida en `vars[i]`. Insertamos esta lista de pares en la posición `i` de la lista `DominioVariables`, que devolvemos al predicado Prolog `showAnswer`.

ii) Utilizando la lista `DominioVariables`, el predicado Prolog se encarga de imprimir por pantalla el dominio de todas las variables lógicas $\mathcal{FD}$.

En la sección 3.2 explicaremos con todo detalle la forma en que un predicado Prolog unifica ciertas variables lógicas $\mathcal{FD}$ contenidas en la lista `Variables` a valores enteros. Este hecho se produce para recuperar la consistencia con sus `IlcIntVar`s asociadas que han sido vinculadas por el resolutor `solver` tras la propagación de una nueva `IlcConstraint`. Esta unificación a un valor entero de la variable lógica $\mathcal{FD}$ provoca que todas sus apariciones sean reemplazadas por dicho valor entero, entre ellas toda aparición de la variable lógica $\mathcal{FD}$ en alguna restricción $\mathcal{FD}$ de la lista `Restricciones`. Una restricción $\mathcal{FD}$ pasa a ser trivial si todas sus variables lógicas $\mathcal{FD}$ involucradas son vinculadas a valores enteros.

Para mostrar las restricciones no triviales, el predicado Prolog `showAnswer` accede al contenido de la lista `Restricciones`. Utiliza el predicado Prolog `term_variables` que, dado un término Prolog, devuelve las variables lógicas que pertenecen a dicho término. Con esto se evita imprimir por pantalla las restricciones básicas (ground).

El objetivo de poder obtener las restricciones $\mathcal{FD}$ en un formato simplificado nos llevó a estudiar en detalle los métodos del API de la clase `IlcConstraint`. Sin embargo pudimos constatar que `solver` no transforma la conjunción de `IlcConstraint`s sobre las que trabaja en una nueva conjunción que represente la estructura simplificada de la solución. Cada `IlcConstraint` mantiene su misma estructura, incluso si se ha convertido en una `IlcConstraint` trivial. Por tanto, utilizar la información contenida en las `IlcConstraint`s no nos ofrece ninguna ventaja a la hora de imprimir por pantalla las restricciones $\mathcal{FD}$. Mantenemos pues la implementación que accede al contenido de `Restricciones` e imprime toda restricción $\mathcal{FD}$ no trivial.

2.2.4. Gestión de una restricción $\mathcal{FD}$ compuesta

Cuando el cálculo de estrechamiento perezoso detecta una restricción $\mathcal{FD}$ en el objetivo $\mathcal{TOY}$ cuya estructura es `L #> R`, transfiere el control de programa al predicado Prolog `$#>(+L,+R,-Out,+Cin,-Cout)` para que se encargue de su traducción y resolución en la aplicación genérica ILOG CP. Para poder realizar las tres fases descritas en las subsecciones anteriores, el predicado `$#>` precisa dos requisitos:

a) Los argumentos del operador aritmético *mayor*, `L` y `R`, deben ser evaluados o bien a una variable lógica $\mathcal{FD}$ o bien a un valor entero.

b) El predicado debe tener acceso a la conjunción de restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ planteado hasta el momento. Para ello debe acceder al elemento `store(Variables,Restricciones)`, contenido en `Cin`.

El requisito b) ya ha sido explicado. Lo resolvemos utilizando el predicado `find_store_in_Cin`, que permite acceder al contenido de la conjunción de restricciones $\mathcal{FD}$ antes de gestionar la nueva restricción $\mathcal{FD}$. Tras gestionar la nueva restricción $\mathcal{FD}$, se almacena el nuevo estado de la conjunción de restricciones $\mathcal{FD}$ en la variable `Cout`.

Para cumplir el requisito a) reutilizamos el predicado `hnf`, contenido en la distribución de $\mathcal{TOY}(\mathcal{FD}s)$. En primer lugar mostramos el código definitivo del predicado `$#>`. A continuación explicamos en detalle el predicado `hnf`, que nos va a permitir utilizar restricciones $\mathcal{FD}$ compuestas en el sistema $\mathcal{TOY}(\mathcal{FD}i)$.

```
% $#>/5(+L,+R,-Out,+Cin,-Cout).

(0)  $#>(L, R, Out, Cin, [store(Vars2,Size2,[HL>HR|Constraints])|Cinter3]):-
(1)    hnf(L, HL, Cin, Cinter1),
(2)    hnf(R, HR, Cinter1, Cinter2),
(3)    find_store_in_Cin(Cinter2,Vars,Size,Constraints,Cinter3),
(4)    manage_constraint_argument(HL,Vars,Size,Vars1,Size1,IsHL,ValHL),
(5)    manage_constraint_argument(HR,Vars1,Size1,Vars2,Size2,IsHR,ValHR),
(6)    post_greater(IsHL,ValHL,IsHR,ValHR,1).
```

Las líneas `(1)` y `(2)` utilizan el predicado Prolog `hnf(+Arg, +Hnf_Arg, +Cin, -Cout)` para calcular la forma normal de cabeza de los argumentos `L` y `R` de la restricción $\mathcal{FD}$ `$#>`. Esto es necesario ya que la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ exige que toda restricción $\mathcal{FD}$ compuesta detectada en el objetivo no sea impuesta directamente sobre la aplicación genérica ILOG CP, sino que se descomponga en un conjunción de restricciones $\mathcal{FD}$ primitivas. Una restricción $\mathcal{FD}$ de aridad n es primitiva si cada uno de sus n argumentos es, o bien una variable lógica $\mathcal{FD}$ o bien un número entero. Imponiendo esta conjunción de restricciones $\mathcal{FD}$ sobre la aplicación genérica ILOG CP se consigue el mismo efecto que imponiendo la restricción $\mathcal{FD}$ compuesta inicial.

Mediante cada llamada al predicado Prolog `hnf(+Arg, +Hnf_Arg, +Cin, -Cout)` el cálculo de estrechamiento computa la forma normal de cabeza del argumento `Arg`, al tiempo que impone al resolutor toda restricción $\mathcal{FD}$ primitiva que se detecta al computar la forma normal de cabeza de `Arg`. Para gestionar una restricción $\mathcal{FD}$ de aridad n, un predicado Prolog debe realizar una llamada a `hnf(+Arg, +Hnf_Arg, +Cin, -Cout)` por cada uno de sus argumentos. De este modo se asegura de:

- La restricción $\mathcal{FD}$ con la que trabaja es primitiva.
- Toda restricción $\mathcal{FD}$ primitiva que se detecte al computar la forma normal de cabeza del argumento es también impuesta al resolutor.

Veámoslo sobre el siguiente ejemplo. Supongamos un objetivo $\mathcal{TOY}$ formado únicamente por la la restricción `(2#*X1 #+ 3#*X2) #> 14`. El cálculo de estrechamiento perezoso detecta esta restricción $\mathcal{FD}$, que transfiere al predicado Prolog `$#>` para su gestión. Concretamente, el predicado Prolog recibe los siguientes argumentos `$#>(2#*X1 #+ 3#*X2, 14, Out, Cin, Cout)`. Para que `$#>` pueda gestionar directamente esta restricción $\mathcal{FD}$, la restricción debe ser primitiva, es decir, `L` y `R` deben ser, o

bien una variable lógica o un valor entero. Aunque `R` es un valor entero, `L` no se encuentra en ninguna de las dos opciones permitidas. Por lo tanto, la restricción `(2#*X1 #+ 3#*X2) #> 14` es compuesta y el predicado `$#>` no puede gestionarla directamente. Hay que descomponer el argumento `L`, por lo que el control de programa pasa al predicado `hnf(L,HL, +Cin, -Cout)`, marcado con `I` en el siguiente fragmento de código.

```
    $#>(2#*X1 #+ 3#*X2, 14, Out, Cin, [store(Vars2,Size2,
                        [_V3#>14 |Constraints])|Cinter3]):-
 I     hnf(2#*X1 #+ 3#*X2, _V3, Cin, Cinter1),
 II    hnf(14, 14, Cinter1, Cinter2),
       find_store_in_Cin(Cinter2,Vars,Size,Constraints,Cinter3),
       manage_constraint_argument(_V3,Vars,Size,Vars1,Size1,Is_V3,Val_V3),
       manage_constraint_argument(14,Vars1,Size1,Vars2,Size2,Is14,Val14),
       post_greater(Is_V3,Val_V3,Is14,Val14,1).
```

`hnf I`

Computando la forma normal de cabeza del argumento `L`, el cálculo de estrechamiento perezoso detecta una restricción $\mathcal{FD}$ `(L1 #+ R1)`, que transfiere al predicado Prolog `$#+`. Concretamente `$#+(2#*X1, 3#*X2, _V3, Cin, Cinter1)`. En este caso, tanto `L1` como `R1` no están en forma normal de cabeza, por lo que la restricción `2#*X1 #+ 3#*X2` es compuesta y el predicado `$#+` no puede gestionarla directamente. Hay que descomponer los argumentos `L1` y `R1`, llamando a `hnf(L1, HL1, +Cin, -Cinter11)` y `hnf(R1, HR1, +Cinter11, -Cinter12)` respectivamente. El control de programa pasa al predicado `hnf(L1, HL1, +Cin, -Cinter11)` (marcado con `III`).

```
    $#+(2#*X1, 3#*X2, _V1, _V3, Cin, [store(Vars13,Size13,
                    [_V1#+_V2 |Constraints11])|Cinter13]):-
 III    hnf(2#*X1, _V1, Cin, Cinter11),
 IV     hnf(3#*X2, _V2, Cinter11, Cinter12),
       find_store_in_Cin(Cinter12,Vars11,Size11,Constraints11,Cinter13),
       manage_constraint_argument(_V1,Vars11,Size11,Vars12,Size12,Is_V1,Val_V1),
       manage_constraint_argument(_V2,Vars12,Size12,Vars13,Size13,Is_V2,Val_V2),
       post_greater(Is_V1,Val_V1,Is_V2,Val_V2,1).
```

hnf III

Computando la forma normal de cabeza del argumento `L1`, el cálculo de estrechamiento perezoso detecta una restricción $\mathcal{FD}$ `(L11 #* R11)`, que transfiere al predicado Prolog `$#*`. Concretamente `$#*(2, X1, _V1, Cin, Cinter11)`. En este caso `L11` es un número entero y `R11` una variable lógica $\mathcal{FD}$. Por lo tanto la restricción `2#*X1` es una restricción $\mathcal{FD}$ primitiva y puede ser impuesta sobre la aplicación genérica ILOG CP.

```
    $#*(2, X1, _V1, Cin, [store(Vars113,Size113,
                [2#*X1 |Constraints111])|Cinter113]):-
       hnf(2, 2, Cin, Cinter111),
       hnf(X1, X1, Cinter111, Cinter112),
       find_store_in_Cin(Cinter112,Vars111,Size111,Constraints111,Cinter113),
       manage_constraint_argument(_V1,Vars111,Size111,Vars112,Size112,Is2,Val2),
       manage_constraint_argument(_V2,Vars112,Size112,Vars113,Size113,IsX1,ValX1),
       post_greater(Is2,Val2,IsX1,ValX1,1).
```

Esta gestión accede al contenido de `store`. Como es la primera restricción $\mathcal{FD}$ primitiva que se detecta en el objetivo, el elemento `store` está vacío. Tanto la lista Variables como Restricciones son vacías. La gestión de la restricción primitiva modifica el contenido de estas listas. Ahora `Restricciones = [2#*X1==_V1]`, y `Variables = [X1,_V1]`. En la aplicación genérica ILOG CP se añaden las `IloIntVars x1` y `_v1` en `vars[0]` y `vars[1]`. Se añade también la nueva `IloConstraint 2*vars[0] == vars[1]` a `model`. El control de programa pasa ahora al predicado `hnf(R1, HR1, +Cinter11, -Cinter12)` (marcado con IV).

hnf IV

Computando la forma normal de cabeza del argumento `R1`, el cálculo de estrechamiento perezoso detecta una restricción $\mathcal{FD}$ `(L12 #* R12)`, que transfiere al predicado Prolog. Concretamente `$#*(3, X2, _V2, Cinter11, Cinter12)`. En este caso `L12` es un número entero y `R12` una variable lógica $\mathcal{FD}$. Por lo tanto la restricción `3#*X2` es una restricción $\mathcal{FD}$ primitiva y puede ser impuesta sobre la aplicación genérica ILOG CP.

```
$#*(3, X2, _V2, Cinter11, [store(Vars123,Size123,
            [3#*X2 |Constraints121])|Cinter123]):-
   hnf(3, 3, Cinter120, Cinter121),
   hnf(X2, X2, Cinter121, Cinter122),
   find_store_in_Cin(Cinter122,Vars121,Size121,Constraints121,Cinter123),
   manage_constraint_argument(3,Vars121,Size121,Vars122,Size122,Is3,Val3),
   manage_constraint_argument(X2,Vars122,Size122,Vars123,Size123,IsX2,ValX2),
   post_greater(Is3,Val3,IsX2,ValX2,1).
```

Esta gestión accede al contenido de `store([2#*X1==_V1], [X1,_V1])`. La gestión de la restricción primitiva modifica el contenido de estas listas. Ahora `Restricciones = [2#*X1==_V1, 3#*X2==_V2]`, y `Variables = [X1, _V1, X2, _V2]`. En la aplicación genérica ILOG CP se añaden las `IloIntVars x2` y `_v2` en `vars[2]` y `vars[3]`, respectivamente. Se añade también la nueva `IloConstraint 3*vars[2] == vars[3]` a `model`. Ahora que `L1 #+ R1` ha computado las formas normales `HL1 = _V1` y `HR1 = _V2` el predicado `$#+` gestiona la restricción $\mathcal{FD}$ `_V1 #+ _V2 == _V3`.

Gestión de L1 más R1

Esta gestión accede al contenido de `store([2#*X1==_V1, 3#*X2==_V2], [X1, _V1, X2, _V2])`. La gestión de la restricción primitiva modifica el contenido de estas listas. Ahora `Restricciones = [2#*X1==_V1, 3#*X2==_V2, _V1#+_V2==_V3]`, y `Variables = [X1, _V1, X2, _V2, _V3]`. En la aplicación genérica ILOG CP se añade la `IloIntVar _v3` en `vars[4]`. Se añade también la nueva `IloConstraint vars[1] + vars[3] == vars[4]` a `model`. Con esto finaliza la gestión del predicado Prolog `hnf(L,HL,Cin,Cout)`, que:

- Ha convertido el argumento `2#*X1 #+ 3#*X2` en `_V3`.
- Ha impuesto sobre la aplicación ILOG CP tantas restricciones $\mathcal{FD}$ primitivas como se han detectado en el cálculo de la forma normal de cabeza de `2#*X1 #+ 3#*X2`.

`hnf II` y resto del predicado *mayor*

El control de programa continúa con la evaluación del argumento `R = 14` (marcado con `II`), cuya forma normal de cabeza ya está en el formato adecuado. Tras finalizar el cómputo de `I` y `II`, la restricción $\mathcal{FD}$ `(2#*X1 #+ 3#*X2) #> 14` ha sido descompuesta a una conjunción de restricciones $\mathcal{FD}$ primitivas, de las cuales todas han sido ya impuestas a la aplicación genérica ILOG CP. La única restricción $\mathcal{FD}$ primitiva aún no impuesta es `_V3 #> 14`, que ahora sí puede ser gestionada por el predicado Prolog `$#>`.

Esta gestión accede al contenido de
`store([2#*X1==_V1, 3#*X2==_V2, _V1#+_V2==_V3], [X1, _V1, X2, _V2, _V3])`.
La gestión de la restricción primitiva modifica el contenido de
`Restricciones = [2#*X1==_V1, 3#*X2==_V2, _V1#+_V2==_V3, _V3 #> 14]`. En la aplicación genérica ILOG CP se añade la nueva `IloConstraint vars[4] >14` a `model`.

Con esto finalizamos este capítulo, donde hemos descrito la interconexión del sistema $\mathcal{TOY}(\mathcal{FD}i)$ y una primera versión mínima de dicho sistema, que soporta ocho tipos de restricciones $\mathcal{FD}$. En el siguiente capítulo describimos una segunda versión del sistema $\mathcal{TOY}(\mathcal{FD}i)$, dotado de una mayor funcionalidad que permite abordar una mayor cantidad de conjunciones de restricciones $\mathcal{FD}$.

Capítulo 3

$\mathcal{TOY}(\mathcal{FD}i)$: Un sistema ampliado

En este capítulo describimos la implementación de nuevas funcionalidades que dotan al sistema $\mathcal{TOY}(\mathcal{FD}i)$ de una mayor operatividad. El capítulo se estructura de la siguiente manera.

- En la primera sección se describe la gestión de restricciones de igualdad y desigualdad sintáctica que involucran alguna variable lógica $\mathcal{FD}$. La gestión de este tipo de restricciones se produce en el sistema de restricciones $\mathcal{H}$. Se estudia como se modifica dicha gestión para poder imponer estas restricciones sobre la aplicación genérica ILOG CP.
- En la segunda sección se describe la inconsistencia entre la conjunción de restricciones $\mathcal{FD}$ `C1,...,Cn,C` y la conjunción de restricciones $\mathcal{FD}$' `c1,...,cn,c` obtenida tras la propagación de la última restricción $\mathcal{FD}$' `c`. Se estudia el modo de restablecer la consistencia.
- En la tercera sección se describe la inconsistencia entre la conjunción de restricciones $\mathcal{FD}$' y el nuevo contenido de la conjunción de restricciones $\mathcal{FD}$ obtenida tras el backtracking sobre una de las funciones que conforman el objetivo $\mathcal{TOY}$. Se estudia el modo de restablecer la consistencia.
- En la cuarta sección se estudia la implementación en $\mathcal{TOY}(\mathcal{FD}i)$ de los procedimientos de búsqueda descritos en la subsección 1.2.1, que permitan encontrar las las soluciones por etiquetado de una conjunción de restricciones $\mathcal{FD}$'.

3.1. Gestión de las restricciones de igualdad y desigualdad

Como dijimos al principio del trabajo, el sistema $\mathcal{TOY}$ soporta programas que incluyen restricciones de dominio finito (restricciones $\mathcal{FD}$), restricciones aritméticas lineales y no lineales (restricciones $\mathcal{R}$), restricciones de igualdad y desigualdad sintáctica (restricciones $\mathcal{H}$) y restricciones de comunicación entre $\mathcal{FD}$, $\mathcal{R}$ y $\mathcal{H}$ (restricciones $\mathcal{M}$). A la hora de resolver objetivos sobre un determinado programa $\mathcal{TOY}$, cada restricción detectada por el cálculo de estrechamiento se envía a su correspondiente sistema de restricciones, donde se gestiona.

La independencia entre los distintos sistemas de restricciones es un pilar fundamental en el funcionamiento de $\mathcal{TOY}$. Las restricciones de igualdad y desigualdad sintáctica se gestionan en el sistema de restricciones $\mathcal{TOY}(\mathcal{H})$. Sin embargo, cualquier implementación

del sistema de restricciones $\mathcal{TOY}(\mathcal{FD})$ guarda un foco de conflicto con el sistema de restricciones $\mathcal{TOY}(\mathcal{H})$. Este conflicto se produce cuando alguna de las variables lógicas involucradas en una restricción $\mathcal{H}$ es una variable lógica $\mathcal{FD}$. En este caso la restricción debe tratarse como restricción $\mathcal{FD}$ y gestionarse en el sistema de restricciones $\mathcal{FD}$.

3.1.1. Gestión de las restricciones de igualdad y desigualdad en $\mathcal{TOY}(\mathcal{FD}s)$

En el caso del sistema $\mathcal{TOY}(\mathcal{FD}s)$, la solución a este conflicto mantiene la gestión en el sistema de restricciones $\mathcal{H}$, incluyendo ligeras modificaciones sobre el código que implementa estas restricciones.

- Restricciones de igualdad sintáctica (==)

  Para el caso de las restricciones de igualdad sintáctica, la gestión se mantiene íntegra en el sistema de restricciones $\mathcal{H}$, sin que el código presente modificación alguna. Esto es debido a que el sistema de restricciones $\mathcal{FD}$, `clpfd`, está integrado en SICStus. Esto permite imponer las restricciones $\mathcal{FD}$ sin necesidad de traducirlas. Las propias restricciones $\mathcal{FD}$ y variables lógicas $\mathcal{FD}$ que detecta el cálculo de estrechamiento perezoso son las restricciones $\mathcal{FD}$ y variables lógicas $\mathcal{FD}$ sobre las que trabaja `clpfd`. Por ello, si el sistema de restricciones $\mathcal{H}$, mediante una restricción de igualdad, unifica alguna variable lógica $\mathcal{FD}$, esta variable lógica $\mathcal{FD}$ aparecerá unificada en cualquiera de sus apariciones en el sistema $\mathcal{TOY}$, en concreto en sus apariciones en las restricciones $\mathcal{FD}$ sobre las que trabaja clpfd. Los tres posibles casos que se presentan son:

  i) La restricción de igualdad sintáctica no involucra ninguna variable lógica $\mathcal{FD}$. Además, ninguna de estas variables lógicas está involucrada en alguna futura restricción $\mathcal{FD}$ detectada por el cálculo de estrechamiento.

  Por ejemplo, en el objetivo `X == Z, Y #> 0, K #> 0`, la restricción `X == Z` no entra en conflicto con el sistema de restricciones `clpfd`, ni durante la propia gestión de `X == Z` ni posteriormente.

  ii) La restricción de igualdad sintáctica no involucra ninguna variable lógica $\mathcal{FD}$. Sin embargo, alguna de estas variables lógicas se verá involucrada en alguna futura restricción $\mathcal{FD}$. Por lo tanto, la variable lógica se convertirá en variable lógica $\mathcal{FD}$, y la restricción de igualdad sintáctica se convertirá a posteriori en una restricción $\mathcal{FD}$.

  Sin embargo, si una restricción de igualdad no involucra en el momento de su gestión a ninguna variable lógica $\mathcal{FD}$, esta restricción no produce ni producirá ningún conflicto con el sistema de restricciones `clpfd`. De nuevo esto se debe al hecho de que `clpfd` está integrado en SICStus.

  Por ejemplo, en el objetivo `X == Y, Y #>= 1, Y #<= 1, X == 2`, la restricción de igualdad `X == Y` se produce entre dos variables lógicas que no son $\mathcal{FD}$. Posteriormente, la restricción `Y #>= 1` convertirá a `Y` en una variable lógica $\mathcal{FD}$, por lo que la restricción `X == Y` se convertirá a posteriori en una restricción $\mathcal{FD}$. Aún así, `clpfd` no precisa que se imponga `X == Y`, ni tener en cuenta a la variable lógica `X`. Si en algún momento `clpfd` unifica a `Y`, mediante la propagación de la conjunción de restricciones $\mathcal{FD}$ sobre las que trabaja, esta `Y` se verá reeemplazada en toda aparición en el sistema, en concreto en la restricción `X == Y` que contine $\mathcal{H}$. En este objetivo, tras la gestión de `Y #<= 1`, `clpfd`

unifica `Y` al valor `1`. Esto reeemplaza las apariciones de `Y` en el sistema de restricciones $\mathcal{H}$, en particular `X == Y`, que pasa a ser `X == 1`. Entonces, $\mathcal{H}$ unifica `X` al valor `1`. La evaluación del objetivo continúa. El cálculo de estrechamiento detecta ahora `1 == 2`, ya que `X` está unificada a `1`, y la impone al sistema de restricciones $\mathcal{H}$, que encuentra insatisfactibilidad. Por tanto este objetivo falla, no por insatisfactibilidad de `clpfd`, sino por insatisfactibilidad de $\mathcal{H}$.

iii) La restricción de igualdad involucra alguna variable lógica $\mathcal{FD}$.
Por ejemplo, en el objetivo `X #> 0, X #> 1, X == 2, X #> 3`, la restricción de igualdad `X == 2` se gestiona en el resolutor $\mathcal{H}$, que unifica a la variable lógica `X` al valor `2`. Esto provoca que toda aparición de `X` en el sistema se reemplace por `2`, en concreto las apariciones de `X` en el sistema de restricciones `clpfd`. Debido a la gestión de `X == 2` en $\mathcal{H}$, el contenido de `clpfd` pasa de ser `X #> 0, X #> 1` a ser `2 #> 0, 2 #> 1`, aunque la restricción `X == 2` no haya sido impuesta sobre él. Como la conjunción de restricciones $\mathcal{FD}$ `2 #> 0, 2 #> 1` es satisfactible, el cálculo continúa la evaluación del objetivo $\mathcal{TOY}$. Detecta la restricción `2 #> 3`, ya que `X` está unificada a `2`, y la impone a `clpfd`, que encuentra insatisfactibilidad.

- Restricciones de desigualdad sintáctica (/=)

  Para el caso de las restricciones de desigualdad sintáctica, la implementación de $\mathcal{TOY}(\mathcal{FD}s)$ modifica ligeramente el código de gestión de $\mathcal{H}$ y el código de gestión de los predicados Prolog de $solve^{FD}$ que gestionan las restricciones $\mathcal{FD}$. Los tres casos que se presentan son:

  i) La restricción de desigualdad sintáctica no involucra ninguna variable lógica $\mathcal{FD}$. Además, ninguna de estas variables lógicas está involucrada en alguna futura restricción $\mathcal{FD}$ detectada por el cálculo de estrechamiento.
  Por ejemplo, en el objetivo `X /= Z, Y #> 0, K #> 0`, la gestión de la restricción `X /= Z` no entra en conflicto con el sistema de restricciones `clpfd`, ya que ni `X` ni `Y` son variables lógicas $\mathcal{FD}$. Sin embargo, $\mathcal{TOY}(\mathcal{FD}s)$ modifica ligeramente el código del predicado Prolog que gestiona `/=` en $\mathcal{H}$. Ahora este predicado Prolog incluirá en el almacén `Cin` los términos `X:[Z]` y `Z:[X]`.
  Los predicados Prolog que gestionan las distintas restricciones $\mathcal{FD}$ deben tener en cuenta estos términos `X:[Z]` y `Z:[X]`, a los que pueden acceder a través de su argumento de entrada `Cin`. Si en la restricción $\mathcal{FD}$ que están gestionando, alguna de las variables lógicas $\mathcal{FD}$ involucradas es `X` o `Z`, entonces las desigualdades sintácticas que se trataron sobre `X` o `Z` se convierten ahora en restricciones $\mathcal{FD}$, que deben ser impuestas a `clpfd`. Para ello, cada predicado Prolog de $solve^{FD}$ utiliza el predicado Prolog `toSolverFD` para cada uno de los argumentos de la restricción $\mathcal{FD}$ que está gestionando. Este predicado `toSolverFD`, busca en `Cin` términos con el formato `arg:[Ineq|RIneq]`. Elimina dicho término de `Cin` e impone las desigualdades sobre `clpfd`.
  Por lo tanto, el predicado Prolog `$#>` utilizará `toSolverFD` para buscar términos `Y:[Ineq|RIneq]` y `K:[Ineq|RIneq]` en la gestión de `Y #> 0` y `K #> 0` respectivamente. Como no encuentra dichos términos, la gestión de `X /= Z` que fue gestionada en $\mathcal{H}$ no será nunca gestionada en `clpfd`.

  ii) La restricción de desigualdad sintáctica no involucra ninguna variable lógica $\mathcal{FD}$. Sin embargo, alguna de estas variables lógicas se verá involucrada en alguna

futura restricción $\mathcal{FD}$. Por lo tanto, la variable lógica se convertirá en variable lógica $\mathcal{FD}$, y la restricción de desigualdad sintáctica se convertirá a posteriori en una restricción $\mathcal{FD}$.

Por ejemplo, en el objetivo `X /= Y, Y #>= 0, Y #=< 1, X == 1`, la restricción de desigualdad `X /= Y` se produce entre dos variables lógicas que no son $\mathcal{FD}$. Por lo tanto, el predicado Prolog de $\mathcal{H}$ no la impone a `clpfd`, pero incluye `X:[Y]` e `Y:[X]` en `Cin`.

Posteriormente, el predicado Prolog `$#>` gestiona la restricción `Y #>= 0`. Al evaluar `toSolverFD` con su argumento `Y` encuentra `Y:[X]` en `Cin`. Por lo tanto impone la restricción `X /= Y` al sistema de restricciones `clpfd`. Además elimina `Y:[X]` de `Cin`. Su otro argumento es `0`, por lo que no cabe sincronización posible. Con esto concluye la sincronización, y comienza la gestión de la restricción $\mathcal{FD}$ `Y #>= 0`, que impone a `clpfd`.

Si no imponemos la desigualdad `X /= Y` a `clpfd`, la respuesta encontrada por `clpfd` para el objetivo `Y #>= 0, Y #=< 1` no vincula `Y` al valor `0`. Por contra, si imponemos `X /= Y`, tras la gestión de `X == 1` en $\mathcal{H}$, la restricción $\mathcal{FD}$ pasará a ser `1 /= Y`, por lo que `clpfd` vinculará `Y` a `0`. Es decir, si no se exige que los predicados Prolog de $solve^{FD}$ sincronicen las desigualdades sintácticas gestionadas por $\mathcal{H}$ que ahora se han convertido en restricciones $\mathcal{FD}$, entonces `clpfd` no será consistente con la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$, por lo que la solución que encuentre no será correcta.

iii) La restricción de desigualdad involucra alguna variable lógica $\mathcal{FD}$.

Por ejemplo, en el objetivo `X #> 1, X /= 2, X #< 4`, la restricción de desigualdad `X /= 2` involucra a la variable lógica $\mathcal{FD}$ `X`, por lo que es impuesta a `clpfd`. Además, como ya ha sido impuesta, no es necesario que se añada `X:[2]` a `Cin`.

De nuevo basta fijarse que el objetivo tiene solución `Y -> 3`, pero que si no imponemos `X /= 2` a `clpfd`, éste encuentra la solución incompleta `Y in 2..3`.

En el caso del sistema $\mathcal{TOY}(\mathcal{FD}i)$, el conflicto entre restricciones $\mathcal{H}$ que en realidad son restricciones $\mathcal{FD}$ precisa una solución más compleja, debido a la comunicación entre SICStus Prolog y C++.

3.1.2. Gestión de la restricción de igualdad en $\mathcal{TOY}(\mathcal{FD}i)$

Como vimos en la sección 2.2, para representar en la aplicación genérica ILOG CP la conjunción de restricciones $\mathcal{FD}$ subyacente a un objetivo $\mathcal{TOY}$, es preciso traducir cada una de las restricciones $\mathcal{FD}$ a una nueva restricción $\mathcal{FD}$' equivalente, cuyo formato de entrada sea el adecuado para la biblioteca ILOG Concert 2.6. La necesidad de traducir cada restricción $\mathcal{FD}$ y sus variables lógicas $\mathcal{FD}$ involucradas nos obliga a gestionar las restricciones de igualdad sintáctica.

En $\mathcal{TOY}(\mathcal{FD}s)$ esa gestión no era necesaria, ya que las unificaciones de variables lógicas $\mathcal{FD}$ a valores enteros quedaban automáticamente reflejadas en `clpfd`. Además, si una restricción de igualdad sintáctica unificaba dos variables lógicas $\mathcal{FD}$, esta restricción se encontraba implícitamente añadida a `clpfd`. En $\mathcal{TOY}(\mathcal{FD}i)$ este proceso no puede ser automático. Si una restricción de igualdad sintáctica gestionada en $\mathcal{H}$ unifica una variable lógica $\mathcal{FD}$ `Var` a un valor entero `Value`, entonces se produce una inconsistencia entre la variable lógica `V` y su `IloIntVar v` asociada. Para recuperar la consistencia se debe

imponer sobre `model` una nueva `IloConstraint` que acote al valor `Value` el dominio de `v`. Por otro lado, si dos variables lógicas $\mathcal{FD}$ `V1` y `V2` están unificadas, entonces se debe imponer explícitamente sobre `model` la `IloConstraint c = v1 == v2`.

La solución más sencilla pasa por modificar el código de $\mathcal{H}$ que gestiona las restricciones de igualdad sintáctica. Sin embargo, el principio de independencia entre los distintos sistemas de restricciones de $\mathcal{TOY}$ nos lleva a intentar evitar esto en la medida de lo posible. El ámbito de este trabajo se reduce a implementar $\mathcal{TOY}(\mathcal{FD}i)$ como una versión de $\mathcal{TOY}(\mathcal{FD})$, pretendiendo que sea lo más autocontenida posible y que no interfiera en los demás sistemas de restricciones del sistema. Por ello sopesamos otras alternativas.

Primer intento con uso de demonios

Guiados por el automatismo con que la unificación gestiona las restricciones de igualdad sintáctica en `clpfd`, nuestra primera alternativa en la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ pasa por utilizar demonios. Estudiamos el escenario que soporta SICStus para el uso de demonios:

1. SICStus Prolog dispone de la biblioteca `atts` para el uso de variables atribuidas. De este modo se puede separar el conjunto de variables lógicas del sistema en dos conjuntos disjuntos, el de las variables lógicas atribuidas y el de las variables lógicas no atribuidas.

2. La biblioteca `atts` dispone del predicado `verify_attributes(-Var, +Value, -Goals)`. Mediante el uso de este predicado se puede extender el algoritmo de unificación de SICStus, permitiendo que los objetivos del cuerpo de `verify_attributes` sean evaluados cada vez que se vincule una variable atribuida `Var`. El argumento `Value` representa el valor al que `Var` se vincula. El argumento `Goals` representa una lista con los objetivos que serán llamados tras la vinculación de `Var` a `Value`. La definición del predicado es fija y no permite ninguna modificación sobre el número y naturaleza de sus argumentos.

El sistema de restricciones $\mathcal{H}$ gestiona restricciones de igualdad sintáctica. En el sistema $\mathcal{TOY}(\mathcal{FD}s)$, cuando $\mathcal{H}$ unificaba una variable lógica $\mathcal{FD}$ este cambio se veía automáticamente reflejado en `clpfd`. Utilizando el escenario soportado por SICStus que acabamos de estudiar nos proponemos:

i) Utilizar la biblioteca `atts` en la implementación de $\mathcal{TOY}(\mathcal{FD}i)$. Utilizar un atributo `fd/0`, que imponemos sobre las variables lógicas $\mathcal{FD}$. El resto de variables lógicas del sistema no serán atribuidas. El predicado `manage_constraint_argument`, explicado en la subsección 2.2.4, detecta cada nueva variable lógica $\mathcal{FD}$ del objetivo $\mathcal{TOY}$, que añade a la lista `Variables` contenida en el elemento `store` del almacén `Cin`. Se modifica la segunda cláusula de este predicado `manage_constraint_argument` imponiendo el atributo `fd/0` a cada nueva variable lógica `V` que es detectada en el sistema mediante `put_atts(V,fd)`.

ii) Modificamos el contenido de `ilog_cp.cpp` e `ilog_cp.pl` añadiendo un nuevo predicado Prolog `post_equal(Index,Value,Feasible)`, que esté implementado mediante la función C++ `post_equal(long Index, long Value, long* feasible)`. Esta función:

- Impone la restricción $\mathcal{FD}$ `vars[Index] == Value` a la conjunción de restricciones $\mathcal{FD}$' contenidas en `model`.
- Se encarga de que `solver` propague la `IloConstraint vars[Index] == Value`.
- Devuelve en `feasible` un valor booleano que representa la satisfactibilidad de la nueva conjunción de restricciones $\mathcal{FD}$" tras la propagación.

iii) Utilizar el predicado Prolog `verify_attributes` para que llame, como único objetivo de su cuerpo, al predicado Prolog `post_equal(Index,Value,Feasible)`. De este modo, cada vez que $\mathcal{H}$ unifique una variable lógica $\mathcal{FD}$ `Var` con un número entero `Value`, el demonio `verify_attributes` se activará automáticamente. Éste llama al predicado `post_equal`, que acota al número entero `Value` la `IloIntVar` asociada a `Var`. Con esto se reproduciría exactamente el comportamiento de $\mathcal{TOY}(\mathcal{FD}s)$, completando con éxito en $\mathcal{TOY}(\mathcal{FD}i)$ la gestión de las restricciones de igualdad sintáctica que unifican alguna variable lógica $\mathcal{FD}$ a un número entero.

Sin embargo este intento fracasa, ya que no es posible implementar el punto iii). El predicado `verify_attributes(Var,Value,Goals)` no puede obtener el valor entero `Index` necesario para la llamada a `post_equal(Index,Value,Feasible)`. El predicado `verify_attributes` recibe como argumento la variable lógica $\mathcal{FD}$ `Var`. Pero necesita acceder a la lista `Variables`, contenida en `store`, para conocer la posición de `vars` en la que se encuentra su `IloIntVar` asociada. Sin poder acceder a la lista `Variables` no se puede referenciar a `Var` en la aplicación genérica ILOG CP, como ya demostramos en el primer intento de comunicación descrito en la subsección 2.2.1. Como los argumentos de `verify_attributes` están fijados, no hay ninguna posibilidad de acceder a `Cin`, y con ello a la lista `Variables` contenida en `store`. Hay que plantear una segunda alternativa para sincronizar en la aplicación genérica ILOG CP las variables lógicas $\mathcal{FD}$ que se vinculan a valores enteros por restricciones de igualdad sintáctica.

Segundo intento modificando la lista `Variables`

La segunda alternativa retoma la idea descrita en $\mathcal{TOY}(\mathcal{FD}s)$ para la imposición en `clpfd` de las restricciones de desigualdad sintáctica que a posteriori se han convertido en restricciones $\mathcal{FD}$. Entonces era necesario que cada predicado Prolog de $solve^{FD}$, antes de gestionar su restricción $\mathcal{FD}$, utilizara el predicado Prolog `toSolverFD` para sincronizar estas posibles desigualdades ahora convertidas en relevantes para `clpfd`. Reproducimos en $\mathcal{TOY}(\mathcal{FD}i)$ esta forma de trabajar para sincronizar toda `IloIntVar v` cuya variable lógica $\mathcal{FD}$ asociada `V` ha sido unificada a un valor entero por $\mathcal{H}$.

Cada vez que el sistema de restricciones $\mathcal{H}$ unifica una variable lógica $\mathcal{FD}$ `V` a un valor entero `Value`, cualquier aparición en el sistema $\mathcal{TOY}$ de `V` se reemplaza por `Value`. En concreto la aparición de `V` en la lista `Variables`, contenida en el elemento `store`.

Por otro lado, todos los predicados Prolog de $solve^{FD}$ que interactúan con la aplicación genérica ILOG CP reciben como argumento de entrada el almacén `Cin`, por lo que tienen acceso a esta lista `Variables`. Por lo tanto, encargamos a estos predicados Prolog la tarea de sincronizar las `IloIntVars v1,...,vn` cuyas variables lógicas $\mathcal{FD}$ `V1,...,Vn` han sido unificadas a valores enteros por $\mathcal{H}$. Cada uno de estos predicados realiza en primer lugar la sincronización. Una vez finalizada esta sincronización, realiza la gestión con la aplicación genérica ILOG CP que tenga encargada.

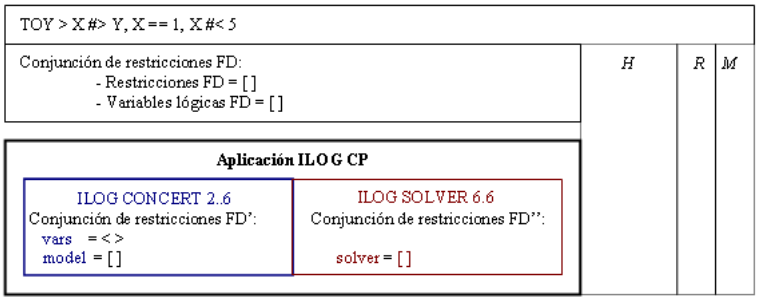

Figura 3.1: Inicio del objetivo.

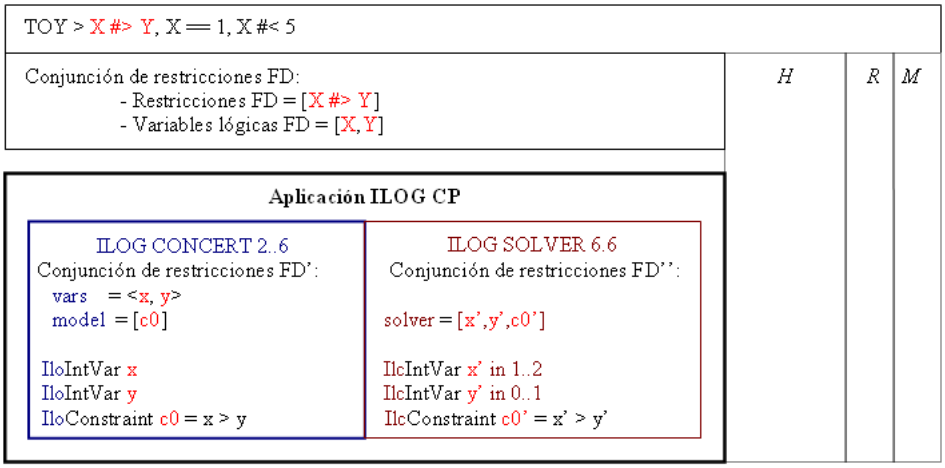

Figura 3.2: Evaluación de la primera restricción.

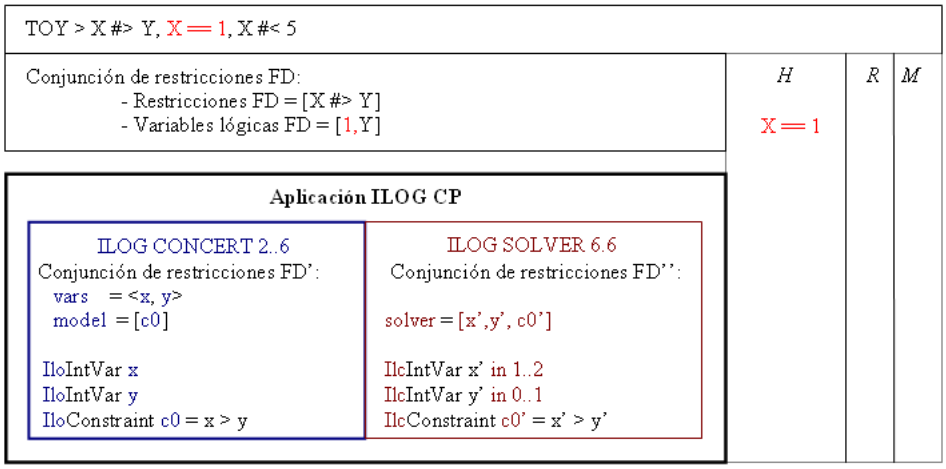

Figura 3.3: Evaluación de la segunda restricción.

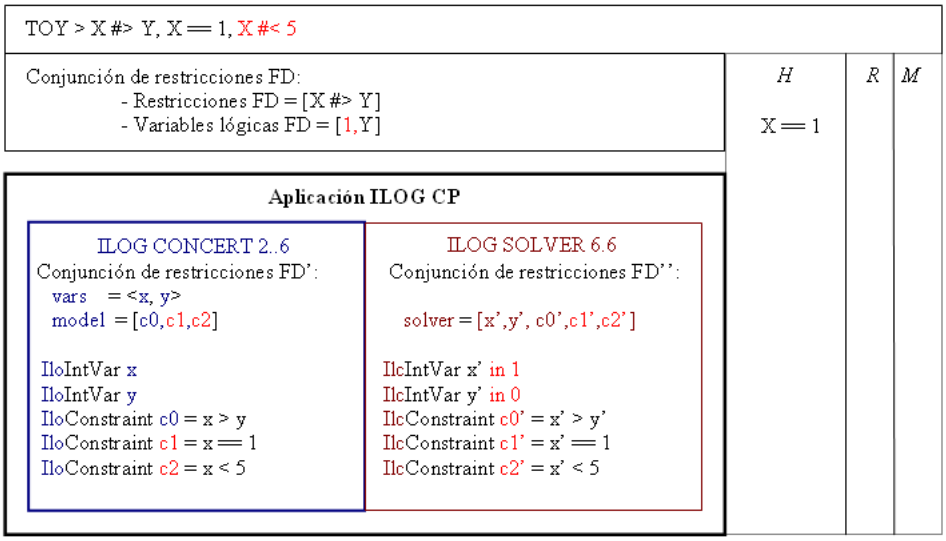

Figura 3.4: Evaluación de la tercera restricción.

Las Figuras 3.1, 3.2, 3.3 y 3.4 representan la modificación de los sistemas de restricciones $\mathcal{FD}$ y $\mathcal{H}$ a medida que se evalúa el objetivo `X #> Y, Y == 1, X #< 5`, donde `X` e `Y` son variables lógicas $\mathcal{FD}$ con dominio `0...2`.

La Figura 3.3 muestra la modificación que sufre la lista `Variables` tras la gestión de `X == 1` en el sistema de restricciones $\mathcal{H}$. Se puede ver que, tras unificar la variable lógica $\mathcal{FD}$ al valor `1`, la aparición de `X` en `Variables` se ve reemplazada por `1`. Sin embargo, su `IloIntVar x` asociada en la aplicación genérica ILOG CP no ve acotado su dominio a `1`, ya que la restricción `IloConstraint c1 = x == 1` aún no se ha impuesto.

En la Figura 3.4 el predicado `$#<` de $solve^{FD}$ se encarga de:

1. Sincronizar la `IloIntVar x`, asociada a la variable lógica $\mathcal{FD}$ `X`. Para ello impone sobre `model` la `IloConstraint c1 = vars[0] == 1`, que acota el dominio de `x` al valor `1`, recuperando la consistencia con `X`, unificada a `1`.

2. Tras finalizar la sincronización, se encarga de gestionar la restricción $\mathcal{FD}$ `X #< 5`.

No sincronizar automáticamente estas `IloIntVars` provoca inconsistencias entre la conjunción de restricciones $\mathcal{FD}$' representada en `model` y la conjunción de restricciones $\mathcal{FD}$ subyacentes al objetivo $\mathcal{TOY}$ evaluado hasta el momento. La Figura 3.3 evidencia este caso. Tras la gestión de `X == 1` en $\mathcal{H}$, si accedemos a la aplicación genérica ILOG CP, requiriendo a `solver` la solución computada al problema representado en `model`, la solución obtenida es `X in 1..2, Y in 0..1`. Esta solución es correcta con respecto a la conjunción de restricciones $\mathcal{FD}$' representada en `model`: `x' > y'`. Pero no lo es con respecto a la conjunción de restricciones $\mathcal{FD}$ subyacente al objetivo $\mathcal{TOY}$ evaluado hasta el momento: `X #> Y, X == 1`, que da pie a la solución: `X -> 1, Y -> 0`.

Este tipo de inconsistencias no debe incomodarnos desde el punto de vista de la corrección de las soluciones computadas, aunque sí desde el punto de vista de la eficiencia en la evaluación de objetivos. El único requisito exigido es que todo predicado Prolog que interaccione con la aplicación genérica ILOG CP se encargue previamente de sincronizar sus `IloIntVars`.

La Figura 3.3 no muestra esta situación. Pero la Figura 3.3 muestra la gestión del predicado Prolog de $\mathcal{H}$ que gestiona la igualdad sintáctica. Este predicado `==` no interacciona con la aplicación genérica ILOG CP, por lo que no debe contrastar si las `IloIntVar`s quedan sincronizadas. Posteriormente la Figura 3.4 muestra la gestión del predicado `$#<` de $solve^{FD}$. Este predicado si que interacciona con la aplicación genérica ILOG CP, imponiendo sobre `model` una nueva `IloConstraint x < 5`, exigiendo su propagación, así como la satisfactibilidad del nuevo conjunto de restricciones $\mathcal{FD}$" resultante. Por ello, antes de hacer ninguna de estas acciones, el predicado Prolog `$#<` debe garantizar que toda `IloIntVar` sea consistente con su variable lógica $\mathcal{FD}$ asociada. Estudiémoslo sobre dos objetivos:

- `X #> 1, X == 5, X #< 3` falla. El fallo se produce en el predicado Prolog `$#<`. Como este predicado sincroniza la `IloIntVar x` asociada a `X` antes de gestionar `X #< 3`, podemos asegurar que la restricción que produce la insatisfactibilidad es la restricción `X #< 3`.

- `X #> 1, X == 0, X #< 3` falla. El fallo se produce en el predicado `$#<`. Este predicado sincroniza la `IloIntVar x` asociada a `X` antes de gestionar `X #< 3`. Al imponer la `IloConstraint c1 = x == 0` el resolutor `solver` encuentra insatisfactibilidad. Por tanto podemos asegurar que la restricción que produce la insatisfactibilidad es la restricción $\mathcal{H}$ `X == 5`. El predicado Prolog no llega siquiera a realizar la gestión de `X #< 3`.

El siguiente objetivo muestra que esta forma de sincronización es correcta pero ineficiente:
`X #> 1, X == 0, X1 = 0, ...X1000 = 0`
El objetivo debería fallar tras la gestión de `X == 0`, que produciría la insatisfactibilidad de la conjunción de restricciones $\mathcal{FD}$" sobre la que trabaja `solver`. Sin embargo, la restricción `X == 0` no es transmitida inmediatamente a `model`. La transmitirá el próximo predicado Prolog que interaccione con la aplicación genérica ILOG CP. Por lo tanto, al no detectarse aún el fallo, la evaluación del objetivo continúa. Las siguientes 1000 restricciones son también gestionadas por $\mathcal{H}$, sin fallo. Por lo tanto la evaluación del objetivo finaliza aparentemente con éxito.

El control de programa llega al predicado `writeSolution` (explicado en la subsección 2.2.3). Este predicado muestra al usuario la solución del objetivo $\mathcal{TOY}$ planteado. Para mostrar esta solución el predicado Prolog precisa interaccionar con la aplicación genérica ILOG CP, obteniendo los dominios de las `IlcIntVar`s sobre las que trabaja `solver`. Antes de interaccionar con la aplicación genérica ILOG CP, el predicado Prolog se encarga de sincronizar sus `IloIntVar`s. La sincronización impone sobre `model` la `IloConstraint c1 = x == 0`, que junto con la `IloConstraint c0 = x >1` provocan insatisfactibilidad, haciendo fallar al objetivo. Pero se han computado inutilmente 1000 restricciones por no sincronizar automáticamente las unificaciones de variables lógicas $\mathcal{FD}$ a valores enteros.

Implementación de la sincronización

Para implementar la sincronización de las restricciones de igualdad sintáctica tal y como la hemos explicado debemos modificar la estructura interna de la lista `Variables`. Ahora esta lista pasa a ser una lista de pares (`[Var,Flag]`). El componente `Var` representa una variable lógica $\mathcal{FD}$. El componente `Flag` es el valor entero 0 o 1. Si `Var` no está unificada a un valor entero, entonces `Flag = 0`. Si `Var` está unificada al valor entero

k, entonces Flag indica si la IloIntVar var está o no acotada a k. Modificar la estructura de Variables para que sea una lista de pares tiene dos efectos colaterales:

- Modificamos la segunda cláusula del predicado Prolog manage_constraint_argument, que puede añadir una nueva variable Arg a la lista Variables. Reemplazamos el objetivo de su cuerpo (append(Vars, [Arg], NewVars) por (append(Vars, [[Arg,0]], NewVars).
- Modificamos la cabeza del predicado is_var_in_vars_list, que ahora busca la variable Var en una lista de pares. Reemplazamos is_var_in_vars_list(V,[X|_],E,E):- por is_var_in_vars_list(V,[[X,_]|_],E,E):-.

Ampliamos el contenido de ilog_cp.cpp e ilog_cp.pl con un nuevo predicado Prolog post_equal implementado en la función C++ post_equal. Este predicado sigue la misma definición que el predicado post_greater explicado en la subsección 2.2.3. Se encarga de imponer una nueva IloConstraint c = v1 == v2 sobre model, propagarla y devolver como resultado si la nueva conjunción de restricciones $\mathcal{FD}$" es o no satisfactible.

Para sincronizar model con las restricciones de igualdad sintáctica impuestas en $\mathcal{H}$ debemos:

- Recorrer la lista Variables, identificando las variables Var que han sido unificadas a un valor enteros Value.
- Por cada una de estas variables imponer sobre model una IloConstraint c = var == Value.
- Indicar a la lista Variables que las variables ya han sido sincronizadas, para no tener que sincronizarlas de nuevo en el futuro.

Creamos una primera versión del predicado Prolog post_fd_unifications, que realiza estas tareas.

```
%post_fd_unifications/2(+VarList,-NewVarList).
%
(0) post_fd_unifications(Vars,NewVars) :-
(1)    get_Herbrand_unifications(Vars,0,ListOfConstraints),
(2)    post_equalities(ListOfConstraints),
(3)    modify_list_of_vars(Vars,NewVars).
```

La línea (0) muestra la cabeza del predicado. Éste modifica la lista Variables, imponiendo sobre model las restricciones de igualdad que sincronizan las IloIntVars. Para ello utiliza tres predicados entre los objetivos de su cuerpo.

El predicado get_Herbrand_unifications(Vars,0,ListOfConstraints) (1) identifica los índices de cada pareja de Variables que encaja con el patrón [Value,0]. Este patrón indica que $\mathcal{H}$ ha unificado la variable lógica $\mathcal{FD}$ V contenida en ese índice de Variables al valor entero Value, y que además, su IloIntVar v asociada no está sincronizada con ese valor. Para cada una de estas parejas [Value,0], contenida en la posición i de Variables, el predicado genera el par [i,Value], que incluye a ListOfConstraints.

```
%get_Herbrand_unifications/3(+VarList,+Index,-PatternIndex).
%
```

```
get_Herbrand_unifications([],_,[]).
%
get_Herbrand_unifications([[X,Y]|Xs],I,[[I,X]|R]) :-
    integer(X),
    Y == 0,
    !,
    NI is I+1,
    get_Herbrand_unifications(Xs,NI,R).
%
get_Herbrand_unifications([[X,Y]|Xs],I,R) :-
    integer(X),
    Y == 1,
    !,
    NI is I+1,
    get_Herbrand_unifications(Xs,NI,R).
%
get_Herbrand_unifications([[X,Y]|Xs],I,R) :-
    var(X),
    Y == 0,
    !,
    NI is I+1,
    get_Herbrand_unifications(Xs,NI,R).
%
get_Herbrand_unifications([[X,Y]|Xs],I,R) :-
    var(X),
    Y == 1,
    !,
    NI is I+1,
    get_Herbrand_unifications(Xs,NI,R).
```

Las cuatro cláusulas especifican los cuatro casos que aceptamos como válidos. Estos cuatro casos hacen fallar a situaciones donde una variable lógica $\mathcal{FD}$ es unificada por $\mathcal{H}$ a una función. Por ejemplo, el objetivo `X == Z, X #> 0, Z == coin` debe fallar. La unificación de `Z` a `coin` provoca que `X` se unifique a `coin`. No es posible que `X` sea al mismo tiempo una variable lógica $\mathcal{FD}$ mayor que `0` y la función `coin`. Como `X ->coin` no es ni una variable lógica ni un valor entero, el predicado `get_constraints_to_solver([[coin,_]|Xs],I,R)` fallará.

El predicado `post_equalities(ListOfConstraints) (2)` impone una a una las restricciones de igualdad generadas en `ListOfConstraints`. Por cada par `[i,Value]` impone sobre `model` la `IloConstraint vars[i] == Value`. Para ello utiliza `post_equal(1,i,0,Value,1)`. Recordamos que imponemos que `Feasible == 1` para exigir que, tras la gestión de la nueva `IloConstraint`, la nueva conjunción de restricciones $\mathcal{FD}$" contenida en `solver` sea satisfactible.

```
%post_equalities/1(+ConstraintsToSolver).
%
post_equalities([]).
%
post_equalities([[I,X]|Xs]) :-
```

```
    post_equal(1,I,0,X,1),
    post_equalities(Xs).
```

El predicado `modify_list_of_vars(Vars,NewVars)` (3) es llamado tras imponer todas las restricciones de igualdad y comprobar que la nueva conjunción de restricciones contenida en `solver` sigue siendo satisfactible. Este predicado modifica la lista `Variables`, modificando todo patrón `[Value,0]` por `[Value,1]`. Esto indica que toda variable lógica $\mathcal{FD}$ `V` unificada a `Value` tiene sincronizada a su `IloIntVar v` asociada.

```
%modify_list_of_vars/2(+List,-NewList).
%
modify_list_of_vars([],[]).
modify_list_of_vars([[V,0]|Xs],[[V,1]|NXs]):-
    integer(V),
    !,
    modify_list_of_vars(Xs,NXs).
%
modify_list_of_vars([[V,0]|Xs],[[V,0]|NXs]):-
    var(V),
    !,
    modify_list_of_vars(Xs,NXs).
%
modify_list_of_vars([[V,1]|Xs],[[V,1]|NXs]):-
    modify_list_of_vars(Xs,NXs).
```

Modificamos los predicados de $solve^{FD}$ que implementamos en la versión de $\mathcal{TOY}(\mathcal{FD}i)$ de la sección 2.2. Mostramos esta modificación sobre el predicado `$#>`:

```
% $#>/5(+L,+R,-Out,+Cin,-Cout).

  $#>(L, R, Out, Cin, [store(Vars3,Size2,[HL>HR|Constraints])|Cinter3]):-
    hnf(L, HL, Cin, Cinter1),
    hnf(R, HR, Cinter1, Cinter2),
    find_store_in_Cin(Cinter2,Vars,Size,Constraints,Cinter3),
**  post_fd_unifications(Vars,Vars1),
    manage_constraint_argument(HL,Vars1,Size,Vars2,Size1,IsHL,ValHL),
    manage_constraint_argument(HR,Vars2,Size1,Vars3,Size2,IsHR,ValHR),
    post_greater(IsHL,ValHL,IsHR,ValHR,1).
```

Observamos el resultado de esta técnica de sincronización sobre el objetivo que mostramos al principio de la subsección en las Figuras 3.1, 3.2, 3.3 y 3.4.

Imposición de las restricciones de igualdad entre dos variables lógicas $\mathcal{FD}$

Además de las unificaciones de variables lógicas $\mathcal{FD}$ a valores enteros, en el sistema $\mathcal{TOY}(\mathcal{FD}i)$ también vamos a tener que gestionar las restricciones de igualdad entre dos variables lógicas $\mathcal{FD}$. En $\mathcal{TOY}(\mathcal{FD}s)$ esa gestión no era necesaria, ya que las unificaciones entre variables lógicas $\mathcal{FD}$ quedaban automáticamente reflejadas en `clpfd`, al ser las mismas variables.

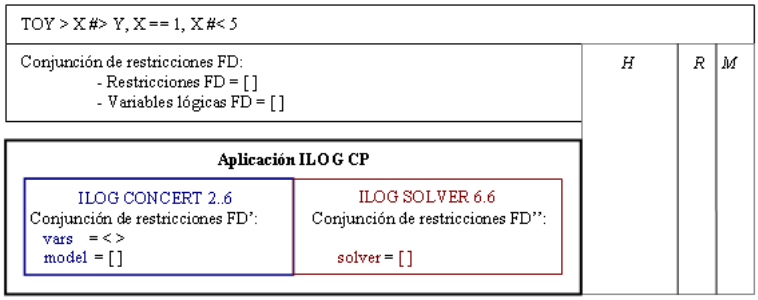

Figura 3.5: Inicio del objetivo.

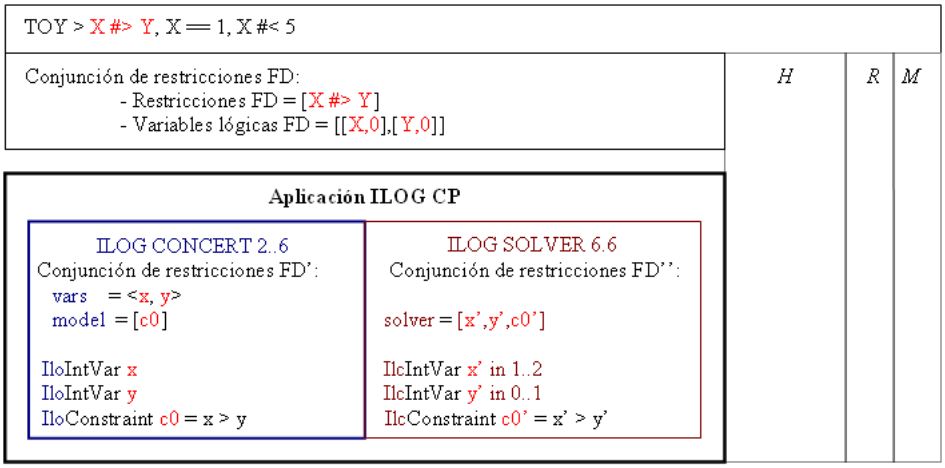

Figura 3.6: Evaluación de la primera restricción.

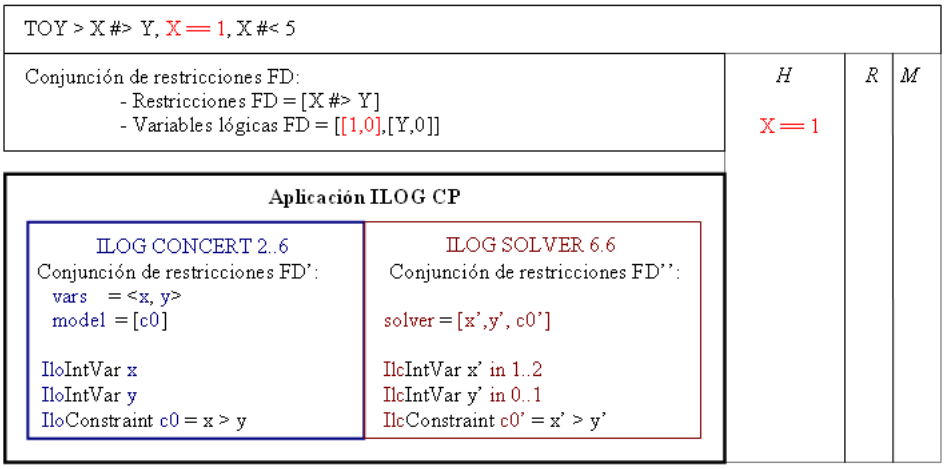

Figura 3.7: Evaluación de la segunda restricción.

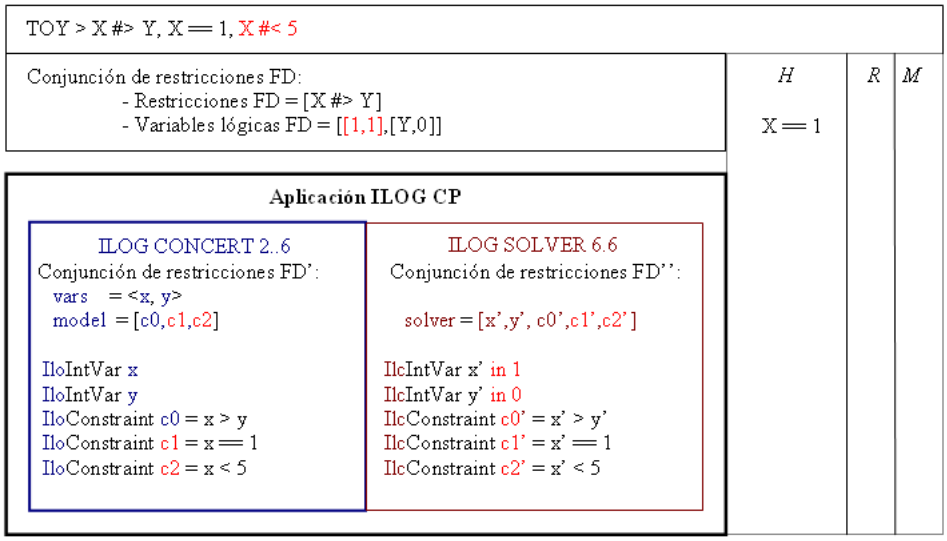

Figura 3.8: Evaluación de la tercera restricción.

En $\mathcal{TOY}(\mathcal{FD}i)$ este proceso no puede ser automático. Si una restricción de igualdad sintáctica gestionada en $\mathcal{H}$ unifica dos variables lógicas $\mathcal{FD}$, entonces se produce una inconsistencia entre estas variables lógicas y sus `IloIntVar`s asociadas. Si no se recupera esta consistencia se puede producir incorrección en las soluciones computadas por `solver`, o lo que es lo mismo, incorrección en la solución a un objetivo $\mathcal{TOY}$. Basta ver el siguiente objetivo `Y #> 7, Y == X #+ 5`, donde suponemos que toda variable lógica tiene el dominio `0...10`. Como describimos en la subsección 2.2.4, el cálculo de estrechamiento perezoso descompone la restricción $\mathcal{FD}$ compuesta `Y == X #+ 5` en las restricciones $\mathcal{FD}$ primitivas `X #+ 5 == _Z, Y == _Z`. En este caso, la restricción `Y == _Z` es una restricción entre dos variables lógicas $\mathcal{FD}$:

a) Si no imponemos la restricción `Y == _Z` sobre `model`, entonces la solución que `solver` encuentra al objetivo es: `X in 0..5, Y in 8..10, _Z in 5..10`. Esta solución es incorrecta.

b) Si imponemos la restricción `Y == _Z` sobre `model`, entonces la solución correcta que `solver` encuentra al objetivo es: `X in 3..5, Y in 8..10, _Z in 8..10`.

Para gestionar estas restricciones de igualdad sintáctica entre dos variables lógicas $\mathcal{FD}$ ampliamos el modo en que cada predicado Prolog que interacciona con la aplicación genérica ILOG CP sincroniza las `IloIntVar`s. Mostramos dos objetivos para explicar exactamente las restricciones de igualdad que queremos sincronizar:

- `X #> Z, Y #> Z, X == Y, K #> 0`. La restricción `X == Y` involucra dos variables lógicas $\mathcal{FD}$. Por lo tanto, el próximo predicado Prolog que interacciona con la aplicación genérica ILOG CP, en este caso, `K #> 0`, debe sincronizar `X == Y` imponiendo sobre `model` la `IloConstraint c2 = x == y`.

- `X #> Z, X == Y, K #> 0, Y #> Z`. La restricción `X == Y` involucra a una variable lógica $\mathcal{FD}$`X` y a una variable lógica `Y`. A continuación, el predicado `$#>` gestiona la restricción `K #> 0`. Esta restricción $\mathcal{FD}$ no convierte a `Y` en variable lógica $\mathcal{FD}$.

Por lo tanto, durante la gestión de `K #> 0` la restricción `X == Y` no es relevante para `model`, ya que no involucra a dos variables lógicas $\mathcal{FD}$. Por lo tanto no se sincroniza. Sin embargo, la restricción `Y #> Z` convierte a `Y` en variable lógica $\mathcal{FD}$, por lo ahora la restricción $\mathcal{FD}$ `X == Y` involucra a dos variables lógicas $\mathcal{FD}$. Por lo tanto es relevante para `model` y se debe sincronizar. El predicado `$#>` que gestiona `Y #> Z` sincroniza previamente esta restricción `X == Y`.

Para realizar la sincronización de dos variables lógicas $\mathcal{FD}$ unificadas en una restricción de igualdad sintáctica ampliamos el contenido de `ilog_cp.cpp` e `ilog_cp.pl`. Creamos un nuevo predicado Prolog `post_Herbrand_equality_between_vars(+Vars, -Feasible)` implementado en la función C++ `post_Herbrand_equality_between_vars(SP_term_ref Vars,long* Feasible)`. La función C++ recibe como argumento el término Prolog que constituye la lista `Variables`. Esta lista `Variables` es tratada como un tipo `SP_term_ref` dentro de la función C++.

Ampliamos el contenido de la aplicación genérica ILOG CP contenida en `ilog_cp.cpp`, incluyendo una nueva variable global `static vector< pair<int,int> > variables_linked_by_herbrand_unification`, que indica las parejas de `vars` que sido sincronizadas debido a la unificación en $\mathcal{H}$ de sus respectivas variables lógicas $\mathcal{FD}$.
Si `variables_linked_by_herbrand_unification = <(1,2),(0,1)>`, entonces sabemos que la `IloConstraint ci = vars[1] == vars[2]` e `IloConstraint cj = vars[0] == vars[1]` están contenidas en `model`.

El algoritmo que realiza la función C++ para sincronizar las `IloIntVar`s es el siguiente:

0. Copia la lista `Variables` en un `SP_term_ref` auxiliar, sobre el que trabaja.
1. Extrae la cabeza de ese `SP_term_ref`, esto es, un par `[Var,Flag]`. Guarda la cola de la lista en este `SP_term_ref`.
2. Comprueba si `Var` es una variable lógica $\mathcal{FD}$. Si no lo es, vuelve al punto 1.
3. Si lo es, efectúa el método `SP_compare(Var,V)` con todas las demas `V` de `Variables`. Al terminar volvemos al punto 1.
4. Si alguna comparación (Var,V) indica que son iguales, comprueba si la pareja de enteros (Índice_Var,Índice_V) está contenida en el vector.
5. Si lo está, entonces no es necesario imponerla sobre `model`. Si no lo está imponemos a `model` la `IloConstraint = vars[Índice_Var] == vars[Índice_V]`. Además añadimos la pareja de índices al vector.
6. Propagamos la restricción. Si `solver` es satisfactible, continuamos comparando con las restantes `V` de `Variables`. Si no, devolvemos fallo.

Modificamos el código Prolog de `post_fd_unifications`, para que ahora también sincronice las restricciones de igualdad entre dos variables lógicas $\mathcal{FD}$. De nuevo, exigimos que `Feasible == 1`, para garantizar que la conjunción de restricciones $\mathcal{FD}$" tras la sincronización sigue siendo satisfactible.

```
post_fd_unifications(Vars,NewVars) :-
    get_Herbrand_unifications(Vars,0,ListOfConstraints),
    post_equalities(ListOfConstraints),
    modify_list_of_vars(Vars,NewVars),
**  post_Herbrand_equality_between_vars(NewVars,1).
```

Con esto finaliza la descripción de la gestión de las restricciones de igualdad sintáctica en el sistema $\mathcal{TOY}(\mathcal{FD}i)$.

3.1.3. Gestión de la restricción de desigualdad en $\mathcal{TOY}(\mathcal{FD}i)$

Las dos técnicas de sincronización que utilizamos para la gestión de las restricciones de igualdad sintáctica no son aplicables a las restricciones de desigualdad sintáctica. En concreto:

1. La restricción de igualdad que unificaba una variable lógica $\mathcal{FD}$ a un valor entero, permitía detectar la aparición de un patrón `[Value,0]` en la lista `Variables`. En cambio una restricción de desigualdad entre una variable lógica $\mathcal{FD}$ y un valor entero no unifica la variable lógica $\mathcal{FD}$, por lo que no podemos detectar ningún patrón en la lista `Variables` que nos indique que se ha producido esta desigualdad.

2. La restricción de igualdad entre dos variables lógicas $\mathcal{FD}$ no dejaba rastro en la lista `Variables`. Sin embargo se podía utilizar el método `SP_compare` entre cada par de variables lógicas $\mathcal{FD}$ para saber si ambas estaban unificadas. En cambio ahora, el hecho de que `SP_compare` indique que dos variables lógicas $\mathcal{FD}$ no están unificadas no implica que exista una restricción de desigualdad impuesta entre ambas variables.

Las restricciones de desigualdad sintáctica `L /= R` son gestionadas por el predicado `notEqual` de $\mathcal{H}$. Queremos imponer sobre el sistema de restricciones $\mathcal{FD}$ las restricciones de desigualdad donde todas sus variables involucradas como argumento son en realidad variables lógicas $\mathcal{FD}$. Por la independencia de los distintos sistemas de restricciones, pretendemos que `notEqual` no se encargue de imponer estas restricciones sobre `model`. Sin embargo, por lo visto en los puntos 1 y 2, no queda más remedio que hacerlo, ya que es imposible identificar estas restricciones si no se modifica el contenido de `notEqual`.

Las modificaciones necesarias son:

i) Creamos un predicado `check_any_var_is_fd_var(V1,V2,Vars,Size,FD)`, que detecta si en una restricción de desigualdad todos los argumentos que sean variables lógicas son en realidad variables lógicas $\mathcal{FD}$. Una variable lógica es $\mathcal{FD}$ si está contenida en la lista `Variables`.

```
%check_any_var_is_fd_var/5(HL,HR,Vars,Size,Result).
%
check_any_var_is_fd_var(HL,HR,Vars,Size,1):-
    var(HL),
    is_var_in_vars_list(HL,Vars,0,IndexHL),
    IndexHL < Size,
    var(HR),
    is_var_in_vars_list(HR,Vars,0,IndexHR),
    IndexHR < Size,
    !.
%
check_any_var_is_fd_var(HL,HR,Vars,Size,1):-
    var(HL),
    is_var_in_vars_list(HL,Vars,0,Index),
    Index < Size,
```

```
    integer(HR),
    !.
%
check_any_var_is_fd_var(HL,HR,Vars,Size,1):-
    integer(HL),
    var(HR),
    is_var_in_vars_list(HR,Vars,0,Index),
    Index < Size,
    !.
%
check_any_var_is_fd_var(_,_,_,_,0).
```

ii) Ampliamos el contenido de los archivos `ilog_cp.cpp` e `ilog_cp.pl`. Creamos un predicado Prolog `post_notEqual` implementado en la función C++ `post_notEqual`. La definición del predicado es equivalente a la de `post_greater`, pero en este caso gestiona una `IloConstraint c = v1 != v2`.

iii) Modificamos el predicado `notEqual`, para que imponga la restricción `HL /= HR` sobre `model` cuando `check_any_var_is_fd_var(HL,HR,Vars,Size,1)`. Estudiamos en detalle el nuevo predicado `notEqual` modificado.

```
notEqual(L,R,Cin,Cout):-
    hnf(L,HL,Cin,Cout1),
    hnf(R,HR,Cout1,Cout2),
    find_storeLSCE_into_Cin(Cout2,Vars,Size,Constraints,Cout3),
(1) check_any_var_is_fd_var(HL,HR,Vars,Size,FD),
(2) ( FD == 1 ->
(3)     (post_fd_unifications(Vars,Vars1),
(4)      manage_argument_of_constraint(HL,Vars1,Size,Vars2,Size1,IHL,VHL),
(4)      manage_argument_of_constraint(HR,Vars2,Size1,Vars3,Size2,IHR,VHR),
(5)      append(Constraints,[HL/=HR],Constraints1),
(6)      post_notEqual(IHL,VHL,IHR,VHR,1),
(7)      append(Cout3,[store(Vars3,Size2,Constraints1)],Cout),
         !
        )
    ;
        (append(Cout3,[store(List,Size,Constraints)],Cout4),
(8)      notEqualHnf(HL,HR,Cout4,Cout),
         !
        )
    ).
```

En `(1)` se determina si todo argumento de la restricción de desigualdad que sea variable es en realidad una variable lógica $\mathcal{FD}$. En caso de que no sea así, gestionamos la restricción en $\mathcal{H}$ mediante el predicado Prolog `notEqualHnf`. En caso de que la restricción sea en realidad una restricción $\mathcal{FD}$, entonces es preciso gestionarla como una restricción $\mathcal{FD}$.

Añadimos como un último punto a tratar la sincronización del sistema de restricciones $\mathcal{FD}$ con las restricciones de desigualdad, que ya fue tratado en la implementación de

$\mathcal{TOY}(\mathcal{FD}s)$, y reproducimos de nuevo para la implementación de $\mathcal{TOY}(\mathcal{FD}i)$. Es el caso que se produce cuando en una restricción de desigualdad sintáctica no todas sus variables lógicas involucradas son variables lógicas $\mathcal{FD}$. Sin embargo, si todas estas variables lógicas involucradas se convierten posteriormente en variables lógicas $\mathcal{FD}$, entonces la restricción de desigualdad sintáctica se convertirá a posteriori en una restricción $\mathcal{FD}$, que debe ser impuesta a `model`.

En el sistema $\mathcal{TOY}(\mathcal{FD}s)$ esto implicaba dos modificaciones:

1. Como explicamos, $\mathcal{TOY}(\mathcal{FD}s)$ modificaba el contenido del predicado `notEqualHnf`. Cuando una restricción de desigualdad se gestiona en $\mathcal{H}$ mediante `notEqualHnf`, se incluyen términos en el almacén `Cin` que permitan a posteriori reconstruir esta restricción. Por ejemplo, en el objetivo `X /= Z, Y #> 0, K #> 0`, la gestión de la restricción `X /= Z` no entra en conflicto con el sistema de restricciones `clpfd`, ya que ni `X` ni `Y` son variables lógicas $\mathcal{FD}$. Sin embargo, $\mathcal{TOY}(\mathcal{FD}s)$ modifica ligeramente el código de `notEqualHnf`, que ahora incluye en el almacén `Cin` los términos `X:[Z]` y `Z:[X]`.

2. $\mathcal{TOY}(\mathcal{FD}s)$ crea el predicado `toSolverFD`, que impone a `clpfd` toda restricción de desigualdad almacenada en `Cin` que ahora se haya convertido en restricción $\mathcal{FD}$.

Respetamos el punto 1 tal y como está implementado en el sistema $\mathcal{TOY}(\mathcal{FD}s)$. Es decir, en la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ utilizamos la versión del predicado Prolog `notEqualHnf` que se utilizó en el sistema $\mathcal{TOY}(\mathcal{FD}i)$.

Adaptamos el punto 2, manteniendo la idea del predicado `toSolverFD` que implementa el sistema $\mathcal{TOY}(\mathcal{FD}s)$, pero siendo conscientes de que la imposición de una nueva restricción $\mathcal{FD}$ de desigualdad modifica la lista `Restricciones`. Estudiemos el objetivo `X /= Y, Y#> 2, X #> 3`. El predicado `$#>(Y,2,0,Cin,Cout)` deberá ahora:

i) Sincronizar las restricciones de desigualdad sintáctica que ahora se conviertan en restricciones $\mathcal{FD}$.

ii) Sincronizar las `IloIntVar`s debido a restricciones de igualdad sintáctica.

iii) Gestionar la restricción `Y #> 2`.

El contenido de `store` al que accede el predicado Prolog `$#>(Y,2,0,Cin,Cout)` es: `Variables = [], Tamaño = 0, Restricciones = []`. Por otro lado `model` y `vars` están vacíos.

El predicado Prolog `$#>(Y,2,Cin,Cout)` procede entonces con el punto i), llamando al predicado `toSolverFD`. Detecta el término `Y:[X]`. Sin embargo, como `X` no es una variable lógica $\mathcal{FD}$ no debe imponer la restricción `Y /= X` sobre `model`. Elimina `Y:[X]` de `Cin`. A continuación el predicado `$#>` gestiona la restricción `Y #> 2`. Esto modifica `Variables = [Y]`, `Tamaño = 1` y `Restricciones = [Y #> 2]`.

La gestión pasa ahora al predicado `$#>(X,3,Cin,Cout)`. En primer lugar llama al predicado `toSolverFD`. Detecta el término `X:[Y]`. Como `Y` es una variable lógica $\mathcal{FD}$ se debe imponer la restricción `X /= Y` sobre `model`. Elimina el elemento `X:[Y]` del almacén `Cin`. Al añadir `X /= Y` se modifica `Variables = [X,Y]`, `Tamaño = 2` y `Restricciones = [X /= Y]`. A continuación el predicado `$#>` gestiona la restricción $\mathcal{FD}$ `X #> 3`, utilizando `Variables = [X,Y]`, `Tamaño = 2` y `Restricciones = [X /= Y]`.

El predicado `toSolverFD` queda así:

```
%toSolverFD/9(+Arg,+Cin,-Cout,+Vars,+Size,+Constraints,-NewVars,-NewSize,
                                                          -NewConstraints).
%
toSolverFD(Arg,Cin,Cout,Vars,Size,Constraints,NewVars,NewSize,
                                                    NewConstraints):-
    var(Arg),
    !,
    extractCtr(Cin,Arg,Cout1,Inequalities_Of_Arg),
    passToSolverFD(Arg,Inequalities_Of_Arg,Cout1,Cout,Vars,Size,Constraints,
                                        NewVars,NewSize,NewConstraints).
%
toSolverFD(Arg,Cin,Cin,Vars,Size,Constraints,Vars,Size,Constraints).
```

La primera cláusula contempla el caso de que `Arg` sea una variable lógica $\mathcal{FD}$. En este caso se llama al predicado `passToSolverFD`.

La segunda cláusula contempla el caso de que `Arg` sea un valor entero, en cuyo caso no hay ninguna restricción de desigualdad que sincronizar.

El predicado `passToSolverFD` se encarga de imponer sobre `model` aquellas restricciones de desigualdad sintáctica `Arg /= V`, donde `V` sea también una variable lógica $\mathcal{FD}$:

```
%passToSolverFD/10(+Var,+Inequalities_Of_Arg,+Cin,-Cout,+Vars,+Size,
                        +Constraints,-NewVars,-NewSize,-NewConstraints).
%
passToSolverFD(_,[],Cin,Cin,Vars,Size,Constraints,NewVars,NewSize,
                                                    NewConstraints).
%
passToSolverFD(X,[Y|R],Cin,Cout,Vars,Size,Constraints,NewVars,NewSize,
                                                    NewConstraints):-
    var(Y),
    is_var_in_vars_list(Y,Vars,0,Index),
    Index < Size,
    !,
    manage_argument_of_constraint(X,Vars1,Size1,Vars2,Size2,IsX,ValX),
(1)    post_unequal(IsX,ValX,1,Index,1),
    append(Constraints,[X\=Y],Constraints1),
(2)    toSolverFD(Y,Cin,Cout1,Vars1,Size1,Constraints1,Vars2,Size2,
                                                    Constraints2),
(3)    passToSolverFD(X,R,Cout1,Cout,Vars2,Size2,Constraints2,NewVars,
                                        NewSize,NewConstraints).
%
passToSolverFD(X,[Y|R],Cin,Cout,Vars,Size,Constraints,NewVars,NewSize,
                                                    NewConstraints):-
    var(Y),
    is_var_in_vars_list(Y,Vars,0,Index),
    Index == Size,
    !,
    passToSolverFD(X,R,Cin,Cout,Vars,Size,Constraints,NewVars,NewSize,
                                                    NewConstraints).
```

```
passToSolverFD(X,[Y|R],Cin,Cout,Vars,Size,Constraints,NewVars,NewSize,
                                                        NewConstraints):-
    integer(Y),
    !,
    manage_argument_of_constraint(X,Vars1,Size1,Vars2,Size2,IsX,ValX),
    post_unequal(IsX,ValX,0,Y,1),
    append(Constraints,[X\=Y],Constraints1),
    passToSolverFD(X,R,Cin,Cout,Vars1,Size1,Constraints1,NewVars,
                                        NewSize,NewConstraints).
```

La primera cláusula indica que no hay impuesta ninguna restricción de desigualdad que involucre a `Arg`. Por lo tanto, no hay nada que sincronizar.

La segunda cláusula indica el caso en que existe una restricción de desigualdad `Arg /= Y`, donde `Y` es una variable lógica $\mathcal{FD}$:

(1) Se sincroniza con `model` la restricción `Arg /= Y`.

(2) Se efectúa `toSolverFD(Y)`, para imponer sobre `model` otras restricciones de desigualdad del tipo `Y /= V`, donde `V` sea una variable lógica $\mathcal{FD}$.

(3) Se continúa la sincronización de restricciones de desigualdad de tipo `Arg /= V`, donde `V` sea una variable lógica $\mathcal{FD}$.

La tercera cláusula indica el caso en que existe una restricción de desigualdad `Arg /= Y`, donde `Y` es una variable lógica que no pertecece a las variables lógicas $\mathcal{FD}$. En este caso no se sincroniza esta restricción y se continúa la búsqueda de desigualdades sobre `Arg`.

La cuarta cláusula indica el caso en que existe una restricción de desigualdad `Arg /= Y`, donde `Y` es un valor entero. Se sincroniza sobre `model` la restricción `Arg /= Y`. Continúa la búsqueda de desigualdades sobre `Arg`.

La versión definitiva del predicado `notEqual HL /= HR` incluye la sincronización de desigualdades sobre los argumentos `HL` y `HR`.

```
notEqual(L,R,Cin,Cout):-
    hnf(L,HL,Cin,Cout1),
    hnf(R,HR,Cout1,Cout2),
    find_storeLSCE_into_Cin(Cout2,Vars,Size,Constraints,Cout3),
(1) check_any_var_is_fd_var(HL,HR,Vars,Size,FD),
(2) ( FD == 1 ->
(3)   (toSolverFD(HL,Cout3,Cout4,Vars,Size,Constraints,Vars0,
                                            Size0,Constraints0),
       toSolverFD(HR,Cout4,Cout5,Vars0,Size0,Constraints0,Vars1,
                                            Size1,Constraints1),
       post_fd_unifications(Vars1,Vars2),
(4)    manage_constraint_argument(HL,Vars2,Size1,Vars3,Size2,IHL,VHL),
(4)    manage_constraint_argument(HR,Vars3,Size2,Vars4,Size3,IHR,VHR),
(5)    append(Constraints1,[HL/=HR],Constraints2),
(6)    post_notEqual(IHL,VHL,IHR,VHR,1),
(7)    append(Cout5,[store(Vars4,Size3,Constraints2)],Cout),
       !
       )
```

```
    ;
        (append(Cout3,[store(List,Size,Constraints)],Cout4),
(8)      notEqualHnf(HL,HR,Cout4,Cout),
         !
        )
    ).
```

Modificamos también los predicados de $solve^{FD}$ que implementamos en la versión de $\mathcal{TOY}(\mathcal{FD}i)$ de la sección 2.2. Mostramos esta modificación sobre el predicado `$#>`:

```
% $#>/5(+L,+R,-Out,+Cin,-Cout).

  $#>(L, R, Out, Cin, [store(Vars5,Size4,[HL>HR|Constraints2])|Cinter5]):-
    hnf(L, HL, Cin, Cinter1),
    hnf(R, HR, Cinter1, Cinter2),
    find_store_in_Cin(Cinter2,Vars,Size,Constraints,Cinter3),
**  toSolverFD(HL,Cinter3,Cinter4,Vars,Size,Constraints,Vars1,Size1,Constraints1),
**  toSolverFD(HR,Cinter4,Cinter5,Vars1,Size1,Constraints1,Vars2,Size2,Constraints2),
    post_fd_unifications(Vars2,Vars3),
    manage_constraint_argument(HL,Vars3,Size2,Vars4,Size3,IsHL,ValHL),
    manage_constraint_argument(HR,Vars4,Size3,Vars5,Size4,IsHR,ValHR),
    post_greater(IsHL,ValHL,IsHR,ValHR,1).
```

3.2. Gestión de las acotaciones deducidas por `solver`

Cuando una variable lógica $\mathcal{FD}$ se unificaba a un valor en el sistema $\mathcal{TOY}(\mathcal{FD}s)$, toda aparición de esa variable en el sistema apuntaba a dicho valor. Por un lado, cuando $\mathcal{H}$ unificaba una variable lógica $\mathcal{FD}$, sus apariciones en clpfd quedaban automáticamente reemplazadas. Por otro lado, cuando clpfd unificaba una variable lógica $\mathcal{FD}$ a un valor entero, cualquier otra aparición de dicha variable lógica $\mathcal{FD}$ en el sistema $\mathcal{TOY}$ quedaba automáticamente reemplazada. En la sección anterior estudiamos cómo implementar el primero de estos casos en el sistema $\mathcal{TOY}(\mathcal{FD}i)$. En esta sección nos centramos en describir la forma de implementar el segundo caso.

Cada nueva `IloConstraint` impuesta sobre `model` debe propagarse. Mediante esta propagación se podan valores de los dominios de las `IlcIntVar`s contenidas en `solver`. Si el dominio de alguna de estas `IlcIntVar` pasa a ser vacío, entonces se afirma que la conjunción de restricciones $\mathcal{FD}$" sobre la que trabaja `solver` es insatisfactible. Desde el punto de vista de SICStus Prolog, si `solver` acota el dominio de una `IlcIntVar` a un único valor, entonces está unificando esa `IlcIntVar` a dicho valor. Por lo tanto, debemos transmitir la acotación de una `IlcIntVar` a un valor `Value`, para que toda aparición en el sistema $\mathcal{TOY}$ de la variable lógica $\mathcal{FD}$ asociada a esta `IlcIntVar` se vea automáticamente reemplazada por `Value`.

Este proceso no puede ser automático en el sistema $\mathcal{TOY}(\mathcal{FD}i)$. Sin embargo, para unificar toda variable lógica $\mathcal{FD}$ contenida en el sistema $\mathcal{TOY}$ basta con unificar la variable lógica $\mathcal{FD}$ contenida en la lista `Variables`. Recordemos que existe una relación entre:

- La variable lógica $\mathcal{FD}$, accesible desde la posición `i` de la lista `Variables`.
- La variable lógica $\mathcal{FD}$', accesible desde la posición `i` del vector `vars`.

- La variable lógica $\mathcal{FD}$", accesible mediante el método `solver.getIntVar(vars[i])`.

Esta relación permite, dada una variable lógica $\mathcal{FD}$, acceder a su objeto `IlcIntVar` asociado, sobre el que trabaja `solver`. Basta con:

1. Obtener el índice `i` que ocupa la variable lógica $\mathcal{FD}$ en la lista `Variables`.
2. Utilizar ese índice para acceder a su `IloIntVar` asociada, contenida en `vars[i]`.
3. Utilizar esa `IloIntVar` para acceder a su `IlcIntVar` asociada, a través del método `IlcIntVar solver.getIntVar(IloIntVar)`.

Los dos siguientes objetivos motivan la necesidad de implementar esta sincronización:

- `X == Y, Y #> 1, Y #< 3, X == 0`. Este objetivo falla. La restricción `X == Y` es gestionada en $\mathcal{H}$. Posteriormente, la gestión de `Y #> 1` o `Y #< 3` no sincroniza esta restricción, ya que `X` no es una variable lógica $\mathcal{FD}$. Tras la propagación de `Y #< 3` `solver` acota al valor `2` el dominio de `IlcIntVar y`, asociada a `Y`.

  - Si no sincronizamos `Y`, la aparición de `Y` en la lista `Variables` no será unificada. Se mantendrá como el par `[Y,0]`.
    El cómputo continúa con la evaluación de `X == 0`, gestionada en $\mathcal{H}$, que unifica `X` a `0`. Como `X == Y`, entonces también unifica `Y` con el valor `0`. En este momento, la variable `Y` de `Variables` está unificada a `0`, su par es `[0,0]`, mientras que su `IlcIntVar y'` asociada está acotada al valor `2`.

    Al haber finalizado (aparentemente) el cómputo del objetivo, se llama al predicado `showAnswer`, que realiza la sincronización del par `[0,0]` de `Variables`, imponiendo la `IloConstraint c2 = y == 0` sobre `model`. Para propagar `c2`, `solver` crea la `IlcConstraint c2' = y' == 0`. La propagación de esta última hace a la nueva conjunción de restricciones $\mathcal{FD}$" insatisfactible, ya que `y'` no puede ser 2 y 0 al mismo tiempo.

  - Si en la gestión de `Y #< 3`, tras acotar `y'` a `2`, unificamos `Y` al valor `2`, entonces, debido a `X == Y`, $\mathcal{H}$ unifica `X` a `2`. Posteriormente la gestión de `X == 0` falla, ya que `X` está unificado a `2`.

  No sincronizar las variables lógicas $\mathcal{FD}$ cuyas `IlcIntVar`s asociadas han sido acotadas puede generar problemas de ineficiencia, al no detectar inmediatamente los fallos que surgen.

- El sistema $\mathcal{FD}$+$\mathcal{R}$ permite que coexistan variables lógicas $\mathcal{FD}$ con variables lógicas $\mathcal{R}$. Un puente puede unificar una variable lógica $\mathcal{FD}$ con una variable lógica $\mathcal{R}$. Si la variable lógica $\mathcal{FD}$ es acotada por `solver` y ese valor no es transmitido a $\mathcal{R}$, entonces pueden producirse inconsistencias en la solución obtenida. `X == Y, Y #> 0, X > 3.6, Y #< 2` debería fallar, ya que `Y` es acotada a `1`, por lo que `X` no puede ser mayor que `3.6`. Sin embargo, si no sincronizamos la variable lógica `Y` contenida en `Variables` con su `IlcIntVar` asociada, este valor no se transmitirá.

Tras motivar que esta sincronización es necesaria, vamos a centrarnos en la forma de llevarla a cabo.

Hasta ahora, el predicado `post_greater` se encarga de:

1. Imponer la `IloConstraint` asociada sobre `model`.
2. Hacer que `solver` propague esta `IloConstraint`, creando primero su `IlcConstraint` asociada.
3. Obtener si la nueva conjunción de `IlcConstraint`s sobre las que trabaja `solver` es o no satisfactible.

Ahora añadimos una nueva tarea. En caso de que la conjunción de `IlcConstraint`s sea satisfactible, la propia función C++ `post_greater` debe:

- Chequear si alguna `IlcIntVar` ha sido acotada fruto de esta última propagación.
- Para toda `IlcIntVar` que esté en esta situación modificar su par (variable lógica $\mathcal{FD}$, flag) asociado en la lista `Variables`.

Esto implica modificar las definiciones del predicado Prolog `post_greater` y de la función C++ `post_greater` que la implementa, a las que añadimos dos nuevos argumentos: la lista `Variables`, que se pasa como argumento de entrada, y la lista `New Variables`, que resulta de sincronizar toda nueva `IlcIntVar` que haya sido acotada en la última propagación. Veamos la nueva definición de `post_greater`, que es llamada dentro del predicado Prolog `$#>`.

- El predicado Prolog
  `post_greater/7(+IsHL,+ValHL,+IsHR,+ValHR,+Vars,-NewVars,-Feasible).`
- La función C++
  `post_greater(long IsHL, long ValHL, long IsHR, long ValHR, SP_term_ref Vars, SP_term_ref NewVars, long* Feasible)`

Una primera alternativa para implementar esto consiste en:

- Utilizar la función `int SP_get_list(SP_term_ref t, SP_term_ref head, SP_term_ref tail)` para procesar de uno en uno los pares (variable lógica $\mathcal{FD}$, flag) de la lista `Variables`.
- Acceder a cada `IlcIntVar` sobre la que trabaja `solver` a través de su variable lógica $\mathcal{FD}$ contenida en el par.
- Utilizar los métodos `int IlcIntVar::isBound()` e `int IlcIntVar::getValue()` para conocer si la `IlcIntVar` está acotada, y en caso de estarlo obtener dicho valor `Value`.
- Si está acotada, modificar el par de `Variables` unificando la variable lógica $\mathcal{FD}$ con `Value`. Para ello se utiliza el método `int SP_unify(SP_term_ref t1, SP_term_ref t2)`. Además hay que modificar `Flag`, que ahora pasa a valer `1`, ya que `solver` conoce que esta variable lógica $\mathcal{FD}$ está unificada.

Sin embargo esta alternativa tiene dos fuentes de ineficiencia. La primera es preguntar a toda variable `IlcIntVar` si está acotada. La segunda es que el hecho de que esté acotada no implica que haya sido acotada debido a la última propagación, lo que puede hacer

que sincronicemos una misma `IlcIntVar` múltiples veces a lo largo de un objetivo. En el objetivo `X #> 0, X #< 2, Y #> 0` sólo queremos sincronizar la `IlcIntVar` asociada a `X` tras la gestión de `X #< 2`, que es donde realmente se ve acotada. No tiene sentido volver a sincronizarla en la gestión de `Y #> 0`, donde de nuevo la `IlcIntVar` estará acotada.

Para mejorar estas dos fuentes de ineficiencia realizamos una segunda versión de la sincronización basada en la detección de eventos y el uso de demonios.

3.2.1. Implementación eficiente de la sincronización

ILOG permite asociar el evento de la acotación de una `IlcIntVar` asociada a una `IloIntVar` con la ejecución automática de una cierta porción de código. Esto se basa en cuatro pilares:

1. La creación de dos vectores que contengan la nueva información a sincronizar y la información previamente sincronizada respectivamente.
2. La creación de una nueva clase de `IlcConstraint`s en `ILOG Solver 6.6` sobre la que `solver` pueda trabajar.
   - 2.1. Definición de la clase implementación `IlcConstraintI`, así como de los distintos métodos que la conforman.
   - 2.2. Definición de un nuevo demonio asociado a un evento de activación y que ejecuta una cierta porción de código.
   - 2.3. Definición de la clase manejador `IlcConstraint`.
3. La asociación de esta nueva clase de `IlcConstraint` a una nueva clase de `IloConstraint`, que pueda ser impuesta sobre `model`.

Utilizando estos tres pilares, cada función C++ `post_greater` encuentra, tras su etapa de propagación, la información que debe sincronizar con la lista `Variables`. Esta información está contenida en uno de los vectores creados en 1. Basta entonces con procesar la lista `Variables` y modificar los elementos indicados por el vector. Esto nos permite sincronizar sólo aquellas `IlcIntVar` que han sido vinculadas por la última propagación, sin necesidad de preguntar a todas si han sido vinculadas y garantizando que no se realicen múltiples sincronizaciones de una misma `IlcIntVar`.

Pasamos a explicar los pilares de esta implementación con más detalle:

1. Se crean los dos siguientes vectores:

```
static vector< pair<int,int> > vars_to_synchronize;
static vector< pair<int,int> > vars_synchronized;
```

   El primer vector contiene la información que cada función `post_greater` debe sincronizar con la lista `Variables`. Esta información es un conjunto de pares de enteros. La primera componente indica el índice de `Variables` que se debe sincronizar. La segunda componente indica el valor al que se debe unificar la variable lógica $\mathcal{FD}$ mediante el método `SP_unify`. Al final de cada función `post_greater` se borra el contenido de este vector.

El segundo vector contiene los pares que ya han sido sincronizados en alguna ocasión con la lista Variables. Su cometido es que estos pares no se sincronizen de nuevo. Cada par candidato a ser almacenado en el vector vars_to_synchronize es contrastado con el contenido del vector vars_synchronized. Si el par ya se encontraba en este último, entonces no se añade al vector vars_to_synchronize. Si el candidato no se encuentra en vars_synchronized, entonces se añade a ambos vectores. Al primero, para que la función post_greater lo sincronice con la lista Variables, y al segundo para que ninguna futura función post_greater lo vuelva a sincronizar.

2. ILOG Solver 6.6 permite crear nuevas clases de restricciones IlcConstraints, sobre las que el resolutor solver pueda trabajar. Nosotros creamos una nueva clase de restricciones IlcCheckWhenBound, de aridad 2, que involucra a una IlcIntVar y al índice que su IloIntVar asociada ocupa en el vector vars. Definimos esta nueva IlcConstraint IlcCheckWhenBound para que ante el evento de que su IlcIntVar involucrada se acote a un determinado valor Value, entonces automáticamente se genere el candidato (Index,Value), para que sea enviado a los vectores creados en 1.

   Para toda IlcIntVar se impone una IlcCheckWhenBound. La función post_greater genera en su etapa de propagación tantos candidatos (Index,Value) como IlcIntVars se vean acotadas. El segundo vector filtra sólo los candidatos relacionados con las IlcIntVars acotadas en esta última propagación.

   Describimos ahora la implementación de esta nueva clase de restricciones. Para cada clase de objetos utilizada, ILOG exige la creación de dos clases de objetos, la clase implementación NombreClase_I, que describe la implementación de la clase, y la clase manejador NombreClase, que es la que se facilita al usuario para su utilización en las aplicaciones ILOG CP.

   2.1 El código que se muestra a continuación define la cabecera de la clase IlcCheckWhenBoundI.

```
class IlcCheckWhenBoundI : public IlcConstraintI {
protected:
    IlcIntVar _x;
    long _index;
public:
    IlcCheckWhenBoundI(IloSolver s, IlcIntVar x, long index):
        IlcConstraintI(s), _x(x), _index(index) {}
    ~IlcCheckWhenBoundI() {}
    virtual void post();
    virtual void propagate() {}
    void varDemon();
long getIndex();
    IlcIntVar getVar();
};
```

   Se puede observar que contiene los dos argumentos de su aridad, la IlcIntVar, así como el índice que su IloIntVar ocupa en vars. Además tiene como argumento el resolutor de restricciones solver que trabaja sobre ella.

Tiene dos métodos, `getIndex` y `getVar`, que permiten acceder a los argumentos de la restricción.

Además, hay que redefinir el método `varDemon`, que será ejecutado automáticamente cuando la `IlcIntVar` se acote. Este método genera el candidato `(Index,Value)` y chequea su inserción en los vectores definidos en 1.

Igualmente hay que redefinir el método `post` que especifica los eventos ante los que esta `IlcCheckWhenBoundI` debe propagar. En nuestro caso queremos que propague cuando la `IlcIntVar` involucrada en la restricción se acote. Para ello, existe el método `IlcIntVar::whenValue(IlcDemon d)`, que activa el demonio `d` cuando la `IlcIntVar` acota su dominio a un valor.

2.2 La propia definición de `whenValue` nos exige la creación de un demonio. Crearemos un demonio que esté asociado al método `varDemon` de la clase de restricciones `IlcCheckWhenBoundI`. Cuando este demonio se active se ejecutará automáticamente el método `varDemon`.

Utilizamos la macro de ILOG para la creación de demonios.

```
ILCCTDEMON0(RealizeVarBound, IlcCheckWhenBoundI, varDemon);
```

Esta macro indica que el demonio tiene 0 argumentos y por lo tanto no tiene que especificar su tipo. Simplemente crea el nuevo tipo de demonios `RealizeVarBound` asociado al método de la clase de restricciones antes mencionado.

2.3 Para asociar una clase manejador con su clase implementación, ILOG proporciona una macro. Utilizamos dicha macro para asociar la clase `IlcCheckWhenBound`, que podrá utilizar el usuario, con la clase `IlcCheckWhenBoundI` que acabamos de definir.

```
IlcConstraint IlcCheckWhenBound(IloSolver s, IlcIntVar x, long index) {
    return new (s.getHeap()) IlcCheckWhenBoundI(s, x, index);
}
```

Esta macro especifica la creación de un nuevo objeto `IlcCheckWhenBoundI` para los objetos `IlcCheckWhenBound` definidos en la aplicación genérica ILOG CP. Obviamente hay que especificar los argumentos que utilizará cada nuevo `IlcCheckWhenBoundI` creado.

3. Por último, como estamos viendo a lo largo del trabajo, cuando se utiliza una aplicación genérica ILOG CP, no se trabaja directamente con ILOG Solver 6.6, sino que se trabaja con ILOG Concert 2.6. Toda información contenida en `model` es traducida (de un modo más o menos eficiente según vimos en la subsección 2.2.2) al resolutor `solver`.

   Por ello, para que `solver` trabaje sobre una `IlcCheckWhenBound(IloSolver solver, IlcIntVar v', long i)` es necesario imponer sobre `model` una `IloCheckWhenBound(IloEnv env, IloIntVar v, long i)`. ILOG proporciona una macro para hacer esta asociación, que nosotros utilizamos a continuación:

```
ILOCPCONSTRAINTWRAPPER2(IloCheckWhenBound, solver, IloIntVar, _v, long, _i) {
    use(solver, _v);
```

```
        return IlcCheckWhenBound(solver, solver.getIntVar(_v), _i);
    }
```

Con esto hemos alcanzado un sistema de eventos que permite detectar cada acotación de una `IlcIntVar` asociada a una `IloIntVar` contenida en `model`. Por último, hay que modificar la función C++ que crea una nueva `IloIntVar` y la introduce en el vector `vars` en la última posición. Esta función C++ deberá ahora imponer sobre `model` la restricción `IloCheckWhenBound()` con la `IloIntVar` y su índice en `vars`.

3.3. Gestión del backtracking

En el sistema $\mathcal{TOY}(\mathcal{FD}i)$ vamos a tener dos fuentes de indeterminismo. La primera fuente viene dada por el etiquetado de un conjunto de variables lógicas $\mathcal{FD}$ cada una de ellas con un conjunto de posibles valores que pueden tomar. Esto da pie a un procedimiento de búsqueda, con el que se pueden encontrar las distintas soluciones como combinaciones de esos posibles valores. Esto será tratado en la sección 3.4. La segunda fuente viene dada por la utilización de funciones indeterministas definidas por varias reglas. Estas funciones obligan al cálculo de estrechamiento a elegir una de las reglas y proceder con su evaluación. Podemos decir que la evaluación de cada una de estas definiciones constituye una rama de cómputo y que la elección entre todas ellas un punto de decisión. Tras explorar una rama de cómputo (hasta encontrar fallo o encontrar una solución) el cálculo de estrechamiento perezoso puede retornar al punto de decisión y continuar evaluando una rama de cómputo diferente. Cuando esto ocurre decimos que se ha producido backtracking o vuelta atrás sobre el punto de decisión que constituye esta función indeterminista. En esta sección nos encargamos de explicar cḿo gestionar el backtracking (o vuelta atrás) en el sistema $\mathcal{TOY}(\mathcal{FD}i)$.

3.3.1. Dificultades a solventar para manejar backtracking en $\mathcal{TOY}(\mathcal{FD}i)$

Describimos mediante un ejemplo la situación que deseamos manejar. El siguiente programa $\mathcal{TOY}$ define una función indeterminista:

```
f1 :: int -> bool
f1 A = true <== A #> 5
f1 A = true <== A #< 5
```

Ejecutamos el objetivo `f1 X, X #> Y, f1 Y`, donde las variables `X` e `Y` tienen el dominio `0...10`. El sistema $\mathcal{TOY}$ debe encontrar las tres siguientes soluciones:

1. `X #> Y, X in 7..10, Y in 6..9`
2. `X #> Y, X in 6..10, Y in 0..4`
3. `X #> Y, X in 1..4, Y in 0..3`

Para encontrar estas tres soluciones, el cálculo de estrechamiento evalúa todas las ramas de cómputo generadas por la función indeterminista `f1`. La Figura 3.9 muestra las distintas ramas evaluadas para computar estas soluciones.

Los círculos representan los puntos de decisión que se generan debido a `f1`. El primer punto obliga al cálculo de estrechamiento a elegir entre evaluar una primera rama de

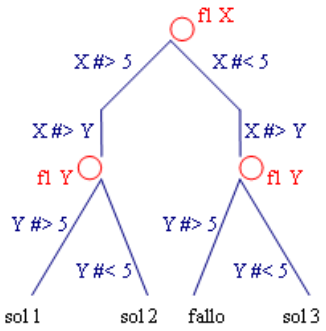

Figura 3.9: Evaluación del objetivo.

cómputo con `X #> 5` o una segunda rama de cómputo con `X #< 5`. Los otros dos puntos de decisión obligan al cálculo de estrechamiento a elegir entre `Y #> 5` o `Y #< 5`. La implementación del cálculo de estrechamiento elige entre las distintas ramas de cómputo en el orden textual en que éstas aparecen en la definición de la función indeterminista. De este modo, en los puntos de decisión descritos en el ejemplo, el cálculo de estrechamiento evalúa en primer lugar la primera rama de cómputo `A #> 5`, y luego efectúa backtracking para evaluar la segunda rama `A #< 5`.

La situación que queremos reproducir en el sistema $\mathcal{TOY}(\mathcal{FD}i)$ es la siguiente:

En todo punto del cómputo deben ser consistentes la conjunción de restricciones $\mathcal{FD}$ (almacenada en el elemento `store` del almacén `Cin`), la conjunción de restricciones $\mathcal{FD}$' (almacenada en `model`) y la conjunción de restricciones $\mathcal{FD}$" (almacenada en `solver`).

En la Figura 3.10 podemos ver que el estado de las tres es consistente al inicio de la evalución del objetivo.

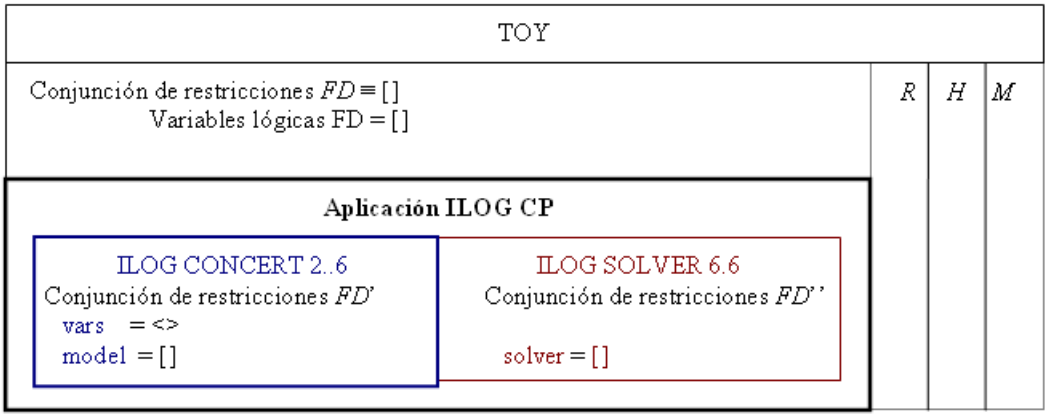

Figura 3.10: Inicio del objetivo.

El cálculo de estrechamiento gestiona la expresión `f1 X`. Esta expresión contiene una función indeterminista, por lo que se encuentra ante un punto de decisión. Debe elegir entre las dos ramas de cómputo que esta función `f1 X` genera. Como evalúa las ramas en orden, en este caso debe gestionar la restricción `X #> 5`. Podemos ver el estado en la

Figura 3.11. Tras gestionar `X #> 5` continúa con la gestión de `X #> Y`. En ese punto el estado del sistema $\mathcal{TOY}(\mathcal{FD}i)$ se puede ver en la Figura 3.12.

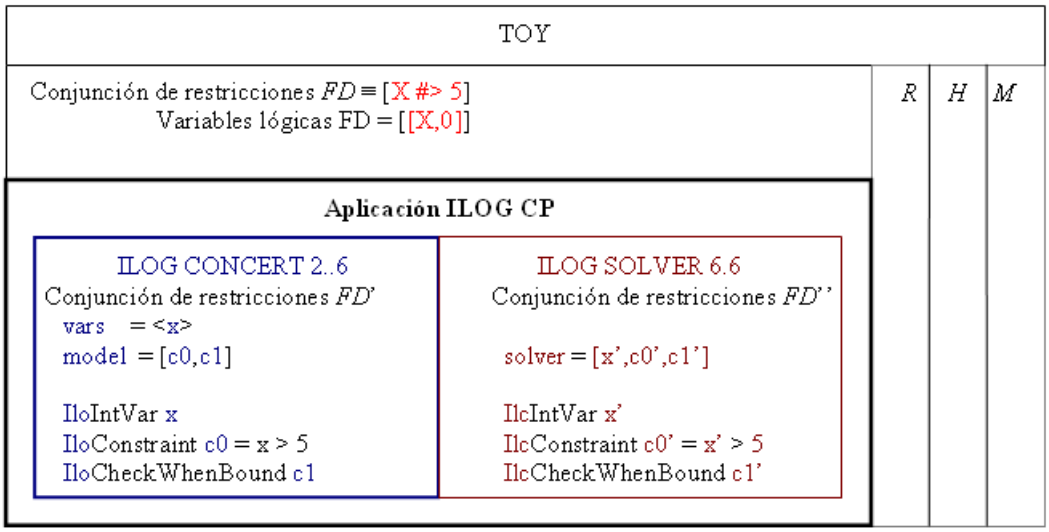

Figura 3.11: Estado tras `X` mayor que 5.

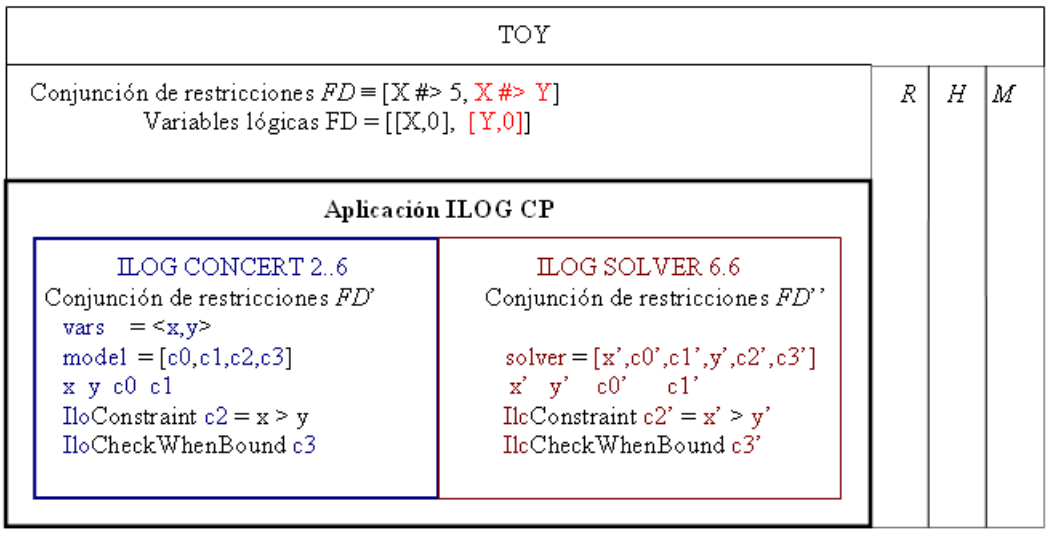

Figura 3.12: Estado tras `X` mayor que `Y`.

El cálculo debe ahora evaluar la expresión `f1 Y`. De nuevo se encuentra ante un punto de elección. Gestiona la primera rama `Y #> 5`. El estado del sistema $\mathcal{TOY}(\mathcal{FD}i)$ puede verse en la Figura 3.13. La evaluación del objetivo concluye con éxito, por lo que el sistema muestra al usuario la solución `X #> Y, X in 7..10, Y in 6..9`.

Si el usuario decide buscar nuevas soluciones para el objetivo entonces se produce backtracking sobre el último punto de decisión, situado antes de la evaluación de la expresión `f1 Y`. En este punto, el estado que el sistema $\mathcal{TOY}(\mathcal{FD}i)$ debería tener se corresponde de

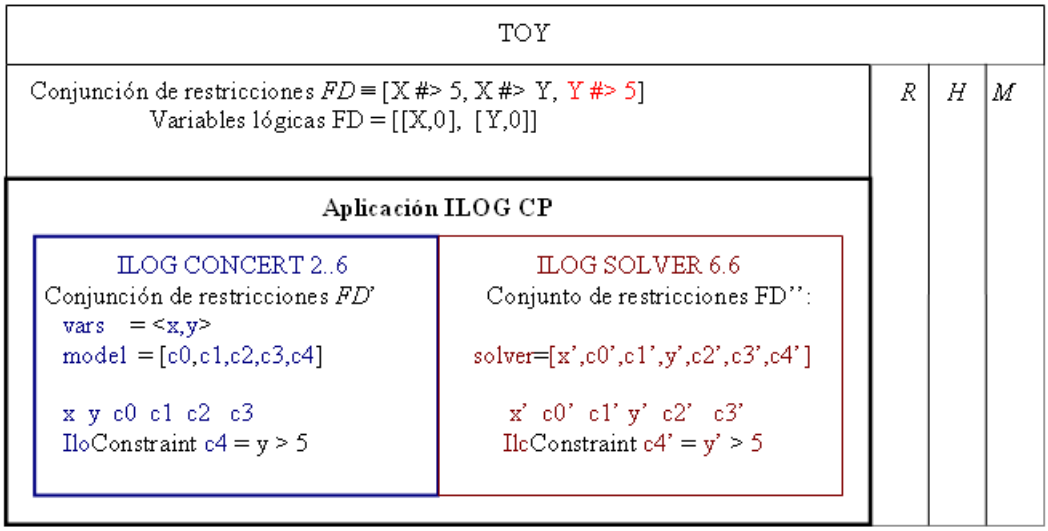

Figura 3.13: Primera solución al objetivo.

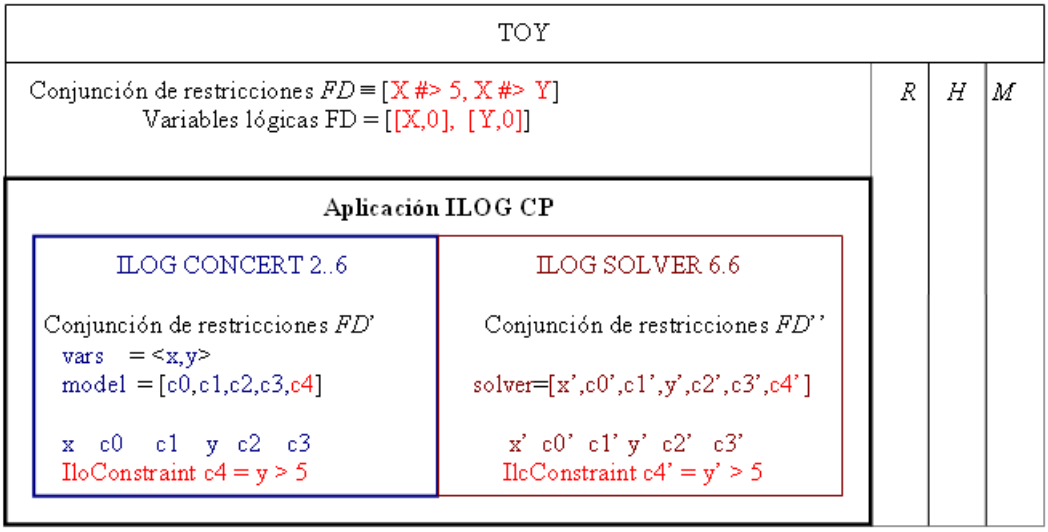

Figura 3.14: Estado erróneo producido por el backtracking.

nuevo con el descrito en la Figura 3.12. Sin embargo, en la Figura 3.14 observamos que el nuevo estado no es exactamente ése:

1. Al ser $\mathcal{TOY}$ un sistema implementado en SICStus Prolog, el backtracking se produce de manera automática. Por ello, si el cálculo de estrechamiento debe explorar una nueva rama de cómputo, el backtracking se produce automáticamente, de manera transparente al usuario. La Figura 3.14 muestra como la conjunción de restricciones $\mathcal{FD}$ recupera automáticamente el estado que esta conjunción de restricciones $\mathcal{FD}$ tenía antes del punto de decisión `f1 Y`.

2. La aplicación genérica ILOG CP está implementada en C++, por lo que el backtracking no puede producirse de manera automática. La Figura 3.14 muestra cómo las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" no se ven afectadas por el backtracking y mantienen el estado que tenían tras evaluar la primera rama de cómputo de `f1 Y`.

Encontramos por tanto que cada vez que se produce backtracking en el sistema $\mathcal{TOY}(\mathcal{FD}i)$, las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" dejan de ser consistentes con el estado actual del cómputo del objetivo $\mathcal{TOY}$. La conjunción de restricciones $\mathcal{FD}$ sí que se mantiene consistente, por lo que se produce una inconsistencia entre el estado de las conjunciones de restricciones $\mathcal{FD}$ y el estado de las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$". Esto implica que las futuras soluciones encontradas para el objetivo $\mathcal{TOY}$ no serán correctas, por lo que la actual implementación de $\mathcal{TOY}(\mathcal{FD}i)$ no es capaz de encontrar la segunda y tercera solución al objetivo. Debemos restablecer explícitamente el estado de las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" para que sean consistentes con el estado de la conjunción de restricciones $\mathcal{FD}$.

3.3.2. Recuperación de consistencia entre conjunciones de restricciones

Describimos las distintas alternativas exploradas para restablecer el estado de las conjunciones $\mathcal{FD}$' y $\mathcal{FD}$" tras producirse backtracking sobre un punto de decisión.

Recuperación utilizando la conjunción de restricciones $\mathcal{FD}$

Cualquier alternativa para llevar a cabo esta tarea utiliza el hecho de que la conjunción de restricciones $\mathcal{FD}$ si está restablecida y es consistente con el actual punto de evaluación del objetivo $\mathcal{TOY}$. Esta conjunción de restricciones se encuentra contenida en el elemento `store(Variables,Restricciones)` del almacén `Cin`. Utilizando esta información se realiza un primer intento de recuperación del estado de las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" basado en los siguientes puntos:

1. Recuperar la consistencia de la conjunción de restricciones $\mathcal{FD}$' con la conjunción de restricciones $\mathcal{FD}$.

   1.1 Recuperar la consistencia entre el vector `vars` y la lista `Variables`.

   1.2 Recuperar la consistencia entre `model` y la lista `Restricciones`.

2. Recuperar la consistencia de la conjunción de restricciones $\mathcal{FD}$" con la conjunción de restricciones $\mathcal{FD}$'.

Para recuperar la consistencia entre la conjunción de restricciones $\mathcal{FD}$' y la conjunción de restricciones $\mathcal{FD}$ eliminamos los últimos elementos contenidos en `vars` y `model`. Esto

es debido a que `Variables`, `vars`, `Restricciones` y `model` almacenan sus elementos en el orden en que estos les son introducidos, esto es, en el orden textual en que aparecen en la evaluación del objetivo $\mathcal{TOY}$. Por otro lado, el backtracking retorna desde un cierto punto de cómputo del objetivo hasta el último punto de decisión evaluado. Para ello deshace todas las restricciones $\mathcal{FD}$ gestionadas desde dicho punto de decisión al punto de evaluación actual. Estas restricciones $\mathcal{FD}$ así como sus variables lógicas $\mathcal{FD}$ involucradas se habían ido añadiendo al final de `Variables`, `vars`, `Restricciones` y `model`. Al producirse el backtracking el estado de `Variables` y `Restricciones` se restaura automáticamente, pero no así el de `vars` y `model`.

La recuperación de la consistencia podría consistir en algo tan simple como la adaptación del tamaño de `vars` y `model` al tamaño de `Variables` y `Restricciones` respectivamente. Esta alternativa de recuperación es válida para `vars` pero no lo es para `model`. Las listas `Variables` y `vars` son consistentes y evolucionan paralelamente como vimos en la subsección 2.2.1. Sin embargo, las listas `Restricciones` y `model` no son consistentes. Hay varias restricciones que son impuestas sobre `model` pero no se añaden a `Restricciones`:

- Como vimos en la subsección 3.2, para sincronizar las `IlcIntVar` acotadas con sus variables lógicas $\mathcal{FD}$ asociadas en la lista `Variables` es necesario imponer una restricción `IloCheckWhenBound` sobre cada `IloIntVar` contenida en `vars`. Estas restricciones son impuestas sobre `model` pero no se añaden a `Restricciones`.

- Las restricciones de igualdad sintáctica entre una variable lógica $\mathcal{FD}$ y un valor o entre dos variables lógicas $\mathcal{FD}$ se gestionan en $\mathcal{H}$. Estas restricciones son impuestas sobre `model` pero no sobre restricciones.

- Restricciones $\mathcal{FD}$, como `domain` o `labeling`, se imponen sobre `model` pero no se añaden a `Restricciones`. Cabe recordar que `Restricciones` fue introducida en `store` para una mayor eficiencia a la hora de mostrar al usuario las soluciones al objetivo $\mathcal{TOY}$ computado. Mediante la utilización de `Restricciones` evitamos analizar las `IlcConstraint`s sobre las que trabaja `solver`.

Por lo tanto, mediante el tamaño de `Variables` podemos restaurar el tamaño de `vars`, ya que el hecho de que el tamaño actual de `Variables` sea `k` unidades menor que el tamaño de `vars` indica que se ha producido backtracking y que se deben eliminar las últimas `k` `IloIntVar`s de `vars`. Sin embargo mediante el tamaño de `Restricciones` no podemos restaurar el tamaño de `model` ya que el tamaño de `Restricciones` es siempre menor que el tamaño de `model`, se haya o no producido backtracking. Además, como hemos visto en el ejemplo descrito, siempre que se produce backtracking el número de restricciones $\mathcal{FD}$ disminuye, mientras que el número de variables lógicas $\mathcal{FD}$ no tiene por qué hacerlo.

Debemos por tanto explorar otras alternativas para recuperar la consistencia entre la conjunción de restricciones $\mathcal{FD}$' y la conjunción de restricciones $\mathcal{FD}$. Estas nuevas alternativas han de ser capaces de restaurar el contenido de `model` con el que tenía antes del punto de decisión sobre el que se ha producido backtracking.

Recuperación ampliando el contenido de `store`

Cada predicado Prolog de $solve^{FD}$ que gestiona una nueva restricción $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ debe tener acceso a dos valores enteros:

- El tamaño que `model` debería tener en este punto del cómputo.

- El tamaño que `model` realmente tiene.

En las Figuras 3.12 y 3.14 vimos que antes de la evaluación de la segunda rama de cómputo de `f1 Y`, el tamaño que debería tener `model` es 4, mientras que el tamaño que realmente tiene es 5, por lo que se detecta que se ha efectuado backtracking.

Para que cada predicado Prolog tenga acceso a estos dos valores es necesario realizar los siguientes cambios:

1. Crear en la aplicación genérica ILOG CP una variable estática `long model_size` que representa el tamaño de `model`.
2. Modificar las funciones C++ de la aplicación genérica ILOG CP, para que cada vez que se imponga una nueva `IloConstraint` sobre `model`, inmediatamente se aumente el tamaño de `model_size`.
3. Crear un nuevo predicado `get_model_size(S)` implementado en una función C++ `get_model_size(long* S)`, que acceda a la aplicación genérica ILOG CP para obtener el tamaño de `model_size`.
4. Modificar el contenido del elemento `store(Variables,Restricciones)` por `store(Variables,Restricciones,ModelSize)`.
5. Modificar el predicado `find_store` para que ahora acceda al nuevo contenido de `store`.

Con estos cambios, cada predicado de $solve^{FD}$ que gestiona una nueva restricción $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ debe realizar las siguientes acciones:

1. Utilizar `find_store(Variables,Restricciones,ModelSize)` para tener acceso al valor `ModelSize`. Este valor representa el tamaño que `solver` debería tener en este punto de cómputo.
2. Llamar a la función `get_model_size(S)` para obtener el tamaño que `model` tiene actualmente.
3. Compararlos.
   - Si `ModelSize <S` entonces es que se ha producido backtracking. Más adelante estudiamos en detalle los pasos que se realizan cuando se detecta backtracking para recuperar la consistencia entre las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$". De momento suponemos que se recupera la consistencia.
   - Si `ModelSize == S` entonces detectamos que no se ha producido backtracking.
4. Realizar la gestión de la nueva restricción $\mathcal{FD}$.
5. Antes de almacenar el nuevo estado de `NuevaVariables` y `NuevaRestricciones`, realizar una nueva llamada a la función `get_model_size(NuevoModelSize)`, para obtener el nuevo tamaño de `model` tras realizar la gestión de la nueva restricción $\mathcal{FD}$.
6. Almacenar el nuevo elemento `store(NuevaVariables,NuevaRestricciones,NuevoModelSize)` para que futuros predicados Prolog que interaccionen con la aplicación genérica ILOG CP tengan acceso al estado del sistema $\mathcal{TOY}(\mathcal{FD}i)$ tras la gestión de esta restricción $\mathcal{FD}$.

A continuación mostramos el predicado Prolog `$#>` de $solve^{FD}$ para la gestión del backtracking.

```
$#>(L, R, Out, Cin, [store(Vars7,Size4,Constraints3,NewModelSize)|Cout5]):-
       hnf(L, HL, Cin, Cout1),
       hnf(R, HR, Cout1, Cout2),
(1)    find_store_into_Cin(Cout2,Vars,Size,Constraints,ModelSize,
                                                        Cout3),
(2)    backtracking(Size,Vars,ModelSize,Vars1,1),
       toSolverFD(HL,Cout3,Cout4,Vars1,Size,Constraints,Vars2,Size1,
            Constraints1),
       toSolverFD(HR,Cout4,Cout5,Vars2,Size1,Constraints1,Vars3,Size2,
            Constraints2),
       sinchronize_solver(Vars3,Vars4,1),
       manage_constraint_argument(HL,Vars4,Size2,Vars5,Size3,IHL,VHL),
       manage_constraint_argument(HR,Vars5,Size3,Vars6,Size4,IHR,VHR),
       post_greater(IHL,VHL,IHR,VHL,Vars6,Vars7,1),
       append(Constraints2,[HL>HR],Constraints3),
(3)    get_model_size(NewModelSize).
```

Estudiemos el comportamiento del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$ sobre el ejemplo `f1 X, X #> Y, f1 Y` que venimos describiendo. Vemos que se amplía el contenido de `ilog_cp.cpp` e `ilog_cp.pl` definiendo un nuevo predicado Prolog `backtracking` implementado en la función C++ `backtracking`. De momento obviamos el comportamiento de la función C++ `backtracking`, que será explicada posteriormente en detalle. Por ahora, basta saber que es la función que compara `ModelSize` con `model_size` y restaura la consistencia entre las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$" en caso de que se haya producido backtracking.

Al inicio del objetivo se evalúa la primera rama de `f1 X`. El predicado Prolog `$#>(X,5,O,Cin,Cout)` realiza la gestión de la restricción $\mathcal{FD}$ `X #> 5`. En (1) el predicado `find_store` no encuentra el elemento `store` en `Cin`, ya que al inicio del objetivo `Cin = []`, por lo que devuelve `ModelSize = 0`. En (2) la función `backtracking` contrasta `ModelSize = 0` con la variable `model_size = 0`. Determina que no hay backtracking y continúa con la evaluación de la restricción $\mathcal{FD}$ `X #> 5`. La gestión de esta restricción impone dos `IloConstraint`s sobre `model`, como vimos en la Figura 3.11. El nuevo tamaño de `model_size = 2`. En (3) la función `get_model_size` obtiene ese nuevo tamaño y lo almacena en `store`.

El predicado `$#>(X,Y,O,Cin,Cout)` gestiona la siguiente restricción `X #> Y` del objetivo. En (1) el predicado `find_store` recoge `ModelSize = 2`. Se gestiona la restricción y se obtiene de la aplicación genérica ILOG CP el nuevo `ModelSize = 4` (3), que se almacena en el elemento `store` (4).

Este `ModelSize = 4` es el tamaño que está almacenado en `store` antes de la evaluación de la expresión `f1 Y`. Por ello, si en algún momento se produce backtracking sobre este punto de decisión, Prolog restaurará automáticamente este `ModelSize = 4`.

La evaluación del objetivo continúa evaluando la primera rama de `f1 Y`. Se gestiona la restricción `Y #> 5`. El predicado Prolog `$#>(Y,5,O,Cin,Cout)` recibe en `find_store` que `ModelSize = 4` (1), como se almacenó en `store` al final del predicado `$#<(Y,5,O,Cin,Cout)`. Se gestiona la restricción $\mathcal{FD}$ `Y #> 5`. Se obtiene el nuevo `model_size = 5` (3) y se almacena en `store` (4). Con esto se finaliza la evaluación del objetivo, encuentrándose una

primera solución que se muestra al usuario.

La búsqueda de nuevas soluciones al objetivo produce backtracking sobre el punto de decisión que se impone con `f1 Y`. En este caso se procede a evaluar la segunda rama, gestionándose la restricción `Y #< 5`. El predicado Prolog `$#<(Y,5,O,Cin,Cout)` recibe en `find_store` que `ModelSize = 4` (1), como se almacenó en `store` al final de `$#>(X,Y,O,Cin,Cout)`. En (2) comprueba que `model_size = 5`, por lo que detecta backtracking y efectúa la restauración de la consistencia entre las conjunciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$". Tras esto gestiona la restricción `Y #< 5`. Encuentra la segunda solución al objetivo y la muestra al usuario.

La búsqueda de nuevas soluciones produce backtracking sobre `f1 Y`. Como todas sus ramas han sido ya exploradas se produce backtracking sobre `f1 X`. Se restablece el estado de `Cin = []` al inicio del objetivo. El predicado `$#<(X,5,O,Cin,Cout)` ejecuta `find_store` que devuelve `ModelSize = 0` (1). En (2) compara `ModelSize = 0` con `model_size = 5` y detecta el backtracking. Restaura de nuevo las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" al estado que tenían al inicio del objetivo y continúa con la evaluación de `Y #< 5`.

De este modo podemos comprobar que con los cambios explicados la implementación del sistema $\mathcal{TOY}(\mathcal{FD}i)$ permite gestionar correctamente el backtracking.

Explicamos ahora en detalle la función C++

```
\texttt{backtracking(long VarsSize, SP\_term\_ref Vars, long ModelSize,
                                  SP\_term\_ref NewVars, long* Feasible)}.
```

Esta función compara `ModelSize` con `model_size` para en caso de que sean iguales realizar el backtracking. En primer lugar realiza los siguientes pasos para recuperar la consistencia entre las conjunciones de restricciones $\mathcal{FD}$ y $\mathcal{FD}$':

- Ajustamos `vars` al tamaño actual de `Variables`, eliminando tantas `IloIntVar`s del final de `vars` como sean necesarias. Para eliminar las últimas `k1 IloIntVar`s del vector `vars` se utiliza el método `vars.remove`, que incluye un argumento que indica la posición desde que se empieza a eliminar, y un segundo argumento que indica el número de `IloIntVar`s a eliminar: `vars.remove(vars.getSize()-k1, k1)`.

- Ajustamos el tamaño de `model`, eliminando sus últimas `IloConstraint`s almacenadas. Para eliminar las últimas `k2 IloConstraint`s de `model` se utiliza un objeto de la clase `IloModel::Iterator`, que recorre en orden los elementos de `model`. Conociendo el tamaño actual de `model` y el tamaño que debido al backtracking debería tener, iteramos por todas las `IloConstraint`s de `model` y utilizamos el método `model.remove(IloConstraint)` sobre las que se deben eliminar.

Una vez recuperada la consistencia entre la conjunción de restricciones $\mathcal{FD}$' y la conjunción de restricciones $\mathcal{FD}$, lo que nos queda es una inconsistencia entre ambas y el contenido de la conjunción de restricciones $\mathcal{FD}$", como puede verse en la Figura 3.15, donde la `IloConstraint c4` no pertenece ya a `model`, pero la `IlcConstraint c4'` sí pertenece a `solver`. La nueva conjunción de restricciones $\mathcal{FD}$" que deseamos obtener es la que vimos en la Figura 3.12. Con respecto a su estado actual, la conjunciones de restricciones $\mathcal{FD}$" deseada presenta tan sólo cambios destructivos. Bajo este contexto la única posibilidad de recuperar la consistencia entre el nuevo estado de `model` y `solver`, es decir, entre la conjunción de restricciones $\mathcal{FD}$' y $\mathcal{FD}$", es efectuar el método `solver.extract(model)`

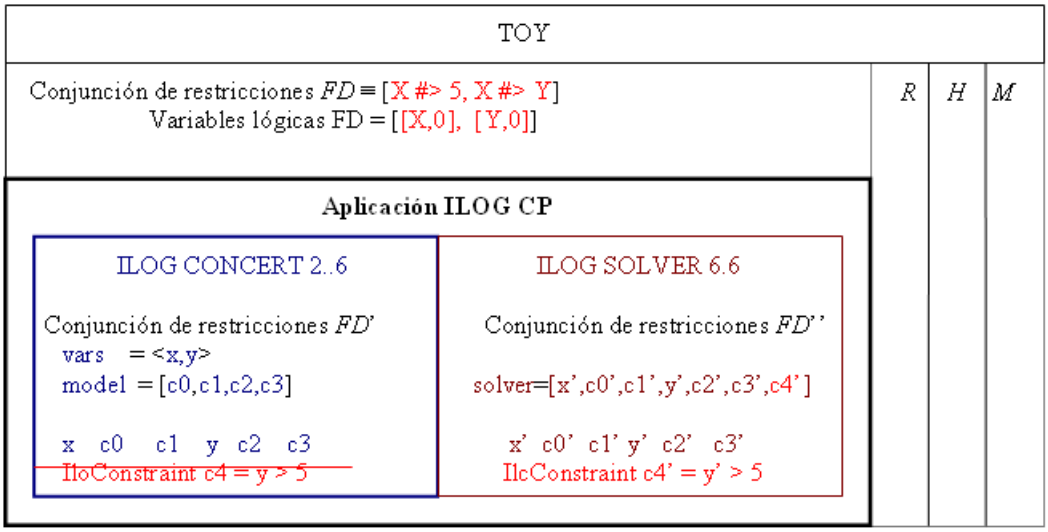

Figura 3.15: Recuperación de la consistencia entre $\mathcal{FD}$ y $\mathcal{FD}$'.

Esta forma de traducción logra nuestro propósito aunque de una forma ineficiente. Recorre en orden todas las `IloConstraints` contenidas en `model`. Para cada una de ellas comprueba si `solver` contiene su `IlcConstraint` asociada, asi como las `IlcIntVars` asociadas a las `IloIntVars` involucradas en dicha `IloConstraint`.

i) Si `solver` ya contiene los objetos asociados a la `IloConstraint` traducida, entonces no se realiza ninguna acción.

ii) Si `solver` no contiene los objetos asociados, entonces los crea.

iii) Si tras finalizar el recorrido sobre `model`, el resolutor `solver` contiene objetos `IlcIntVar` o `IlcConstraint` que no han sido referenciados desde `model` entonces `solver` los elimina y deja de trabajar sobre ellos.

Para toda `IloConstraint` de `model`, el estado actual de `solver` se encuentra en el punto i). Sólo tras recorrer todas las `IloConstraints` de `model` el resolutor `solver` realiza el cambio que queríamos hacer sobre la conjunción de restricciones $\mathcal{FD}$": eliminar aquellas `IlcIntVar` e `IlcConstraint` sobre las que ya no se debe trabajar debido al backtracking. La Figura 3.16 muestra el estado de $\mathcal{TOY}(\mathcal{FD}i)$ tras recuperar la consistencia entre las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$".

Como la función `backtracking` modifica la conjunción de restricciones $\mathcal{FD}$" sobre la que trabaja `solver`, la propia función `backtracking` se debe encargar también de propagar esta nueva conjunción de restricciones $\mathcal{FD}$". Para ello utiliza el método `solver.propagate()`, que propaga una a una todas las `IlcConstraints` sobre las que trabaja `solver`. Esta propagación puede acotar algunas `IlcIntVars` que deben sincronizarse con la lista `Variables`, como vimos en la sección 3.2.
Al detectar backtracking es preciso borrar el contenido del vector `vars_sichronized`, ya que puede contener información erronea.
Veámoslo con el siguiente objetivo `domain [X] 0 1, f2 X` donde `f2 X` tiene la siguiente definición:

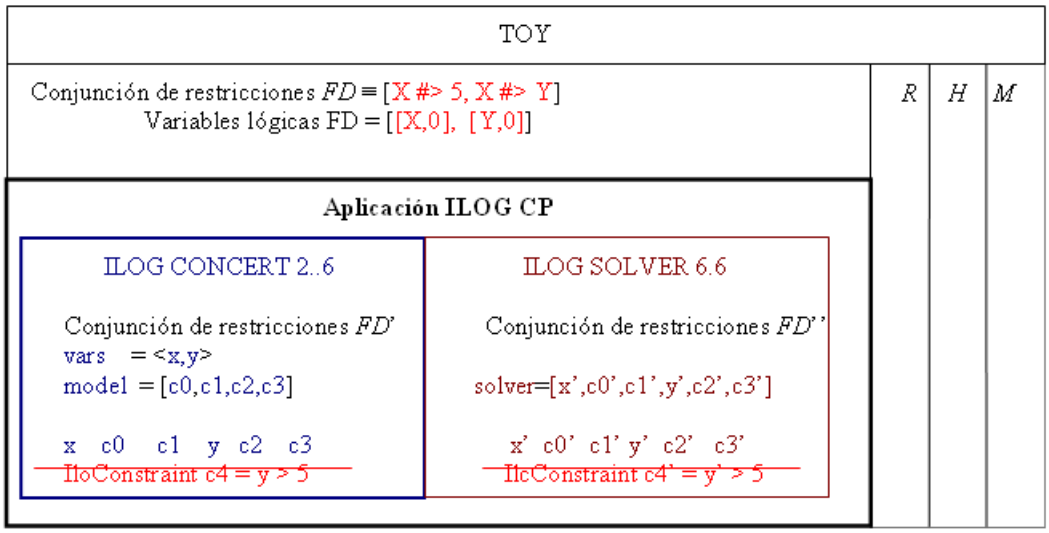

Figura 3.16: Recuperación de la consistencia entre $\mathcal{FD}$ $\mathcal{FD}$' y $\mathcal{FD}$".

```
f2 :: int -> bool
f2 A = true <== A #> 0
f2 A = true <== A #< 1
```

Al evaluar la primera rama de `f2 X`, el resolutor `solver` acota al valor 1 la `IlcIntVar v0'` asociada a `X`. Esto añade el par al vector `vars_sinchronized`. La búsqueda de nuevas soluciones produce backtracking sobre `f2 X`. Esta segunda rama acota al valor 0 la `IlcIntVar v0'` asociada a `X`. Sin embargo, si al detectar el backtracking no eliminamos el contenido del vector `vars_sinchonized` entonces esta segunda acotación de `v0'` no será sincronizada con la lista `Variables`, ya que `v0'` ya se encuentra contenida en `vars_sinchonized` por su acotación durante la exploración de la primera rama de `f2 X`.

La decisión de borrar el contenido del vector `vars_sinchronized` hace que pares que no han sido deshechos por el backtracking se borren igualmente, por lo que es posible que alguna variable lógica $\mathcal{FD}$ se sincronice de nuevo. Como vimos en la sección 3.2 esto no supone ningún problema en cuanto a la corrección de las soluciones encontradas, tan sólo una pérdida de tiempo innecesaria. Una alternativa a este problema sería almacenar en `store` el tamaño del vector `vars_sinchronized`. Sin embargo, respetamos la decisión de borrar completamente el vector con cada backtracking para no recargar excesivamente el contenido de `store`.

En este momento las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$" son de nuevo consistentes entre ellas y consistentes con el estado que contenía el sistema $\mathcal{TOY}(\mathcal{FD}i)$ antes del punto de decisión sobre el que se ha efectuado backtracking. Si se continúa ahora con la evaluación del objetivo se puede asegurar que las soluciones que se encuentren serán correctas.

3.3.3. `model` como paso intermedio entre $\mathcal{TOY}$ y `solver`

En la subsección 2.2.2 quedó pendiente una mayor explicación de los motivos por los que se utilizaba `model` como paso intermedio entre $\mathcal{TOY}$ y `solver`. Cabía preguntarse si

no sería más razonable trabajar exclusivamente con las conjunciones de restricciones $\mathcal{FD}$ y $\mathcal{FD}$", realizando una traducción directamente desde $\mathcal{TOY}$ hasta `solver`.

Además de los motivos de simpleza y estandarización explicados en la subsección 2.2.2 podemos añadir ahora un tercer motivo: la eficiencia en la resolución de objetivos que involucren backtracking.

Las dos formas que utilizamos para añadir una nueva `IlcConstraint` a `solver` utilizan objetos de la biblioteca ILOG Concert 2.6:

- El método que utilizamos habitualmente es `solver.propagate(IloConstraint)`. Utiliza el objeto `IloConstraint` de la biblioteca ILOG Concert 2.6.
- El método utilizado cuando se produce backtracking es `solver.extract(model)`. Utiliza el objeto `model` de la biblioteca ILOG Concert 2.6.

La única forma de añadir una `IlcConstraint` directamente sobre `model` es mediante el uso de `IloGoals`. Se debe crear un `IloGoal` para cada clase de restricción que se quiera añadir.

Sin embargo no es posible eliminar directamente una `IlcConstraint` de `solver`. La API facilitada por ILOG Solver 6.6 no lo permite. La única forma de realizar el backtracking trabajando directamente sobre `solver` consiste en:

- Utilizar un vector de `IlcIntVar`s paralelo a la lista `Variables`, similar al vector `vars` utilizado actualmente.
- Utilizar un vector de pares(`IlcConstraint`,int) paralelo al contenido de `solver`. Cada vez que se añade una `IlcConstraint` a `solver` se añade también al vector. Además, el valor entero indica el tipo de `IlcConstraint` que es, necesario para saber el tipo de `IloGoal` que es necesario utilizar para introducirlo en `solver`.
- Almacenar en el elemento `store` el tamaño de este vector, del mismo modo que lo hacemos actualmente con `model_size`.
- Cuando se detecta backtracking:
  - Borrar completamente el contenido de `solver` mediante el método `solver.clear()`.
  - Restaurar el tamaño del vector de `IlcIntVar`s y del vector de pares, del mismo modo que lo hacemos ahora.
  - Añadir de nuevo todas las `IlcConstraint`s a `solver`, utilizando para cada `IlcConstraint` el `IloGoal` concreto que se encarga de añadir cada tipo de `IlcConstraint` a `solver`.

Esto supone dos cosas:

- La necesidad de utilizar dos objetos de tipo vector muy similares a `vars` y `model`. Para acabar utilizando estos objetos parece más razonable utilizar `model` y `vars`, propuestos por ILOG y que incluyen un extenso API.
- El backtracking supone rehacer de nuevo la conjunción de restricciones impuestas desde el inicio del objetivo hasta el punto de elección en el que se encuentra actualmente la evaluación del objetivo $\mathcal{TOY}$.

Por todo ello, mantenemos la elección de utilizar la biblioteca ILOG Concert 2.6 para modelar la conjunción de restricciones $\mathcal{FD}$ propuesta por el objetivo $\mathcal{TOY}$, y continuamos realizando la doble traducción entre $\mathcal{TOY}$ y `model` y entre `model` y `solver`.

3.4. Gestión del etiquetado

Nuestra actual implementación del sistema $\mathcal{TOY}(\mathcal{FD}i)$ ofrece una única solución por intervalos para cada conjunción de restricciones $\mathcal{FD}$ propuesta en un objetivo $\mathcal{TOY}$ (en caso de que esta solución exista). Por ejemplo, para el objetivo `domain [X,Y] 0 2, X #> Y` tanto el sistema $\mathcal{TOY}(\mathcal{FD}s)$ como el sistema $\mathcal{TOY}(\mathcal{FD}i)$ ofrecen como la solución intensional `X #> Y, X in 1..2, Y in 0..1`.

Sin embargo, existen casos en los que nos puede interesar obtener una solución extensional, donde conozcamos el valor exacto que toman todas o algunas de las variables lógicas $\mathcal{FD}$ involucradas en la conjunción de restricciones $\mathcal{FD}$. La restriccción $\mathcal{FD}$ `labeling Strategy VarsLab` utilizada en $\mathcal{TOY}(\mathcal{FD}s)$ exigía que cada solución mostrada al usuario acotara cada variable lógica $\mathcal{FD}$ de `VarsLab` a un único valor. La restricción $\mathcal{FD}$`labeling Strategy VarsLab` es una restricción $\mathcal{FD}$ de etiquetado sobre las variables lógicas $\mathcal{FD}$ de `VarsLab`.

Aplicando una restricción $\mathcal{FD}$ de etiquetado sobre `X`, el sistema $\mathcal{TOY}(\mathcal{FD}s)$ encuentra las dos siguientes soluciones para el anterior objetivo
`domain [X,Y] 0 2, X #> Y, labeling [] [X]`:

- `sol1: X -> 1, Y -> 0.`
- `sol2: X -> 2, Y -> 1.`

Veamos otro ejemplo más. Definimos la función indeterminista `f3` como:

```
f3 :: int -> bool
f3 A = true <== domain [A] 0 1
f3 A = true <== domain [A] 5 6
```

Si evaluamos ahora el objetivo `f3 X`, tanto el sistema $\mathcal{TOY}(\mathcal{FD}s)$ como el sistema $\mathcal{TOY}(\mathcal{FD}i)$ encuentran las dos siguientes soluciones al objetivo:

- `sol1: X in 0..1`
- `sol2: X in 5..6`

Aplicando una restricción $\mathcal{FD}$ de etiquetado sobre `X`, el sistema $\mathcal{TOY}(\mathcal{FD}s)$ encuentra las cuatro siguientes soluciones para el anterior objetivo `f3 X, labeling [] [X]`:

- `sol1: X -> 0.`
- `sol2: X -> 1.`
- `sol3: X -> 5.`
- `sol4: X -> 6.`

Si nos abstraemos del sistema de restricciones externo utilizado, cuando se impone una restricción $\mathcal{FD}$ de etiquetado sobre un sistema de restricciones externo se da pie a un procedimiento de búsqueda de soluciones. Este procedimiento etiqueta cada una de las variables lógicas $\mathcal{FD}$ de `VarsLab`, generando un candidato a solución. Existen tantos candidatos como combinaciones de valores se pueden efectuar sobre las variables lógicas $\mathcal{FD}$ de `VarsLab`. El orden en que se generan los candidatos viene determinado por `Strategy`. Si un candidato no satisface la conjunción de restricciones $\mathcal{FD}$ impuesta sobre el sistema

de restricciones entonces este candidato se descarta y se genera el siguiente candidato. Si un candidato satisface la conjunción de restricciones $\mathcal{FD}$ entonces se ha encontrado una solución. Si se requiere una nueva solución entonces se busca el siguiente candidato.

Además, los procedimientos de búsqueda de clpfd e ILOG Solver 6.6 añaden propagación de restricciones durante la búsqueda. Gracias a esta propagación se puede detectar previamente algunas combinaciones de valores que no respetarán la conjunción de restricciones $\mathcal{FD}$. Con esto se evita la generación de algunos candidatos fallidos haciendo más eficaz al procedimiento de búsqueda.

En esta sección describimos la implementación de la restricción $\mathcal{FD}$ `labeling Strategy VarsLab` en el sistema $\mathcal{TOY}(\mathcal{FD}i)$, adaptandonos al marco propuesto por ILOG para los procedimientos de búsqueda. En esta primera implementación de $\mathcal{TOY}(\mathcal{FD}i)$ tan sólo definimos dos estrategias de búsqueda sobre el orden en que se etiquetan las variables de `VarsLab`. La primera es una estrategia estática que etiqueta las variables en el orden textual en que aparecen en `VarsLab`. La segunda es la estrategia dinámica `first_fail` que etiqueta en cada momento la variable que contiene menor número de valores en su dominio. Ambas estrategias etiquetan los valores del dominio de cada variable en orden ascendente. La primera estrategia se identificará como `Strategy = []`. La segunda como `Strategy = [ff]`.

3.4.1. Implementación del `labeling`

Describimos por separado la implementación del predicado de $solve^{FD}$ `labeling Strategy VarsLab` y el marco de trabajo que ofrece la biblioteca ILOG Solver 6.6 para implementar procedimientos de búsqueda.

Implementación del predicado `labeling`

La restricción `labeling Strategy VarsLab` es impuesta directamente sobre `solver`, que realiza el etiquetado de las `IlcIntVar`s asociadas a `VarsLab`. Las soluciones encontradas a este etiquetado respetarán la conjunción de `IlcConstraint`s sobre las que trabaja `solver`. Por ello es importante que el etiquetado se produzca tras haber añadido todas las restricciones $\mathcal{FD}$ de la conjunción de restricciones. Colocaremos por tanto la instrucción `labeling` al final de los objetivos $\mathcal{TOY}$, como última expresión $\mathcal{FD}$ a evaluar.

Para poder mostrar las diferentes soluciones de etiquetado, el predicado Prolog de $solve^{FD}$ `labeling Strategy VarsLab` debe comportarse como una función indeterminista. Mientras puedan existir más soluciones al etiquetado, el predicado `labeling` debe ofrecer una nueva rama de cómputo. Cuando se determine que no existen más soluciones al etiquetado, el predicado `labeling` debe fallar y no ofrecer nuevas ramas de cómputo.

Para implementar este comportamiento creamos el predicado Prolog `find_solution(Strat,Indexes,Vars,NewVars,F)` implementado en la función C++ `find_solution(long Strat, SP_term_ref Indexes, SP_term_ref Vars, SP_term_ref NewVars, long* Found)`.

Esta función C++ genera la búsqueda de un candidato solución. Para ello etiqueta valores sobre las `IlcIntVar`s asociadas a las variables lógicas $\mathcal{FD}$ de `VarsLab` especificadas la restricción $\mathcal{FD}$ `labeling Strategy VarsLab`. Estos valores se etiquetan siguiendo la estrategia `Strategy`. Además, si encuentra una solución, sincroniza las variables lógicas de `Variables` cuyas `IlcIntVar`s asociadas hayan resultado acotadas. La función devuelve además en `Found` un entero indicando si ha encontrado una nueva solución o no.

Con esto, el siguiente fragmento de código dota al predicado `labeling` del comportamiento indeterminista antes descrito:

```
repeat,
(find_solution(ST,Indexes,Vars,NewVars,1) ->
    (true)
;
    (!,fail)
)
```

El predicado Prolog `repeat` consiste en un bucle que genera una lista infinita de elecciones para el backtracking. El contenido del bucle es sencillo.

Si la función C++ `find_solution` encuentra una nueva solución al etiquetado, entonces se ejecuta el predicado Prolog `true`, que siempre tiene éxito. Se almacena el nuevo contenido de `store` y finaliza la gestión de la restricción $\mathcal{FD}$ `labeling`. Como esta restricción se encuentra al final del objetivo se ejecuta el predicado Prolog `showAnswer`, que muestra la solución obtenida al usuario.

Si se requieren nuevas soluciones se produce backtracking sobre el bucle `repeat` de `labeling`, y se recupera el contenido que tenía `store` al inicio del bucle `repeat`. En este caso nos beneficiamos de que el backtracking no se produzca de manera automática en la aplicación ILOG CP. Como el estado de esta aplicación no se restaura, el propio procedimiento de búsqueda de ILOG reanudará la búsqueda a partir del candidato encontrado como solución en la última ocasión.

Si la función C++ `find_solution` no encuentra una nueva solución al etiquetado, entonces podemos asegurar que se han explorado ya todos los candidatos del etiquetado y que no quedan más soluciones. Entonces se ejecuta `(!,fail)`. La instrucción de corte (!) se produce para salir del bucle `repeat`, es decir, para que no se exploren por backtracking nuevas ramas de cómputo en el predicado Prolog `labeling`. La instrucción `fail` se produce para hacer fallar a esta rama de cómputo. Al no haberse encontrado una nueva solución, esta rama de cómputo es fallida, por lo que no se debe finalizar con éxito la gestión de `labeling`. El fallo en `labeling` producirá backtracking sobre el anterior punto de decisión colocado en el objetivo $\mathcal{TOY}$, en caso de que exista alguno.

Los argumentos de entrada de `find_solution` se obtienen de la siguiente manera. Para `long Strat`, se chequea si `Strategy` es `[]` o `[ff]`, generando `Strat` = 0 o 1, respectivamente. Para `SP_term_ref VarsToLab` se crea un nuevo predicado `labeling_manage_vars(VarsLab,Vars,Size,Indexes)` que genera la lista de enteros `Indexes`. Esta lista contiene los índices de las `IloIntVar`s asociadas a `VarsLab`. Será pasada como argumento `SP_term_ref` a `find_solution`.

```
%labeling_manage_vars/4(+VarsLab,+Vars,+Size,-Indexes).
%
labeling_manage_vars([],_,_,[]).
%
labeling_manage_vars([V|RV],_,_,RP) :-
    integer(V),
    !,
    labeling_manage_vars(RV,RP).
%
labeling_manage_vars([V|RV],Vars,Size,RP) :-
```

```
    var(V),
    is_var_in_vars_list(V,Vars,0,Index),
    Index == Size,
    !,
    labeling_manage_vars(RV,RP).
%
labeling_manage_vars([V|RV],Vars,Size,[Index|RP]) :-
    var(V),
    is_var_in_vars_list(V,Vars,0,Index),
    Index < Size,
    !,
    labeling_manage_vars(RV,RP).
```

Si una variable lógica $\mathcal{FD}$ ha sido acotada a un valor entero previamente a la gestión de la restricción `labeling` entonces esta variable no debe ser implicada en el etiquetado. Por ejemplo, en el objetivo
`domain [X,Y] 0 1, X #> 0, labeling [] [X,Y]` no tiene sentido hacer un etiquetado sobre la variable `X`, ya que su dominio se reduce únicamente al valor `1`. La segunda cláusula de `labeling_manage_vars` gestiona esta situación.

No tiene sentido hacer referencia a nuevas variables lógicas $\mathcal{FD}$ en `labeling`. Cada variable lógica $\mathcal{FD}$ debe estar impuesta sobre alguna restricción $\mathcal{FD}$ de la conjunción de restricciones $\mathcal{FD}$. La tercera cláusula de `labeling_manage_vars` gestiona esta situación.

La cuarta cláusula añade a `Indexes` el índice de cada variable lógica de `Variables` sobre las que la restricción `labeling` ha solicitado efectuar el etiquetado.

Por lo tanto, el código definitivo del predicado Prolog de $solve^{FD}$ `labeling Strategy VarsLab` queda como sigue:

```
'$labeling'(Strategy, VarsLab, true, Cin, Cout) :-
    nf(Strategy, NFStrategy, Cin, Cout1),
    toyListToPrologList(NFStrategy, HNFStrategy),
    (HNFStrategy == [ff] ->
        (Strat = 1)
    ;
        ( (HNFStrategy == [] ->
            (Strat = 0)
          ;
            (fail)
          )
        )
    ),
    nf(VarsLab, NFVarsLab, Cout1, Cout2),
    toyListToPrologList(NFVarsLab, HNFVarsLab),

    find_store_into_Cin(Cout2,Vars,Size,Constraints,ModelSize,Cout3),
    listToSolverFD(HNFVarsLab,Cout3,Cout4,Vars,Size,Constraints,ModelSize,
            Vars0,Size0,Constraints0,ModelSize1),
    sinchronize_solver(Vars0,Vars1,1),
    labeling_manage_vars(HNFVarsLab,Vars1,Size0,Indexes),
    repeat,
```

```
    (find_solution(Strat,Indexes,Vars1,Vars2,1) ->
      (true)
    ;
      (!,fail)
    ),
  append(Cout4,[store(Vars2,Size0,Constraints0,ModelSize1)],Cout).
```

Implementación de la función C++ `find_solution`

En la sección 1.2.1 describimos en detalle los procedimientos de búsqueda soportados por la biblioteca ILOG Solver 6.6.

El contexto utilizado por `labeling` exige que cada vez que llamemos a la función `find_solution(Strategy,Indexes,Vars,NewVars,Founded)` se busque la siguiente solución al etiquetado. Además se debe devolver como argumento si se ha encontrado dicha solución o no. Esto implica la siguiente forma de proceder:

- Creamos una nueva variable estática `static long flag_labeling(0)`. Ésta indica si se está gestionando una restricción $\mathcal{FD}$ `labeling` o no. Además creamos el objeto `static IloIntVarArray vars_labeling(env)` que almacena las `IloIntVar`s que se van a etiquetar en el procedimiento de búsqueda.
- La variable `flag_labeling` permanece a 0 a lo largo de la evaluación de todo el objetivo. Cuando se gestiona una restricción `labeling` se detecta por `flag_labeling` si es la primera llamada a `find_solution` que se produce. Si `flag_labeling == 0`, entonces:
  - Se crea el procedimiento de búsqueda `g`.
    - Se accede a la lista `Indexes` y para cada índice `i` se ejecuta el método `vars_labeling.add(vars[i])`
    - Se accede al entero `Strategy` para seleccionar entre la estrategia `IloChooseFirstUnboundInt` o `IloChooseMinSizeInt`.
  - Se vincula `g` a `solver` mediante el método `solver.startNewSearch(g)`.
  - Se modifica `flag_labeling = 1`, para que la próxima llamada a `find_solution` no realice estas tareas.
- Se ejecuta la instrucción `feasible = solver.next()`. Se devuelve `feasible` como indicador de si se ha encontrado o no la solución.
  - Si `feasible = 0` entonces es que la búsqueda ha finalizado sin encontrar una nueva solución. Esto producirá que el predicado Prolog `labeling` falle y se produzca backtracking sobre el último punto de decisión del objetivo $\mathcal{TOY}$. Por lo tanto debemos:
    - Eliminar las estructuras relacionadas con la búsqueda mediante el método `solver.endSearch()`
    - Eliminar el contenido de `vars_labeling` mediante el método `vars_labeling.remove(0,vars_labeling.getSize())`
    - Restaurar `flag_labeling = 0`, para indicar que ya no se está gestionando una restricción `labeling`.

    Imaginemos que hemos definido la siguiente función indeterminista `f4`:

```
f4:: [int] -> bool
f4 A = true <== A == [X], domain A 0 1
f4 A = true <== A == [X,Y], domain A 0 1
```

Si ahora evaluamos el siguiente objetivo `f4 T, labeling [] T` se encontrarán las siguientes seis soluciones: `sol1. X -> 0`, `sol2. X -> 1`, `sol3. X -> 0, Y -> 0`, `sol4. X -> 0, Y -> 1`, `sol5. X -> 1, Y -> 0` y `sol6. X -> 1, Y -> 1`.

La primera rama de cómputo de `f4 T` gestiona la restricción $\mathcal{FD}$ `labeling [] [X]`, que obtiene las dos primeras soluciones. Tras descubrir que no hay una tercera solución finalizamos la búsqueda, borramos el contenido de `vars_labeling` y restauramos `flag_labeling = 0`. Si no hicieramos esto no podríamos gestionar más adelante la restricción $\mathcal{FD}$ `labeling [] [X,Y]`, que permitirá encontrar las siguientes cuatro soluciones al objetivo $\mathcal{TOY}$.

- Si `feasible = 1` entonces se ha encontrado una nueva solución al etiquetado. Sincronizamos las variables lógicas de `Variables` cuyas `IlcIntVar`s asociadas hayan sido acotadas por el procedimiento de búsqueda.

Con respecto a este último punto es preciso hacer una matización. Ilustrémoslo mediante el objetivo
`domain [X] 5 12, domain [Y] 2 17, X #+ Y == 17, X #- Y == 5, labeling [] [X,Y]`.
El procedimiento de búsqueda que utiliza `solver` puede verse en la Figura 3.17.

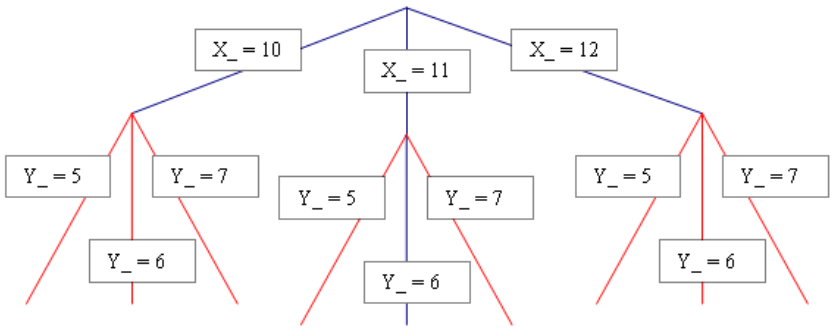

Figura 3.17: Procedimiento de búsqueda.

Mediante `solver.next()` se encuentra una primera solución `X -> 11, Y -> 6`. Sin embargo, para encontrar esta solución, `solver` etiqueta en primer lugar `X = 10`. Esto acota al valor `10` la `IlcIntVar` asociada a `X` y a la `IloIntVar` contenida en `vars[0]`, por lo que se introduce el par `(0,10)` en el vector `vars_to_sichronize` y en el vector `vars_sichronized`. La propagación de restricciones durante la búsqueda determina que esa rama no contiene ningún candidato que deba ser explorado. Por lo tanto se explora otra rama y `solver` etiqueta `X = 11`. Esto acota al valor 11 la `IlcIntVar` asociada a `X` y `vars[0]`, por lo que se introduce el par `(0,11)` en los vectores. Tras la propagación de restricciones se etiqueta `Y = 6`, por lo que se añade el par `(1,6)` en los vectores. La propagación de restricciones determina a `X = 11, Y = 6` como una solución al etiquetado. Sin embargo, el contenido del vector `vars_to_sinchronize = <(0,10),(0,11),(1,6)>`. Si

sincronizamos las variables lógicas de `Variables` con el contenido del vector `vars_to_sinchronize` se unificará a `X` con el valor `10` y posteriormente con el valor `11`, lo que provocará fallo.

Para arreglar esta situación debemos efectuar dos cambios:

- Un cambio en la definición de la clase `IlcCheckWhenBound`. Ahora el ḿetodo `post()` solo propagará si `flag_labeling == 0`. Esto impedirá que durante el procedimiento de búsqueda (donde flag_labeling = 1) se añadan pares a los vectores.
- Al no añadir los pares a los vectores evitamos fallos en la sincronización de `Variables`, pero esto nos obliga a sincronizar `Variables` con el método ineficiente que explicamos en la subsección 2.2.2. Esto supone preguntar una a una a las `IlcConstraint`s sobre las que trabaja `solver` si están acotadas, y en caso de estarlo unificar su variable lógica $\mathcal{FD}$ asociada al valor al que están acotadas.

Capítulo 4

Ejemplos

En este capítulo se evalúa el rendimiento del sistema $\mathcal{TOY}(\mathcal{FD}i)$ sobre tres ejemplos. Se comparan los tiempos de resolución (expresados en milisegundos) de los sistemas $\mathcal{TOY}(\mathcal{FD}i)$, $\mathcal{TOY}(\mathcal{FD}s)$ e ILOG CP 1.4. Por cada uno de estos ejemplos utilizamos un programa $\mathcal{TOY}$ `ej.toy`, que se ejecuta en los sistemas $\mathcal{TOY}(\mathcal{FD}i)$ y $\mathcal{TOY}(\mathcal{FD}s)$, y una aplicación ILOG CP concreta `ej.cpp`, que modela y resuelve una conjunción de restricciones equivalente a la propuesta en el programa `ej.toy`. Para cada ejemplo y sistema se ha realizado una única ejecución.

Los programas se han ejecutado en una máquina con un procesador Intel Dual Core 2.4Ghz con 4GB RAM de memoria. El sistema operativo ha sido Windows XP SP3. Se han utilizado los sistemas SICStus Prolog 3.12.8, ILOG CP 1.4 y las herramientas de Microsoft Visual Studio 2005.

4.1. Sistemas de ecuaciones

Se usan dos programas paramétricos de prueba que modelan sistemas de `n` ecuaciones linealmente independientes de la forma $A*X = b$. Sus `n` variables enteras `[X1, ..., Xn]`, de dominio {`1..n`}, tienen una única solución. La matriz `n` × `n` A está fijada y el vector b depende de la solución que se haya predeterminado. La matriz A toma el valor i en los coeficientes de su diagonal $A_{i,i}$ y el valor 1 para el resto de sus coeficientes.

Veamos un ejemplo con 5 variables, donde cada variable tiene dominio 1...5. Para este caso nuestro sistema de ecuaciones es:

```
(1 1 1 1 1)   (X1)   (b1)
(1 2 1 1 1)   (X2)   (b2)
(1 1 3 1 1) * (X3) = (b3)
(1 1 1 4 1)   (X4)   (b4)
(1 1 1 1 5)   (X5)   (b5)
```

Los valores de `bi` los escogemos arbitrariamente de modo que satisfagan una serie de soluciones. Escogemos los valores que queremos que tome nuestra solución, es decir, escogemos unos valores para `X1,X2,X3,X4,X5`. A continuación resolvemos el sistema $A*X$ para conocer los valores de `b1,b2,b3,b4,b5`. Supongamos que escogemos `X1 = 1, X2 = 1, X3 = 1, X4 = 1, X5 = 1`. Entonces resolvemos:

```
(1 1 1 1 1)   (1)   (b1)
(1 2 1 1 1)   (1)   (b2)
```

```
(1 1 3 1 1) * (1) = (b3)
(1 1 1 4 1)   (1)   (b4)
(1 1 1 1 5)   (1)   (b5)
```

Obtenemos que `b1 = 5, b2 = 6, b3 = 7, b4 = 8, b5 = 9`.

El programa $\mathcal{TOY}$ `ej.toy` plantea el sistema de ecuaciones al sistema de restricciones externo para que, conociendo `A` y `b`, busque por etiquetado una solución al sistema. En nuestro caso plantea:

```
(1 1 1 1 1)   (X1)   (5)
(1 2 1 1 1)   (X2)   (6)
(1 1 3 1 1) * (X3) = (7)
(1 1 1 4 1)   (X4)   (8)
(1 1 1 1 5)   (X5)   (9)
```

El sistema de restricciones obtendrá como única solución
`X1 = 1, X2 = 1, X3 = 1, X4 = 1, X5 = 1`.

Para nuestro análisis comparamos los tiempos de resolución para instancias del problema con 5, 12 y 15 variables, respectivamente. En cada caso distinguimos entre el uso del procedimiento de búsqueda estático que selecciona las variables en el orden textual en que aparecen en el programa y el procedimiento de búsqueda dinámico 'First Fail'(denotado por ^{ff}). Ambos etiquetan los valores del dominio de cada variable en sentido ascendente.

Además, mostramos la ganancia de velocidad de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ e ILOG CP, respectivamente. Específicamente, denotamos:

- (a) a la ganancia de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ utilizando el procedimiento de búsqueda estático.
- (b) a la ganancia de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ utilizando el procedimiento de búsqueda dinámico.
- (c) a la ganancia de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a ILOG CP utilizando el procedimiento de búsqueda estático.
- (d) a la ganancia de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a ILOG CP utilizando el procedimiento de búsqueda dinámico.

Primer sistema

La solución `[X1,...,Xn]` cumple: $\forall$`i` $\in \{1 \ldots n\}$ `Xi` $= i$. En la Tabla 4.1 se muestran los tiempos de resolución precisados por $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}i)$ e ILOG CP para encontrar la solución al sistema.

`n`	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
5	0	0	32	0	15	15	-	-	2.13	0
12	47	1,782	235	469	15	281	5	0.26	15.67	1.67
15	297	307,782	563	32,549	63	20,578	1.90	0.11	8.94	1.58

Tabla 4.1: Resultados para el primer sistema de ecuaciones

Para la estrategia de búsqueda estática $\mathcal{TOY}(\mathcal{FD}i)$ tiene un menor rendimiento que $\mathcal{TOY}(\mathcal{FD}s)$, mientras que para la estrategia de búsqueda dinámica $\mathcal{TOY}(\mathcal{FD}i)$ tiene un mayor rendimiento que $\mathcal{TOY}(\mathcal{FD}s)$. Esta tendencia aumenta cuanto mayor es el número de variables del problema. Observando los dominios de las variables `[X1,...,Xn]` tras la propagación inicial de las restricciones, concluimos que la estructura de la solución $\forall i \in \{1 \ldots n\}$ `Xi` $= i$ favorece notablemente a la estrategia de búsqueda estática y perjudica gravemente a la estrategia de búsqueda 'First Fail'. Esto indica que, en problemas donde se requiere escasa exploración para encontrar una solución, la penalización que sufre $\mathcal{TOY}(\mathcal{FD}i)$ por mantener la consistencia entre las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$" hace que sea menos eficiente que $\mathcal{TOY}(\mathcal{FD}s)$. Sin embargo, a medida que la exploración necesaria para encontrar la solución es mayor, esta penalización se compensa, haciendo que $\mathcal{TOY}(\mathcal{FD}i)$ sea más eficiente que $\mathcal{TOY}(\mathcal{FD}s)$.

Segundo sistema

La solución `[X1, ..., Xn]` cumple: $\forall$`i` $\in \{1 \ldots n\}$ `Xi` $= n-(i-1)$. En la Tabla 4.2 se muestran los tiempos de resolución precisados por $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}i)$ e ILOG CP para encontrar la solución al sistema.

`n`	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
5	15	15	15	0	31	15	1	0	0.48	0
12	484	235	469	249	109	63	0.97	1.06	4.30	3.95
15	16,000	16,047	10,546	2,609	843	1,765	0.66	0.16	12.51	1.48

Tabla 4.2: Resultados para el segundo sistema de ecuaciones

La tendencia apuntada en el ejemplo anterior se confirma con este segundo ejemplo, donde $\mathcal{TOY}(\mathcal{FD}i)$ tiene un mayor rendimiento que $\mathcal{TOY}(\mathcal{FD}s)$ para ambas estrategias. La estructura de la solución $\forall$`i` $\in \{1 \ldots n\}$ `Xi` $= n-(i-1)$ perjudica gravemente a la estrategia de búsqueda estática, mientras que ni perjudica ni beneficia excesivamente a la estrategia 'First Fail'. Para la estrategia estática $\mathcal{TOY}(\mathcal{FD}i)$ obtiene un ligero mejor tiempo de resolución. Para la estrategia dinámica, mucho más representativa al no tratar un caso extremo, los tiempos de resolución de $\mathcal{TOY}(\mathcal{FD}i)$ son mejores un orden de magnitud frente a los de $\mathcal{TOY}(\mathcal{FD}s)$.

4.2. Problema de las n reinas

Planteamos ahora el clásico problema de las `n` reinas, que intenta colocar `n` reinas en un tablero de ajedrez cuyas dimensiones son `n` $\times$ `n` de manera que las reinas no se ataquen entre ellas. Para $\mathcal{TOY}(\mathcal{FD}i)$ y $\mathcal{TOY}(\mathcal{FD}s)$ utilizamos el ejemplo `queens.toy`, contenido en la distribución del sistema $\mathcal{TOY}$. Para el caso de la aplicación ILOG CP, modificamos la aplicación `queens.cpp` contenida en la distribución de ILOG CP 1.4, para adaptarla a nuestras estrategias de búsqueda. Utilizamos instancias del problema con `n = 5, 12 y 15`. En cada uno de los casos, comparamos el tiempo precisado para encontrar la primera, segunda y tercera solución, respectivamente. En las Tablas 4.3, 4.4 y 4.5 se muestran los

tiempos de resolución precisados por $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}i)$ e ILOG CP para encontrar la primera, segunda y tercera solución al problema de las reinas, respectivamente.

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
5	16	16	46	16	15	15	2.87	1	3.06	1.06
12	47	62	515	438	31	31	10.96	7.06	16.61	14.13
15	109	79	1359	1312	31	46	12.46	16.60	43.84	28.52

Tabla 4.3: Primera solución

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
5	0	15	0	0	15	15	-	0	0	0
12	0	16	16	0	31	31	-	0	0.52	0
15	93	31	78	47	62	46	0.83	1.52	1.26	1.02

Tabla 4.4: Segunda solución

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
5	0	0	0	15	31	15	-	-	0	1
12	16	15	16	0	31	46	1	0	0.52	0
15	47	16	47	47	93	62	1	2.94	0.51	0.76

Tabla 4.5: Tercera solución

Como podemos comprobar, el sistema $\mathcal{TOY}(\mathcal{FD}i)$ no es competitivo con el sistema $\mathcal{TOY}(\mathcal{FD}s)$ para este ejemplo. Para una instancia del problema con diez reinas, el tiempo de resolución de $\mathcal{TOY}(\mathcal{FD}i)$ es un orden de magnitud peor que el de $\mathcal{TOY}(\mathcal{FD}s)$. Sin embargo, los tiempos de ILOG CP sí que son competitivos con respecto a los de $\mathcal{TOY}(\mathcal{FD}s)$. Aunque el sistema $\mathcal{TOY}(\mathcal{FD}i)$ no podrá nunca alcanzar los tiempos de ILOG CP, concluimos que existe un amplio margen de mejora en el rendimiento de $\mathcal{TOY}(\mathcal{FD}i)$. En el siguiente capítulo enumeramos los diferentes factores de $\mathcal{TOY}(\mathcal{FD}i)$ que actualmente están produciendo una penalización en el rendimiento del sistema. Como trabajo futuro, nos proponemos realizar una nueva implementación más eficiente de $\mathcal{TOY}(\mathcal{FD}i)$ para superar estos factores.

Capítulo 5

Conclusiones y trabajo futuro

Concluimos este trabajo enunciando las principales conclusiones obtenidas que permiten trazar del trabajo futuro que deberá seguir nuestra línea de investigación.

5.1. Conclusiones

- Hemos integrado la tecnología de resolución de restricciones sobre dominios finitos ILOG CP 1.4 dentro del sistema lógico funcional con restricciones $\mathcal{TOY}$. Hasta ahora, el sistema $\mathcal{TOY}$ utilizaba el sistema de restricciones $\mathcal{FD}$ externo proporcionado por la biblioteca clpfd de SICStus Prolog, dando lugar al sistema $\mathcal{TOY}(\mathcal{FD}s)$. Con la integración descrita en este trabajo, $\mathcal{TOY}$ puede utilizar como sistema de restricciones $\mathcal{FD}$ externo el proporcionado por las bibliotecas ILOG Concert 2.6 e ILOG Solver 6.6, dando lugar al sistema $\mathcal{TOY}(\mathcal{FD}i)$.

- Mientras que la comunicación entre $\mathcal{TOY}$ y clpfd se producía de manera natural, al estar ambos implementados en SICStus Prolog, la comunicación entre $\mathcal{TOY}$ e ILOG CP requiere gran cantidad de código interfaz, ya que ILOG Concert e ILOG Solver precisan un acesso mediante bibliotecas C++.

- Se requiere código interfaz para la interconexión de la arquitectura de componentes del sistema $\mathcal{TOY}(\mathcal{FD}i)$. SICStus permite comunicar a una aplicación SICStus (en nuestro caso $\mathcal{TOY}$) con una aplicación C++ (ILOG CP) mediante la creación de un recurso ajeno que vincule predicados Prolog a funciones C++.

  La creación de este tipo de recursos ajenos no es intuitiva, ya que requiere comprender la conversión de parámetros entre SICStus y C++ y requiere conocer los comandos internos utilizados en la creación de un recurso ajeno. Precisa un alto conocimiento tanto del sistema $\mathcal{TOY}$ como de la aplicación ILOG CP. Esto permite determinar:

  - Los parámetros concretos con los que configurar la creación del recurso ajeno.
  - La manera exacta de cargar este recurso ajeno en el sistema $\mathcal{TOY}$.

- Se requiere código interfaz para gestionar las restricciones $\mathcal{FD}$ del objetivo $\mathcal{TOY}$ utilizando la tecnología ILOG CP. Es fundamental una doble mentalidad, que permita compaginar el funcionamiento declarativo de un sistema como SICStus con el desarrollo de la aplicación imperativa ILOG CP. Esto puede verse a muy diferentes niveles:

1. Diferencia de filosofía.

   Por un lado, ILOG está diseñado para un aislamiento entre la fase de modelado y la fase de resolución. Por otro lado, la semántica operacional de $\mathcal{TOY}$ se basa en un cálculo de estrechamiento perezoso. Así, cada vez que se gestiona una nueva restricción $\mathcal{FD}$ de un objetivo $\mathcal{TOY}$, se produce su modelado y resolución en la aplicación ILOG CP. Hemos descartado el procedimiento por defecto utilizado por ILOG para realizar el modelado y resolución y hemos optado por utilizar métodos alternativos que se adapten mejor a la semántica operacional de $\mathcal{TOY}$.

2. Carencias en la comunicación.

   Aunque es posible utilizar cualquier término Prolog dentro de una función C++, existen casos en los que es imposible transmitir información desde $\mathcal{TOY}$ a la aplicación ILOG CP. Es el caso de la traducción y transmisión de restricciones $\mathcal{FD}$, donde la propia aplicación ILOG CP no es capaz de reconocer una misma variable lógica $\mathcal{FD}$ involucrada en dos restricciones diferentes.
   Para solventar estas carencias hemos almacenado en $\mathcal{TOY}$ la información imposible de transmitir. De este modo, cada predicado Prolog que gestiona una restricción $\mathcal{FD}$ del objetivo $\mathcal{TOY}$:
   - Accede a la conjunción de restricciones $\mathcal{FD}$ detectada hasta el momento por el cálculo de estrechamiento perezoso.
   - Utiliza esa información para traducir y transmitir a la aplicación ILOG CP la restricción que está gestionando.
   - Añade la restricción gestionada a la conjunción de restricciones $\mathcal{FD}$ y almacena esta nueva conjunción para que sea utilizada por futuros predicados Prolog.

   Para alcanzar este comportamiento, en el sistema $\mathcal{TOY}(\mathcal{FD}i)$ hemos reutilizado el almacén `Cin` incluido en la implementación del sistema $\mathcal{TOY}(\mathcal{FD}s)$.

   Además, como el sistema $\mathcal{TOY}$ permite utilizar funciones indeterministas, hemos comprobado que para poder efectuar el backtracking sobre la aplicación ILOG CP, necesitamos actualizar el contenido de las conjunciones de restricciones $\mathcal{FD}$' y $\mathcal{FD}$" a partir del nuevo contenido de la conjunción de restricciones $\mathcal{FD}$. Cuando se produce el backtracking a un punto de elección anterior, se restaura automáticamente el contenido de la conjunción $\mathcal{FD}$, implementada en Prolog. Las conjunciones $\mathcal{FD}$ y $\mathcal{FD}$', implementadas en C++, no se actualizan automáticamente, sino que las debemos modificar para mantener la consistencia con la nueva conjunción $\mathcal{FD}$.

   Aunque esta conjunción $\mathcal{FD}$ es necesaria, supone una mayor dificultad en el desarrollo y mantenimiento del sistema $\mathcal{TOY}(\mathcal{FD}i)$, además de una penalización en su rendimiento.

   Justificada la necesidad de la conjunción de restricciones $\mathcal{FD}$, hemos justificado la replicación de información entre las conjunciones de restricciones $\mathcal{FD}$ y

$\mathcal{FD}$', contenidas en la aplicación ILOG CP. Hemos comparado esta alternativa con otra que utilice únicamente la conjunción de restricciones $\mathcal{FD}$". Hemos concluido que por eficiencia, simpleza y estandarización conviene utilizar las conjunciones $\mathcal{FD}$' y $\mathcal{FD}$".

- Hemos mantenido la independencia entre los distintos sistemas de restricciones del sistema $\mathcal{TOY}$. De este modo, hemos mantenido las restricciones de igualdad y desigualdad sintáctica en el sistema de restricciones $\mathcal{H}$, y hemos optado por sincronizar posteriormente con la aplicación ILOG CP las restricciones de igualdad y desigualdad sintáctica que involucran exclusivamente variables lógicas $\mathcal{FD}$.

  Esto ha supuesto un esfuerzo importante, que se ha traducido en una reforma en las estructuras de datos de la conjunción de restricciones $\mathcal{FD}$. Además, por cada nueva restricción $\mathcal{FD}$ que se gestiona, debemos determinar las posibles restricciones de igualdad y desigualdad que sea necesario sincronizar. Concretamente, por cada nueva restricción $\mathcal{FD}$ a gestionar comprobamos:

  i) Posibles restricciones de desigualdad a sincronizar.

  ii) Posibles restricciones de unificación entre una variable lógica $\mathcal{FD}$ y un valor entero.

  iii) Posibles restricciones de unificación entre dos variables lógicas $\mathcal{FD}$.

  Tras estudiar el rendimiento sobre algunos ejemplos, concluimos que esta forma de proceder no merece la pena, ya que:

  - Supone una altísima penalización en el rendimiento del sistema $\mathcal{TOY}(\mathcal{FD}i)$.
  - Existen ejemplos en los que el número de sincronizaciones realizadas es muy escaso, o nulo.

- Para mantener la consistencia entre las conjunciones de restricciones $\mathcal{FD}$, $\mathcal{FD}$' y $\mathcal{FD}$" debemos sincronizar las `IlcIntVar`s acotadas con sus variables lógicas $\mathcal{FD}$ asociadas. Para ello hemos ampliado el repertorio de restricciones de ILOG. Hemos estudiado los elementos que componen la creación de una nueva restricción y los pasos necesarios para llevarlo a cabo.

  Aunque el repertorio de restricciones ofrecido en la herramienta ILOG OPL Studio es muy amplio, el repertorio de restricciones existente en las bibliotecas ILOG Concert e ILOG Solver es mucho menor. Concluimos que la creación de nuevas clases de `IloConstraint`s es una posible forma de aumentar la expresividad del sistema $\mathcal{TOY}(\mathcal{FD}i)$, haciéndola comparable a la del sistema $\mathcal{TOY}(\mathcal{FD}s)$.

- Hemos integrado en el sistema $\mathcal{TOY}(\mathcal{FD}i)$ dos procedimientos de búsqueda predefinidos por ILOG Concert e ILOG Solver. Hemos adaptado los métodos proporcionados por ILOG para la búsqueda de soluciones a nuestro contexto de ejecución. De este modo, encontramos las distintas soluciones al etiquetado a medida que son solicitadas por el usuario.

- Concluimos por tanto que la versión del sistema $\mathcal{TOY}(\mathcal{FD}i)$ descrita en este trabajo representa una propuesta sólida para la utilización de ILOG CP 1.4 como sistema de restricciones $\mathcal{FD}$ externo del sistema $\mathcal{TOY}$. Sin embargo, el punto en que se encuentra la implementación de $\mathcal{TOY}(\mathcal{FD}i)$ es aún muy incipiente como para obtener un

rendimiento competitivo con el del sistema $\mathcal{TOY}(\mathcal{FD}s)$. Se precisa una nueva versión mejorada, que goce de una mayor expresividad para abordar un mayor repertorio de ejemplos y una gestión de sus restricciones más eficiente, para obtener unos tiempos competitivos en la resolución de estos ejemplos.

5.2. Trabajo futuro

- Como trabajo inmediato nos planteamos implementar esta nueva versión del sistema $\mathcal{TOY}(\mathcal{FD}i)$:
  - Implementaremos nuevas clases de `IloConstraint`s que amplíen la expresividad del sistema $\mathcal{TOY}(\mathcal{FD}i)$, para poder hacer frente a un mayor número de ejemplos.
  - Implementaremos nuevos predicados para las restricciones de igualdad y desigualdad sintáctica, que permitan imponer directamente sobre `model` las restricciones que involucren exclusivamente variables lógicas $\mathcal{FD}$.
  - Realizaremos un estudio comparativo para la sincronización de las `IloIntVar`s acotadas con sus variables lógicas $\mathcal{FD}$ asociadas. Comprobaremos si es más eficiente realizar esta sincronización en C++ sobre variables `SP_term_ref` o si es mejor realizar esta sincronización definiendo un nuevo predicado Prolog.
  - Añadiremos el tamaño del vector `vars_synchronized` a la información contenida en la conjunción de restricciones $\mathcal{FD}$, para hacer más eficaz el backtracking.
- Ampliaremos el desarrollo de los procedimientos de búsqueda utilizados. Por un lado, permitiremos definir al usuario su propio procedimiento de búsqueda, para explotar al máximo el conocimiento que pueda tener del problema a resolver. Por otro lado, utilizaremos el procedimiento de búsqueda como caja transparente, de manera que el usuario pueda intervenir dinámicamente en las decisiones adoptadas por el resolutor durante la búsqueda.
- Haremos uso de las clases para depuración que ofrece ILOG, de manera que cuando una conjunción de restricciones $\mathcal{FD}$" no sea satisfactible, se pueda informar al usuario de las restricciones $\mathcal{FD}$ asociadas que están impidiendo esa satisfactibilidad.
- Más adelante, integraremos los resolutores ILOG Scheduler 6.6 e ILOG Dispatcher 4.6, pertencecientes también a la tecnología ILOG CP 1.4. De este modo resolveremos de una manera más específica conjunciones de restricciones $\mathcal{FD}$ orientadas a la planificación de calendarios para proyectos y a la planificación de rutas.
- Siguiendo el esquema propuesto en [8] para la combinación de los distintos resolutores del sistema $\mathcal{TOY}$ a la hora de modelar y resolver problemas heterogéneos, integraremos la tecnología ILOG CPLEX 12.1 en el sistema de restricciones $\mathcal{TOY}(\mathcal{R})$. Esto nos permitirá reproducir el esquema propuesto en [8], trabajando sobre la tecnología ILOG en lugar de SICStus Prolog. Además de la mejora de rendimiento que ILOG ofrece, utilizaremos los procedimientos de búsqueda de como caja transparente de ILOG para ampliar la intervención del usuario en la cooperación de los distintos resolutores.

- Finalmente, estudiaremos la implementación de un nuevo resolutor especializado en el cálculo simbólico de lambda expresiones sencillas. Nuestro objetivo será integrarlo dentro de nuestro sistema $\mathcal{TOY}$ y estudiar ejemplos que relacionen estas lambda expresiones simbólicas con el resolutor $\mathcal{TOY}(\mathcal{R})$.

Bibliografía

[1] P. Arenas, S. Estévez, A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo, and F. Sáenz-Pérez. $\mathcal{TOY}$. a multiparadigm declarative language. version 2.3.1., 2007. R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.

[2] B-Prolog. B-Prolog. `http://www.probp.com/`.

[3] F. Baader and K. Schulz. On the combination of symbolic constraints, solution domains and constraints solvers. In *CP'95*, volume 976 of *LNCS*, pages 380–397. Springer, 1995.

[4] F. Benhamou. Heterogeneous constraint solving. In *ALP'96*, volume 1139 of *LNCS*, pages 62–76. Springer, 1996.

[5] Ciao. Ciao. `http://clip.dia.fi.upm.es/Software/Ciao/`.

[6] R. G. del Campo and F. Sáenz-Pérez. Programmed search in a timetabling problem over finite domains. *Electr. Notes Theor. Comput. Sci.*, 177: pages 253–267, 2007.

[7] ECLiPSe. ECLiPSe. `http://87.230.22.228/`.

[8] S. Estévez-Martín, A. Fernández, M. Hortalá-González, F. Sáenz-Pérez, M. Rodríguez-Artalejo, and R. del Vado-Vírseda. On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. *Theory Pract. Log. Program.*, 9(4): pages 415–527, 2009.

[9] S. Estévez-Martín, A. J. Fernández, and F. Sáenz-Pérez. About implementing a constraint functional logic programming system with solver cooperation. In *proc. of CICLOPS'07*, pages 57–71, 2007.

[10] A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *Theory Pract. Log. Program.*, 7(5): pages 537–582, 2007.

[11] Gecode. Gecode. `http://www.gecode.org/`.

[12] GNU Prolog. GNU Prolog. `http://www.gprolog.org/`.

[13] L. Granvilliers, E. Monfroy, and F. Benhamou. Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter*, 14(2), 2001.

[14] M. Henz and T. Müller. An overview of finite domain constraint programming. In *5th Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore, 2000.

[15] P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Technischen Universität Dresden, Fakultät Informatik, 2001.

[16] P. Hofstedt and P. Pepper. Integration of declarative and constraint programming. *Theory Pract. Log. Program.*, 7(1-2): pages 93–121, 2007.

[17] ILOG. ILOG Solver 6.6, Reference Manual, 2008.

[18] ILOG. ILOG OPL Studio 6.1, Reference Manual, 2009.

[19] ILOGCP14. ILOGCP14. `http://www.ilog.com/products/cp/`.

[20] J. Jaffar and J. Lassez. Constraint logic programming. In *Proc. POPL'87*, pages 111–119. ACM Press, 1987.

[21] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19&20: pages 503–581, 1994.

[22] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. Semantics of constraints logic programs. *Journal of Logic Programming*, 37(1-3): pages 1–46, 1998.

[23] F. López-Fraguas, M. Rodríguez-Artalejo, and R. del Vado-Virseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *PPDP'04*, pages 43–54. ACM Press, 2004.

[24] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *RTA*, volume 1631 of *LNCS*, pages 244–247. Springer, 1999.

[25] M. Marin, T. Ida, and W. Schreiner. CFLP: a Mathematica Implementation of a Distributed Constraint Solving System. In *IMS'99*, volume 8, pages 287–300. WIT Press, 2001.

[26] K. Marriot and P. J. Stuckey. *Programming with constraints*. The MIT Press, Cambridge, Massachusetts, 1998.

[27] Microsoft, 2005. `http://msdn.microsoft.com/en-us/visualc/default.aspx`.

[28] Minion. Minion. `http://minion.sourceforge.net/`.

[29] E. Monfroy. *Solver collaboration for constraint logic programming*. PhD thesis, Centre de Recherche en Informatique de Nancy, 1996.

[30] E. Monfroy. A Solver Collaboration in BALI. In *Proc. of JCSLP'98*, pages 349–350. MIT Press, 1998.

[31] E. Monfroy and C. Castro. A component language for hybrid solver cooperations. In *ADVIS'04*, volume 3261 of *LNCS*, pages 192–202. Springer, 2004.

[32] SICStus Prolog. Using SICStus Prolog with newer Microsoft C compilers. `http://www.sics.se/isl/sicstuswww/site/dontpanic.html`.

[33] SICStus Prolog, 2007. `http://www.sics.se/isl/sicstus`.

[34] SWI. SWI. `http://www.swi-prolog.org/`.